A Learning Guide to R

Beginner to intermediate skills in data analysis, visualization, and manipulation

Remko Duursma, Jeff Powell & Glenn Stone

WESTERN SYDNEY UNIVERSITY

Hawkesbury Institute for the Environment

Prologue

This book is not a complete introduction to statistical theory. It should also not be the first statistics book you read. Instead, this book shows you how to implement many useful data analysis routines in **R**. Sometimes we explain a bit of theory behind the method, but this is an exception. We really assume that you have already learned statistics elsewhere and are here to learn how to actually do the analyses.

We have learned from experience that data practictioners, that means scientists and everyone else who has an interest in learning from data, do not learn statistical analyses by studying the underlying theory. They learn from doing, and from using examples written by others who have learned by doing. For this reason the book in front of you is largely a compendium of examples. We have chosen the examples and the exercises in the hope that they resemble real-world problems.

We have designed this book so it can be used for self-study. The exercises at the end of each chapter aim to test important skills learned in the chapter. Before you start on the exercises, read through the text, try to run the examples, and play around with the code making small modifications. We have also placed many *Try it yourself* boxes throughout the text to give you some ideas on what to try, but try to be creative and modify the examples to see what happens.

You can download the example datasets used in this book (see Appendix A), as well as the solutions to the exercises at `bit.ly/RDataCourse`.

An index with the functions and packages used is provided at the end of this book.

What is data analysis?

Most scientists would answer that this is the part of the work where the p-values are calculated, perhaps alongside a host of other metrics and tables. We take the view that *data analysis includes every step from raw data to the outputs*, where outputs include figures, tables and statistics. This view is summarized in the figure on the next page.

Because every step from raw data to output affects the outcome, every step should be well documented, and reproducible. By 'reproducible', we mean it should be possible for some other person (even if this is the future you) to re-run the analysis and get the same results. We agree with many others that **R** is particularly good for reproducible research of this kind [1], because it has utilities for all steps of data analysis.

We stress that it is absolutely vital to a successful data analysis to visualize at every step of the analysis [2]. When analyzing data, you should apply a continuous loop of statistical inference and visualizing the data (Find a significant effect you did not expect? Visualize it!; Visually find something interesting? Analyze it!).

Assumed skills of the reader

This book is written for scientists and data analysts who do not have a background in computer science, mathematics, or statistics. The astute **R** programmer might notice that some terminology has been simplified, for reasons of clarity. Experience with programming in any language is therefore not required. We do assume that the reader has completed at least an *Introduction to Statistics* course, as this book does not explain the fundamentals of statistical analysis.

[1]Gandrud, C. Reproducible Research with R and R Studio. CRC Press, 2015. 2nd Edition.
[2]Anscombe 1973 "Graphs in Statistical analysis", The American Statistician 27:17-21

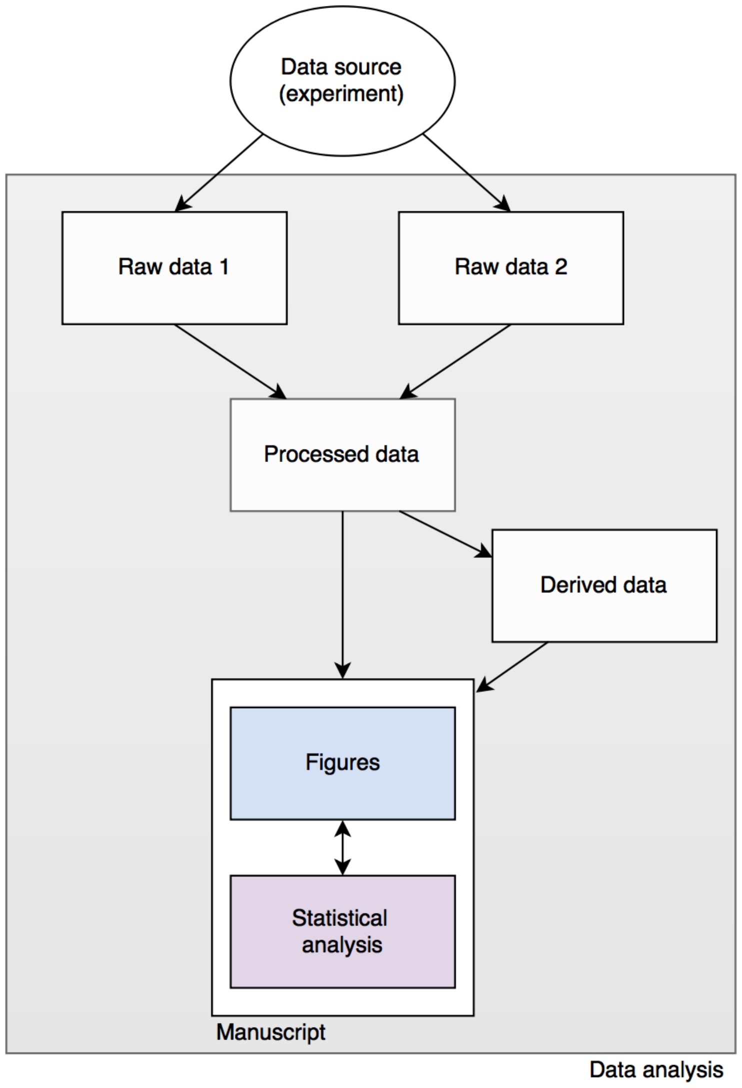


Figure 1: A broader view of data analysis. The view we take in this book is that all operations on the data should be viewed as part of data analysis. Usually, only the statistical analysis is viewed as the analysis, but in many applications there are numerous steps from the initial raw data (produced from some data source, perhaps an experiment or an observational study) leading to the eventual figures and statistics. One of these steps is the combining of several raw data files into a processed dataset, which may also include data cleaning, calculation of new variables, renaming of columns, reshaping, and so on. Derived datasets are common as well, and these include summary variables (e.g. averages by treatment, totals over some time period). The process of visualizing the data via figures, and testing for significant effects and other statistical procedures is a two-way street: figures inform the statistics to be applied and vice versa (uncovering statistical significance on some effect mandates visualizing the effect). We stress to all data practitioners that all steps of data analysis are implemented in such a way to be fully reproducable, and documented in its entirety.

If you have used **R** before, you will very likely still learn many new skills, and perhaps understand the more basic skills a little better.

About the Authors

Remko Duursma was an Associate Professor at the Hawkesbury Institute for the Environment, Western Sydney University, and is now searching for a new direction as an independent data scientist. His research at HIE focused on understanding vegetation function and structure in response to the environment, using models and data. In his research, he routinely analyses large datasets arising from sensor networks, climate change experiments, literature syntheses, and other sources. He has been using R since 2001.

Jeff Powell is an Associate Professor based at the Hawkesbury Institute for the Environment at Western Sydney University. His research interests centre on understanding community assembly processes, particularly in the context of soil ecology and plant-microbe interactions, and relating these to ecosystem function. He utilises a variety of approaches to ecological informatics approach in his research, developing new tools and extending existing tools to test ecological hypotheses with large datasets from surveys of ecological communities, ecosystem properties, and species and environmental attributes. He has been using R since 2005.

Glenn Stone is a statistician and data scientist currently working in the private sector. Previously he has been a Professor of Data Science in the Centre for Research in Mathematics at Western Sydney University and, before that, a Principal Research Analyst with Insurance Australia Group and a Senior Research Statistician with the CSIRO. His research interests are in computational statistics and data science, as applied to a wide variety of scientific and commercial applications. He has been using R since 2003.

Acknowledgments

This book accompanies the five-day course 'Data analysis and visualization with R', taught twice a year at the Hawkesbury Institute for the Environment (`www.westernsydney.edu.au/hie`). The PDF of this book is available at `www.westernsydney.edu.au/rmanual`. The class website can be reached at `bit.ly/RDataCourse`.

We have used `knitr` (with Sweave and LaTeX) (see `http://yihui.name/knitr/`) to produce this book, from within Rstudio (`www.rstudio.com`). We also use numerous add-on packages. Thanks to the developers for making these tools freely available.

Thanks also to those who have contributed original datasets that are used throughout this book (see Appendix A).

September 17, 2019

Contents

Chapter 1

Basics of R

1.1 Installing R and so on

If you are reading this at the HIE R course, the computers in this room already have **R** and RStudio installed. There is no need to update. Do take note of the recommended settings in RStudio discussed in Section 1.1.2.

1.1.1 R installation

Throughout this book, we assume you use RStudio. However, you still have to install **R**. RStudio is a program that runs **R** for us, and adds lots of functionality. You normally don't have to open the **R** program directly, just use RStudio (see next Section 1.1.2).

To install **R** on your system, go here : `http://cran.r-project.org`, click on the download link at the top, and then 'base', and finally download the installer.

Run the installer, and simply press OK on all windows (since you will be using RStudio, the settings here don't matter).

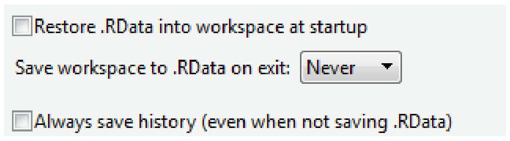
It is a good idea to install newer versions of **R** when they become available. Simply repeat this process or see another solution in Section 1.11.

1.1.2 Using and installing RStudio

We will use RStudio throughout this course (but see note above, you need to install **R** first). To download RStudio, go here: `www.rstudio.org` to download it (Windows or Mac).

Take some time to familiarize yourself with RStudio. When you are using a new installation of RStudio, the default behaviour is to save all your objects to an 'RData' file when you exit, and loads the same objects when you open RStudio. This is very dangerous behaviour, and you must turn it off. See Section 1.13 for more information on 'RData' files. For now, make sure you go to Tools > Global Options... and on the General tab, make sure the settings are like the figure below.

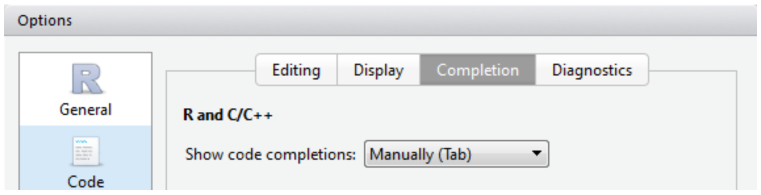
Another feature you may want to turn off is the automatic code completion, which is now a standard feature in RStudio. If you are using an older version of RStudio, this won't apply (and you won't be able to find the settings in the figure below).


Restore .RData into workspace at startup

Save workspace to .RData on exit: Never

Always save history (even when not saving .RData)

Figure 1.1: Settings in Tools/Global Options/General, to prevent automatically loading objects from a previous session. This contaminates your workspace, causing problems that are difficult to spot.



R and C/C++

Show code completions: Manually (Tab)

Figure 1.2: Settings in Tools > Global Options > Code > Completion to avoid automatic completion of your code, which opens hint windows as you type. Some people find this helpful, others find it annoying and distracting.

1.2 Basic operations

1.2.1 R is a calculator

When you open the **R** console, all you see is a friendly > staring at you. You can simply type code, hit Enter, and **R** will write output to the screen.

Throughout this tutorial, **R** code will be shown together with output in the following way:

```
# I want to add two numbers:
1 + 1

## [1] 2
```

Here, we typed `1 + 1`, hit Enter, and **R** produced 2. The `[1]` means that the result only has one element (the number '2').

In this book, the **R** output is shown after `##`. Every example can be run by you, simply copy the section (use the text selection tool in Adobe reader), and paste it into the console (with Ctrl+Enter on a Windows machine, or Cmd+Enter on a Mac).

We can do all sorts of basic calculator operations. Consider the following examples:

```
# Arithmetic
12 * (10 + 1)

## [1] 132

# Scientific notation
3.5E03 + 4E-01

## [1] 3500.4

# pi is a built-in constant
sin(pi/2)

## [1] 1

# Absolute value
```

```
abs(-10)

## [1] 10

# Yes, you can divide by zero
1001/0

## [1] Inf

# Square root
sqrt(225)

## [1] 15

# Exponents
15^2

## [1] 225

# Round down to nearest integer (and ceiling() for up or round() for closest)
floor(3.1415)

## [1] 3
```

Try typing `?Math` for description of more mathematical functions.

Also note the use of # for comments: anything after this symbol on the same line is *not* read by **R**. Throughout this book, comments are shown in green.

1.3 Working with scripts and markdown files

When working with **R**, you can type everything into the console, just as you did in the last few examples. However, you've probably already noticed this has some disadvantages. It's easy to make mistakes, and annoying to type everything over again to just correct one letter. It's also easy to loose track of what you've written. As you move on to working on larger, more complicated projects (in other words, your own data!) you will quickly find that you need a better way to keep track of your analyses. (After all, have you ever opened up a spreadsheet full of data six months after you started it, and spent the whole day trying to reconstruct just what units you used in column D?)

Luckily **R** and RStudio provide a number of good ways to do this. In this course we will focus on two, scripts and markdown documents.

1.3.1 R scripts

Scripts offer the simplest form of repeatable analysis in **R**. Scripts are just text files that contain code and comments. Script files should end in `.R`.

In RStudio, open a new script using the `File` menu: File > New File > R Script, and save it in your current working directory with an appropriate name (for example, 'rcourse_monday.R'). (Use File > Save, note that your working directory is the default location).

A brand new script is completely empty. It's a good idea to start out a new script by writing a few comments:

```
# HIE R Course - Monday
# Notes by <your name here>
# <today's date>
# So far we've learned that R is a big calculator!
1 + 1
```

As we mentioned in Section 1.2.1, the # symbol indicates a comment. On each line, **R** does not evaluate anything that comes after #. Comments are great for organizing your code, and essential for reminding yourself just what it was you were intending. If you collaborate with colleagues (or your supervisor), you'll also be very grateful when you see comments in their code – it helps you understand what they are doing.

Try this yourself Sometimes, you might want to write code directly into the console, and add it to a script later, once it actually works as expected. You can use the History tab in RStudio to save some time. There, you can select one or more lines, and the button To Source will copy it to your script.
Try it now. Select one line by clicking on it, and send it to your script file using `To Source`.
You can use this feature to add all of the examples we typed in Section 1.2.1 to your notes. To select more than one line at a time, hold down Shift. You can then send all the examples to your new script file with one click.

1.3.2 Using scripts to program with objects

It gets more interesting when we introduce objects. Objects are named variables that can hold different values. Once you have to keep track of objects, you'll naturally want to use scripts so that you can easily remember what value has been assigned to what object. Try adding these examples to your sample script:

```
# Working with objects

# Define two objects
x <- 5
y <- 2 * x

# ... and use them in basic a calculation.
x + y
```

Note the use of `<-` : this is an operator that assigns some content to an 'object'. The arrow points from the content to the object. We constructed two objects, `x` and `y`.

Now let's run the script. You can run your entire script by clicking the `Source` button in the top right. Or, more conveniently, you can select sections of your script with the mouse, and click Run, or by pressing Ctrl+Enter (Cmd+Enter on a Mac). If nothing is selected, the current line of code will be executed. The results from running the script above look like this (note that the comments are sent to the console along with the code):

```
# Working with objects

# Define two objects
x <- 5
y <- 2 * x

# ... and use them in a basic calculation.
x + y
```

```
## [1] 15
```

Objects you've created (in this case, x and y) remain in memory, so we can reuse these objects until we close **R**. Notice that the code and comments are echoed to the console, and shown along with the final results.

Let's add some more calculations using x and y:

```
# ... a few more calculations:
x * y
y / x
(x * y) / x
```

Run your script by clicking source or highlighting a few lines and typing Ctrl-Enter or Cmd-Enter. You should get these results:

```
## [1] 50
## [1] 2
## [1] 10
```

If you like, try adding some of your own calculations using x and y, or creating new objects of your own.

You can also assign something else to the object, effectively overwriting the old `x`.

```
# Reassigning an object
# Before:
message(x)

## 5

x <- "Hello world"
# After:
message(x)

## Hello world
```

Here we assigned a character string to `x`, which previously held an numerical value. Note that it is not a problem to overwrite an existing object with a different type of data (unlike some other programming languages).

Try this yourself Note that RStudio has four panes: the R script, the R console, Files/Plots/Packages/Help and Environment/History. You can change the placement of the panes in Tools > Global options... > Pane layout. Change the layout to your liking.
You can also change the appearance of code in the script pane and the R console pane. Take a look at the styles in Tools > Global Options... > Appearance, and then pick one in the Editor theme list.

1.3.3 Working with markdown files

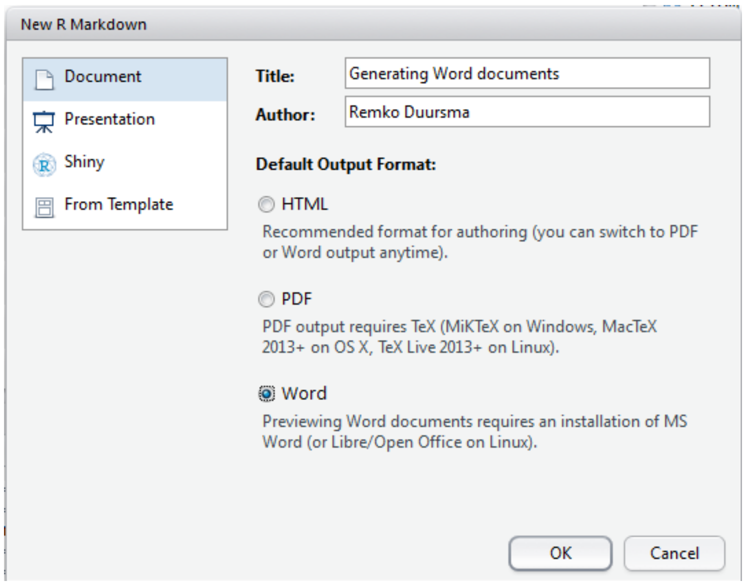
Script files are good for simple analyses, and they are great for storing small amounts of code that you would like to use in lots of different projects. But, they are not the best way to share your results with others. Most of all, R scripts, no matter how well commented, are not suitable for submission as journal articles. For that, we need to incorporate all our exciting results and beautiful figures into a document with an introduction, methods, results, and discussion and conclusions.

Fortunately, there is an easy way to do this. It also makes a great way to keep track of your work as you are developing an analysis. This is know as an R markdown file. Markdown is a simple set of rules for formatting text files so that they are both human-readable and processable by software. The `knitr` package (as used by RStudio) will take a markdown-formatted text file and produce nicely-formatted output. The output can be a Word document, HTML page, or PDF file. If you need to install `knitr`, instructions for installing new packages can be found in Section 1.10. In the next example, we will show you how to use R markdown to output a word file. Word files are excellent for collaborating with your colleagues when writing manuscripts. HTML files can also be very useful – RStudio produces them quickly, and they are great for emailing and viewing online. They can also be used to present results in an informal meeting. PDF files are good for publishing finished projects. However, producing PDFs using RStudio and knitr requires installing other packages, which we won't cover here (see the Further Reading for more suggestions if you are interested in doing this.)

Further reading To make PDFs using R markdown, you will need to install additional software. This software interfaces with a typesetting environment called LaTeX. On Windows, you will need to install MiKTeX (`miktex.org`, for instructions see `miktex.org/howto/install-miktex`). On Mac OS, you will need MacTeX (for both instructions and a download link see `tug.org/mactex/mactex-download.html`). On Linux, you will need to install TeX Live (see `www.tug.org/texlive/`). The other packages needed are automatically downloaded when you install RStudio. Once you have this software, `http://rmarkdown.rstudio.com/pdf_document_format.html` has a nice explanation of different features you can add to your document using this format.

In addition to Word, HTML, and PDF, it is also possible to create many other types of files with R markdown. You can for instance, make slideshows (select File > New File > R Markdown... then choose Presentation from the list of file types on the left). You can also make interactive documents, websites, and many other things using templates available in various **R** packages. See `http://rmarkdown.rstudio.com/` for tutorials on how to take advantage of this simple, but powerful way of writing documents.

RStudio offers a handy editor for markdown files. Start a new markdown file by choosing File > New File > R Markdown.... You will see the following dialogue box:



Title: Generating Word documents

Author: Remko Duursma

Default Output Format:

HTML
Recommended format for authoring (you can switch to PDF or Word output anytime).

PDF
PDF output requires TeX (MiKTeX on Windows, MacTeX 2013+ on OS X, TeX Live 2013+ on Linux).

[✓ Word]
Previewing Word documents requires an installation of MS Word (or Libre/Open Office on Linux).

Figure 1.3: Go to File > New File > R Markdown, enter a title for your document and select Word document.

The new R markdown document (which is just a text file with the extension `.Rmd`) already contains some example code. Run this example by clicking the button just above the markdown document `Knit Word` (it will ask you to save the document first if you haven't already).

The example code will generate a new document, which opens in MS Word. You can see that the output contains text, R code, and even a plot.

We suggest using R markdown to organize your notes throughout this course, and especially to organize any analyses that you are producing for scientific projects.

Let's take a look at the example markdown file and see what things need to be changed to make it your own. The first thing in the file is the header. This probably already includes your name, a title, the date you created the file, and the type of output you would like to produce (in this case, a Word document).

```
---
title: "Basic R calculations in markdown"
author: "Remko Duursma"
date: "16 September 2015"
output: word_document
---
```

After the header, you'll see two kinds of text: chunks of R code and regular text.

1.3.4 R code in markdown

The first thing you will see under the header in your new markdown document is a grey box:

~~~
```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
```
~~~

You can delete the rest of the example code that came with the file, but keep this box. It contains a chunk of R code to set the options for the markdown document and the software that processes it (a package known as `knitr`). Don't worry too much about these options right now. Instead, notice that the chunk starts and ends with three accent characters (found to the left of the numeric keys on QWERTY keyboards):

~~~
```
~~~

This tells the software that you are starting a chunk of R code. You can use this syntax to add your own chunks of code to the markdown file. At the start of the chunk, you have several options – you can give the chunk a name (for example, `setup`), you can set whether to show or not show the code (`echo`), and whether or not to evaluate code (`eval`). There are other options, but these are the ones you will use the most.

Table 1.1: Options for code chunks within an markdown document.

Option	What it sets	Possible values
`echo`	Should the code be shown in the final document?	`TRUE, FALSE`
`eval`	Should the results be shown in the final document?	`TRUE, FALSE`

Try this yourself Make code chunks that vary the options `echo` and `eval`. For instance, try `{r echo=TRUE, eval=FALSE}` and compare the results with `{r echo=FALSE, eval=TRUE}`. Can you imagine situations where you would want to use one or the other?

Within the code chunks themselves, you can type R code just as you normally would. `#Comments` are treated as comments, objects are treated as objects, and expressions are evaluated and values are returned.

1.3.5 Body text in markdown

What makes markdown distinct from a script is the flexibility you have to create much larger and more descriptive text blocks (and in RStudio, you can spell check them, too!) This allows you to write documents that contain complex analyses and results that a ready to share.

All of the text outside of the R code chunks is interpreted as body text. Markdown is designed to be simple to use, but you may find the first few symbols you see a bit mysterious. Here's a quick guide to what's going on:

Table 1.2: Basic markdown formatting

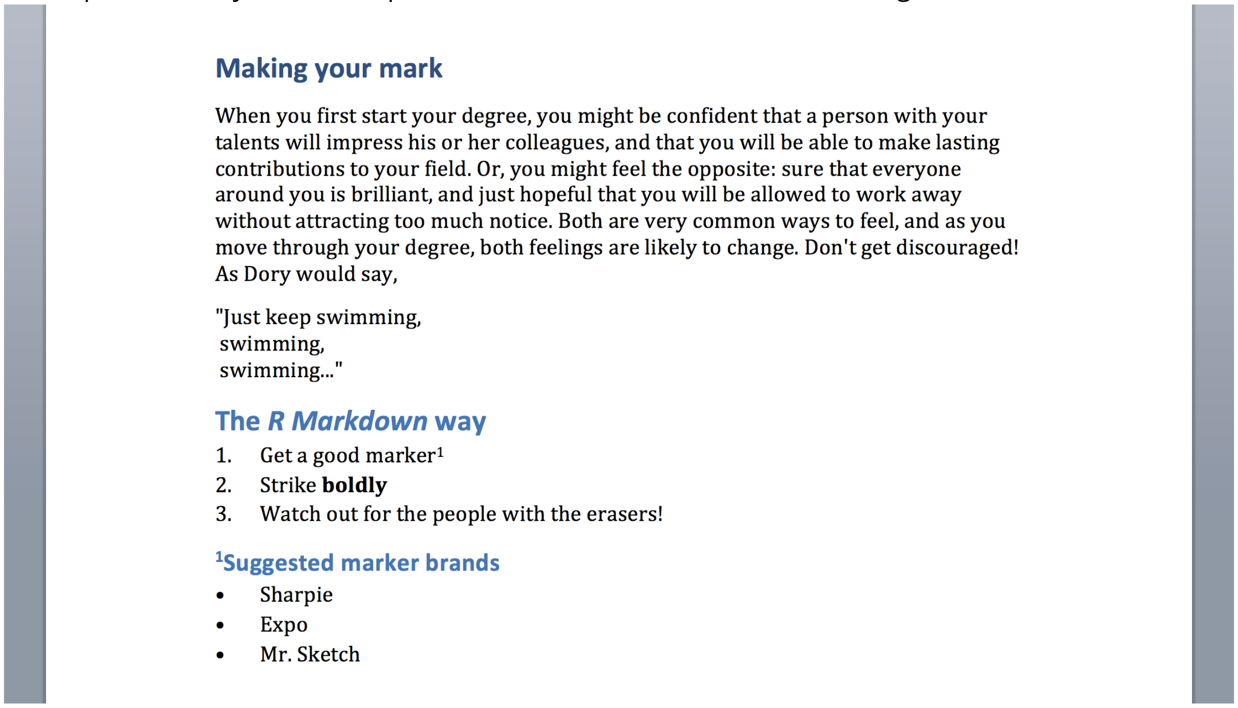
Formatting option	Symbols	Example
Headings	`#`	`#Example Heading`
Subheadings	`##`	`##Subheading`
Bold	`**`	`**bold text**`
Italic	`*`	`*italic text*`
Strike through	`~~`	`~~crossed-out text~~`
Superscript	`^`	`x^2^`
Subscript	`~`	`CO~2~`
Bulleted lists	`*`	`* A list item`
		`* Another list item`
		`* Yet another list item`
Numbered lists	`1.`	`1.  First list item`
		`2.  Second list item`
		`3.  Third list item`
Horizontal rule	three or more -	`----`
Line break	two or more spaces plus return	

Here's a silly example that combines headings, emphasis, lists, superscripts, and line breaking:

```
# Making your mark

When you first start your degree, you might be confident that a person with your talents
will impress his or her colleagues, and that you will be able to make lasting
contributions to your field. Or, you might feel the opposite: sure that everyone around
you is brilliant, and just hopeful that you will be allowed to work away without
attracting too much notice. Both are very common ways to feel, and as you move through
your degree, both feelings are likely to change. Don't get discouraged! As Dory would
say,
   "Just keep swimming,
   swimming,
   swimming..."

## The *R Markdown* way
1. Get a good marker^^1^^
2. Strike **boldly**
3. Watch out for the people with the erasers!
```

```
###^^1^^Suggested marker brands
* Sharpie
* Expo
* Mr. Sketch
```

When processed by knitr and opened in Word, the results look something like this:



Making your mark

When you first start your degree, you might be confident that a person with your talents will impress his or her colleagues, and that you will be able to make lasting contributions to your field. Or, you might feel the opposite: sure that everyone around you is brilliant, and just hopeful that you will be allowed to work away without attracting too much notice. Both are very common ways to feel, and as you move through your degree, both feelings are likely to change. Don't get discouraged! As Dory would say,

"Just keep swimming,
swimming,
swimming..."

The *R Markdown* way

1. Get a good marker^1
2. Strike **boldly**
3. Watch out for the people with the erasers!

^{1}Suggested marker brands

- Sharpie
- Expo
- Mr. Sketch

Figure 1.4: A short example of a Word document formatted using markdown.

Try this yourself Experiment with the different types of markdown formatting. If you would like a guide, the file `Ch1_Basics_of_R.Rmd` can be found with your course notes and contains the examples shown in this section. Try rearranging or changing the code in this file.

1.3.6 Putting it together: notebooks in markdown

The real power of markdown, though, is to be able to build notebooks that you can use to combine code and text.

1.4 Working with vectors

A very useful type of object is the `vector`, which is basically a string of numbers or bits of text (but not a combination of both). The power of **R** is that most functions can use a vector directly as input, which greatly simplifies coding in many applications.

Let's construct an example vector with 7 numbers:

```
nums1 <- c(1,4,2,8,11,100,8)
```

We can now do basic arithmetic with this `numeric vector`:

```
# Get the sum of a vector:
sum(nums1)

## [1] 134

# Get mean, standard deviation, number of observations (length):
mean(nums1)

## [1] 19.14286

sd(nums1)

## [1] 35.83494

length(nums1)

## [1] 7

# Some functions result in a new vector, for example:
rev(nums1)  # reverse elements

## [1]   8 100  11   8   2   4   1

cumsum(nums1)  # cumulative sum

## [1]   1   5   7  15  26 126 134
```

There are many more functions you can use directly on vectors. See Table 1.3 for a few useful ones.

Table 1.3: A few useful functions for vectors. Keep in mind that **R** is case-sensitive!

Function	What it does	Example
`length`	Returns the length of the vector	`length(nums1)`
`rev`	Reverses the elements of a vector	`rev(nums1)`
`sort`	Sorts the elements of a vector	`sort(nums1)`
`order`	Returns the order of elements in a vector	`order(nums1)`
`head`	Prints the first few elements of a vector	`head(nums1, 3)`
`max`	Returns the maximum value in a vector	`max(nums1)`
`min`	Returns the minimum value in a vector	`min(nums1)`
`which.max`	Which element of the vector contains the max value?	`which.max(nums1)`
`which.min`	Which element of the vector contains the min value?	`which.min(nums1)`
`mean`	Computes the mean of a numeric vector	`mean(nums1)`
`median`	Computes the median of a numeric vector	`median(nums1)`
`var`	Computes the variance of a vector	`var(nums1)`
`sd`	Computes the standard deviation of a vector	`sd(nums1)`
`cumsum`	Returns the cumulative sum of a vector	`cumsum(nums1)`
`diff`	Sequential difference between elements of a vector	`diff(1:10)`
`unique`	Lists all the unique values of a vector	`unique(c(5,5,10,10,11))`
`round`	Rounds numbers to a specified number of decimal points	`round(2.1341,2)`

Try this yourself Some of the functions in Table 1.3 result in a single number, others give a vector of the same length as the input, and for some functions it depends. Try to guess what the result should look like for the listed functions, and test some of them on the `nums1` vector that we constructed above.

Vectorized operations

In the above section, we introduced a number of functions that you can use to do calculations on a vector of numbers. In **R**, a number of operations can be done on two vectors, and the result is a vector itself. Basically, **R** knows to apply these operations one element at a time. This is best illustrated by some examples:

```
# Make two vectors,
vec1 <- c(1,2,3,4,5)
vec2 <- c(11,12,13,14,15)

# Add a number, element-wise
vec1 + 10

## [1] 11 12 13 14 15

# Element-wise quadratic:
vec1^2

## [1]  1  4  9 16 25

# Pair-wise multiplication:
vec1 * vec2

## [1] 11 24 39 56 75

# Pair-wise division
vec1 / vec2

## [1] 0.09090909 0.16666667 0.23076923 0.28571429 0.33333333

# Pair-wise difference:
vec2 - vec1

## [1] 10 10 10 10 10

# Pair-wise sum:
vec1 + vec2

## [1] 12 14 16 18 20

# Compare the pair-wise sum to the sum of both vectors:
sum(vec1) + sum(vec2)

## [1] 80
```

In each of the above examples, the operators (like + and so on) 'know' to make the calculations one element at a time (if one vector), or pair-wise (when two vectors). Clearly, for all examples where two vectors were used, the two vectors need to be the same length (i.e., have the same number of elements).

Applying multiple functions at once

In **R**, we can apply functions and operators in combination. In the following examples, the *innermost* expressions are always evaluated first. This will make sense after some examples:

```
# Mean of the vector 'vec1', *after* squaring the values:
mean(vec1^2)

## [1] 11

# Mean of the vector, *then* square it:
```

```
mean(vec1)^2

## [1] 9

# Standard deviation of 'vec1' after subtracting 1.2:
sd(vec1 - 1.2)

## [1] 1.581139

# Standard deviation of vec1, minus 1.2:
sd(vec1) - 1.2

## [1] 0.3811388

# Mean of the log of vec2:
mean(log(vec2))

## [1] 2.558972

# Log of the mean of vec2:
log(mean(vec2))

## [1] 2.564949

# Mininum value of the square root of pair-wise sum of vec1 and vec2:
min(sqrt(vec1 + vec2))

## [1] 3.464102

# The sum of squared deviations from the sample mean:
sum((vec1 - mean(vec1))^2)

## [1] 10

# The sample variance:
sum((vec1 - mean(vec1))^2) / (length(vec1) - 1)

## [1] 2.5
```

Try this yourself Confirm that the last example is equal to the sample variance that **R** calculates (use `var`).

Character vectors

A vector can also consist of `character` elements. Character vectors are constructed like this (don't forget the quotes, otherwise **R** will look for objects with these names - see Exercise 1.15.5).

```
words <- c("pet","elk","star","apple","the letter r")
```

Many of the functions summarized in the table above also apply to `character vectors` with the exception of obviously numerical ones. For example, the `sort` function to alphabetize a character vector:

```
sort(words)

## [1] "apple"        "elk"          "pet"          "star"
## [5] "the letter r"
```

And count the number of characters in each of the elements with `nchar`,

```
nchar(words)

## [1]  3  3  4  5 12
```

We will take a closer look at working with character strings in a later chapter (see Section 3.5)

1.5 Working with matrices

Another type of object is a matrix. A matrix is like a vector, except that it contains both rows and columns (a bit like an Excel spreadsheet.) Like vectors, matrices can contain only one type of data at a time.

The easiest way to construct a matrix is to make a vector, then give it dimensions:

```
# Notice that, by default, R fills down columns, from left to right:
mat1 <- matrix(c(1,2,3,4,5,6,7,8,9), nrow=3, ncol=3)

# A matrix can also be filled by rows:
mat2 <-  matrix(c(1,2,3,4,5,6,7,8,9), nrow=3, ncol=3, byrow=TRUE)
```

As with vectors, we can do basic arithmetic with a `numeric matrix`:

```
# Get the sum of a vector:
sum(mat1)

## [1] 45

# Get mean, standard deviation, and number of observations (length):
mean(mat1)

## [1] 5

sd(mat1)

## [1] 2.738613

length(mat1)

## [1] 9
```

We can also do arithmetic on matrices. As with vectors, simple arithmetic will be applied to each element at a time:

```
# Addition
# Add a single number to all the elements of a matrix:
mat1 + 10

##      [,1] [,2] [,3]
## [1,]   11   14   17
## [2,]   12   15   18
## [3,]   13   16   19

# Multiplication
# Each element is multiplied by 10:
mat2 * 10

##      [,1] [,2] [,3]
## [1,]   10   20   30
## [2,]   40   50   60
## [3,]   70   80   90
```

Again as with vectors, if we do arithmetic on two matrices at a time, each corresponding element in the two matrices is used:

```
# Multiply two matrices; this multiplies each element with each other
# Obviously, the matrices have to have the same dimensions for this to work:
mat1 * mat2

##      [,1] [,2] [,3]
## [1,]    1    8   21
## [2,]    8   25   48
## [3,]   21   48   81
```

In addition to basic arithmetic, there are many functions designed specifically to work with matrices. For instance, `diag` returns a vector with the diagonal elements of a matrix:

```
diag(mat1)

## [1] 1 5 9
```

The functions `rowSums` and `colSums` calculate the sums of rows and columns,

```
# Sums of rows and columns:
rowSums(mat1)

## [1] 12 15 18

colSums(mat1)

## [1]  6 15 24
```

and corresponding functions `rowMeans` and `colMeans` calculate the means.

One of the special operations that applies only to matrices is the `t` function, which stands for transpose. To transpose a matrix is to flip it so that the rows become the columns, and the columns become the rows:

```
# Transpose a matrix
t(mat1)

##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
```

Matrices can be a useful way to organize data, but in **R** it is generally more common to use dataframes, which allow you to combine more than one type of data in a single object. Nonetheless, some basic skills in matrices are useful as a number of built-in functions in **R** return matrices as output.

Further reading The Quick-R website has a good explanation of matrix-related functions `http://www.statmethods.net/advstats/matrix.html`, as well as many other resources and cheat sheets. If you are well-versed in linear algebra and would like to use **R** to do these type of calculations, `http://www.ats.ucla.edu/stat/r/library/matrix_alg.htm` has a helpful guide.

1.6 Generating data

1.6.1 Sequences of numbers

Let's look at a few ways to generate sequences of numbers that we can use in the examples and exercises. There are also a number of real-world situations where you want to use these functions.

First, as we saw already, we can use c() to 'concatenate' (link together) a series of numbers. We can also combine existing vectors in this way, for example:

```
a <- c(1,2,3)
b <- c(4,5,6)
c(a,b)

## [1] 1 2 3 4 5 6
```

We can generate sequences of numbers using `:`, `seq` and `rep`, like so:

```
# Sequences of integer numbers using the ":" operator:
1:10   # Numbers 1 through 10

##  [1]  1  2  3  4  5  6  7  8  9 10

5:-5   # From 5 to -5, backwards

##  [1]  5  4  3  2  1  0 -1 -2 -3 -4 -5

# Examples using seq()
seq(from=10,to=100,by=10)

##  [1]  10  20  30  40  50  60  70  80  90 100

seq(from=23, by=2, length=12)

##  [1] 23 25 27 29 31 33 35 37 39 41 43 45

# Replicate numbers:
rep(2, times = 10)

##  [1] 2 2 2 2 2 2 2 2 2 2

rep(c(4,5), each=3)

## [1] 4 4 4 5 5 5
```

The `rep` function works with any type of vector. For example, character vectors:

```
# Simple replication
rep("a", times = 3)

## [1] "a" "a" "a"

# Repeat elements of a vector
rep(c("E. tereticornis","E. saligna"), each=3)

## [1] "E. tereticornis" "E. tereticornis" "E. tereticornis" "E. saligna"
## [5] "E. saligna"      "E. saligna"
```

1.6.2 Random numbers

We can draw random numbers using the `runif` function. The `runif` function draws from a uniform distribution, meaning there is an equal probability of any number being chosen.

```
# Ten random numbers between 0 and 1
runif(10)

##  [1] 0.06728766 0.23638268 0.97409961 0.48043094 0.85442766 0.05184931
##  [7] 0.40691283 0.20467773 0.75007979 0.69036761

# Five random numbers between 100 and 1000
runif(5, 100, 1000)
```

```
## [1] 260.3831 459.2554 872.7460 645.6936 767.4211
```

Try this yourself The `runif` function is part of a much larger class of functions, each of which returns numbers from a different probability distribution. Inspect the help pages of the functions `rnorm` for the normal distribution, and `rexp` for the exponential distribution. Try generating some data from a normal distribution with a mean of 100, and a standard deviation of 10.

Next, we will `sample` numbers from an existing vector.

```
numbers <- 1:15
sample(numbers, size=20, replace=TRUE)

##  [1] 11  1  9  2  6  2 14 10  3  5 13  8  1 13  3  2 13  3 15 12
```

This command samples 20 numbers from the `numbers` vector, with replacement.

1.7 Objects in the workspace

In the examples above, we have created a few new objects. These objects are kept in memory for the remainder of your session (that is, until you close **R**).

In RStudio, you can browse all objects that are currently loaded in memory. Objects that are currently loaded in memory make up your *workspace*. Find the window that has the tab 'Environment'. Here you see a list of all the objects you have created in this **R** session. When you click on an object, a window opens that shows the contents of the object.

Alternatively, to see which objects you currently have in your workspace, use the following command:

```
ls()

## [1] "nums1"   "vec1"    "vec2"    "words"   "x"       "y"       "a"
## [8] "b"       "numbers"
```

To remove objects,

```
rm(nums1, nums2)
```

And to remove all objects that are currently loaded, use this command. **Note:** you want to use this wisely!

```
rm(list=ls())
```

Finally, when you are ending your **R** session, but you want to continue exactly at this point the next time, make sure to save the current workspace. In RStudio, find the menu Session (at the top). Here you can save the current workspace, and also load a previously saved one. Alternatively, you can use the `save.image` function (see also Section 1.13).

1.8 Files in the working directory

Each time you run **R**, it 'sees' one of your folders ('directories') and all the files in it. This folder is called the *working directory*. You can change the working directory in RStudio in a couple of ways.

The first option is to go to the menu Session > Set Working Directory > Choose Directory... .

Or, find the 'Files' tab in one of the RStudio windows (usually bottom-right). Here you can browse to the directory you want to use (by clicking on the small ... button on the right), and click More > Set as working directory.

You can also set or query the current working directory by typing the following code:

```r
# Set working directory to C:/myR
setwd("C:/myR")

# What is the current working directory?
getwd()
```

Note: For Windows users, use a forward slash (that is, /), not a back slash!

Finally, you can see which files are available in the current working directory, as well as in subdirectories, using the `dir` function (*Note*: a synonym for this function is `list.files`).

```r
# Show files in the working directory:
dir()

# List files in some other directory:
dir("c:/work/projects/data/")

# Show files in the subdirectory "data":
dir("data")

# Show files in the working directory that end in csv.
# (The ignore.case=TRUE assures that we find files that end in 'CSV' as well as 'csv',
# and the '[]' is necessary to find the '.' in '.csv'.
dir(pattern="[.]csv", ignore.case=TRUE)
```

Try this yourself List the R scripts in your current working directory (*Hint*: an R script should have the extension `.R`).

1.9 Keep a clean memory

When you start **R**, it checks whether a file called `.RData` is available in the default working directory. If it is, it loads the contents from that file automatically. This means that, over time, objects can accumulate in your memory, cluttering your workspace and potentially causing problems, especially if you are using old objects without knowing it.

We recommend you collect all your working code in scripts or markdown files (see Section 1.3). For each new project, add a line of code to the top that guarantees you start out with a clean memory, as in the following example. It is also good practice to set the working directory at the top of your script, with `setwd` (see Section 1.8), to make sure all the relevant files can be accessed.

```r
# Set working directory
setwd("C:/projects/data")

# Clean workspace
rm(list=ls())
```

This sort of workflow avoids common problems where old objects are being used unintentionally. In summary, **always:**

- Make sure you are in the correct working directory
- Make sure your workspace is clean, or contains objects you know
- Write scripts or markdown files that contain your entire workflow
- Once you have a working script (or markdown file), run the full file to make the final output

1.10 Packages

Throughout this tutorial, we will focus on the basic functionality of **R**, but we will also call on a number of add-on 'packages' that include additional functions. We will tell you which packages you need as they are called for. If, after running code from the book, you get "Error: could not find function 'example'", you can look up the function in the table at the end of the chapter to find out which package it comes from. You can find a full list of packages available for R on the CRAN site (`http://cran.r-project.org/`, navigate to 'Packages'), but be warned – it's a very long list! A more palatable index of packages is provided at `http://r-pkg.org/`.

In RStudio, click on the Packages tab in the lower-right hand panel. There you can see which packages are already installed, and you can install more by clicking on the Install button.

Alternatively, to install a particular package, simply type:

```
install.packages("gplots")
```

You will be asked to select a mirror (select a site closest to your location), and the package will be installed to your local library. The package needs to be installed only once. To use the package in your current **R** session, type:

```
library(gplots)
```

This loads the gplots package from your library into working memory. You can now use the functions available in the `gplots` package. If you close and reopen **R**, you will need to load the package again.

To quickly learn about the functions included in a package, type:

```
library(help=gplots)
```

If you are using packages in a script file, it is usually considered a good idea to load any packages you will be using at the *start* of the script.

1.10.1 Using packages in markdown files

Unfortunately, it is not possible to install new packages at the beginning of a markdown file. Any attempt to knit a file with `install.packages` will fail miserably. But, if you want the code in your file to run correctly, you need to have any relevant packages not only installed, but also loaded using the `library()` function. We may also not want any of the package startup messages to appear in our final document. What should we do?

We suggest two different ways of handling this. The first is simply to make sure that any needed packages are already installed before knitting, and include a block at the start of your markdown document that loads these packages:

```
---
title: "Basic R calculations in markdown"
```

~~~
author: "Remko Duursma"
date: "16 September 2015"
output: word_document
---

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
library(importantpackage1)
library(importantpackage2)
library(importantpackage3)
~~~

## R Markdown
Add the rest of the text and code here...
```

Alternatively, you can use a package called `pacman`. This package is great for documents that you will be sharing – it means that your collaborators will not have to spend time finding any packages they don't have. It has a function `p_load` that checks whether a package is installed, installs it if it is not, and makes sure it is loaded. Using `p_load` in a markdown document will not cause knitting to fail. With `pacman`, the start of a markdown document would look like this:

```
---
title: "Basic R calculations in markdown"
author: "Remko Duursma"
date: "16 September 2015"
output: word_document

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
library(pacman)
p_load(importantpackage1)
p_load(importantpackage2)
p_load(importantpackage3)
```

## R Markdown
Add the rest of the text and code here...
```

## 1.11 Updating R and packages

It is generally a good idea to always have the latest version of **R** installed on your computer. You can follow the installation instructions described in Section 1.1.1, or alternatively use the `installr` package (for Windows) or the `updateR` package (for Mac):

```
On Windows:
library(installr)

Downloads and installs the newest R version, if available.
Just follow the instructions and click OK.
```
```

```
updateR()

# On Mac (requires the 'devtools' package is installed):
devtools::install_github("AndreaCirilloAC/updateR")
library(updateR)

# Downloads and installs the newest R version, if available.
updateR(admin_password = "YOUR_PASSWORD_HERE")
```

The various contributed packages are routinely updated by the maintainers. Again, it is a very good idea to keep these up to date. If your code depends on an old package version, it may prevent your code from working on another computer. To update all packages, run the following command. *Note*: make sure to run this when you have just opened RStudio.

```
update.packages(ask=FALSE)
```

We recommend updating your packages about once a month.

1.12 Accessing the help files

Every function that you use in **R** has its own built-in help file. For example, to access the help file for the arithmetic mean, type:

```
?mean
```

This opens up the help file in your default browser (but you do not need to be online to read help files).

Functions added by loading new packages also have built-in help files. For example, to read about the function `bandplot` in the `gplots` package, type:

```
library(gplots)
?bandplot
```

Don't get overwhelmed when looking at the help files. Much of **R**'s reputation for a steep learning curve has to do with the rather obscure and often confusing help files. A good tip for beginners is to not read the help files, but skip straight to the Example section at the bottom of the help file. The first line of the help file, which shows what kind of input, or arguments, the function takes, can also be helpful.

If you know what type of function that you are looking for but do not know its name, use `??`. This searches the help files for a given keyword:

```
??ANOVA
```

1.13 Save a session in RData format

It is often useful to copy the results of an **R** session, so that you can continue later exactly where you left off - without having to execute all the code again.

We do this using `save.image`, which writes all objects that are currently in the workspace in a special file (an '.Rdata' file).

```
# Save the entire workspace:
save.image("august8.RData")
```

```
# So you can return to where you were, like this:
load("august8.RData")
```

You can also write one or more selected objects an '.RData' file, using `save`.

```
# Save one object, for later use.
save(myobject, file="myobject.RData")

# and load it (this will make the object available in the workspace again).
load("myobject.RData")

# or save multiple objects:
save(obj1, obj2, obj3, file="someobjects.RData")
```

1.14 Functions used in this chapter

For functions not listed here, please refer to the index at the end of this book.

Function	What it does	Example use

`:`	A handy little function that generates a sequence of numbers, counting by 1, backwards or forwards.	`0:18`
		`48:-8`

`?`	Finds help files for a specific function, package or dataset.	`?max`
`??`	Finds help files that match a given topic	`??correlation`

`c`	Combines input to form a vector. Input must be of the same data type (e.g., only letters, only numbers, etc.)	`c(1,2,5,9)`
		`c("imagine","that")`

`cbind`	Can be used to combine two matrices horizontally, keeping the same number of rows but adding more columns.	`cbind(matrix(1:4,2,2),`
		`matrix(1:4,2,2))`

`ceiling`	Rounds numbers in its input to the next highest whole number. Returns a vector.	`ceiling(c(4.999,4.001))`
`colMeans`	Returns means for each column of a matrix.	`colMeans(matrix(1:4,2,2))`
`colSums`	Returns sums for each column of a matrix.	`colSums(matrix(1:4,2,2))`
`cor`	Returns the correlation between two vectors, or between the columns of a matrix.	`cor(c(1,3,5,7),c(4,2,8,6))`
`cumsum`	Calculates the cumulative sum of its input. For vectors, it returns a vector. For matrices, it adds down a column first, then continues with the next column. The result is returned as a vector.	`cumsum(c(5,10,2,9))`
`diag`	Returns a vector with the diagonal elements of matrix.	`diag(matrix(1:25,5,5))`
`diff`	Calculates the sequential difference between a series of numbers. The result is returned as a vector.	`diff(c(2,4,5,8))`
`dir`	Lists the files in a directory or folder. Default is the working directory, but other locations can be provided. The same as `list.files`.	`dir()` `dir("C:/")` *Windows* `dir("/Users/")` *Mac*
`example`	Runs the example code at the bottom of the help page for a given function.	`example(cumsum)`
`floor`	Rounds numbers in its input to the next lowest whole number. Returns a vector.	`floor(c(4.999,4.001))`
`head`	Shows the first six elements of a vector or the first six rows of a matrix.	`head(letters)`
`intersect`	Returns the matching elements in two vectors.	`intersect(c(2,4),c(1,2))`
`length`	Returns the length of a vector. For a matrix, it returns the total number of elements.	`length(LETTERS)`

Function	What it does	Example use
`LETTERS`, `letters`	Not really functions, but vectors of the uppercase and lowercase English alphabet, always loaded and ready to use.	`LETTERS` `letters`
`list.files`	The same as `dir`; provides a list of files in a directory.	`list.files()`
`ls`	Gives the names of all the objects currently in working memory.	`ls()`
`max`	Returns the largest value in its input.	`max(1:100)`
`mean`	Returns the arithmetic average of its input.	`mean(c(1,2,3,10))`
`median`	Returns the middle value of its input.	`median(c(1,2,3,10)`
`min`	Returns the smallest value in its input.	`min(1:100)`
`nchar`	Gives the number of characters in each element of a vector containing strings of text.	`nchar(c("imagine","that"))`
`order`	Returns the order of elements in a vector. Note that this does not sort the vector. Instead, it tells you how the elements of the vector would need to be rearranged to sort them.	`order(c(9,5,8,2))`
`rbind`	A function that can be used to combine two matrices vertically, keeping the same number of columns but adding more rows.	`rbind(matrix(1:4,2,2),` `matrix(1:4,2,2))`
`rep`	Repeats its input a given number of times. Can be used on numbers, characters, and other kinds of data. Takes the arguments `times`: a number of times to repeat the input, `each`: a number of times to repeat each element of the input, and `length.out`: useful if you need output of an exact length.	`rep(1, times=5)` `rep(1:3, times=5)` `rep(1:3, each=5)` `rep(1:3, times=5, each=2)` `rep(1:3, length.out=5)` `rep("okay",10)`
`rev`	Reverses the contents of its input, and returns them as a vector.	`rev(c("is","it","cool"))`
`rexp`	Generates random data with an exponential distribution. The first argument is the number of data points to generate, the second argument is the rate parameter. The rate parameter changes the steepness of the distribution. The default is one. It can be set as 1/the desired mean of the data.	`(rexp(1000, rate=(1/5)))`
`rm`	Can be used to remove objects in the current workspace. To remove all objects, give `ls()` as the first argument.	`rm(x)` `rm(list=ls())`
`rnorm`	Generates random data with a normal distribution. The first argument is the number of points to generate, the second argument is the mean, and the third argument is the standard deviation. Default for `mean` is 0, and default for `sd` is 1.	`rnorm(100)` `rnorm(10, mean=50, sd=25)`

Function	**What it does**	**Example use**

`round`	Takes a vector of values, and rounds them to a given number of digits. The default for `digits` is 0.	`round(3.1415)`
		`round(99.99999, digits=2)`

`rowMeans`	Returns means for each row of a matrix.	`rowMeans(matrix(1:4,2,2))`
`rowSums`	Returns sums for each row of a matrix.	`rowSums(matrix(1:4,2,2))`

`runif`	Generates random data with a uniform distribution. The first argument is the number of points to generate, the second argument is the maximum value, and the third argument is the minimum. Default for `max` is 1, and default for `min` is 0.	`runif(10)`
		`runif(10, min=1, max=6)`

`sample`	Takes a given number of samples from a vector or a matrix. Can be set to sample with or without replacement. With replacement means that the same element can be sampled more than once. Without replacement allows each element to be sampled only once. Default is without replacement.	`sample(1:6, size=6)`
		`sample(1:6, 6, replace=TRUE)`

`save`	Saves an object to disk for later use.	`save(x, file="My_object")`
`save.image`	Saves a copy of the current workspace, which can be loaded at a later time. You can supply a file name.	`save.image(file="Aug8.RData")`
`sd`	Returns the standard deviation of the values in a vector or matrix.	`sd(c(99,85,50,87,89))`

`seq`	Generates a sequence of numbers from a given value to another given value. Can be set to count by 1s, 2s, 4s, etc.	`seq(1, 10)`
		`seq(1, 10, by=2.5)`

`setdiff`	Takes two vectors of numbers as arguments. Returns *only* the numbers that are found in the first vector, but not in the second.	`setdiff(1:5,4:8)`
		`setdiff(4:8,1:5)`

`setwd`	Sets the current working directory. The working directory is the folder that **R** can see. By default, **R** will assume files are located in the working directory.	`setwd("C:/RProject")` *Windows* `setwd("~/RProject")` *Mac*
`sort`	Sorts a vector in ascending or descending order.	`sort(1:5, decreasing=TRUE)`
`t`	Stands for 'transpose'. Can be used to transpose a matrix – that is, to flip it so that rows become columns, and columns become rows.	`t(matrix(c(1,2,3,1),2,2))`
`union`	Produces the union of two vectors, without duplicates.	`union(1:5,4:8)`
`unique`	Returns only the unique values in a vector.	`unique(c(1,2,1,5,5,2))`
`var`	Returns the variance of a vector or matrix.	`var(c(99,85,50,87,89))`

Function	What it does	Example use
`which.max`	Returns the position of the largest value in a vector. Contrast with `max`.	`which.max(c(99,85,50,87,89))`
`which.min`	Returns the position of the smallest value in a vector. Contrast with `min`.	`which.min(c(99,85,50,87,89))`

1.15 Exercises

In these exercises, we use the following colour codes:

■ **Easy**: make sure you complete some of these before moving on. These exercises will follow examples in the text very closely.

◆ **Intermediate**: a bit harder. You will often have to combine functions to solve the exercise in two steps.

▲ **Hard**: difficult exercises! These exercises will require multiple steps, and significant departure from examples in the text.

We suggest you complete these exercises in an **R** markdown file. This will allow you to combine code chunks, graphical output, and written answers in a single, easy-to-read file.

1.15.1 Calculating

Calculate the following quantities:

1. ■ The sum of `100.1`, `234.9` and `12.01`
2. ■ The square root of `256`
3. ■ Calculate the 10-based logarithm of `100`, and multiply the result with the cosine of π. *Hint:* see `?log` and `?pi`.
4. ■ Calculate the cumulative sum ('running total') of the numbers 2,3,4,5,6.
5. ◆ Calculate the cumulative sum of those numbers, but in reverse order. *Hint:* use the `rev` function.
6. ◆ Find 10 random numbers between 0 and 100, rounded to the nearest whole number (*Hint:* you can use either `sample` or a combination of `round` and `runif`).

1.15.2 Simple objects

Type the following code, which assigns numbers to objects `x` and `y`.

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

1. ■ Calculate the product of `x` and `y`
2. ■ Store the result in a new object called `z`
3. ■ Inspect your workspace by typing `ls()`, and by clicking the `Environment` tab in Rstudio, and find the three objects you created.
4. ■ Make a vector of the objects `x`, `y` and `z`. Use this command,

   ```
   myvec <- c(x,y,z)
   ```
5. ■ Find the minimum, maximum, length, and variance of `myvec`.
6. ■ Remove the `myvec` object from your workspace.

1.15.3 Working with a single vector

1. ■ The numbers below are the first ten days of rainfall amounts in 1996. Read them into a vector using the c() function (recall Section 1.4 on p. 17).

   ```
   0.1  0.6 33.8  1.9  9.6  4.3 33.7  0.3  0.0  0.1
   ```

   Inspect Table 1.3 on page 18, and answer the following questions:

2. ■ What was the mean rainfall, how about the standard deviation?
3. ■ Calculate the cumulative rainfall ('running total') over these ten days. Confirm that the last value of the vector that this produces is equal to the total sum of the rainfall.
4. ■ Which day saw the highest rainfall (write code to get the answer)?

1.15.4 Scripts

This exercise will make sure you are able to make a 'reproducable script', that is, a script that will allow you to repeat an analysis without having to start over from scratch. First, set up an **R** script (see Section 1.3 on page 11), and save it in your current working directory.

1. ■ Find the `History` tab in Rstudio. Copy a few lines of history that you would like to keep to the script you just opened, by selecting the line with the mouse and clicking `To Source`.
2. ■ Tidy up your R script by writing a few comments starting with `#`.
3. ■ Now make sure your script works completely (that is, it is entirely *reproducible*). First clear the workspace (`rm(list=ls())` or click `Clear` from the `Environment` tab). Then, run the entire script (by clicking `Source` in the script window, top-right).

1.15.5 To quote or not to quote

This short exercise points out the use of quotes in **R**.

1. ■ Run the following code, which makes two numeric objects.

   ```
   one <- 1
   two <- 2
   ```

2. ♦ Run the following two lines of code, and look at the resulting two vectors. The first line makes a character vector, the second line a numeric vector by recalling the objects you just constructed. Make sure you understand the difference.

   ```
   vector1 <- c("one","two")
   vector2 <- c(one, two)
   ```

3. ♦ The following lines of code contain some common errors that prevent them from being evaluated properly or result in error messages. Look at the code without running it and see if you can identify the errors and correct them all. Also execute the faulty code by copying and pasting the text into the console (not typing it, R studio will attempt to avoid these errors by default) so you get to know some common error messages (but not all of these result in errors!).

   ```
   vector1 <- c('one', 'two', 'three', 'four, 'five', 'seven')

   vec.var <- var(c(1, 3, 5, 3, 5, 1)
   vec.mean <- mean(c(1, 3, 5, 3, 5, 1))
   ```

```
vec.Min <- Min(c(1, 3, 5, 3, 5, 1))

Vector2 <- c('a', 'b', 'f', 'g')
vector2
```

1.15.6 Working with two vectors

1. 🟩 You have measured five cylinders, their lengths are:

   `2.1, 3.4, 2.5, 2.7, 2.9`

   and the diameters are :

   `0.3, 0.5, 0.6, 0.9, 1.1`

   Read these data into two vectors (give the vectors appropriate names).
2. 🟩 Calculate the correlation between lengths and diameters (use the `cor` function).
3. 🟩 Calculate the volume of each cylinder (V = length * pi * (diameter / 2)^2).
4. 🟩 Calculate the mean, standard deviation, and coefficient of variation of the volumes.
5. ♦ Assume your measurements are in centimetres. Recalculate the volumes so that their units are in cubic millimetres. Calculate the mean, standard deviation, and coefficient of variation of these new volumes.
6. ♦ You have measured the same five cylinders, but this time were distracted and wrote one of the measurements down twice:

   `2.1, 3.4, 2.5, 2.7, 2.9`

   and the diameters are :

   `0.3, 0.5, 0.6, 0.6, 0.9, 1.1`

   Read these data into two vectors (give the vectors appropriate names). As above, calculate the correlation between the vectors and store in a new vector. Also generate a vector of volumes based on these vectors and then calculate the mean and standard deviations of the volumes. Note that some steps result in errors, others in warnings, and some run perfectly fine. Why were some vectors created and others were not?

1.15.7 Alphabet aerobics 1

For the second question, you need to know that the 26 letters of the Roman alphabet are conveniently accessible in R via `letters` and `LETTERS`. These are not functions, but vectors that are always loaded.

1. ♦ Read in a vector that contains "A", "B", "C" and "D" (use the `c()` function). Using `rep`, produce this:

   ```
   "A" "A" "A" "B" "B" "B" "C" "C" "C" "D" "D" "D"
   ```

   and this:

   ```
   "A" "B" "C" "D" "A" "B" "C" "D" "A" "B" "C" "D"
   ```
2. ♦ Draw 10 random letters from the lowercase alphabet, and sort them alphabetically (*Hint:* use `sample` and `sort`). The solution can be one line of code.

3. ♦ Draw 5 random letters from each of the lowercase and uppercase alphabets, incorporating both into a single vector, and sort it alphabetically.
4. ♦ Repeat the above exercise but sort the vector alphabetically in descending order.

1.15.8 Comparing and combining vectors

Inspect the help page `union`, and note the useful functions `union`, `setdiff` and `intersect`. These can be used to compare and combine two vectors. Make two vectors :

```
x <- c(1,2,5,9,11)
y <- c(2,5,1,0,23)
```

Experiment with the three functions to find solutions to these questions.

1. ♦ Find values that are contained in both `x` and `y`
2. ♦ Find values that are in `x` but not `y` (and vice versa).
3. ♦ Construct a vector that contains all values contained in either `x` or `y`, and compare this vector to `c(x,y)`.

1.15.9 Into the matrix

In this exercise you will practice some basic skills with matrices. Recall Section 1.5 on p. 21.

1. ■ Construct a matrix with 10 columns and 10 rows, all filled with random numbers between 0 and 1 (see Section 1.6.2 on p. 23).
2. ■ Calculate the row means of this matrix (*Hint:* use `rowMeans`). Also calculate the standard deviation across the row means (now also use `sd`).
3. ▲ Now remake the above matrix with 100 columns, and 10 rows. Then calculate the column means (using, of course, `colMeans`), and plot a frequency diagram (a 'histogram') using `hist`. We will see this function in more detail in a later chapter, but it is easy enough to use as you just do `hist(myvector)`, where `myvector` is any numeric vector (like the column means). What sort of shape of the histogram do you expect? Now repeat the above with more rows, and more columns.

1.15.10 Packages

This exercise makes sure you know how to install packages, and load them. First, read Section 1.10 (p. 26).

1. ■ Install the `car` package (you only have to do this once for any computer).
2. ■ Load the `car` package (you have to do this every time you open Rstudio).
3. ■ Look at the help file for `densityPlot` (Read Section 1.12 on p. 28)
4. ■ Run the example for `densityPlot` (at the bottom of the help file), by copy-pasting the example into a script, and then executing it.
5. ■ Run the example for `densityPlot` again, but this time use the `example` function:

   ```
   example(densityPlot)
   ```

   Follow the instructions to cycle through the different steps.

6. ♦ Explore the contents of the `car` package by clicking first on the Packages tab, then finding the `car` package, and clicking on that. This way, you can find out about all functions a package contains (which, normally, you hope to avoid, but sometimes it is the only way to find what you are looking for). The same list can be obtained with the command `library(help=car)`, but that displays a list that is not clickable, so probably not as useful.

1.15.11 Save your work

We strongly encourage the use of R markdown files or scripts that contain your entire workflow. In most cases, you can just rerun the script to reproduce all outputs of the analysis. Sometimes, however, it is useful to save all resulting objects to a file, for later use. First read Section 1.13 on p. 28.

Before you start this exercise, first make sure you have a reproducable script as recommended in the third exercise to this chapter.

1. ■ Run the script, save the workspace, give the file an appropriate name (it may be especially useful to use the date as part of the filename, for example 'results_2015-02-27.RData'.
2. ■ Now close and reopen Rstudio. If you are in the correct working directory (it should become a habit to check this with the `getwd` function, do it now!), you can load the file into the workspace with the `load` function. Alternatively, in Rstudio, go to `File/Open File...` and load the workspace that way.

Chapter 2

Reading and subsetting data

2.1 Reading data

There are many ways to read data into **R**, but we are going to keep things simple and show only a couple of options. Let's assume you have a fairly standard dataset, with variables organized in columns, and individual records in rows, and individual fields separated by a comma (a comma-separated values (CSV) file). This is a very convenient and safe way to read in data. Most programs can save data to this format if you choose File > Save As... and select Comma Separated Values from the drop-down Format menu. If your data is not in a CSV file, see Section 2.1.2 on how to read other formats into R.

2.1.1 Reading CSV files

Use the function `read.csv` to read in the first example dataset ('Allometry.csv'). This assumes that the file 'Allometry.csv' is in your current working directory. Make sure you fully understand the concept of a working directory (see Section 1.8) before continuing.

```
allom <- read.csv("Allometry.csv")
```

If the file is stored elsewhere, you can specify the entire path (this is known as an *absolute* path).

```
allom <- read.csv("c:/projects/data/Allometry.csv")
```

Or, if the file is stored in a sub-directory of your working directory, you can specify the *relative* path.

```
allom <- read.csv("data/Allometry.csv")
```

The latter option is probably useful to keep your data files separate from your scripts and outputs in your working directory. We will not discuss how to organize your files here, but return to this important topic in Chapter 9.

The previous examples read in an entire dataset, and stored it in an object I called `allom`. This type of object is called a *dataframe*. We will be using dataframes a lot throughout this book. Like matrices, dataframes have two dimensions: they contain both columns and rows. Unlike matrices, each column can hold a different type of data. This is very useful for keeping track of different types of information about data points. For example, you might use one column to hold height measurements, and another to hold the matching species IDs. When you read in a file using `read.csv`, the data is automatically stored in a dataframe. Dataframes can also be created using the function `data.frame`. More on this in Section 2.1.2.

To read a description of this example dataset, see Section A.1 on page 247.

To look at the entire dataset after reading, simply type the name of the dataframe. This is called *printing* an object.

```
allom
```

Alternatively, in RStudio, find the dataframe in the `Environment` tab. If you click on it, you can inspect the dataframe with a built-in viewer (opens up in a separate window).

It is usually a better idea to print only the first (or last) few rows of the dataframe. R offers two convenient functions, `head` and `tail`, for doing this.

```
head(allom)

##   species diameter height   leafarea branchmass
## 1    PSME    54.61  27.04 338.485622  410.24638
## 2    PSME    34.80  27.42 122.157864   83.65030
## 3    PSME    24.89  21.23   3.958274    3.51270
## 4    PSME    28.70  24.96  86.350653   73.13027
## 5    PSME    34.80  29.99  63.350906   62.39044
## 6    PSME    37.85  28.07  61.372765   53.86594

tail(allom)

##    species diameter height   leafarea branchmass
## 58    PIMO    73.66  44.64 277.494360  275.71655
## 59    PIMO    28.19  22.59 131.856837   91.76231
## 60    PIMO    61.47  44.99 121.428976  199.86339
## 61    PIMO    51.56  40.23 212.443589  220.55688
## 62    PIMO    18.29  12.98  82.093031   28.04785
## 63    PIMO     8.38   4.95   6.551044    4.36969
```

Note that the row numbers are shown on the left. These can be accessed with `rownames(allom)`.

The function `read.csv` has many options, let's look at some of them. We can skip a number of rows from being read, and only read a fixed number of rows. For example, use this command to read rows 10-15, skipping the header line (which is in the first line of the file) and the next 9 lines. *Note:* you have to skip 10 rows to read rows 10-15, because the header line (which is ignored) counts as a row in the text file!

```
allomsmall <- read.csv("Allometry.csv", skip=10, nrows=5, header=FALSE)
```

2.1.2 Reading other data

Excel spreadsheets

A frequently asked question is: "How can I read an Excel spreadsheet into **R**?" The shortest answer is: *don't do it*. It is generally good practice to store your raw data in comma-separated values (CSV) files, as these are simple text files that can be read by any type of software. Excel spreadsheets may contain formulas and formatting, which we don't need, and usually require Excel to read.

In this book, we assume you always first save an XLS or XLSX file as a CSV. In Excel, select File > Save as..., click on the button next to 'Save as type...' and find 'CSV (Comma delimited) (*.csv)'.

If you do need to read an XLS or XLSX file, the `readxl` package works very well. *Note:* avoid older implementations like the `xlsx` package and `read.xls` in the `gtools` package, which are less reliable.

Tab-delimited text files

Sometimes, data files are provided as text files that are TAB-delimited. To read these files, use the following command:

```
mydata <- read.table("sometabdelimdata.txt", header=TRUE)
```

When using `read.table`, you must specify whether a header (i.e., a row with column names) is present in the dataset.

If you have a text file with some other delimiter, for example `;`, use the `sep` argument:

```
mydata <- read.table("somedelimdata.txt", header=TRUE, sep=";")
```

Reading typed data

You can also write the dataset in a text file, and read it as in the following example. This is useful if you have a small dataset that you typed in by hand (this example is from the help file for `read.table`).

```
read.table(header=TRUE, text="
a b
1 2
3 4
")
## a b
## 1 1 2
## 2 3 4
```

Reading data from the clipboard

A very quick way to read a dataset from Excel is to use your clipboard. In Excel, select the data you want to read (including the header names), and press Ctrl-C (Windows), or Cmd-C (Mac). Then, in **R**, type:

```
# for Windows:
mydata <- read.delim("clipboard", header=TRUE)

# or for Mac:
mydata <- read.delim(pipe("pbpaste"), header=TRUE)
```

This is not a long-term solution to reading data, but is a very quick way to read (part of) a messy spreadsheet that someone shared with you.

Other foreign formats

Finally, if you have a dataset in some unusual format, consider the `foreign` package, which provides a number of tools to read in other formats (such as SAS, SPSS, etc.).

Convert vectors into a dataframe

Suppose you have two or more vectors (of the same length), and you want to include these in a new dataframe. You can use the function `data.frame`. Here is a simple example:

```
vec1 <- c(9,10,1,2,45)
vec2 <- 1:5

data.frame(x=vec1, y=vec2)
```

```
##    x y
## 1  9 1
## 2 10 2
## 3  1 3
## 4  2 4
## 5 45 5
```

Here, we made a dataframe with columns named `x` and `y`. *Note:* take care to ensure that the vectors have the same length, otherwise it won't work!

Try this yourself Modify the previous example so that the two vectors are *not* the same length. Then, attempt to combine them in a dataframe and inspect the resulting error message.

2.2 Working with dataframes

This book focuses heavily on dataframes, because this is the object you will use most of the time in data analysis. The following sections provide a brief introduction, but we will see many examples using dataframes throughout this manual.

2.2.1 Variables in the dataframe

Let's first read the `allom` data, if you have not done so already.

```
allom <- read.csv("Allometry.csv")
```

After reading the dataframe, it is good practice to always quickly inspect the dataframe to see if anything went wrong. I routinely look at the first few rows with `head`. Then, to check the types of variables in the dataframe, use the `str` function (short for 'structure'). This function is useful for other objects as well, to view in detail what the object contains.

```
head(allom)
```

```
##   species diameter height   leafarea branchmass
## 1    PSME    54.61  27.04 338.485622  410.24638
## 2    PSME    34.80  27.42 122.157864   83.65030
## 3    PSME    24.89  21.23   3.958274    3.51270
## 4    PSME    28.70  24.96  86.350653   73.13027
## 5    PSME    34.80  29.99  63.350906   62.39044
## 6    PSME    37.85  28.07  61.372765   53.86594
```

```
str(allom)
```

```
## 'data.frame': 63 obs. of  5 variables:
##  $ species   : Factor w/ 3 levels "PIMO","PIPO",..: 3 3 3 3 3 3 3 3 3 3 ...
##  $ diameter  : num  54.6 34.8 24.9 28.7 34.8 ...
##  $ height    : num  27 27.4 21.2 25 30 ...
##  $ leafarea  : num  338.49 122.16 3.96 86.35 63.35 ...
##  $ branchmass: num  410.25 83.65 3.51 73.13 62.39 ...
```

Individual variables in a dataframe can be extracted using the dollar $ sign. Let's print all the tree diameters here, after rounding to one decimal point:

```
round(allom$diameter,1)
```

```
##  [1] 54.6 34.8 24.9 28.7 34.8 37.9 22.6 39.4 39.9 26.2 43.7 69.8 44.5 56.6
## [15] 54.6  5.3  6.1  7.4  8.3 13.5 51.3 22.4 69.6 58.4 33.3 44.2 30.5 27.4
## [29] 43.2 38.9 52.6 20.8 24.1 24.9 46.0 35.0 23.9 60.2 12.4  4.8 70.6 11.4
## [43] 11.9 60.2 60.7 70.6 57.7 43.1 18.3 43.4 18.5 12.9 37.9 26.9 38.6  6.5
## [57] 31.8 73.7 28.2 61.5 51.6 18.3  8.4
```

It is also straightforward to add new variables to a dataframe. Let's convert the tree diameter to inches, and add it to the dataframe as a new variable:

```
allom$diameterInch <- allom$diameter / 2.54
```

Instead of using the $-notation every time (which can result in lengthy, messy code, especially when your variable names are long) you can use `with` to indicate where the variables are stored. Let's add a new variable called `volindex`, a volume index defined as the square of tree diameter times height:

```
allom$volindex <- with(allom, diameter^2 * height)
```

For comparison, here is the same code using the $-notation. Note that the previous example is probably easier to read.

```
allom$volindex <- allom$diameter^2 * allom$height
```

The `with` function allows for more readable code, while at the same time making sure that the variables `diameter` and `height` are read from the dataframe `allom`.

Try this yourself The above examples are important – many times throughout this book you will be required to perform similar operations. As an exercise, also add the ratio of height to diameter to the dataframe.

After adding new variables, it is good practice to look at the result, either by printing the entire dataframe, or only the first few rows:

```
head(allom)
```

```
##   species diameter height   leafarea branchmass diameterInch volindex
## 1    PSME    54.61  27.04 338.485622  410.24638    21.500000 80640.10
## 2    PSME    34.80  27.42 122.157864   83.65030    13.700787 33206.72
## 3    PSME    24.89  21.23   3.958274    3.51270     9.799213 13152.24
## 4    PSME    28.70  24.96  86.350653   73.13027    11.299213 20559.30
## 5    PSME    34.80  29.99  63.350906   62.39044    13.700787 36319.09
## 6    PSME    37.85  28.07  61.372765   53.86594    14.901575 40213.71
```

A simple summary of the dataframe can be printed with the `summary` function:

```
summary(allom)
```

```
##  species      diameter          height          leafarea
##  PIMO:19   Min.   : 4.83   Min.   : 3.57   Min.   :  2.636
##  PIPO:22   1st Qu.:21.59   1st Qu.:21.26   1st Qu.: 28.581
##  PSME:22   Median :34.80   Median :28.40   Median : 86.351
##            Mean   :35.56   Mean   :26.01   Mean   :113.425
##            3rd Qu.:51.44   3rd Qu.:33.93   3rd Qu.:157.459
##            Max.   :73.66   Max.   :44.99   Max.   :417.209
##    branchmass        diameterInch       volindex
##  Min.   :   1.778   Min.   : 1.902   Min.   :    83.28
##  1st Qu.:  16.878   1st Qu.: 8.500   1st Qu.:  9906.35
##  Median :  72.029   Median :13.701   Median : 34889.47
##  Mean   : 145.011   Mean   :13.999   Mean   : 55269.08
##  3rd Qu.: 162.750   3rd Qu.:20.250   3rd Qu.: 84154.77
##  Max.   :1182.422   Max.   :29.000   Max.   :242207.52
```

For the numeric variables, the minimum, 1st quantile, median, mean, 3rd quantile, and maximum values are printed. For so-called 'factor' variables (i.e., categorical variables), a simple table is printed (in this case, for the `species` variable). We will come back to factors in Section 3.2. If the variables have missing values, the number of missing values is printed as well (see Section 3.4).

To see how many rows and columns your dataframe contains (handy for double-checking you read the data correctly),

```
nrow(allom)

## [1] 63

ncol(allom)

## [1] 7
```

2.2.2 Changing column names in dataframes

To access the names of a dataframe as a vector, use the `names` function. You can also use this to change the names. Consider this example:

```
# read names:
names(allom)

## [1] "species"    "diameter"   "height"     "leafarea"   "branchmass"

# rename all (make sure vector is same length as number of columns!)
names(allom) <- c("spec","diam","ht","leafarea","branchm")
```

We can also change some of the names, using simple indexing (see Section 2.3.1)

```
# rename Second one to 'Diam'
names(allom)[2] <- "Diam"

# rename 1st and 2nd:
names(allom)[1:2] <- c("SP","D")
```

2.3 Extracting data from vectors and dataframes

2.3.1 Vectors

Let's look at reordering or taking subsets of a vector, or `indexing` as it is commonly called. This is an important skill to learn, so we will look at several examples.

Let's recall the our last two `numeric vectors`:

```
nums1 <- c(1,4,2,8,11,100,8)
nums2 <- c(3.3,8.1,2.5,9.8,21.2,13.8,0.9)
```

Individual elements of a vector can be extracted using square brackets, `[ ]`. For example, to extract the first and then the fifth element of a vector:

```
nums1[1]

## [1] 1

nums1[5]

## [1] 11
```

You can also use another object to do the indexing, as long as it contains a integer number. For example,

```
# Get last element:
nelements <- length(nums1)
nums1[nelements]

## [1] 8
```

This last example extracts the last element of a vector. To do this, we first found the length of the vector, and used that to *index* the vector to extract the last element.

We can also select multiple elements, by *indexing* the vector with another vector. Recall how to construct sequences of numbers, explained in Section 1.6.1.

```
# Select the first 3:
nums1[1:3]

## [1] 1 4 2

# Select a few elements of a vector:
selectthese <- c(1,5,2)
nums1[selectthese]

## [1]  1 11  4

# Select every other element:
everyother <- seq(1,7,by=2)
nums1[everyother]

## [1]  1  2 11  8

# Select five random elements:
ranels <- sample(1:length(nums2), 5)
nums2[ranels]

## [1]  2.5 13.8  0.9 21.2  9.8

# Remove the first element:
nums1[-1]

## [1]   4   2   8  11 100   8

# Remove the first and last element:
nums1[-c(1, length(nums1))]

## [1]   4   2   8  11 100
```

Next, we can look at selecting elements of a vector based on the values in that vector. Suppose we want to make a new vector, based on vector `nums2` but only where the value within certain bounds. We can use simple logical statements to index a vector.

```
# Subset of nums2, where value is at least 10 :
nums2[nums2 > 10]
```

```
## [1] 21.2 13.8

# Subset of nums2, where value is between 5 and 10:
nums2[nums2 > 5 & nums2 < 10]

## [1] 8.1 9.8

# Subset of nums2, where value is smaller than 1, or larger than 20:
nums2[nums2 < 1 | nums2 > 20]

## [1] 21.2  0.9

# Subset of nums1, where value is exactly 8:
nums1[nums1 == 8]

## [1] 8 8

# Subset nums1 where number is NOT equal to 100
nums1[nums1 != 100]

## [1]  1  4  2  8 11  8

# Subset of nums1, where value is one of 1,4 or 11:
nums1[nums1 %in% c(1,4,11)]

## [1]  1  4 11

# Subset of nums1, where value is NOT 1,4 or 11:
nums1[!(nums1 %in% c(1,4,11))]

## [1]   2   8 100   8
```

These examples showed you several new logical operators. These operators are summarized in Table 2.1. See the help page `?Logic` for more details on logical operators. If any are unclear, don't worry. We will return to logical data in Section 3.3.

Table 2.1: Logical operators.

Operator	Meaning
>	greater than
<	less than
&	AND
==	equal to
|	OR
%in%	is an element of
!	NOT

Assigning new values to subsets

All of this becomes very useful if we realize that new values can be easily assigned to subsets. This works for any of the examples above. For instance,

```
# Where nums1 was 100, make it -100
nums1[nums1 == 100] <- -100

# Where nums2 was less than 5, make it zero
nums2[nums2 < 5] <- 0
```

Try this yourself Using the first set of examples in this section, practice assigning new values to subsets of vectors.

2.3.2 Subsetting dataframes

There are two ways to take a subset of a dataframe: using the square bracket notation (`[]`) as in the above examples, or using the `subset` function. We will learn both, as they are both useful from time to time.

Dataframes can be indexed with row and column numbers, like this:

`mydataframe[row,column]`

Here, `row` refers to the row number (which can be a vector of any length), and `column` to the column number (which can also be a vector). You can also refer to the column by its *name* rather than its number, which can be very useful. All this will become clearer after some examples.

Let's look at a few examples using the Allometry dataset (see Section A.1 for a description of the dataset).

```
# Read data
allom <- read.csv("allometry.csv")

# Recall the names of the variables, the number of columns, and number of rows:
names(allom)

## [1] "species"    "diameter"   "height"     "leafarea"   "branchmass"

nrow(allom)

## [1] 63

ncol(allom)

## [1] 5

# Extract tree diameters: take the 4th observation of the 2nd variable:
allom[4,2]

## [1] 28.7

# We can also index the dataframe by its variable name:
allom[4,"diameter"]

## [1] 28.7

# Extract the first 3 rows of 'height':
allom[1:3, "height"]

## [1] 27.04 27.42 21.23

# Extract the first 5 rows, of ALL variables
# Note the use of the comma followed by nothing
# This means 'every column' and is very useful!
allom[1:5,]

##   species diameter height   leafarea branchmass
## 1    PSME    54.61  27.04 338.485622  410.24638
## 2    PSME    34.80  27.42 122.157864   83.65030
## 3    PSME    24.89  21.23   3.958274    3.51270
## 4    PSME    28.70  24.96  86.350653   73.13027
## 5    PSME    34.80  29.99  63.350906   62.39044

# Extract the fourth column
# Here we use nothing, followed by a comma,
# to indicate 'every row'
allom[,4]

##  [1] 338.485622 122.157864   3.958274  86.350653  63.350906  61.372765
##  [7]  32.077794 147.270523 141.787332  45.020041 145.809802 349.057010
## [13] 176.029213 319.507115 234.368784   4.851567   7.595163  11.502851
## [19]  25.380647  65.698749 160.839174  31.780702 189.733007 253.308265
## [25]  91.538428  90.453658  99.736790  34.464685  68.150309  46.326060
## [31] 160.993131   9.806496  20.743280  21.649603  66.633675  54.267994
## [37]  19.844680 131.727303  22.365837   2.636336 411.160376  15.476022
## [43]  14.493428 169.053644 139.651156 376.308256 417.209436 103.672633
## [49]  33.713580 116.154916  44.934469  18.855867 154.078625  70.602797
## [55] 169.163842   7.650902  93.072006 277.494360 131.856837 121.428976
## [61] 212.443589  82.093031   6.551044

# Select only 'height' and 'diameter', store in new dataframe:
allomhd <- allom[,c("height", "diameter")]
```

As we saw when working with vectors (see Section 2.3.1), we can use expressions to extract data. Because each column in a dataframe is a vector, we can apply the same techniques to dataframes, as in the following examples.

We can also use one vector in a dataframe to find subsets of another. For example, what if we want to find the value of one vector, if another vector has a particular value?

```
# Extract diameters larger than 60
allom$diameter[allom$diameter > 60]

## [1] 69.85 69.60 60.20 70.61 60.20 60.71 70.61 73.66 61.47

# Extract all rows of allom where diameter is larger than 60.
# Make sure you understand the difference with the above example!
allom[allom$diameter > 60,]

##    species diameter height leafarea branchmass
## 12    PSME    69.85  31.35 349.0570   543.9731
## 23    PIPO    69.60  39.37 189.7330   452.4246
## 38    PIPO    60.20  31.73 131.7273   408.3383
## 41    PIPO    70.61  31.93 411.1604  1182.4222
## 44    PIPO    60.20  35.14 169.0536   658.2397
## 45    PIMO    60.71  39.84 139.6512   139.2559
## 46    PIMO    70.61  40.66 376.3083   541.3062
## 58    PIMO    73.66  44.64 277.4944   275.7165
....

# We can use one vector to index another. For example, find the height of the tree
# that has the largest diameter, we can do:
allom$height[which.max(allom$diameter)]

## [1] 44.64

# Recalling the previous section, this is identical to:
allom[which.max(allom$diameter), "height"]

## [1] 44.64
```

```
# Get 10 random observations of 'leafarea'. Here, we make a new vector
# on the fly with sample(), which we use to index the dataframe.
allom[sample(1:nrow(allom),10),"leafarea"]
```

```
##  [1]  45.02004  32.07779  68.15031 121.42898 169.05364  33.71358  14.49343
##  [8]  20.74328 234.36878 131.85684
```

```
# As we did with vectors, we can also use %in% to select a subset.
# This example selects only two species in the dataframe.
allom[allom$species %in% c("PIMO","PIPO"),]
```

```
##    species diameter    height   leafarea branchmass
## 23    PIPO    69.60 39.369999 189.733007  452.42455
## 24    PIPO    58.42 35.810000 253.308265  595.64015
## 25    PIPO    33.27 20.800001  91.538428  160.44416
## 26    PIPO    44.20 29.110001  90.453658  149.72883
## 27    PIPO    30.48 22.399999  99.736790   44.13532
## 28    PIPO    27.43 27.690001  34.464685   22.98360
## 29    PIPO    43.18 35.580000  68.150309  106.40410
## 30    PIPO    38.86 33.120001  46.326060   58.24071
....
```

```
# Extract tree diameters for the PIMO species, as long as diameter > 50
allom$diameter[allom$species == "PIMO" & allom$diameter > 50]
```

```
## [1] 60.71 70.61 57.66 73.66 61.47 51.56
```

```
# (not all output shown)
```

Try this yourself As with vectors, we can quickly assign new values to subsets of data using the <- operator. Try this on some of the examples above.

Using `subset()`

While the above method to index dataframes is very flexible and concise, sometimes it leads to code that is difficult to understand. It is also easy to make mistakes when you subset dataframes by the column or row number (imagine the situation where the dataset has changed and you redo the analysis). Consider the `subset` function as a convenient and safe alternative.

With the `subset` function, you can select rows that meet a certain criterion, and columns as well. This example uses the pupae data, see Section A.6.

```
# Read data
pupae <- read.csv("pupae.csv")

# Take subset of pupae, ambient temperature treatment and CO2 is 280.
subset(pupae, T_treatment == "ambient" & CO2_treatment == 280)
```

```
##    T_treatment CO2_treatment Gender PupalWeight Frass
## 1      ambient           280      0       0.244 1.900
## 2      ambient           280      1       0.319 2.770
## 3      ambient           280      0       0.221    NA
## 4      ambient           280      0       0.280 1.996
## 5      ambient           280      0       0.257 1.069
....
```

```
# (not all output shown)

# Take subset where Frass is larger than 2.9.
# Also, keep only variables 'PupalWeight' and 'Frass'.
# Note that you don't quote the column names when using 'subset'.
subset(pupae, Frass > 2.6, select=c(PupalWeight,Frass))
```

```
##    PupalWeight Frass
## 2        0.319 2.770
## 18       0.384 2.672
## 20       0.385 2.603
## 25       0.405 3.117
## 29       0.473 2.631
....
```

Let's look at another example, using the cereal data (see Section A.7). Here, we use `%in%`, which we already saw in Section 2.3.1.

```
# Read data
cereal <- read.csv("cereals.csv")

# What are the Manufacturers in this dataset?
levels(cereal$Manufacturer)
```

```
## [1] "A" "G" "K" "N" "P" "Q" "R"
```

```
# Take a subset of the data with only 'A', 'N' and 'Q' manufacturers,
# keep only 'Cereal.name' and 'calories'.
cerealsubs <- subset(cereal, Manufacturer %in% c("A","N","Q"), select=c(Cereal.name,calories))
cerealsubs
```

```
##                  Cereal.name calories
## 1                  100%_Bran       70
## 2          100%_Natural_Bran      120
## 11               Cap'n'Crunch      120
## 21     Cream_of_Wheat_(Quick)      100
## 36           Honey_Graham_Ohs      120
## 42                       Life      100
## 44                      Maypo      100
## 55                Puffed_Rice       50
## 56               Puffed_Wheat       50
## 57         Quaker_Oat_Squares      100
## 58             Quaker_Oatmeal      100
## 64             Shredded_Wheat       80
## 65     Shredded_Wheat_'n'Bran       90
## 66 Shredded_Wheat_spoon_size       90
## 69    Strawberry_Fruit_Wheats       90
```

Another example using `subset` is provided in Section 3.2.

Try this yourself You can also delete a single variable from a dataframe using the `select` argument in the `subset` function. Look at the second in `?subset` to see how, and try this out with the cereal data we used in the above examples.

2.3.3 Difference between `[]` and `subset()`

We have seen two methods for indexing dataframes. Usually, both the square brackets and `subset` behave in the same way, except when only one variable is selected.

In that case, indexing with `[]` returns a vector, and `subset` returns a dataframe with one column. Like this,

```
# A short dataframe
dfr <- data.frame(a=-5:0, b=10:15)

# Select one column, with subscripting or subset.
dfr[,"a"]
```

```
## [1] -5 -4 -3 -2 -1  0
```

```
subset(dfr, select=a)
```

```
##    a
## 1 -5
## 2 -4
## 3 -3
## 4 -2
## 5 -1
## 6  0
```

In some cases, a vector might be preferred. In other cases, a dataframe. The behaviour of the two methods also differs when there are missing values in the vector. We will return to this point in Section 3.4.3.

2.3.4 Deleting columns from a dataframe

It is rarely necessary to delete columns from a dataframe, unless you want to save a copy of the dataframe to disk (see Section). Instead of deleting columns, you can take a subset and make a new dataframe to continue with. Also, it should not be necessary to delete columns from the dataframe that you have accidentally created in a reproducible script: when things go wrong, simply clear the workspace and run the entire script again.

That aside, you have the following options to delete a column from a dataframe.

```
# A simple example dataframe
dfr <- data.frame(a=-5:0, b=10:15)

# Delete the second column (make a new dataframe 'dfr2' that does not include that column)
dfr2 <- dfr[,-2]

# Use subset to remove a column
# Note: this does not work using square-bracket notation!
dfr2 <- subset(dfr, select = -b)

# Finally, this strange command deletes a column as well.
# In this case, we really delete the column from the existing dataframe,
# whereas the two examples above create a new subset *without* that column.
dfr$b <- NULL
```

2.4 Exporting data

To write a dataframe to a comma-separated values (CSV) file, use the `write.csv` function. For example,

```
# Some data
dfr <- data.frame(x=1:3, y=2:4)

# Write to disk (row names are generally not wanted in the CSV file).
write.csv(dfr,"somedata.csv", row.names=FALSE)
```

If you want more options, or a different delimiter (such as TAB), look at the `write.table` function.

2.5 Functions used in this chapter

For functions not listed here, please refer to the index at the end of this book.

Function	What it does	Example use

`!`	A logical operator meaning NOT. NOT returns the *opposite* of other functions it is used with. Compare to other examples in this table. Returns TRUE or FALSE.	`1 != 2`
		`"a" !%in% LETTERS`

`==`	A logical operator meaning equal to. Returns TRUE or FALSE.	`1 == 2`
		`letters[1] == "a"`

`$`	An operator that allows you to extract or change a single variables from a dataframe. The result can be treated like a vector.	`mtcars$wt`
		`mtcars$kg <- mtcars$wt*454`

`%in%`	A logical operator that asks whether its first arguments is an element of its second argument. Returns TRUE or FALSE.	`"a" %in% letters`
		`"a" %in% LETTERS`

`&`	A logical operator meaning AND. Returns TRUE or FALSE.	`TRUE & TRUE`
		`(100 > 10) & (10 > 100)`

`|`	A logical operator meaning OR. Returns TRUE or FALSE.	`FALSE | FALSE`
		`(100 > 10) | (10 > 100)`

`>`	A logical operator meaning greater than. Returns TRUE or FALSE.	`10 > 100`
		`100 > 10`

`<`	A logical operator meaning less than. Returns TRUE or FALSE.	`10 < 100`
		`100 < 10`

`any`	Given a vector of logical values, returns a single logical value. `TRUE` if there is at least one `TRUE` value, `FALSE` if there is not.	`any(c(TRUE, FALSE,`
		`    FALSE, FALSE)`
		`any(letters == "?")`

`data.frame`	Creates dataframes, one of the most convenient ways to store data in **R**. Data frames have columns, each of which represents a different variable, and rows, each of which represents a different observation of that variable. While each colummn must contain only a single data type, different columns can contain different data types.	`dfr <- data.frame(`
		`    a=-5:0, b=10:15)`

`duplicated`	Given a vector, returns a vector of logical values indicating which elements appear more than once.	`duplicated(c(1,1,2,3,4,4))`

`head`	Shows the first rows of a dataframe. By default, shows the first six rows.	`head(mtcars)`
		`head(mtcars, n=10)`

`LETTERS`, `letters`	Not really functions, but vectors of the uppercase and lowercase English alphabet, always loaded and ready to use.	`LETTERS` `letters`

Function	**What it does**	**Example use**
`ls`	Gives the names of all the objects currently in working memory.	`ls()`

`names`	Allows you to view or change the names of columns in a dataframe.	`names(BOD)`
		`names(BOD) <- c("T","D")`

`ncol`	Gives the number of columns in a dataframe.	`ncol(mtcars)`
`nrow`	Gives the number of rows in a dataframe.	`nrow(mtcars)`

`order`	Can be used to find the sort order for a single column of a dataframe. By default, sorts in increasing order, but can be set to sort in decreasing order.	`order(mtcars$wt)`
		`order(mtcars$wt, decreasing = TRUE)`

`read.csv`	Reads data from comma-separated variable (CSV) files. Important arguments include the file name (or complete file path) and whether or not the file has a `header`. For more details, see Section 2.1.1.	`read.csv("Allometry.csv")`
		`read.csv("Allometry.csv", header=TRUE)`

`read.table`	A flexible function for reading data with different separating characters. To use `read.table`, you need to the character used to separate the data (`sep`) and whether or not the file has a header row (`header`). There are also many other options you can set to customize this function; for more see `?read.table`.	`read.table("tab.txt", header=TRUE)`
		`read.table("semi.txt", sep=";",header=TRUE)`

`rm`	Can be used to remove objects from the current workspace. To remove all objects, give `ls()` as the first argument.	`rm(x)`
		`rm(list=ls())`

`str`	Gives the structure of an object. Very useful for finding out what data types an object contains.	`str(warpbreaks)`

`subset`	Allows the selection of part of a dataframe, using logical operators to define which rows or columns should be included.	`subset(mtcars, wt > 3)`
		`subset(mtcars, wt > 3 & hp < 100)`

`summary`	Provides a simple summary of the variables in a dataframe. Statistical information is given for numerical variables, and a list of counts is given for factor variables.	`summary(warpbreaks)`
`which.max`	For a single variable from a dataframe, gives the position of the largest value.	`which.max(mtcars$wt)`
`which.min`	For a single variable from a dataframe, gives the position of the smallest value.	`which.min(mtcars$wt)`
`with`	Can be used to extract certain variables from a dataframe. A useful alternative to $, particularly when doing math involving multiple variables.	`with(mtcars, mpg/wt)`
`write.csv`	Can be used to output a dataframe as a CSV file. This is a file in which values are separated by commas, and which can be read by Excel and other programs. You must provide a dataframe and a file name. Output will be written to your working directory.	`write.csv(mtcars, file="Motor_cars.csv")`

Function	What it does	Example use
`write.table`	Similar to `write.csv`, but can be used to write files with other separating characters, such as tabs or semicolons. You must provide a dataframe and a file name. Output will be written to your working directory.	`write.table(mtcars, file="Motor_cars.txt", sep=";")`

2.6 Exercises

In these exercises, we use the following colour codes:

🟩 **Easy**: make sure you complete some of these before moving on. These exercises will follow examples in the text very closely.

♦ **Intermediate**: a bit harder. You will often have to combine functions to solve the exercise in two steps.

▲ **Hard**: difficult exercises! These exercises will require multiple steps, and significant departure from examples in the text.

We suggest you complete these exercises in an **R** markdown file. This will allow you to combine code chunks, graphical output, and written answers in a single, easy-to-read file.

2.6.1 Working with a single vector 2

Recall Exercise 1.15.3 on p. 35. Read in the `rainfall` data once more. We now practice subsetting a vector (see Section 2.3.1, p. 44).

1. 🟩 Take a subset of the rainfall data where rain is larger than 20.
2. 🟩 What is the mean rainfall for days where the rainfall was at least 4?
3. 🟩 Subset the vector where it is either exactly zero, or exactly 0.6.

2.6.2 Alphabet aerobics 2

The 26 letters of the Roman alphabet are conveniently accessible in R via `letters` and `LETTERS`. These are not functions, but vectors that are always loaded.

1. 🟩 What is the 18th letter of the alphabet?
2. 🟩 What is the last letter of the alphabet (don't guess, write code)?
3. ♦ Use `?sample` to figure out how to sample with replacement. Generate a random subset of fifteen letters. Are any letters in the subset duplicated? *Hint:* use the `any` and `duplicated` functions. Which letters?

2.6.3 Basic operations with the Cereal data

For this exercise, we will use the Cereal data, see Section A.7 (p. 249) for a description of the dataset.

1. 🟩 Read in the dataset, look at the first few rows with `head` and inspect the data types of the variables in the dataframe with `str`.
2. 🟩 Add a new variable to the dataset called 'totalcarb', which is the sum of 'carbo' and 'sugars'. Recall Section 2.2 (p. 42).
3. 🟩 How many cereals in the dataframe are 'hot' cereals? *Hint:* take an appropriate subset of the data, and then count the number of observations in the subset.
4. ♦ How many unique manufacturers are included in the dataset? *Hint:* use `length` and `unique`.
5. 🟩 Take a subset of the dataframe with only the Manufacturer 'K' (Kellogg's).

6. ■ Take a subset of the dataframe of all cereals that have less than 80 calories, AND have more than 20 units of vitamins.
7. ■ Take a subset of the dataframe containing cereals that contain at least 1 unit of sugar, and keep only the variables 'Cereal.name', 'calories' and 'vitamins'. Then inspect the first few rows of the dataframe with `head`.
8. ■ For one of the above subsets, write a new CSV file to disk using `write.csv` (see Section 2.4 on p. 52).
9. ■ Rename the column 'Manufacturer' to 'Producer' (see Section 2.2.2, p. 44).

2.6.4 A short dataset

1. ■ Read the following data into **R** (number of honeyeaters seen at the EucFACE site seen in a week). Give the resulting dataframe a reasonable name. *Hint:*To read this dataset, look at Section 2.1.2 (p. 40) (there are at least two ways to read this dataset, or you can type it into Excel and save as a CSV file if you prefer).

```
Day nrbirds
sunday 3
monday 2
tuesday 5
wednesday 0
thursday 8
friday 1
saturday 2
```

2. ■ Add a day number to the dataset you read in above (sunday=1, saturday=7). Recall the `seq` function (Section 1.6.1, p. 22).
3. ■ Delete the 'Day' variable (to only keep the `daynumber` that you added above).
4. ♦ On which `daynumber` did you observe the most honeyeaters? *Hint:* use `which.max`, in combination with indexing.
5. ♦ Sort the dataset by number of birds seen. *Hint:* use the `order` function to find the order of the number of birds, then use this vector to index the dataframe.

2.6.5 Titanic

1. ■ Read the data described in Section A.12 (p. 251)
2. ■ Make two new dataframes : a subset of male survivors, and a subset of female survivors. Recall Section 2.3.2 (p. 47) (you can use either the square brackets, or `subset` to make the subsets).
3. ♦ Based on the previous question, what was the name of the oldest surviving male? In what class was the youngest surviving female? *Hint:* use `which.max`, `which.min` on the subsets you just created.
4. ▲ Take 15 random names of passengers from the Titanic, and sort them alphabetically. *Hint:* use `sort`.

2.6.6 Managing your workspace

Before you start this exercise, first make sure you have a reproducible script.

1. 🟩 You should now have a cluttered workspace. Generate a vector that contains the names of all objects in the workspace.
2. ⧫ Generate a vector that contains the names of all objects in the workspace, with the exception of `titanic`. *Hint:* use the `ls` function.
3. ⧫ Look at the help page for `rm`. Use the `list` argument to delete all objects from the workspace except for `titanic`. Use the `ls` function to confirm that your code worked.

Chapter 3

Special data types

3.1 Types of data

For the purpose of this tutorial, a dataframe can contain six types of data. These are summarized in the table below:

Data type	Description	Example	Section
`numeric`	Any number	`c(1, 12.3491, 10/2, 10*6)`	
`character`	Character strings	`c("E. saligna", "HFE", "a b c")`	3.5
`factor`	Categorical variable	`factor(c("Control","Fertilized","Irrigated"))`	3.2
`logical`	Either TRUE or FALSE	`10 == 100/10`	3.3
`Date`	Special Date class	`as.Date("2010-6-21")`	3.6.1
`POSIXct`	Special Date-time class	`Sys.time()`	3.6.2

Also, **R** has a very useful built-in data type to represent missing values. This is represented by `NA` (Not Available) (see Section 3.4).

We will show how to convert between data types at the end of this chapter (Section 3.7).

3.2 Working with factors

The *factor* data type is used to represent qualitative, categorical data.

When reading data from file, for example with `read.csv`, **R** will automatically convert any variable to a factor if it is unable to convert it to a numeric variable. If a variable is actually numeric, but you want to treat it as a factor, you can use `as.factor` to convert it, as in the following example.

```
# Read pupae data
pupae <- read.csv("pupae.csv")

# This dataset contains a temperature (T_treatment) and CO2 treatment (CO2_treatment).
# Both should logically be factors, however, CO2_treatment is read as numeric:
str(pupae)

## 'data.frame': 84 obs. of  5 variables:
##  $ T_treatment  : Factor w/ 2 levels "ambient","elevated": 1 1 1 1 1 1 1 1 1 1 ...
##  $ CO2_treatment: int  280 280 280 280 280 280 280 280 280 280 ...
##  $ Gender       : int  0 1 0 0 0 1 0 1 0 1 ...
##  $ PupalWeight  : num  0.244 0.319 0.221 0.28 0.257 0.333 0.275 0.312 0.254 0.356 ...
##  $ Frass        : num  1.9 2.77 NA 2 1.07 ...

# To convert it to a factor, we use:
pupae$CO2_treatment <- as.factor(pupae$CO2_treatment)

# Compare with the above,
str(pupae)

## 'data.frame': 84 obs. of  5 variables:
##  $ T_treatment  : Factor w/ 2 levels "ambient","elevated": 1 1 1 1 1 1 1 1 1 1 ...
##  $ CO2_treatment: Factor w/ 2 levels "280","400": 1 1 1 1 1 1 1 1 1 1 ...
##  $ Gender       : int  0 1 0 0 0 1 0 1 0 1 ...
##  $ PupalWeight  : num  0.244 0.319 0.221 0.28 0.257 0.333 0.275 0.312 0.254 0.356 ...
##  $ Frass        : num  1.9 2.77 NA 2 1.07 ...
```

In the `allom` example dataset, the `species` variable is a good example of a factor. A factor variable has a number of 'levels', which are the text values that the variable has in the dataset. Factors can also represent treatments of an experimental study. For example,

```
levels(allom$species)

## [1] "PIMO" "PIPO" "PSME"
```

Shows the three species in this dataset. We can also count the number of rows in the dataframe for each species, like this:

```
table(allom$species)

##
## PIMO PIPO PSME
##   19   22   22
```

Note that the three species are always shown in the order of the `levels` of the factor: when the dataframe was read, these levels were assigned based on alphabetical order. Often, this is not a very logical order, and you may want to rearrange the levels to get more meaningful results.

In our example, let's shuffle the levels around, using `factor`.

```
allom$species <- factor(allom$species, levels=c("PSME","PIMO","PIPO"))
```

Now revisit the commands above, and note that the results are the same, but the order of the `levels` of the `factor` is different. You can also reorder the levels of a factor by the values of another variable, see the example in Section 6.2.5.

We can also generate new factors, and add them to the dataframe. This is a common application:

```
# Add a new variable to allom:  'small' when diameter is less than 10, 'large' otherwise.
allom$treeSizeClass <- factor(ifelse(allom$diameter < 10, "small", "large"))

# Now, look how many trees fall in each class.
# Note that somewhat confusingly, 'large' is printed before 'small'.
# Once again, this is because the order of the factor levels is alphabetical by default.
table(allom$treeSizeClass)

##
## large small
##    56     7
```

What if we want to add a new factor based on a numeric variable with more than two levels? In that case, we cannot use `ifelse`. We must find a different method. Look at this example using `cut`.

```
# The cut function takes a numeric vectors and cuts it into a categorical variable.
# Continuing the example above, let's make 'small','medium' and 'large' tree size classes:
allom$treeSizeClass <- cut(allom$diameter, breaks=c(0,25,50,75),
                           labels=c("small","medium","large"))

# And the results,
table(allom$treeSizeClass)

##
##  small medium  large
##     22     24     17
```

Empty factor levels

It is important to understand how factors are used in **R**: they are not simply text variables, or 'character strings'. Each unique value of a factor variable is assigned a *level*, which is used every time you summarize your data by the factor variable.

Crucially, even when you delete data, the original factor *level* is still present. Although this behaviour might seem strange, it makes a lot of sense in many cases (zero observations for a particular factor level can be quite informative, for example species presence/absence data).

Sometimes it is more convenient to drop empty factor levels with the `droplevels` function. Consider this example:

```
# Read the Pupae data:
pupae <- read.csv("pupae.csv")

# Note that 'T_treatment' (temperature treatment) is a factor with two levels,
# with 37 and 47 observations in total:
table(pupae$T_treatment)

##
##  ambient elevated
##       37       47

# Suppose we decide to keep only the ambient treatment:
pupae_amb <- subset(pupae, T_treatment == "ambient")

# Now, the level is still present, although empty:
table(pupae_amb$T_treatment)

##
##  ambient elevated
##       37        0

# In this case, we don't want to keep the empty factor level.
# Use droplevels to get rid of any empty levels:
pupae_amb2 <- droplevels(pupae_amb)
```

Try this yourself Compare the `summary` of `pupae_amb` and `pupae_amb2`, and note the differences.

3.2.1 Changing the levels of a factor

Sometimes you may want to change the levels of a factor, for example to replace abbreviations with more readable labels. To do this, we can assign new values with the `levels` function, as in the following example using the pupae data:

```
# Change the levels of T_treatment by assigning a character vector to the levels.
levels(pupae$T_treatment) <- c("Ambient","Ambient + 3C")

# Or only change the first level, using subscripting.
levels(pupae$T_treatment)[1] <- "Control"
```

Try this yourself Using the method above, you can also merge levels of a factor, simply by assigning the same new level to both old levels. Try this on a dataset of your choice (for example, in the `allom` data, you can assign new species levels, 'Douglas-fir' for PSME, and 'Pine' for both PIMO and PIPO). Then check the results with `levels()`.

3.3 Working with logical data

Some data can only take two values: true, or false. For data like these, **R** has the *logical* data type. Logical data are coded by integer numbers (0 = `FALSE`, 1 = `TRUE`), but normally you don't see this, since **R** will only *print* `TRUE` and `FALSE` 'labels'. However, once you know this, some analyses become even easier. Let's look at some examples,

```
# Answers to (in)equalities are always logical:
10 > 5

## [1] TRUE

101 == 100 + 1

## [1] TRUE

# ... or use objects for comparison:
apple <- 2
pear <- 3
apple == pear

## [1] FALSE

# NOT equal to.
apple != pear

## [1] TRUE

# Logical comparisons like these also work for vectors, for example:
nums <- c(10,21,5,6,0,1,12)
nums > 5

## [1]  TRUE  TRUE FALSE  TRUE FALSE FALSE  TRUE

# Find which of the numbers are larger than 5:
which(nums > 5)

## [1] 1 2 4 7

# Other useful functions are 'any' and 'all':
# Are any numbers larger than 25?
```

```
any(nums > 25)

## [1] FALSE

# Are all numbers less than or equal to 10?
all(nums <= 10)

## [1] FALSE

# Use & for AND, for example to take subsets where two conditions are met:
subset(pupae, PupalWeight > 0.4 & Frass > 3)

##    T_treatment CO2_treatment Gender PupalWeight Frass
## 25     ambient           400      1       0.405 3.117

# Use | for OR
nums[nums < 2 | nums > 20]

## [1] 21  0  1

# How many numbers are larger than 5?
#- Short solution
sum(nums > 5)

## [1] 4

#- Long solution
length(nums[nums > 5])

## [1] 4
```

We saw a number of 'logical operators', like == (is equal to), and > (is greater than) when we indexed vectors (see Section 2.3.1 and Table 2.1). The help page `?Syntax` has a comprehensive list of operators in **R** (including the logical operators).

3.4 Working with missing values

3.4.1 Basics

In **R**, missing values are represented with `NA`, a special data type that indicates the data is simply *Not Available*.

Warning: Because `NA` represents a missing value, make sure you never use 'NA' as an abbreviation for anything (like North America).

Many functions can handle missing data, usually in different ways. For example, suppose we have the following vector:

```
myvec1 <- c(11,13,5,6,NA,9)
```

In order to calculate the mean, we might want to either exclude the missing value (and calculate the mean of the remaining five numbers), or we might want `mean(myvec1)` to fail (produce an error). This last case is useful if we don't expect missing values, and want **R** to only calculate the mean when there are no `NA`'s in the dataset.

These two options are shown in this example:

```
# Calculate mean: this fails if there are missing values
mean(myvec1)
```

```
## [1] NA

# Calculate mean after removing the missing values
mean(myvec1, na.rm=TRUE)

## [1] 8.8
```

Many functions have an argument `na.rm`, or similar. Refer to the help page of the function to learn about the various options (if any) for dealing with missing values. For example, see the help pages `?lm` and `?sd`.

The function `is.na` returns `TRUE` when a value is missing, which can be useful to see which values are missing, or how many,

```
# Is a value missing? (TRUE or FALSE)
is.na(myvec1)

## [1] FALSE FALSE FALSE FALSE  TRUE FALSE

# Which of the elements of a vector is missing?
which(is.na(myvec1))

## [1] 5

# How many values in a vector are NA?
sum(is.na(myvec1))

## [1] 1
```

Making missing values

In many cases it is useful to change some bad data values to `NA`. We can use our indexing skills to do so,

```
# Some vector that contains bad values coded as -9999
datavec <- c(2,-9999,100,3,-9999,5)

# Assign NA to the values that were -9999
datavec[datavec == -9999] <- NA
```

In many cases, missing values may arise when certain operations did not produce the desired result. Consider this example,

```
# A character vector, some of these look like numbers:
myvec <- c("101","289","12.3","abc","99")

# Convert the vector to numeric:
as.numeric(myvec)

## Warning:  NAs introduced by coercion

## [1] 101.0 289.0  12.3    NA  99.0
```

The warning message `NAs introduced by coercion` means that missing values were produced by when we tried to turn one data type (character) to another (numeric).

Not A Number

Another type of missing value is the result of calculations that went wrong, for example:

```
# Attempt to take the logarithm of a negative number:
log(-1)
## Warning in log(-1):  NaNs produced
## [1] NaN
```

The result is `NaN`, short for *Not A Number*.

Dividing by zero is not usually meaningful, but **R** does not produce a missing value:

```
1000/0
## [1] Inf
```

It produces 'Infinity' instead.

3.4.2 Missing values in dataframes

When working with dataframes, you often want to remove missing values for a particular analysis. We'll use the `pupae` dataset for the following examples. See Section A.6 for a description of this dataset.

```
# Read the data
pupae <- read.csv("pupae.csv")

# Look at a summary to see if there are missing values:
summary(pupae)
##    T_treatment CO2_treatment       Gender        PupalWeight
##  ambient :37   Min.   :280.0   Min.   :0.0000   Min.   :0.1720
##  elevated:47   1st Qu.:280.0   1st Qu.:0.0000   1st Qu.:0.2562
##                Median :400.0   Median :0.0000   Median :0.2975
##                Mean   :344.3   Mean   :0.4487   Mean   :0.3110
##                3rd Qu.:400.0   3rd Qu.:1.0000   3rd Qu.:0.3560
##                Max.   :400.0   Max.   :1.0000   Max.   :0.4730
##                                NA's   :6
##      Frass
##  Min.   :0.986
##  1st Qu.:1.515
##  Median :1.818
##  Mean   :1.846
##  3rd Qu.:2.095
##  Max.   :3.117
##  NA's   :1
# Notice there are 6 NA's (missing values) for Gender, and 1 for Frass.

# Option 1: take subset of data where Gender is not missing:
pupae_subs1 <- subset(pupae, !is.na(Gender))

# Option 2: take subset of data where Frass AND Gender are not missing
pupae_subs2 <- subset(pupae, !is.na(Frass) & !is.na(Gender))

# A more rigorous subset: remove all rows from a dataset where ANY variable
# has a missing value:
pupae_nona <- pupae[complete.cases(pupae),]
```

3.4.3 Subsetting when there are missing values

When there are missing values in a vector, and you take a subset (for example all data larger than some value), should the missing values be included or dropped? There is no one answer to this, but it is important to know that `subset` drops them, but the square bracket method (`[]`) keeps them. See also Section 2.3.3, where we showed that these two methods to subset data differ in their behaviour.

Consider this example, and especially the use of `which` to drop missing values when subsetting.

```
# A small dataframe
dfr <- data.frame(a=1:4, b=c(4,NA,6,NA))

# subset drops all missing values
subset(dfr, b > 4, select=b)

##   b
## 3 6

# square bracket notation keeps them
dfr[dfr$b > 4,"b"]

## [1] NA  6 NA

# ... but drops them when we use 'which'
dfr[which(dfr$b > 4),"b"]

## [1] 6
```

3.5 Working with text

3.5.1 Basics

Many datasets include variables that are text only (think of comments, species names, locations, sample codes, and so on), it is useful to learn how to modify, extract, and analyse text-based ('character') variables.

Consider the following simple examples when working with a single character string:

```
# Count number of characters in a bit of text:
sentence <- "Not a very long sentence."
nchar(sentence)

## [1] 25

# Extract the first 3 characters:
substr(sentence, 1, 3)

## [1] "Not"
```

It gets more interesting when we have character vectors, for example,

```
# Substring all elements of a vector
substr(c("good","good riddance","good on ya"),1,4)

## [1] "good" "good" "good"

# Number of characters of all elements of a vector
nchar(c("hey","hi","how","ya","doin"))
```

```
## [1] 3 2 3 2 4
```

To glue bits of text together, use the `paste` function, like so:

```
# Add a suffix to each text element of a vector:
txt <- c("apple","pear","banana")
paste(txt, "-fruit")

## [1] "apple -fruit"  "pear -fruit"   "banana -fruit"

# Glue them all together into a single string using the collapse argument
paste(txt, collapse="-")

## [1] "apple-pear-banana"

# Combine numbers and text:
paste("Question", 1:3)

## [1] "Question 1" "Question 2" "Question 3"

# This can be of use to make new variables in a dataframe,
# as in this example where we combine two factors to create a new one:
pupae$T_CO2 <- with(pupae, paste(T_treatment, CO2_treatment, sep="-"))
head(pupae$T_CO2)

## [1] "ambient-280" "ambient-280" "ambient-280" "ambient-280" "ambient-280"
## [6] "ambient-280"
```

Try this yourself Run the final example above, and inspect the variable `T_CO2` (with `str`) that we added to the dataframe. Make it into a factor variable using `as.factor`, and inspect the variable again.

3.5.2 Column names

Vectors, dataframes and list can all have `names` that can be used to find rows or columns in your data. We already saw how you can use column names to index a dataframe in Section 2.3.2. Also consider the following examples:

```
# Change the names of a dataframe:
hydro <- read.csv("hydro.csv")
names(hydro)  # first print the old names

## [1] "Date"    "storage"

names(hydro) <- c("Date","Dam_Storage") # then change the names

# Change only the first name (you can index names() just like you can a vector!)
names(hydro)[1] <- "Datum"
```

Sometimes it is useful to find out which columns have particular names. We can use the `match` function to do this:

```
match(c("diameter","leafarea"), names(allom))

## [1] 2 4
```

Shows that the 2nd and 4th column have those names.

3.5.3 Text in dataframes and `grep`

When you read in a dataset (with `read.csv`, `read.table` or similar), any variable that **R** cannot convert to numeric *is automatically converted to a factor*. This means that if a column has even just one value that is text (or some garble that does not represent a number), the column cannot be numeric.

While we know that factors are very useful, sometimes we want a variable to be treated like text. For example, if we plan to analyse text directly, or extract numbers or other information from bits of text. Let's look at a few examples using the `cereal` dataset, described in Section A.7.

```
# Read data, tell R to treat the first variable ('Cereal.name') as character, not factor
cereal <- read.csv("cereals.csv", stringsAsFactors=FALSE)

# Make sure that the Cereal name is really a character vector:
is.character(cereal$Cereal.name)

## [1] TRUE

# The above example avoids converting any variable to a factor,
# what if we want to just convert one variable to character?
cereal <- read.csv("cereals.csv")
cereal$Cereal.name <- as.character(cereal$Cereal.name)
```

Here, the argument `stringsAsFactors=FALSE` avoided the automatic conversion of character variables to factors.

The following example uses `grep`, a very powerful function. This function can make use of *regular expressions*, a flexible tool for text processing.

```
# Extract cereal names (for convenience).
cerealnames <- cereal$Cereal.name

# Find the cereals that have 'Raisin' in them.
# grep() returns the index of values that contain Raisin
grep("Raisin",cerealnames)

##  [1] 23 45 46 50 52 53 59 60 61 71

# grepl() returns TRUE or FALSE
grepl("Raisin",cerealnames)

##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [23]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [34] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [45]  TRUE  TRUE FALSE FALSE FALSE  TRUE FALSE  TRUE  TRUE FALSE FALSE
## [56] FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE
## [67] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE

# That result just gives you the indices of the vector that have 'Raisin' in them.
# these are the corresponding names:
cerealnames[grep("Raisin",cerealnames)]

##  [1] "Crispy_Wheat_&_Raisins"
##  [2] "Muesli_Raisins,_Dates,_&_Almonds"
##  [3] "Muesli_Raisins,_Peaches,_&_Pecans"
##  [4] "Nutri-Grain_Almond-Raisin"
##  [5] "Oatmeal_Raisin_Crisp"
##  [6] "Post_Nat._Raisin_Bran"
##  [7] "Raisin_Bran"
##  [8] "Raisin_Nut_Bran"
##  [9] "Raisin_Squares"
## [10] "Total_Raisin_Bran"
```

```
# Now find the cereals whose name starts with Raisin.
# The ^ symbol is part of a 'regular expression', it indicates 'starts with':
grep("^Raisin",cerealnames)
```

```
## [1] 59 60 61
```

```
# Or end with 'Bran'
# The $ symbol is part of a 'regular expression', and indicates 'ends with':
grep("Bran$", cerealnames)
```

```
##  [1]  1  2  3 20 29 53 59 60 65 71
```

As mentioned, `grep` can do a *lot* of different things, so don't be alarmed if you find the help page *a bit overwhelming*. However, there are a few options worth knowing about. One very useful option is to turn off the case-sensitivity, for example:

```
grep("bran",cerealnames,ignore.case=TRUE)
```

```
##  [1]  1  2  3  4  9 10 20 29 53 59 60 65 71
```

finds `Bran` and `bran` and `BRAN`.

Finally, using the above tools, let's add a new variable to the `cereal` dataset that is `TRUE` when the name of the cereal ends in 'Bran', otherwise it is `FALSE`. For this example, the `grepl` function is more useful (because it returns TRUE and FALSE).

```
# grepl will return FALSE when Bran is not found, TRUE otherwise
cereal$BranOrNot <- grepl("Bran$", cerealnames)

# Quick summary:
summary(cereal$BranOrNot)
```

```
##    Mode   FALSE    TRUE
## logical      67      10
```

Regular expressions are very useful, and with the right knowledge, you can specify almost any possible text string. Below we've included a quick cheat sheet that shows some of the ways they can be used. For more examples and explanation, please see the *Further Reading* at the end of this section.

Table 3.1: A regular expression cheat sheet.

Symbol	**What it means**	**Example**	**Matches**	**Doesn't match**
^*abc*	Matches any string that starts with *abc*	`^eco`	`economics, ecology`	`evo-eco`
abc$	Matches any string that ends with *abc*	`ology$`	`biology, ecology`	`ologies`
.	Matches any character except newline.	`b.t`	`bat, bet, bit`	`beet`
a|*b*	Matches string *a* or string *b*.	`green|blue`	`green, blue`	`red`

Table 3.1: A regular expression cheat sheet.

Symbol	**What it means**	**Example**	**Matches**	**Doesn't match**
`*`	Matches a given string 0 or more times.	`ab*a`	`aa, abba, abbba, abbbba`	`ba, ab`
`?`	Matches a given string 0 times or 1 time – no more.	`ab?a`	`aa, aba`	`abba, abbbba`
(*abc*)	Groups a string to be matched as part of an expression.	`(4570)`	`4570 1125, 4570 1617`	`9772 6585, 9772 6100`
[*abc*]	Matches any of the characters inside the brackets.	`gr[ae]y`	`gray, grey`	`greey, graey`
[^*abc*]	Matches any characters *not* inside the brackets.	`b[^e]`	`bid, ban`	`bees, bed`
-	Within brackets, gives a range of characters to be matched.	`[a-c]t`	`at, bt, ct`	`dt, et`
{*n*}	Used to give the number of times to match the preceding group.	`ab{2}a`	`abba`	`aba, abbba`
`\\`	Used to "escape" a special character so it can be matched as part of an expression. Note than in **R**, you must use a double \\.	`why\\?`	`why?, why ask why?`	`why not`
`\\s`	Matches any whitespace character.	`day:\\s[A-z]`	`day: monday, day: Fri`	`day:Fri`
`\\w`	Matches any whole word.	`my \\w`	`my hobby, my name`	`his program`
`\\d`	Matches any digit.	`\\d years old`	`5 years old, 101.6 years old`	`six years old`

Further reading "Some people, when confronted with a problem, think 'I know, I'll use regular expressions.' Now they have two problems." – Jamie Zawinski, comp.emacs.xemacs, 1997
If *you* would like to know more about how regular expressions work, you can start with the Wikipedia page `http://en.wikipedia.org/wiki/Regular_expressions`. Jan Goyvaerts (who seems to have dedicated his life to explaining regular expressions) has a helpful quick start guide at `http://www.regular-expressions.info/quickstart.html`. The same page also has a tutorial. Also worth a look is Robin Lovelace's introduction specifically for **R** and RStudio at `https://www.r-bloggers.com/regular-expressions-in-r-vs-rstudio/`. Finally, a very useful and more detailed cheat sheet can be found at `http://regexlib.com/CheatSheet.aspx`.

3.6 Working with dates and times

Admittedly, working with dates and times in **R** is somewhat annoying at first. The built-in help files on this subject describe all aspects of this special data type, but do not offer much for the beginning **R** user. This section covers basic operations that you may need when analysing and formatting datasets.

For working with dates, we use the `lubridate` package, which simplifies it tremendously.

3.6.1 Reading dates

The built-in `Date` class in **R** is encoded as an integer number representing the number of days since 1-1-1970 (but this actual origin does not matter for the user). Converting a character string to a date with `as.Date` is straightforward if you use the standard order of dates: `YYYY-MM-DD`. So, for example,

```
as.Date("2008-5-22")

## [1] "2008-05-22"
```

The output here is not interesting, **R** simply prints the date. Because dates are represented as numbers in **R**, we can do basic arithmetic:

```
# First load lubridate when working with dates or date-time combinations.
# (although as.Date is part of the base package)
library(lubridate)

##
## Attaching package:  'lubridate'

## The following object is masked from 'package:base':
##
##     date

# A date, 7 days later:
as.Date("2011-5-12") + 7

## [1] "2011-05-19"

# Difference between dates.
as.Date("2009-7-1") - as.Date("2008-12-1")

## Time difference of 212 days

# With difftime, you can specify the units:
difftime(as.Date("2009-7-1"), as.Date("2008-12-1"), units = "weeks")

## Time difference of 30.28571 weeks

# To add other timespans, use functions months(), years() or weeks() to
# avoid problems with leap years
as.Date("2013-8-18") + years(10) + months(1)

## [1] "2023-09-18"
```

Try this yourself The previous example showed a very useful trick for adding numbers to a `Date` to get a new `Date` a few days later. Confirm for yourself that this method accounts for leap years. That is, the day before `2011-3-1` should be `2011-2-28` (2011 is not a leap year). But what about `2012-3-1`?

Often, text strings representing the date are not in the standard format. Fortunately, it is possible to convert any reasonable sequence to a `Date` object in **R**. All we have to do is provide a character string to `as.Date` and tell the function the order of the fields.

To convert any format to a Date, we can use the `lubridate` package, which contains the functions `ymd`, `mdy`, and all other combinations of y, m, and d. These functions are pretty smart, as can be seen in these examples:

```
# Load lubridate
library(lubridate)

# Day / month / year
as.Date(dmy("31/12/1991"))

## [1] "1991-12-31"

# Month - day - year (note, only two digits for the year)
as.Date(mdy("4-17-92"))

## [1] "1992-04-17"

# Year month day
as.Date(ymd("1976-5-22"))

## [1] "1976-05-22"

#-- Unusual formatting can be read in with the 'format' argument
# in as.Date. See ?strptime for a list of codes.
# For example, Year and day of year ("%j" stands for 'Julian date')
as.Date("2011 121", format="%Y %j")

## [1] "2011-05-01"
```

Another method to construct date objects is when you do not have a character string as in the above example, but separate numeric variables for year, month and day. In this case, use the `ISOdate` function:

```
as.Date(ISOdate(2008,12,2))

## [1] "2008-12-02"
```

Finally, here is a simple way to find the number of days since you were born, using `today` from the lubridate package.

```
# Today's date (and time) can be with the today() function
today()

## [1] "2019-09-17"

# We can now simply subtract your birthday from today's date.
today() - as.Date("1976-5-22")

## Time difference of 15823 days
```

Example: using dates in a dataframe

The `as.Date` function that we met in the previous section also works with vectors of dates, and the `Date` class can also be part of a dataframe. Let's take a look at the Hydro data (see description in Section A.3), to practice working with dates.

```
# Read dataset
hydro <- read.csv("Hydro.csv")
```

```
# Now convert this to a Date variable.
# If you first inspect head(hydro$Date), you will see the format is DD/MM/YYYY
hydro$Date <- as.Date(dmy(hydro$Date))
```

If any of the date conversions go wrong, the `dmy` function (or its equivalents) should print a message letting you know. You can double check if `any` of the converted dates is `NA` like this:

```
any(is.na(hydro$Date))

## [1] FALSE
```

We now have successfully read in the date variable. The `min` and `max` functions are useful to check the range of dates in the dataset:

```
# Minimum and maximum date (that is, oldest and most recent),
min(hydro$Date)

## [1] "2005-08-08"

max(hydro$Date)

## [1] "2011-08-08"

#... and length of measurement period:
max(hydro$Date) - min(hydro$Date)

## Time difference of 2191 days
```

Finally, the `Date` class is very handy when plotting. Let's make a simple graph of the Hydro dataset. The following code produces Fig. 3.1. Note how the `X` axis is automatically formatted to display the date in a (fairly) pretty way.

```
with(hydro, plot(Date, storage, type='l')) # note plot type is letter 'l', not number '1'
```

3.6.2 Date-Time combinations

For dates that include the time, **R** has a special class called `POSIXct`. The `lubridate` package makes it easy to work with this class.

Internally, a date-time is represented as the number of seconds since the 1st of January, 1970. Time zones are also supported, but we will not use this functionality in this book (as it can be quite confusing).

From the lubridate package, we can use any combination of (y)ear,(m)onth,(d)ay, (h)our, (m)inutes, (s)econds. For example `ymd_hms` converts a character string in that order.

Let's look at some examples,

```
# Load lubridate
library(lubridate)

# The standard format is YYYY-MM-DD HH:MM:SS
ymd_hms("2012-9-16 13:05:00")

## [1] "2012-09-16 13:05:00 UTC"

# Read two times (note the first has no seconds, so we can use ymd_hm)
time1 <- ymd_hm("2008-5-21 9:05")
time2 <- ymd_hms("2012-9-16 13:05:00")
```

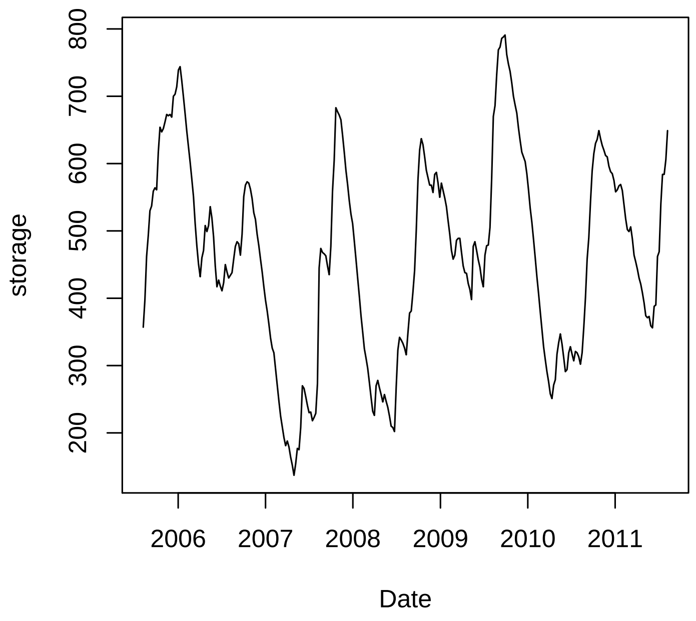


Figure 3.1: A simple plot of the hydro data.

```
# Time difference:
time2 - time1

## Time difference of 1579.167 days

# And an example with a different format, DD/M/YY H:MM
dmy_hm("23-1-89 4:30")

## [1] "1989-01-23 04:30:00 UTC"

# To convert a date-time to a Date, you can also use the as.Date function,
# which will simply drop the time.
as.Date(time1)

## [1] "2008-05-21"
```

As with `Date` objects, we can calculate timespans using a few handy functions.

```
# What time is it 3 hours and 15 minutes from now?
now() + hours(3) + minutes(15)

## [1] "2019-09-17 15:37:29 AEST"
```

Try this yourself The 2012 Sydney marathon started at 7:20AM on September 16th. The winner completed the race in 2 hours, 11 minutes and 50 seconds. What was the time when the racer crossed the finish line? Using the `weekdays` function, which day was the race held?

Example: date-times in a dataframe

Now let's use a real dataset to practice the use of date-times. We also introduce the functions `month`, `yday`, `hour` and `minute` to conveniently extract components of date-time objects.

The last command produces Fig. 3.2.

```
# Read the 2008 met dataset from the HFE.
hfemet <- read.csv("HFEmet2008.csv")

# Convert 'DateTime' to POSIXct class.
# The order of the original data is MM/DD/YYYY HH:MM
hfemet$DateTime <- mdy_hm(hfemet$DateTime)

# It is often useful to add the Date as a separate variable
hfemet$Date <- as.Date(hfemet$DateTime)

# Make sure they all converted OK (if not, NAs would be produced)
any(is.na(hfemet$DateTime))

## [1] FALSE

# FALSE is good here!

# Add day of year
hfemet$DOY <- yday(hfemet$DateTime)

# Add the hour, minute and month to the dataframe:
hfemet$hour <- hour(hfemet$DateTime)
hfemet$minute <- minute(hfemet$DateTime)

# Add the month. See ?month to adjust the formatting of the month
# (it can be the full month name, for example)
hfemet$month <- month(hfemet$DateTime)

# Now produce a plot of air temperature for the month of June.
with(subset(hfemet, month==6), plot(DateTime, Tair, type='l'))
```

```
# We can also take a subset of just one day, using the Date variable we added:
hfemet_oneday <- subset(hfemet, Date == as.Date("2008-11-1"))
```

3.6.2.1 Sequences of dates and times

It is often useful to generate sequences of dates. We can use `seq` as we do for numeric variables (as we already saw in Section 1.6.1).

```
# A series of dates, by day:
seq(from=as.Date("2011-1-1"), to=as.Date("2011-1-10"), by="day")

##  [1] "2011-01-01" "2011-01-02" "2011-01-03" "2011-01-04" "2011-01-05"
##  [6] "2011-01-06" "2011-01-07" "2011-01-08" "2011-01-09" "2011-01-10"

# Two-weekly dates:
seq(from=as.Date("2011-1-1"), length=10, by="2 weeks")

##  [1] "2011-01-01" "2011-01-15" "2011-01-29" "2011-02-12" "2011-02-26"
##  [6] "2011-03-12" "2011-03-26" "2011-04-09" "2011-04-23" "2011-05-07"
```

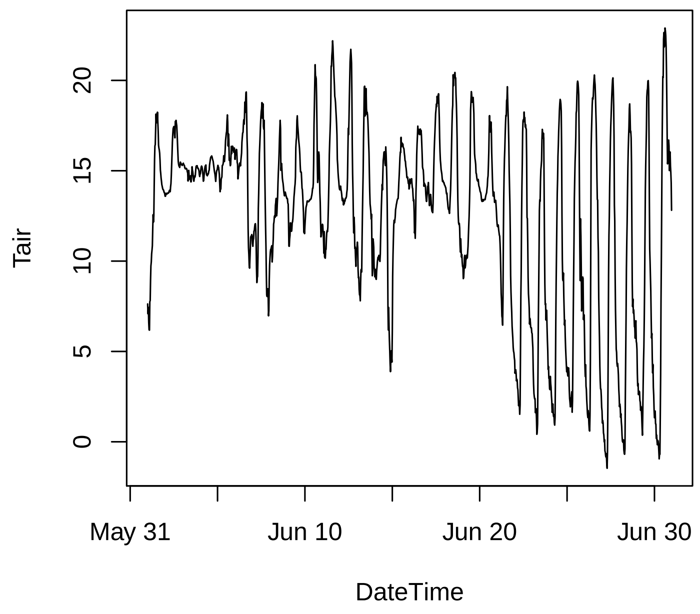


Figure 3.2: Air temperature for June at the HFE

```
# Monthly:
seq(from=as.Date("2011-12-13"), length=5, by="months")

## [1] "2011-12-13" "2012-01-13" "2012-02-13" "2012-03-13" "2012-04-13"
```

Similarly, you can generate sequences of date-times.

```
# Generate a sequence with 30 min timestep:
# (Note that if we don't specify the time, it assumes midnight!)
# Here, the 'by' field specifies the timestep in seconds.
fromtime <- ymd_hms("2012-6-1 0:00:00")
halfhours <- seq(from=fromtime, length=12, by=30*60)
```

3.7 Converting between data types

It is often useful, or even necessary, to convert from one data type to another. For example, when you read in data with `read.csv` or `read.table`, any column that contains some non-numeric values (that is, values that cannot be converted to a number) will be converted to a factor variable. Sometimes you actually want to convert it to numeric, which will result in some missing values (`NA`) when the value could not be converted to a number.

Another common example is when one of your variables should be read in as a factor variable (for example, a column with treatment codes), but because all the values are numeric, **R** will simply assume it is a numeric column.

Before we learn how to convert, it is useful to make sure you know what type of data you have to begin with. To find out what type of data a particular vector is, we use `str` (this is also useful for any other object in **R**).

```
# Numeric
y <- c(1,100,10)
str(y)

##  num [1:3] 1 100 10

# This example also shows the dimension of the vector ([1:3]).

# Character
x <- "sometext"
str(x)

##  chr "sometext"

# Factor
p <- factor(c("apple","banana"))
str(p)

##  Factor w/ 2 levels "apple","banana": 1 2

# Logical
z <- c(TRUE,FALSE)
str(z)

##  logi [1:2] TRUE FALSE

# Date
sometime <- as.Date("1979-9-16")
str(sometime)

##  Date[1:1], format: "1979-09-16"

# Date-Time
library(lubridate)
onceupon <- ymd_hm("1969-8-18 09:00")
str(onceupon)

##  POSIXct[1:1], format: "1969-08-18 09:00:00"
```

Try this yourself Confirm that `str` gives more information than simply printing the object. Try printing some of the objects above (simply by typing the name of the object), and compare the output to `str`.

To test for a particular type of data, use the `is.` functions, which give `TRUE` if the object is of that type, for example:

```
# Test for numeric data type:
is.numeric(c(10,189))

## [1] TRUE

# Test for character:
is.character("HIE")

## [1] TRUE
```

Try this yourself Try the functions `is.factor` and `is.logical` yourself, on one the previous examples. Also try `is.Date` on a `Date` variable (note that you must first load the `lubridate` package for this to work).

We can convert between types with the `as.`*`something`* class of functions.

```
# First we make six example values that we will use to convert
mynum <- 1001
mychar <- c("1001","100 apples")
myfac <- factor(c("280","400","650"))
mylog <- c(TRUE,FALSE,FALSE,TRUE)
mydate <- as.Date("2015-03-18")
mydatetime <- ymd_hm("2011-8-11 16:00")

# A few examples:

# Convert to character
as.character(mynum)

## [1] "1001"

as.character(myfac)

## [1] "280" "400" "650"

# Convert to numeric
# Note that one missing value is created
as.numeric(mychar)

## Warning:  NAs introduced by coercion

## [1] 1001   NA

# Warning!!!
# When converting from a factor to numeric, first convert to character
# !!!
as.numeric(as.character(myfac))

## [1] 280 400 650

# Convert to Date
as.Date(mydatetime)

## [1] "2011-08-11"

# Convert to factor
as.factor(mylog)

## [1] TRUE  FALSE FALSE TRUE
## Levels: FALSE TRUE
```

Try this yourself Try some more conversions using the above examples, and check the results with `str` and the `is.`*`something`* functions (e.g. `is.character`). In particular, see what happens when you convert a logical value to numeric, and a factor variable to numeric (without converting to character first).

When you want to change the type of a variable in a dataframe, simply store the converted variable in the dataframe, using the same name. For example, here we make the `CO2_treatment` variable in the pupae dataset a factor:

```
pupae <- read.csv("pupae.csv")
pupae$CO2_treatment <- as.factor(pupae$CO2_treatment)
```

3.8 Functions used in this chapter

For functions not listed here, please refer to the index at the end of this book.

Function	What it does	Example use

`all`	Given a vector of logical values, returns `TRUE` if they are all true, `FALSE` if at least one is false.	`all(c(TRUE, TRUE, FALSE)`
		`all(letters == "a")`

`any`	Given a vector of logical values, returns a single logical value. `TRUE` if there is at least one `TRUE` value, `FALSE` if there is not.	`any(c(TRUE, FALSE, FALSE)`
		`any(letters == "a")`

`as.character`	Converts a variable to the `character` data type. This data type can hold either single characters (letters and symbols) or strings of characters (words and text). Particular useful for converting between `factor` and `numeric` data types.	`as.character(c(0,0,0,1))`
		`as.character(esoph$alcgp)`

`as.Date`	Used to convert a character string to the `Date` data type. Default order is YYYY-MM-DD. You can also specify a custom format.	`as.Date("2014-02-07")`
		`as.Date("07/02/14",`
		`    format="%d/%m/%y")`

`as.factor`	Converts a variable to the `factor` data type. Factors are categorical variables (for example, "Treatment" and "Control".) For converting numeric variables, see Section 3.2.	`as.factor(rep(c("F","M"),`
		`    each=5))`

`as.logical`	Converts a variable to the `logical` data type. Logical values can hold only `TRUE` and `FALSE` values, represented by underlying `0`s and `1`s.	`as.logical(c(0,0,0,1))`

`as.numeric`	Used to convert a variable to the `numeric` data type. Can hold whole or decimal numbers. Use caution when converting a `factor`. The result will be the underlying numeric representation, and may not match numerical values in the level names.	`as.numeric(c("1.5","2.6"))`
		`as.numeric(factor(`
		`    c("25","30","35")))`

`cat`	Glues together bits of text and outputs them as a file. You need to specify the file name (`file`) and a character to separate the bits of text (`sep`.) Use `sep=" "` for spaces, and `sep="\n"` for newline.	`cat("Roses are red",`
		`    "violets are blue",`
		`    "you love R",`
		`    "R loves you too!",`
		`    file="poem.txt",`
		`    sep="\n")`

Function	**What it does**	**Example use**
`complete.cases`	Finds all complete cases in a dataframe, returning TRUE for complete rows and FALSE for rows with an NA. Can be used to remove all rows with NAs by subsetting the dataset for `complete.cases`.	`airquality[ complete.cases( airquality),]`
`cut`	Can be used to divide a numerical variable into bins, in order to turn it into a factor variable. You must provide data to be broken and a number of `breaks`. Optionally, you can also provide `labels`.	`grades <- cut(50:100, 4, labels=c("P","C", "D","HD"))`
`day`	From the `lubridate` package. When given a `Date`, returns the day of the month.	`day(as.Date("2016 121", format="%Y %j")`
`days`	From the `lubridate` package. Easily creates time periods of a given number of days, which can be used for `Date` arithmetic.	`as.Date("2012-2-28") + days(2)`
`describe`	From the `Hmisc` package. Gives a statistical summary of a data set.	`describe(mtcars)` `describe(airquality)`
`diff`	Calculates the sequential difference between a series of numbers. The result is returned as a vector.	`diff(airquality$Wind)`
`difftime`	Given two times, calculates the difference between them. You can specify which `units` you would like the result returned in (`secs, mins, hours, days, weeks`).	`difftime(now(), dmy("19011980"), units="weeks")`
`dmy`	From the `lubridate` package. Searches for a day, month and year (in that order) in a text string or a vector of strings. Can handle oddly formatted input, as long as the order is correct. Functions `mdy`, `ymd` and all other variations also exist.	`dmy("Born 12 July 1970")` `dmy("170914")`
`droplevels`	Drops unused factor levels.	`droplevels(subset( esoph, tobgp != "30+"))`
`factor`	Can be used to create a new factor variable. When creating a new factor, `levels` specifies possible values for your factor levels, while `labels` specifies what the `levels` should be called. It is also possible to specify whether your factor `ordered` or not. Factors are ordered by the order given in `levels`.	`factor(quakes$stations)` `factor(rep(1:3, each=5), levels=c(1,2,3))` `factor(rep(1:3, each=5), labels=c("B","A","C"), ordered=TRUE)`

Function	What it does	Example use

`grep`	Used to search for text strings using regular expressions. For a case-insensitive search, use `ignore.case=TRUE`. By default, returns the position of matches, but can also be set to return matching strings using `value=TRUE`. For more on regular expressions, see Section 3.5.3 and Table 3.1.	`grep("r", letters)`
		`grep("R", letters)`
		`grep("R", letters, ignore.case=TRUE)`
		`grep("^Merc", row.names(mtcars), value=TRUE)`

`grepl`	The same as `grep`, but returns a vector of logical values (matches are TRUE, non-matches are FALSE). For more on regular expressions, see Section 3.5.3 and Table 3.1.	`grepl("a", c("a","b","a"))`
		`grepl("[H-P]", letters, ignore.case=TRUE)`
		`grepl("^Merc", row.names(mtcars))`

`gsub`	Uses regular expressions to search for matching strings, then replace them. Replaces all matches found. Case sensitivity can be set using `ignore.case`. For more on regular expressions, see Section 3.5.3 and Table 3.1.	`gsub("b[^e]", "be", c("bid", "ban", "bun"))`
		`gsub("gray", "grey", ignore.case=TRUE, "Gray clouds reflected in gray puddles")`

`hour`	From the `lubridate` package. When given a date-time object, returns the hour. Uses a 24 hour clock.	`hour(now())`
`hours`	From the `lubridate` package. Easily creates time periods of a given number of hours, which can be used for `Date` arithmetic.	`now() + hours(2)`
`ifelse`	Takes a logical test, a value to return if TRUE (`yes`), and a value to return if FALSE (`no`).	`ifelse(mtcars$wt <= 3, yes="light", no="heavy")`
`is.character`	Tests whether a given variable is the `character` data type. Returns `TRUE` or `FALSE`.	`is.character("?")`
`is.Date`	Tests whether a given variable is the `Date` data type. Returns `TRUE` or `FALSE`.	`is.Date(today())`
`is.factor`	Tests whether a given variable is the `factor` data type. Returns `TRUE` or `FALSE`.	`is.factor(esoph$agegp)`
`is.logical`	Tests whether a given variable is the `logical` data type. Returns `TRUE` or `FALSE`.	`is.logical(c(TRUE,FALSE))`
`is.na`	When given a list of values, returns `TRUE` if any are `NA`. Note that this is the only way to get logical values for an `NA`; `NA == NA` returns `NA`.	`is.na(c(1,1,2,NA))`
`is.numeric`	Tests whether a given variable is the `numeric` data type. Returns `TRUE` or `FALSE`.	`is.numeric(mtcars$wt)`

Function	**What it does**	**Example use**
`ISOdate`	Given a numerical `year`, `month`, and `day`, returns a date-time object. There is also an `ISOdatetime` function, which adds `hours`, `min` and `sec`.	`ISOdate(2017, 1, 30)` `ISOdatetime(2014, 2, 7, 13, 31, 0)`
`levels`	Used to get or set levels of a factor variable.	`levels(esoph$tobgp)` `levels(esoph$tobgp) <- c("little","light", "medium","heavy")`
`match`	Looks for matches in a list of values. Returns the position of the first match found, in order.	`match(c("t","o","p"), letters)` `match(c("p","o","t"), letters)`
`max`	Returns the largest value in its input. Can also be used with dates.	`max(ymd(010119),`
mdy(010119))		
`mdy`	From the `lubridate` package. Searches for a month, day and year in a text string or a vector of strings. Can handle oddly formatted input, as long as the order is correct. Functions `dmy`, `ymd` and all other variations also exist.	`mdy("Created June 1 2015")` `mdy("122516")`
`min`	Returns the smallest value in its input. Can also be used with dates.	`min(ymd(010119),`
mdy(010119))		
`minute`	From the `lubridate` package. When given a date-time, returns the minute of the hour.	`minute(ymd_hm( "2012-09-16 7:20"))`
`minutes`	From the `lubridate` package. Easily creates time periods of a given number of minutes, which can be used for date-time arithmetic.	`ymd_hm("2012-09-16 7:20") + minutes(20)`
`month`	From the `lubridate` package. When given a `Date`, returns the month of the year.	`month(dmy("24 Dec 2016"))`
`months`	From the `lubridate` package. Easily creates time periods of a given number of months, which can be used for `Date` arithmetic.	`mdy("060708") + months(5)`
`names`	Used to get or set the name or names of a variable.	`names(iris)` `names(iris) <- c("slen", "swid","plen", "pwid","spp")`
`nchar`	Used to find the number of characters in each element of a character vector.	`nchar(names(mtcars))`

Function	What it does	Example use
`nlevels`	Used to find the number of levels for a factor variable.	`nlevels(iris$Species)`
`now`	From the `lubridate` package. Returns the current time and date as a date-time object.	`now() + minutes(45)`

`paste`	Used to glue bits of text together, turning them into a single string. Can be used on vectors. You must specify a character to separate the bits of text (`sep`). Use `sep=""` for no separation. Can also turn a vector of strings into a single string using `collapse`. Set collapse equal to the characters to be used to glue the elements together.	`paste("t","o","p", sep="")`
		`paste("Iris",`
		`    levels(iris$Species),`
		`        sep=" ")`
		`paste("Iris",`
		`    levels(iris$Species),`
		`        collapse=", ")`

`read.csv`	Reads in data from comma-separated variable (CSV) files. Important arguments include the file name (or complete file path) and whether or not the file has a `header`. For more details, see Section 2.1.1.	`read.csv("Allometry.csv")`
		`read.csv("Allometry.csv",`
		`          header=TRUE)`

`read.table`	A flexible function for reading data with different separating characters. To use `read.table`, you need to specify the character used to separate the data (`sep`) and whether the file has a header row (`header`). There are also many other options you can set to customize this function; for more see `?read.table`.	`read.table("tab.txt",`
		`    header=TRUE)`
		`read.table("semi.txt",`
		`    sep=";", header=TRUE)`

`readLines`	Reads each line of a text file as a separate string and creates a character vector. (The first part of this example makes a new text file; the second part shows how `readLines` is used to read it in.)	`cat("Roses are red",`
		`    "violets are blue",`
		`    "you love R",`
		`    "R loves you too!",`
		`    file="poem.txt",`
		`    sep="\n")`
		`readLines("poem.txt")`

`rownames`	Used to get or set the names of rows in a dataframe.	`rownames(mtcars)`
		`rownames(mtcars) <-`
		`    gsub(" ", ".",`
		`        rownames(mtcars))`

`sample`	Takes a given number of samples from a vector or a matrix. Can be set to sample with or without replacement. Default is without replacement.	`sample(1:6, size=6)`
		`sample(1:6, 6,`
		`    replace=TRUE)`

`seq`	Generates a sequence from a given value to another value. Can be used on numbers, dates, or date-times.	`seq(1, 10, by=2.5)`
		`seq(from=ymd("170130"),`
		`    length=6, by="months")`

Function	What it does	Example use

`sort`	Sorts a variable in ascending or descending order. Returns the reordered variable.	`sort(10:1, decreasing=FALSE)`
		`sort(LETTERS, decreasing=TRUE)`

`str`	Gives information about the structure of an object. Can be used to find out which types of data a dataframe contains.	`str(iris)`
		`str(esoph)`

`strsplit`	Splits a text string into multiple smaller strings. The first argument, a string, is split by searching for the second argument, a regular expression. For more on regular expressions, see Section 3.5.3 and Table 3.1.	`strsplit(`
		`    "Roses are red", " ")`

`subset`	Allows the selection of part of a dataframe, using logical operators to define which rows or columns should be included.	`subset(mtcars, wt > 3)`

`substr`	Can be used to extract or replace part of a string. Takes a string, which character to start on, and which the character to stop on.	`substr("Roses are red",11,13)`
		`r.txt <- c("Roses","are","red")`
		`  substr(r.txt,1,2) <- c("")`
		`  r.txt`

`summary`	Provides a summary of the variables in a dataframe. Statistics are given for numerical variables, and counts are given for factor variables.	`summary(warpbreaks)`
`table`	Given a factor variable, can create a table of counts.	`table(esoph$agegp)`
`today`	Returns today's date (as a `Date`). Can be used for `Date` arithmetic	`today() + days(10)`
`unique`	Returns only the unique values in a vector.	`unique(c(1,2,1,5,5,2))`
`weekdays`	Can be used to get the day of the week from a `Date` object.	`weekdays(today())`

`which`	Given a vector of logical values, returns the positions of any that are TRUE. Can be combined with other logical tests.	`which(c(FALSE, TRUE, TRUE))`
		`which(letters == "s")`
		`which(is.na(c(1,2,NA,2,NA)))`

`yday`	From the `lubridate` package. Gets the day of the year from a date object.	`yday(today())`
`year`	From the `lubridate` package. When given a `Date` object, returns the year.	`year(now())`

`years`	From the `lubridate` package. Easily creates time periods of a given number of years, which can be used for `Date` arithmetic.	`today()`
		`    + years(2)`

Function	What it does	Example use
`ymd`	From the `lubridate` package. Searches for a year, month and day (in that order) in a text string or a vector of strings. Can handle oddly formatted input, as long as the order is correct. Functions `mdy`, `dmy` and all other variations also exist.	`ymd("Copyright 2000 June 20")` `ymd("17890714")`

3.9 Exercises

In these exercises, we use the following colour codes:

🟩 **Easy**: make sure you complete some of these before moving on. These exercises will follow examples in the text very closely.

♦ **Intermediate**: a bit harder. You will often have to combine functions to solve the exercise in two steps.

▲ **Hard**: difficult exercises! These exercises will require multiple steps, and significant departure from examples in the text.

We suggest you complete these exercises in an **R** markdown file. This will allow you to combine code chunks, graphical output, and written answers in a single, easy-to-read file.

3.9.1 Titanic

For this section, read the data described in Section A.12 (p. 251). *Note*: the data are TAB-delimited, use `read.table` as shown on p. 251.

1. 🟩 Convert the 'Name' (passenger name) variable to a 'character' variable, and store it in the dataframe. See Section 3.5.3 (p. 68).
2. 🟩 How many observations of 'Age' are missing from the dataframe? See examples in Section 3.4 (p. 63).
3. 🟩 Make a new variable called 'Status', based on the 'Survived' variable already in the dataset. For passengers that did not survive, Status should be 'dead', for those who did, Status should be 'alive'. Make sure this new variable is a factor. See the example with the `ifelse` function in Section 3.2.
4. 🟩 Count the number of passengers in each class (1st, 2nd, 3rd). *Hint:* use `table` as shown in Section 3.2 (p. 59).
5. ♦ Using `grep`, find the six passengers with the last name 'Fortune'. Make this subset into a new dataframe. Did they all survive? *Hint:* to do this, make sure you recall how to use one vector to index a dataframe (see Section 2.3.2). Also, the `all` function might be useful here (see Section 3.3, p. 62).
6. ♦ As in 2., for what proportion of the passengers is the age unknown? Was this proportion higher for 3rd class than 1st and 2nd? *Hint:* First make a subset of the dataframe where age is missing (see Section 3.4.2 on p. 65), and then use `table`, as well as `nrow`.

3.9.2 Hydro dam

Use the hydro dam data as described in Section A.3.

1. 🟩 Start by reading in the data. Change the first variable to a `Date` class (see Section 3.6.1, p. 71).
2. ♦ Are the successive measurements in the dataset always exactly one week apart? *Hint:* use `diff`).
3. ♦ Assume that a dangerously low level of the dam is 235 Gwh. How many weeks was the dam level equal to or lower than this value?
4. ▲ For question 2., how many times did `storage` decrease below 235 (regardless of how long it remained below 235)? *Hint:* use `diff` and `subset`).

3.9.3 HFE tree measurements

Use the data for the HFE irrigation x fertilisation experiment (see Section A.15, p. 252).

1. ■ Read the data and look at various summaries of the dataset. Use `summary`, `str` and `describe` (the latter is in the `Hmisc` package).
2. ■ From these summaries, find out how many missing values there are for `height` and `diameter`. Also count the number of missing values as shown in Section 3.4 (p. 63).
3. ■ Inspect the levels of the treatment (`treat`), with the `levels` function. Also count the number of levels with the `nlevels` function. Now assign new levels to the factor, replacing the abbreviations with a more informative label. Follow the example in Section 3.2.1 (p. 62).
4. ■ Using `table`, count the number of observations by `treat`, to check if the dataset is balanced. Be aware that `table` simply counts the number of rows, regardless of missing values. Now take a subset of the dataset where `height` is not missing, and check the number of observations again.
5. ⧫ For which dates do missing values occur in `height` in this dataset? *Hint:* use a combination of `is.na` and `unique`.

3.9.4 Flux data

In this exercise, you will practice useful skills with the flux tower dataset. See Section A.8 (p. 249) for a description of the dataset.

1. ■ Read the dataframe. Rename the first column to 'DateTime' (recall Section 3.5.2 on p. 67).
2. ■ Convert DateTime to a `POSIXct` class. Beware of the formatting (recall Section 3.6.2 on p. 73).
3. ⧫ Did the above action produce any missing values? Were these already missing in the original dataset?
4. ■ Add a variable to the dataset called 'Quality'. This variable should be 'bad' when the variable 'ustar' is less than 0.15, and 'good' otherwise. Recall the example in Section 3.2 (p. 59).
5. ■ Add a 'month' column to the dataset, as well as 'year'.
6. ⧫ Look at the 'Rain' column. There are some problems; re-read the data or find another way to display `NA` whenever the data have an invalid value. *Hint:* look at the argument `na.strings` in `read.table`.

3.9.5 Alphabet Aerobics 3

In this exercise you will practice a bit more working with text, using the lyrics of the song 'Alphabet Aerobics' by Blackalicious. The lyrics are provided as a text file, which we can most conveniently read into a vector with `readLines`, like this,

```
lyric <- readLines("alphabet.txt")
```

1. ■ Read the text file into a character vector like above. Count the number of characters in each line (*Hint* : use `nchar`).
2. ⧫ Extract the first character of each line (recall examples in Section 3.5 on p. 66), and store it in a vector. Now sort this vector alphabetically and compare it to the unsorted vector. Are they the same? (*Hint* : use the == operator to compare the two vectors). How many are the same, that is, how many of the first letters are actually in alphabetical order already?

3. ▲ Find the most frequent word used in the lyrics. To do this, first paste all lyrics together into one string, then split the string into words, remove commas, then count the words. You will need to use a new function that we will see again in Section 8.2.2. *Hint* : use a combination of `paste`, `strsplit`, `gsub`, `table` and `sort`.

3.9.6 DNA Aerobics

DNA sequences can also be represented using text strings. In this exercise, you will make an artificial DNA sequence.

1. ▲ Make a random DNA sequence, consisting of a 100 random selections of the letters `C`,`A`,`G`,`T`, and paste the result together into one character string (*Hint* : use `sample` as in Section 1.6.2, p. 23 with replacement, and use `paste` as shown in Section 3.5, p. 66). Write it in one line of R code.

Chapter 4

Visualizing data

4.1 The R graphics system

The graphics system in **R** is very flexible: just about every aspect of your plot can be precisely controlled. However, this brings with it a serious learning curve - especially when you want to produce high quality polished figures for publication. In this chapter, you will learn to make simple plots, and also to control different aspects of formatting plots. The following focus on the basic built-in graphics system in **R**, known as the `base` graphics system.

4.2 Plotting in RStudio

By default, when you generate plots in RStudio, they are displayed in the built-in plotting window (normally, the bottom-right). This window has a few useful options.

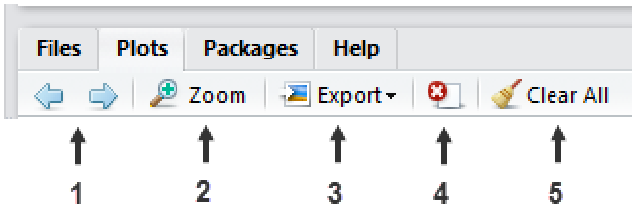


Figure 4.1: The plotting window in RStudio. 1 : cycle through plotting history, 2 : displays the plot in a large window, 3 : Export the plot to an image or PDF, 4 : delete the current plot from the plotting history, 5 : clear all plotting history

4.3 Choosing a plot type

The following table summarizes a few important plot types. More specialized plot types, as well as very brief introductions to alternative plotting packages, are given in Section 4.6.

Function	Graph type
`plot`	Scatter plots and various others (Section 4.3.1)
`barplot`	Bar plot (including stacked and grouped bar plots) (Section 4.3.2)
`hist`	Histograms and (relative) frequency diagrams (Section 4.3.3)
`curve`	Curves of mathematical expressions (Section 4.3.3)
`pie`	Pie charts (for less scientific uses) (Section 4.3.4)
`boxplot`	Box-and-whisker plots (Section 4.3.5)
`symbols`	Like scatter plot, but symbols are sized by another variable (Section 4.6.1)

4.3.1 Using the `plot` function

There are two alternative ways to make a plot of two variables `X` and `Y` that are contained in a dataframe called `dfr` (both are used interchangeably):

```
# Option 1: plot of X and Y
with(dfr, plot(X,Y))

# Option 2: formula interface (Y 'as a function of' X)
plot(Y ~ X, data=dfr)
```

The type of plot produced by this basic command, using the generic `plot` function, depends on the variable type of `X` and `Y`. If both are numeric, a simple scatter plot is produced (see Section 4.4). If `Y` is numeric but `X` is a factor, a boxplot is produced (see Section 4.3.5). If only one argument is provided and it is a data frame with only numeric columns, all possible scatter plots are produced (not further described but see `?plot.data.frame`). And finally, if both `X` and `Y` are factor variables, a mosaic plot is produced (not further described in detail but see Section 7.5.1 and `plot.factor`).

Each of these four options is shown in Fig. 4.2 (code not included).

4.3.2 Bar plots

While simple barplots are easy in **R**, advanced ones (multiple panels, groups of bars, error bars) are more difficult. For those of you who are skilled at making complex barplots in some other software package, you may not want to switch to **R** immediately. Ultimately, though, **R** has greater flexibility, and it is easier to make sure plots are consistent.

The following code makes a simple bar plot, using the default settings (Fig. 4.3):

```
nums <- c(2.1,3.4,3.8,3.9,2.9,5)
barplot(nums)
```

Very often, we like standard errors on our bar plots. Unfortunately, this is slightly awkward in basic **R**.

The easiest option is to use `barplot2` in the `gplots` package, but it provides limited customization of the error bars. Another, more flexible, solution is given in the example below. Here, we use `barplot` for the bars, and `plotCI` for the error bars (from the `plotrix` package). This example also introduces the `tapply` function to make simple tables, we will return to this in Section 6.2.1.

If you want the means and error bars to be computed by the function itself, rather than provide them as in the example below, refer to Section 4.6.2.

The following code produces Fig. 4.4.

```
# Read data, if you haven't already
cereal <- read.csv("Cereals.csv")

```

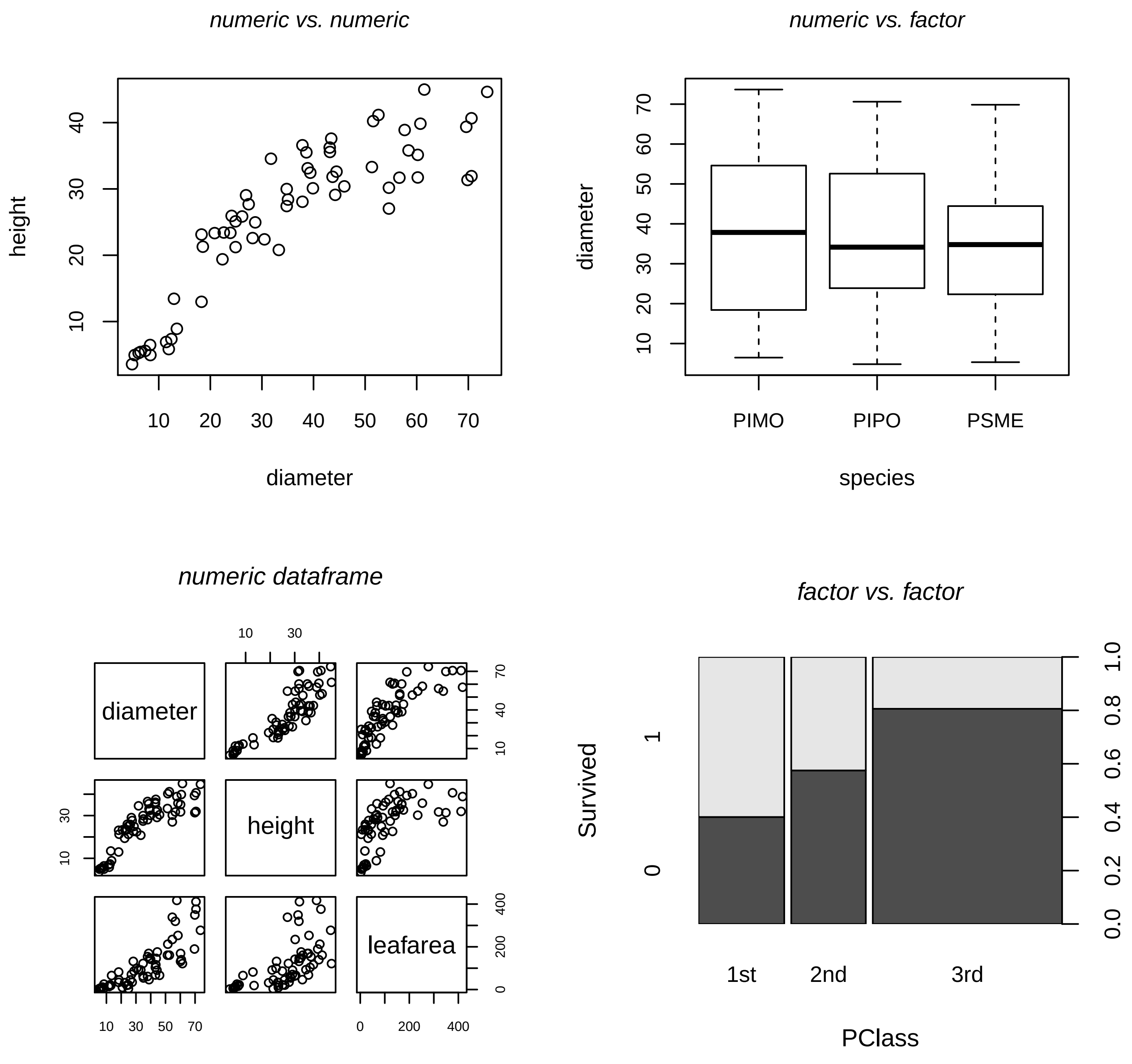


Figure 4.2: Four possible outcomes of a basic call to `plot`, depending on whether the Y and X variables are numeric or factor variables. The term 'numeric dataframe' means a dataframe where all columns are numeric.

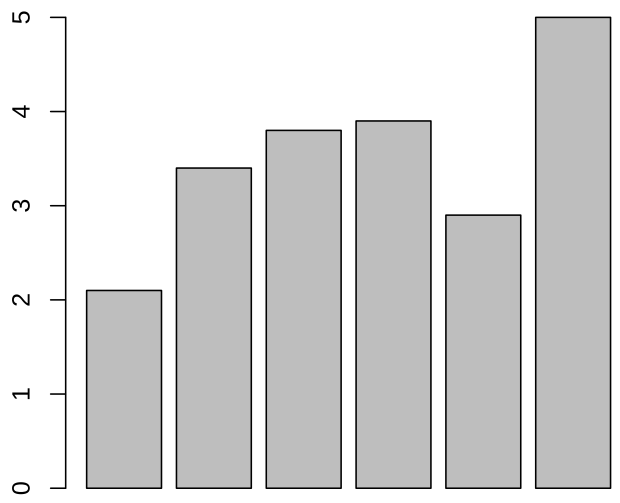


Figure 4.3: Simple bar plot with default settings

```r
# Gets means and standard deviation of rating by cereal manufacturer:
ratingManufacturer <- with(cereal, tapply(rating, Manufacturer, FUN=mean))
ratingManufacturerSD <- with(cereal, tapply(rating, Manufacturer, FUN=sd))

# Make bar plot with error bars. Note that we show means +/- one standard deviation.
# Read the help function ?plotCI to see what the additional arguments mean (pch, add).
library(plotrix)
b <- barplot(ratingManufacturer, col="white", width=0.5, space=0.5, ylim=c(0,80))
plotCI(b, ratingManufacturer, uiw=ratingManufacturerSD, add=TRUE, pch=NA)
```

Finally, to make 'stacked' bar plots, see the example in Section 4.3.4.

4.3.3 Histograms and curves

The `hist` function by default plots a frequency diagram, and computes the breaks automatically. It is also possible to plot a density function, so that the total area under the bars sums to unity, as in a probability distribution. The latter is useful if you want to add a curve representing a univariate distribution.

Consider the following example, which plots both a frequency diagram and a density plot for a sample of random numbers from a normal distribution. (See chapter 5 for background on the functions `dnorm` and `rnorm`.)

This example uses the `par` function to change the layout of the plot. We'll return to this in Section 4.4.8.

This code produces Fig. 4.5.

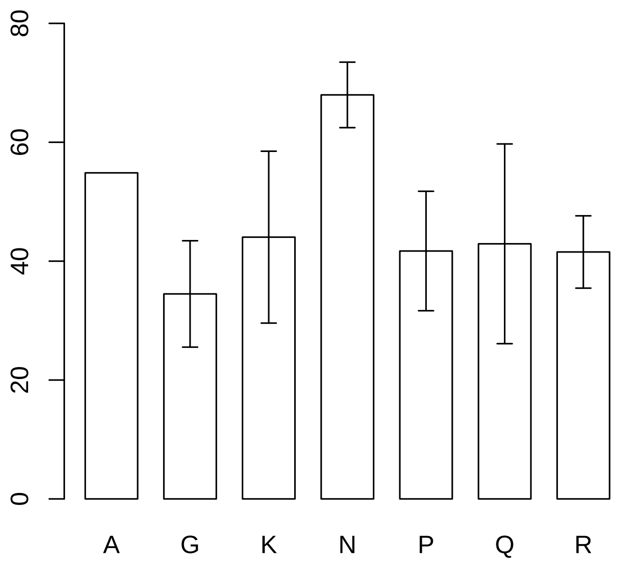


Figure 4.4: Simple bar plot with error bars (shown are means +/- one standard deviation).

```r
# Some random numbers:
rannorm <- rnorm(500)

# Sets up two plots side-by-side.
# The mfrow argument makes one row of plots, and two columns, see ?par.
par(mfrow=c(1,2))

# A frequency diagram
hist(rannorm, freq=TRUE, main="")

# A density plot with a normal curve
hist(rannorm, freq=FALSE, main="", ylim=c(0,0.4))
curve(dnorm(x), add=TRUE, col="blue")
```

Try this yourself We also introduced the `curve` function here. It can plot a curve that represents a function of `x`. For example, try this code yourself :
`curve(sin(x), from=0, to=2*pi)`

4.3.4 Pie charts

The final two basic plotting types are pie charts and box plots. It is recommended you don't use pie charts for scientific publications, because the human eye is bad at judging relative area, but much better at judging relative lengths. So, barplots are preferred in just about any situation. Take the following example, showing the polls for 12 political parties in the Netherlands (see Section A.17).

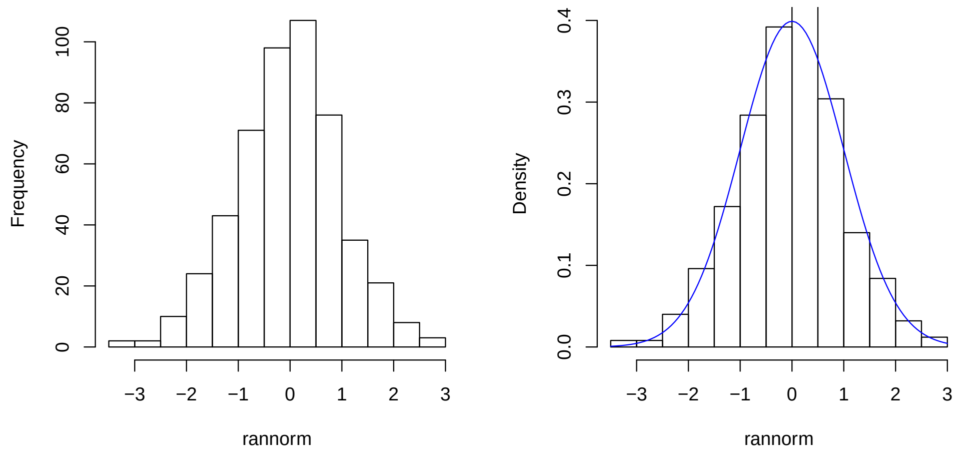


Figure 4.5: Two histogram examples for some normally distributed data.

Most people will say that the fraction of voters for the smaller parties is exaggerated in the pie chart.

This code produces Fig. 4.6.

```r
# Election poll data
election <- read.csv("dutchelection.csv")

# A subset for one date only (Note: unlist makes this into a vector)
percentages <- unlist(election[6, 2:ncol(election)])

# Set up two plots side-by-side
par(mfrow=c(1,2))

# A 'stacked' bar plot.
# the matrix() bit is necessary here (see ?barplot)
# Beside=FALSE makes this a stacked barplot.
barplot(matrix(percentages), beside=FALSE, col=rainbow(12), ylim=c(0,100))

# And a pie chart
pie(percentages, col=rainbow(12))
```

Try this yourself Run the code above to generate a stacked barplot. Now, set `beside=TRUE` and compare the result.

Now what if we want to compare different timepoints to see whether the support for each political party is changing? To produce multiple stacked barplots using the `barplot` function, we need to convert the way that the data are stored from a dataframe to a matrix. Then, because the `barplot` function creates each stack in the barplot from the columns of the matrix, we need to transpose the matrix so that the data for different dates are in separate columns and the data for different parties are in separate rows.

```r
# A subset for the first and last dates
percentages2 <- election[c(1, nrow(election)), -1]
```

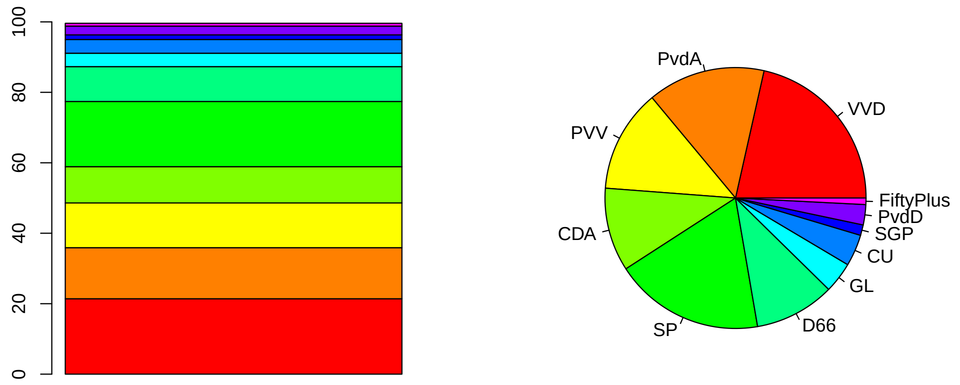


Figure 4.6: A stacked bar and pie chart for the election poll data.

```
# Convert the resulting dataframe to a matrix
percentages2 <- as.matrix(percentages2)

# change the rownames to represent the dates at those two timepoints
rownames(percentages2) <- election[c(1, nrow(election)), 'Date']

# A 'stacked' bar plot for two timepoints.
# the t() bit transposes the party data from columns to rows
# beside=FALSE makes this a stacked barplot.
# the xlim argument creates extra space on the right of the plot for the legend
barplot(t(percentages2), beside=FALSE, col=rainbow(12), ylim=c(0,100),
        xlim=c(0, 4), legend=colnames(percentages2))
```

4.3.5 Box plots

Box plots are convenient for a quick inspection of your data. Consider this example, where we use formula notation to quickly plot means and ranges for the sodium content of cereals by manufacturer.

This code produces Fig. 4.8.

```
cereal <- read.csv("Cereals.csv")

boxplot(sodium ~ Manufacturer, data=cereal, ylab="Sodium content", xlab="Manufacturer")
```

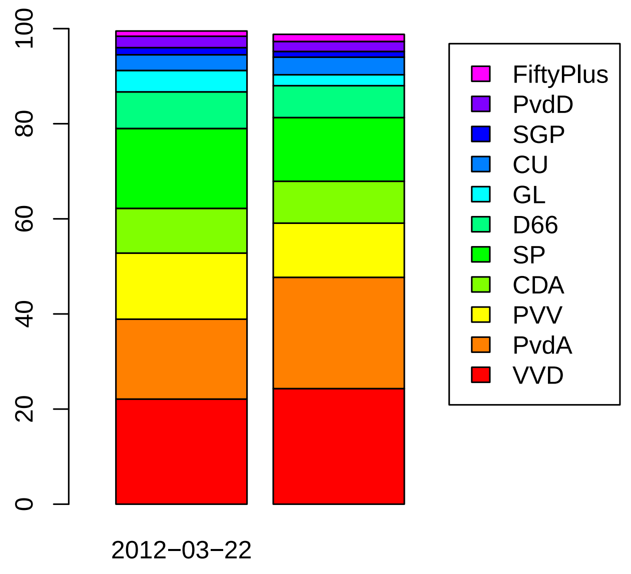


Figure 4.7: A stacked bar chart for the election poll data at two timepoints.

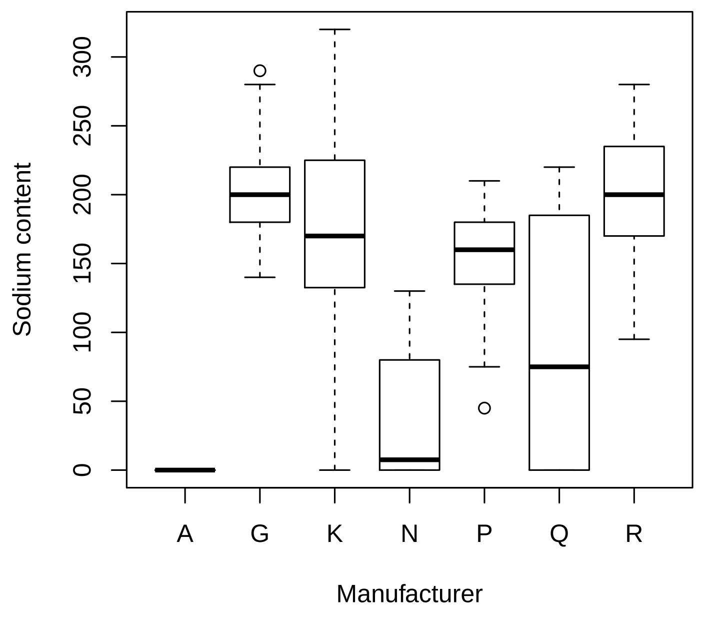


Figure 4.8: A simple box plot for the cereal data.

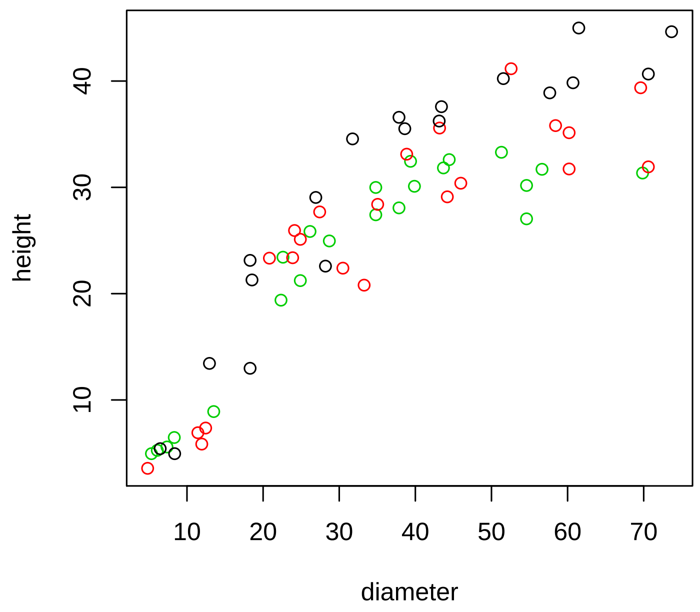


Figure 4.9: Simple scatter plot with default settings

4.4 Fine-tuning the formatting of plots

4.4.1 A quick example

We'll start with an example using the `allom` data. First, we use the default formatting, and then we change a few aspects of the plot, one at a time. We will explain the settings introduced here in more detail as we go along. This code produces Fig. 4.9, a simple scatter plot:

```
# Read data
allom <- read.csv("allometry.csv")

# Default scatter plot
with(allom, plot(diameter, height, col=species))
```

Now let's make sure the axis ranges start at zero, use a more friendly set of colours, and a different plotting symbol.

This code produces Fig. 4.10. Here, we introduce `palette`, a handy way of storing colours to be used in a plot. We return to this in the next section.

```
palette(c("blue","red","forestgreen"))
with(allom, plot(diameter, height, col=species,
                 pch=15, xlim=c(0,80), ylim=c(0,50)))
```

Notice that we use `species` (a factor variable) to code the colour of the plotting symbols. More about this later in this section.

For this figure it is useful to have the zero start exactly in the corner (compare the origin to previous

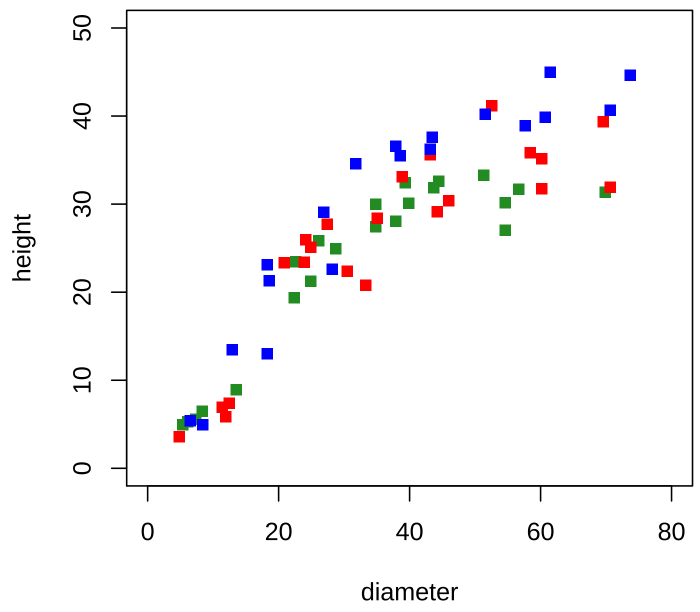


Figure 4.10: Simple scatter plot : changed plotting symbol and X and Y axis ranges.

figure.) To do this we use `xaxs` and `yaxs`. Let's also make the X-axis and Y-axis labels larger (with `cex.lab`) and print nicer labels (with `xlab` and `ylab`).

Finally, we also add a legend (with the `legend` function.)

This code produces Fig. 4.11.

```
par(xaxs="i", yaxs="i",  cex.lab=1.4)
palette(c("blue","red","forestgreen"))
plot(height ~ diameter, col=species, data=allom,
                 pch=15, xlim=c(0,80), ylim=c(0,50),
                 xlab="Diameter (cm)",
                 ylab="Height (m)")
# Add a legend
legend("topleft", levels(allom$species), pch=15, col=palette(), title="Species")
```

4.4.2 Customizing and choosing colours

You can change the colour of any part of your figure, including the symbols, axes, labels, and background. **R** has many built-in colours, as well as a number of ways to generate your own set of colours.

As we saw in the example above, it is quite handy to store a set of nice colours in the `palette` function. Once you have stored these colours they are automatically used when you colour a plot by a factor. As the default set of colours in **R** is very ugly, do choose your own set of colours before plotting. You can also choose specific colors from your palette. The following example would plot symbols using the 3rd colour in your palette (resulting graph not shown).

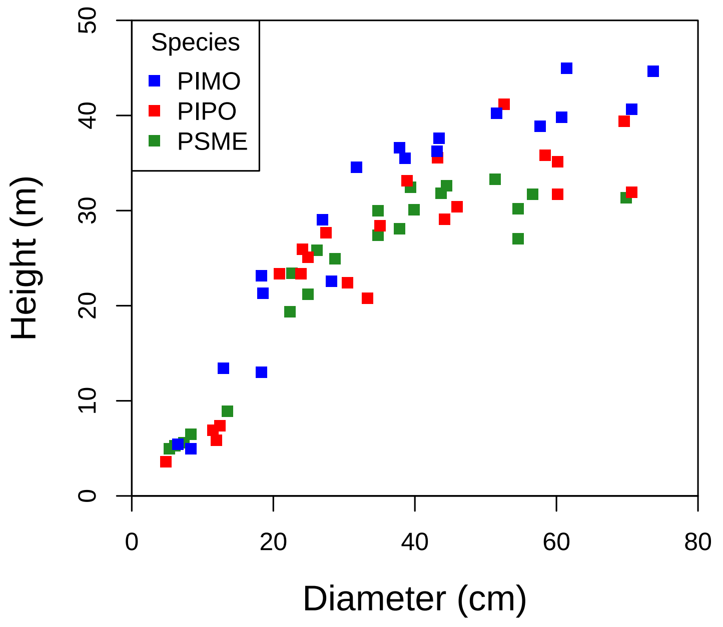


Figure 4.11: Simple scatter plot : many settings customized.

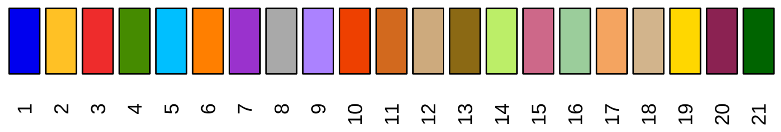
```r
plot(x,y, col=3)
```

To pick one or more of the 657 built-in colours in **R**, this website is very useful : `http://research.stowers-institute.org/efg/R/colour/Chart/`, *especially* the link to a PDF at the top, which lists the colours by name. You can also use the built-in `demo` function to show the colours,

```r
# Follow instructions in the console when running this command.
demo(colors)
```

The following is an example `palette`, with some hand-picked colours. It also shows one way to plot your current palette.

```r
palette(c("blue2","goldenrod1","firebrick2","chartreuse4",
"deepskyblue1","darkorange1","darkorchid3","darkgrey",
"mediumpurple1","orangered2","chocolate","burlywood3",
"goldenrod4","darkolivegreen2","palevioletred3",
"darkseagreen3","sandybrown","tan",
"gold","violetred4","darkgreen"))

# A simple graph showing the colours.
par(cex.axis=0.8, las=3)
n <- length(palette())
barplot(rep(1,n),
        col=1:n,
        names.arg=1:n,axes=FALSE)
```



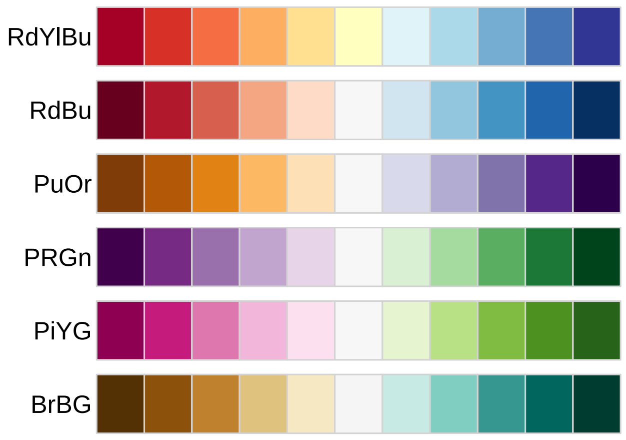
Ranges of colours

R also has a few built-in options to set a range of colours, for example from light-grey to dark-grey, or colours of the rainbow.

A very useful function is `colorRampPalette`, see its help page and the following examples:

```
# Generate a palette function, with colours in between red and blue:
redbluefun <- colorRampPalette(c("red","blue"))

# The result is a function that returns any number of colours interpolated
# between red and blue.
# For example, set the palette to 10 colouts ranging from red to blue:
palette(redbluefun(10))
```

Other functions that are useful are `rainbow`, `heat.colors`, `grey`, and so on (see help page for `rainbow` for a few more). The `RColorBrewer` package also contains the `brewer.pal` function, which helps to generate palettes that are suitable for different types of graphics requiring scales of light and dark colours or highly contrasting colours. It even has functionality for selecting colour schemes that are 'colour-blind friendly', for example:



Try this yourself Look at `?grey` and figure out how to generate a palette of 50 shades of grey.

Here is an example where it might be handy to use a range of colours. Take a look at the HFE weather data (Section A.10). What is the relationship between air temperature (`Tair`), vapour pressure deficit (`VPD`) and relative humidity? Here is a nice way to visualize it.

This code produces Fig. 4.12. Here we use also the `cut` function, which was introduced in Section 3.2.

```
# Read data.
hfemet <- read.csv("hfemet2008.csv")

# Make a factor variable with 10 levels of relative humidity.
hfemet$RHbin <- cut(hfemet$RH, breaks=10)

# Look at the levels: they are from low RH to high RH
levels(hfemet$RHbin)

##  [1] "(14.8,23.1]" "(23.1,31.4]" "(31.4,39.7]" "(39.7,48]"   "(48,56.2]"
##  [6] "(56.2,64.5]" "(64.5,72.8]" "(72.8,81]"   "(81,89.3]"   "(89.3,97.7]"

# Set colours correspondingly, from red to blue.
blueredfun <- colorRampPalette(c("red","blue"))
palette(blueredfun(10))

# Plot VPD and Tair, with the colour of the symbol varying by RH.
# Also set small plotting symbols.
par(cex.lab=1.3)
with(hfemet, plot(Tair, VPD, pch=19, cex=0.5, col=RHbin))

# Finally, add a legend:
legend("topleft", levels(hfemet$RHbin), fill=palette(), title="RH")
```

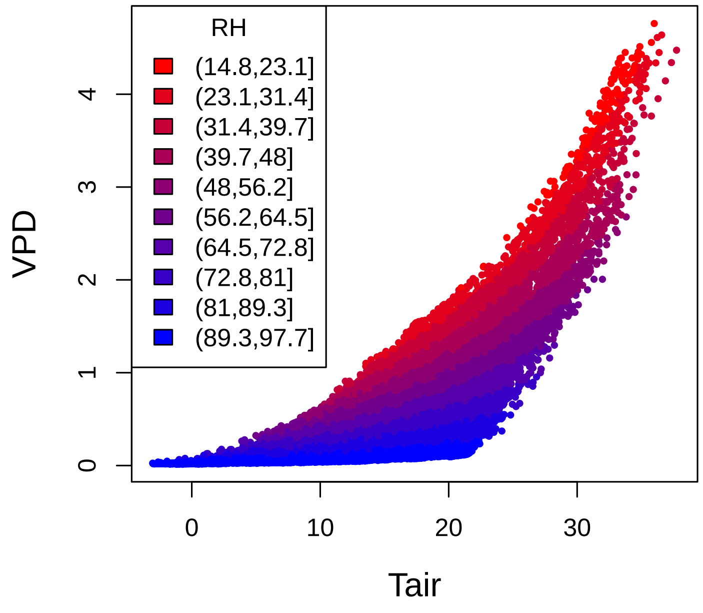


Figure 4.12: Relationship between air temperature, vapour pressure deficit and relative humidity at the HFE.

Semi-transparent colours

If your plot contains a lot of data points, a semi-transparent colour can be useful, so you can see points that would otherwise be obscured.

This example makes Fig. 4.13, using the `scales` package.

```r
# Load the scales package for the 'alpha' function.
library(scales)

# Make a large dataset with random numbers
x <- rnorm(1000)
y1 <- x + rnorm(1000, sd=0.5)
y2 <- -x + rnorm(1000, sd=0.6)

# Use alpha() like this to make a colour transparent,
# the numeric value indicates how transparent the result should be
# (lower values = more transparent)
plot(x,y1,pch=19,col=alpha("blue",0.3))
points(x,y2, pch=19, col=alpha("red",0.3))
```

4.4.3 Customizing symbols and lines

Using the `plot` function, you can make both scatter plots and line plots, and combinations of both. Line types (solid, dashed, thickness, etc.) and symbols (circles, squares, etc.) can be customized.

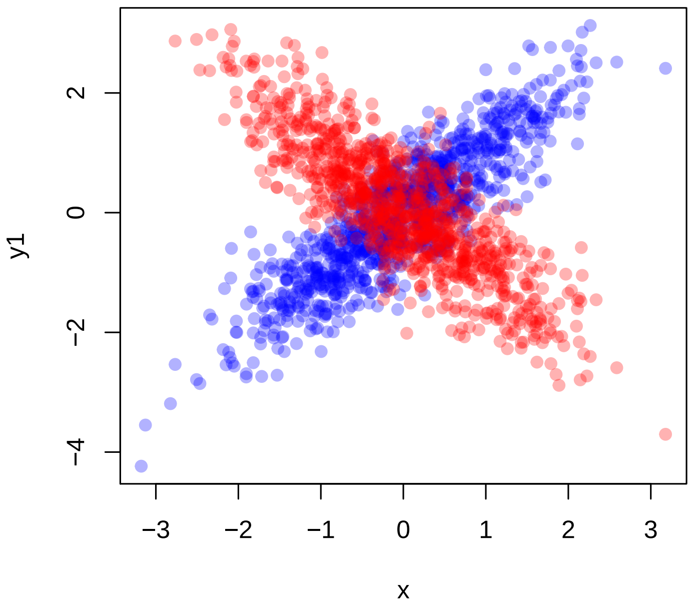


Figure 4.13: A plot of some random numbers, using a semi-transparent colour.

Consider these options when plotting a vector of observations (makes Fig. 4.14).

```
X <- 1:8
Y <- c(4,5.5,6.1,5.2,3.1,4,2.1,0)
par(mfrow=c(3,2))
plot(X,Y, type='p', main="type='p'")
plot(X,Y, type='o', main="type='o'")
plot(X,Y, type='b', main="type='b'")
plot(X,Y, type='l', main="type='l'")
plot(X,Y, type='h', main="type='h'")
plot(X,Y, type='s', main="type='s'")
```

For symbols, use the `pch` argument in `plot`. These are most quickly set with a number, see the table on the help page `?points`.

A few examples with the `pch` argument are shown here. This code produces Fig. 4.15.

```
par(mfrow=c(2,2))

# Open triangles:
with(allom, plot(diameter, height, pch=2, col="red"))

# Red solid squares:
with(allom, plot(diameter, height, pch=15, col="red"))

# Filled circles, with a black edge, and a grey fill colour:
with(allom, plot(diameter, height, pch=21, bg="grey", col="black"))

# Custom plotting symbol (any text works - but only one character)
```

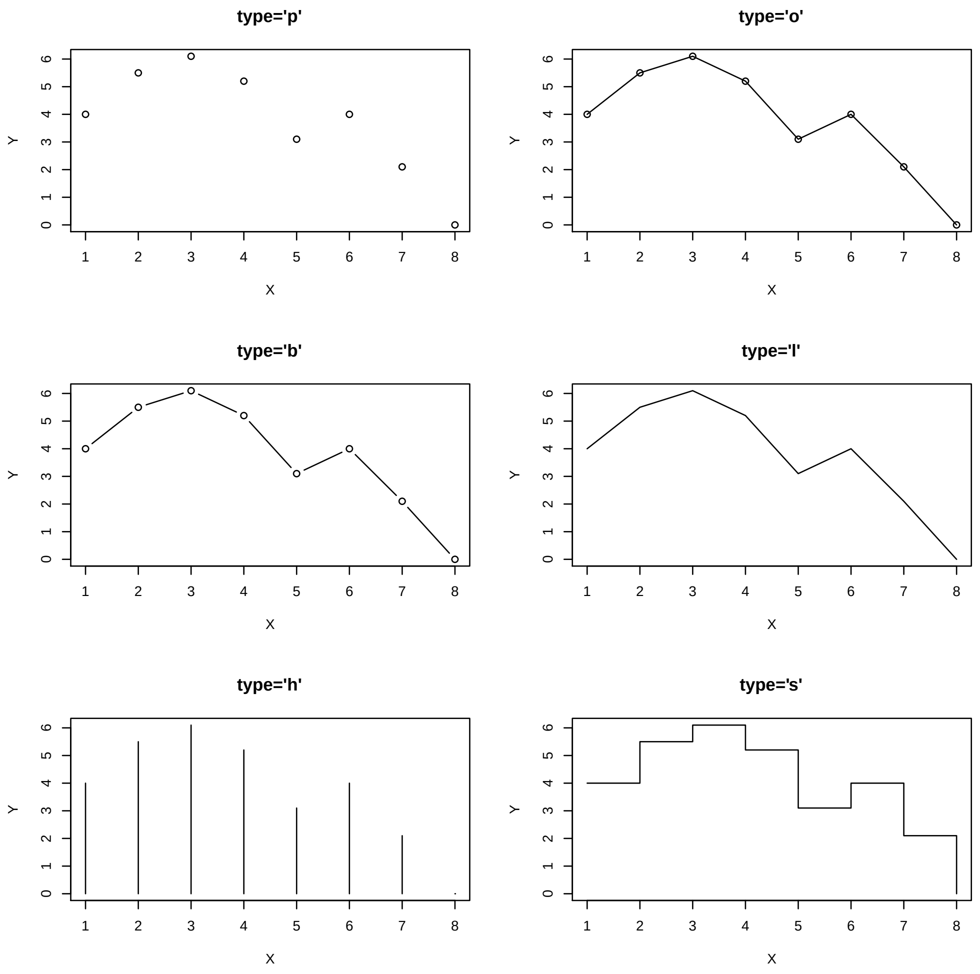


Figure 4.14: Four options for the plotting type (with type=).

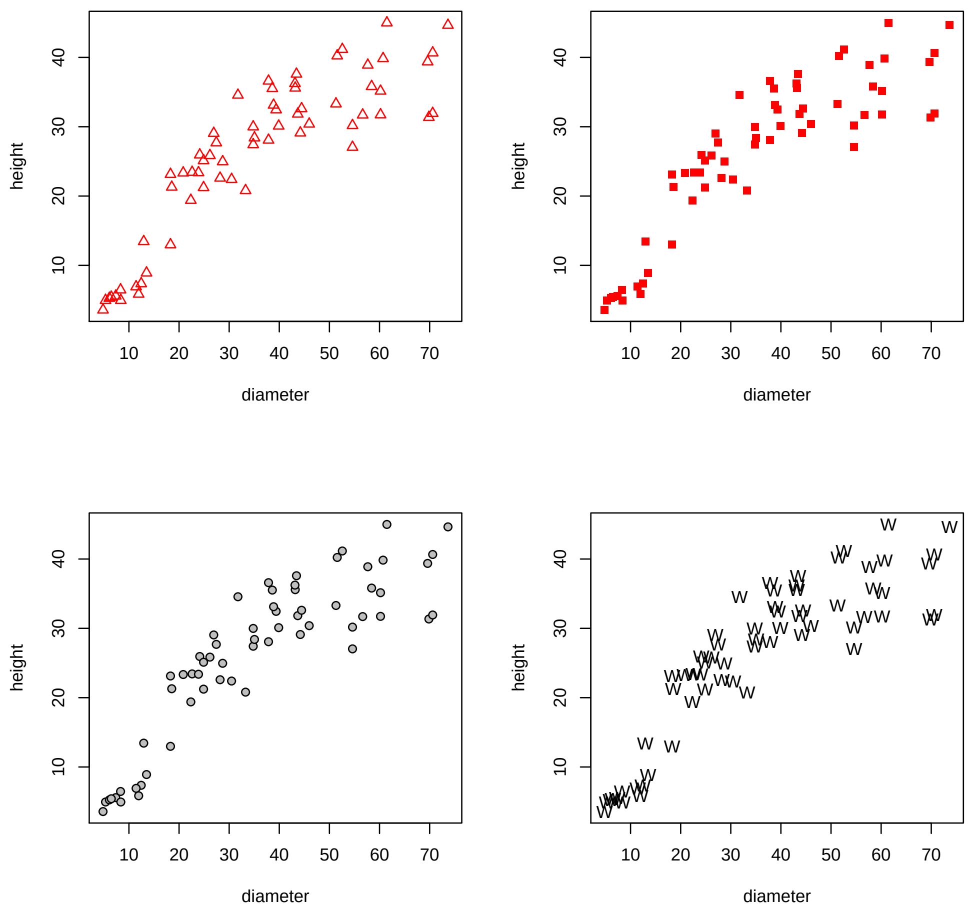


Figure 4.15: Four options for the plotting symbols (with pch).

```r
with(allom, plot(diameter, height, pch="W"))
```

Setting symbol type by a factor level

Finally, it gets more interesting if we vary the plotting symbol by a factor, like we did with colours in the previous section. Look at this simple example that extends Fig. 4.10.

This code produces Figure 4.16. Note how we set up a vector of plotting symbols (1,2 and 15), and use the factor `species` to index it. To recall indexing, read Section 2.3.1.

```r
palette(c("blue","red","forestgreen"))
with(allom, plot(diameter, height,
                 col=species,
                 pch=c(1,2,15)[species],
                 xlim=c(0,80), ylim=c(0,50)))
```

Try this yourself Modify the code above so that the three species are plotted in three different shades of `grey`, with filled circles. Also add a `legend`, and increase the size of the axis labels.

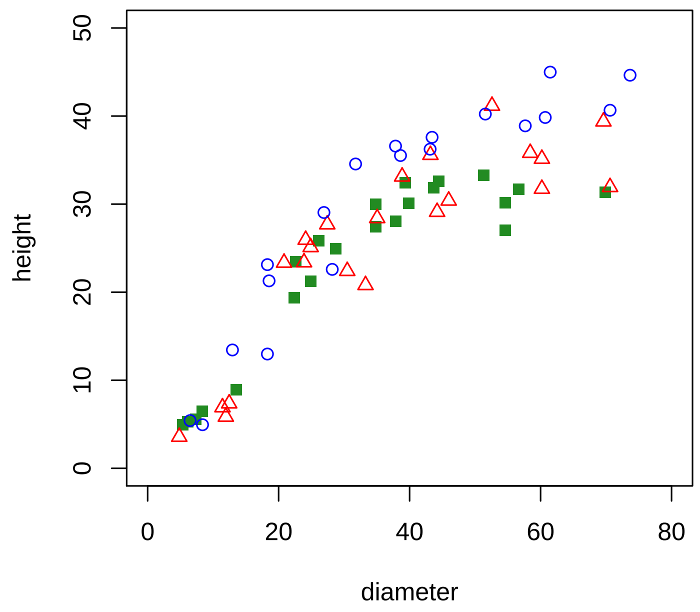


Figure 4.16: Scatter plot : vary colour and symbol by species

4.4.4 Formatting units, equations and special symbols

In scientific publications we frequently need sub- and superscripts in our axis labels (as in m^{-2}). Luckily, **R** has a flexible way of specifying all sorts of expressions in axis labels, titles, and legend texts. We have included a few examples here, and a full reference is given in the help page `?plotmath`. It is especially helpful to run `demo(plotmath)`. Recall that in RStudio, you can look at all your previous plots using the arrows near the top-left of the figures.

```
expression(Infected~area~(cm^2))
```

Infected area (cm^2)

```
expression(Photosynthesis~ ~(mu*mol~m^-2~s^-1))
```

Photosynthesis $(\mu\mathrm{mol}\ \mathrm{m}^{-2}\ \mathrm{s}^{-1})$

```
expression(Temperature~ ~(degree*C))
```

Temperature $(^\circ\mathrm{C})$

Try this yourself The above examples of expressions can be directly used in plots. Try making a scatter plot, and using two of the above expressions for the X and Y axis labels, by using this template:
`plot(x,y, xlab=expression(...))`
You may find that the margins of the plot need to be larger. To increase them use this command (before plotting):
`par(mar=c(5,5,2,2))`

Try this yourself As shown on the help page `?plotmath`, you can make subscripts in labels (using `expression`) with square brackets (`[]`). Try making a label with a subscript.

Special text symbols

If you need a very special character – one that is not a Greek letter or a subscript, or is otherwise covered by `plotmath` – anything can be plotted if you know the Unicode number (`http://en.wikipedia.org/wiki/Unicode_symbols`). You can plot those with this short example (results not shown):

```
expression(Permille~"\u2030")
```

The four digit (or letter) Unicode follows 'u'.

4.4.5 Resetting the graphical parameters

Every time you use `par` to set some graphical parameter, it keeps this setting for all subsequent plots. In RStudio, the following command can be used to reset `par` to the default values. *Warning:* this will delete all the plots you have saved in the plot history.

```
dev.off()
```

4.4.6 Changing the font

There are (only) three basic built-in fonts in **R**, but you can load many more with the help of `windowsFonts` (on Windows only). The fonts can be accessed with the `family` argument. You can set this argument in `plot` (for axis titles), `text` (for adding text to plots), as well as `par` (to change the font permanently for all text).

Note that in `?par`, the `font` argument refers to "italic", "bold", and so on (see below).

```
# A font with a serif (looks a bit like Times New Roman)
plot(x,y, xlab="some label", family="serif")
```

Using `windowsFonts`, you can use *any* font that is loaded in Windows (that is, all the fonts you can see in MS Word). *Note*: this does not work when making a PDF. For more flexibility, consider using the `showtext` package.

```
# Define font(s)
windowsFonts(courier=windowsFont("Courier New"),
             verdana=windowsFont("Verdana"))

# Then you can use,
plot(x,y, xlab="some label", family="courier")
```

Try this yourself Run one of the examples in Section 4.4, and change the font of the axis labels to one of your favourite MS Word fonts.

Further reading If you are interested in using a wider range of fonts, the `showtext` package can help you out. There is a description of what `showtext` does from its creator, Yixuan Qiu, at `https://www.r-project.org/nosvn/pandoc/showtext.html`. Yixuan has also posted a helpful guide to using `showtext` in markdown documents on his blog: `http://statr.me/2014/07/showtext-with-knitr/`.

Italic and bold

The `font` argument can be used to set text to normal face (1), bold (2), italic (3) or bold italic (4). Simply use the following code, changing the number as appropriate:

```
# A plot with a title in italics
plot(x,y, main="Italic text", font=3)
```

4.4.7 Adding to a current plot

Suppose you have made a plot, and want to add points or lines to the current plot, without opening a new window. For this, we can use the `points` function (*Note:* this can also be used to add lines, using the `type='l'` setting).

Consider the following example. Instead of plotting all the data at once, we plot the data for one group, and then add the data for each following group using `points`.

The following code produces Fig. 4.17.

```
# Read the Dutch election poll data
election <- read.csv("dutchelection.csv")
election$Date <- as.Date(election$Date)

# Plot the first variable (make sure to set the Y axis range
# wide enough for all the other data!)
plot(VVD ~ Date, data=election, type='l', col="blue", ylim=c(0,40),
     ylab="Poll result (%)")

# Then add the rest of the results, one at a time.
points(PvdA ~ Date, data=election, type='l', col="red")
points(SP ~ Date, data=election, type='l', col="red", lty=5)
points(GL ~ Date, data=election, type='l', col="forestgreen")
```

Straight lines and text

To place straight lines on a plot, use the following examples (results not shown). For the `abline` function, you may use settings like `lwd` for the thickness of the line, `lty` for line type (dashed, etc.), and `col` for colour.

Using `abline`, it is also straightforward to add regression lines to a plot, as we will see in Section 5.5.

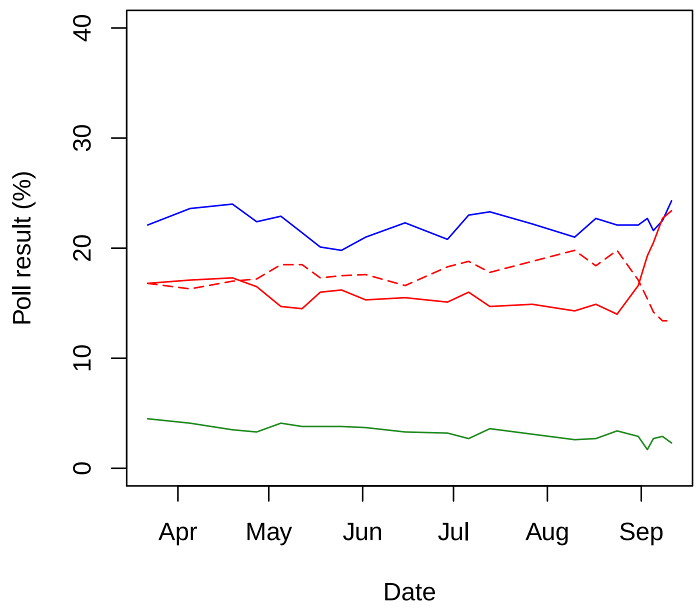


Figure 4.17: Adding lines to an existing plot.

```r
# Add a vertical line at x=0
abline(v=0)

# Add a horizontal line at y=50
abline(h=50)

# Add a line with an intercept of 0 and a slope of 1
# (known as a 1:1 line)
abline(0,1)
```

Adding text to a plot is achieved with the `text` function. Text can also be added to the margin of a plot with `mtext`. You may also use `expression` as the text to be added (see Section 4.4.4). The drawback of using `text` is that you need to specify the X,Y coordinates manually. What if you just want a quick label on the top left of your figure?

Try this:

```r
# Add a bold label 'A' to an existing plot:
legend("topleft", expression(bold(A)), bty='n', inset=0.01)
```

Try this yourself Test the above code with any of the examples in this chapter.

4.4.8 Changing the layout

We have already seen several examples that combine multiple figures in one plot, using `par(mfrow = c(`*`rows`*`, `*`columns`*`))`. Here is another example, which generates the plot in Fig. 4.18. See `?par` for more

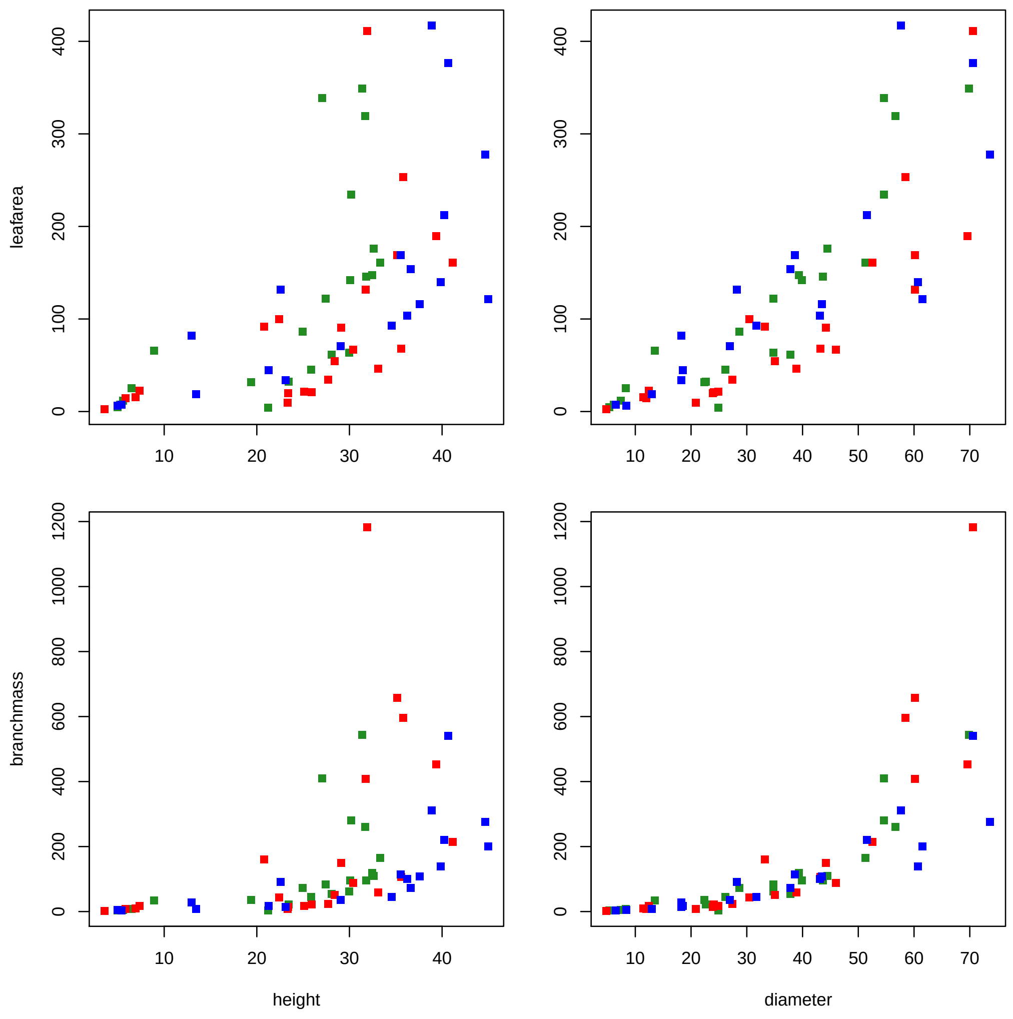


Figure 4.18: Multiple plots within a single plot window.

details.

```
# Set up 2 rows of plots, and 2 columns using 'mfrow':
par(mfrow=c(2,2),mar=c(4.1,4.1,0.1,0.1))
plot(leafarea~height,data=allom,col=species,xlab='', pch=15)
plot(leafarea~diameter,data=allom,col=species,xlab='',ylab='',pch=15)
plot(branchmass~height,data=allom,col=species,pch=15)
plot(branchmass~diameter,data=allom,col=species,ylab='',pch=15)
```

This is a relatively simple way to change the layout of plots. We can generate more complex layouts using the `layout` function. The main argument is a matrix indicating the locations of individual plots in space. Using the code below (resulting plot not shown), the first plot will fill the the left side of the plot window, while the next three calls to plot will fill each section on the right side of the window in the order that they are labelled. Further arguments allow for varying the heights and widths of boxes within the layout.

```
l<-layout(matrix(c(1,1,1,2,3,4),nrow=3,ncol=2,byrow=F))
layout.show(l)
```

Try this yourself Run the code from the above example. Then, run only the first bit to set up a fresh layout. Then make a series of plots, and see how they fill the layout (*Hint:* you can just run `plot(1)` several times in a row to see what happens).

4.4.9 Finding out about more options

To get the most out of plotting in **R**, you need a working knowledge of `par` options, which are used to set graphical parameters. We've used some of these already. Here is a summary of a few of the most useful settings, including a few new ones:

Table 4.1: Setting graphical parameters using `par`

Graphical parameter	Description
`pch`	Sets the type of symbols used in the plot; see `points()` for a list of options.
`type`	Sets whether to plot points, lines, both, or something else (see `?plot`.)
`col`	Sets the colour of plotting symbols and lines.
`lty`	Sets the line type (1=solid, 2=dashed, etc.)
`lwd`	Sets the line width
`cex`	Controls the size of text and points in the plot area. Short for 'character expansion', it acts as a multiplier of the default value.
`cex.axis, cex.lab`	Character expansion of axes and the labels.
`cex.main`	Character expansion of the title of the plot.
`family`	Sets the font for labels and titles. Varies by system, but 'serif','sans' and 'mono' should always work.
`bty`	Sets the type of box, if any, to be drawn around the plot. Use `bty='n'` for none.
`las`	Sets the orientation of the text labels relative to the axis
`mar`	Sets the number of lines in each margin, in the order bottom, left, top, right.
`xaxs, yaxs`	Preset functions for calculating axis intervals.
`xaxp, yaxp`	Sets the coordinates for tick marks on each axis.
`xaxt, yaxt`	Sets axis type, but can also suppress plotting axes by specifying 'n'.

You can choose to set an option with the `par` function, which will apply that setting to any new plots (until you change it again). Alternatively, you can use any of these settings when calling `plot` or `points` to change only the current plot. See this example (output not shown try this yourself).

```r
# Two ways of setting the size of the X and Y axis labels:
# 1.
plot(1:10, 1:10, cex.lab=1.2)

# 2.
par(cex.lab=2)
plot(1:10,1:10)

# For the latter, the setting is maintained for the next plot as well.
plot(1:3, 1:3)
```

Further reading Keeping on top of all the options associated with `par` can be difficult. The `?par` help page has been dramatically improved in recent years, but can still be a bit too much information. It's very helpful to have some quick references at your fingertips. Take a look http://www.statmethods.net/advgraphs/parameters.html from the Quick-R website, and Gaston Sanchez's handy cheat sheet at http://gastonsanchez.com/resources/2015/09/22/R-cheat-sheet-graphical-parameters/.

4.5 Formatting examples

4.5.1 Vessel data

This example will use a lot of what we learned in this section to fine-tune the formatting of a plot. We will use the `vessel` data (see Section A.13). In this dataset, the expectation was that xylem vessel diameters are smaller in the top of the tree than at the bottom. Rather than going straight to statistical analyses (ANOVAs and so on) it is wise to visualize the data first.

The normal workflow for optimizing your plots is to first use the default settings to make sure you are plotting the correct data, and then fine-tuning one aspect of the plot at a time. This can take a while, but the end result should be worth it. Here we show a default histogram, and the fine-tuned version after some work.

The following code produces Fig. 4.19.

```
# Read vessel data, and make two datasets (one for 'base' data, one for 'apex' data).
vessel <- read.csv("vessel.csv")
vesselBase <- subset(vessel, position=="base")
vesselApex <- subset(vessel, position=="apex")

# Set up two figures next to each other:
par(mfrow=c(1,2))

# Simple histograms, default settings.
hist(vesselBase$vesseldiam)
hist(vesselApex$vesseldiam)
```

Next, we make the two histograms again, but customize the settings to produce a high quality plot. Try to figure out yourself what the options mean by inspecting the help files `?hist`, and `?par`.

This code produces Fig. 4.20. (To run this code, make sure to read the vessel data first, as shown in the previous example).

```
# Fine tune formatting with par()
par(mfrow=c(1,2), mar=c(5,5,4,1), cex.lab=1.3, xaxs="i", yaxs="i")

# First panel
hist(vesselBase$vesseldiam,
     main="Base",
     col="darkgrey",
     xlim=c(0,160), breaks=seq(0,160,by=10),
     xlab=expression(Vessel~diameter~ ~(mu*m)),
     ylab="Number of vessels")

# Second panel
```

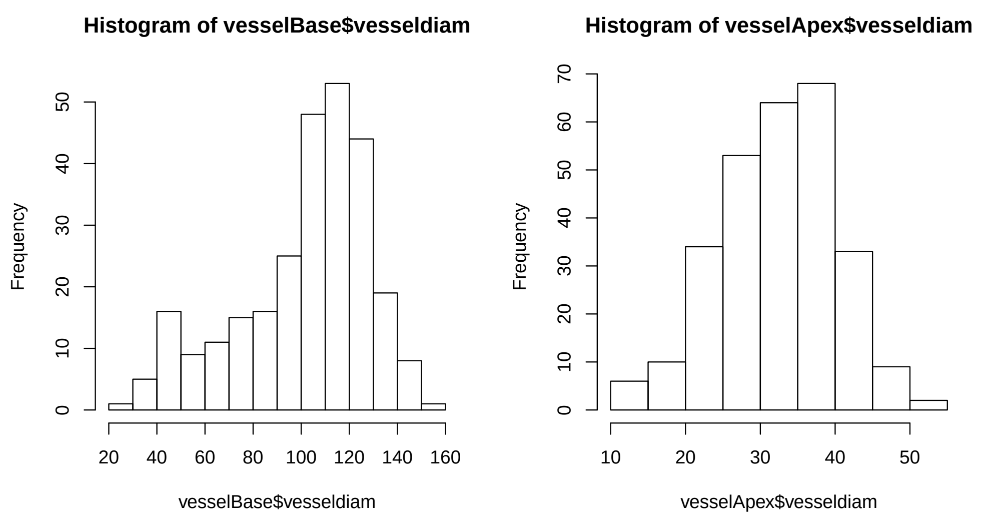


Figure 4.19: Two simple default histograms

```
hist(vesselApex$vesseldiam,
     main="Apex",
     col="lightgrey",
     xlim=c(0,160), breaks=seq(0,160,by=10),
     xlab=expression(Vessel~diameter~ ~(mu*m)),
     ylab="Number of vessels")
```

4.5.2 Weather data

Suppose you want to plot more than one variable in a plot, but the units (or ranges) are very different. So, you decide to use two axes. Let's look at an example. For more information on how to deal with the date-time class, see Section 3.6.2.

This code produces Fig. 4.21.

```
# Read the hfemet data. Avoid conversion to factors.
hfemet <- read.csv("HFEmet2008.csv", stringsAsFactors=FALSE)

# Convert to a proper DateTime class:
library(lubridate)
hfemet$DateTime <- mdy_hm(hfemet$DateTime)

# Add the Date :
hfemet$Date <- as.Date(hfemet$DateTime)

# Select one day (a cloudy day in June).
hfemetsubs <- subset(hfemet, Date==as.Date("2008-6-1"))

# Plot Air temperature and PAR (radiation) in one plot.
# First we make a 'vanilla' plot with the default formatting.
with(hfemetsubs, plot(DateTime, Tair, type='l'))
par(new=TRUE)
```

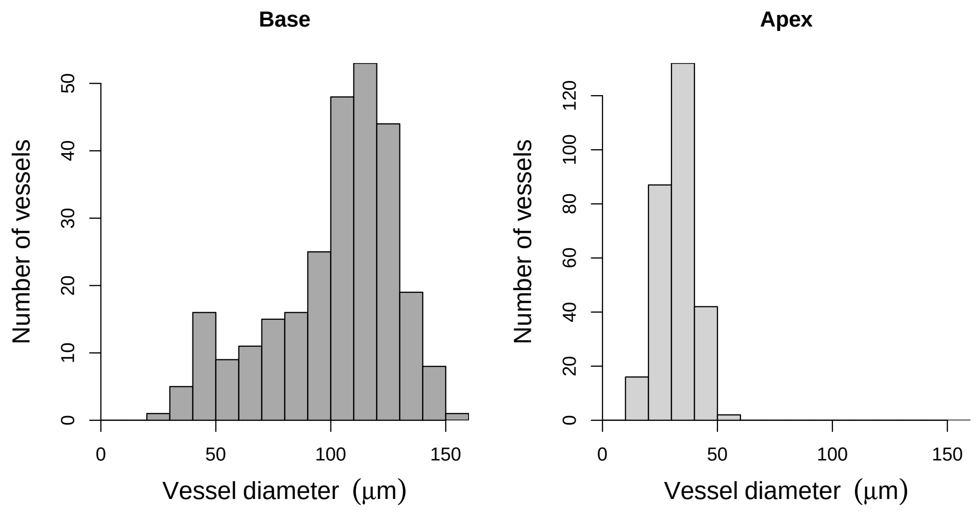


Figure 4.20: Two customized histograms

```
with(hfemetsubs, plot(DateTime, PAR, type='l', col="red",
                      axes=FALSE, ann=FALSE))
```

```
# The key here is to use par(new=TRUE), it produces the next
# plot right on top of the old one.
```

Next, we make the same plot again but with better formatting. Try to figure out what everything means by inspecting the help pages `?par`, `?legend`, and `?mtext`, or by changing the parameters yourself, one at a time.

This code produces Fig. 4.22.

```
par(mar=c(5,5,2,5), cex.lab=1.2, cex.axis=0.9)
with(hfemetsubs, plot(DateTime, Tair, type='l',
                      ylim=c(0,20), lwd=2, col="blue",
                      xlab="Time",
                      ylab=expression(T[air]~ ~(""^"o"*C))))
par(new=TRUE)
with(hfemetsubs, plot(DateTime, PAR, type='l', col="red",
                      lwd=2,
                      ylim=c(0,1000),
                      axes=FALSE, ann=FALSE))
axis(4)
mtext(expression(PAR~ ~(mu*mol~m^-2~s^-1)), side=4, line=3, cex=1.2)
legend("topleft", c(expression(T[air]),"PAR"), lwd=2, col=c("blue","red"),
       bty='n')
```

4.6 Special plots

In this section, we show some examples of special plots that might be useful. There are also a large number of packages that specialize in special plot types, take a look at the `plotrix` and `gplots` packages, for example.

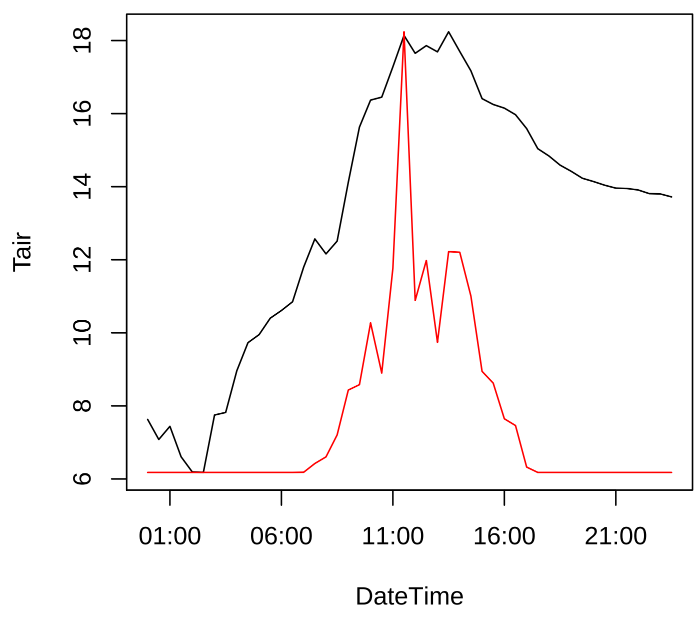


Figure 4.21: A default plot with two axes

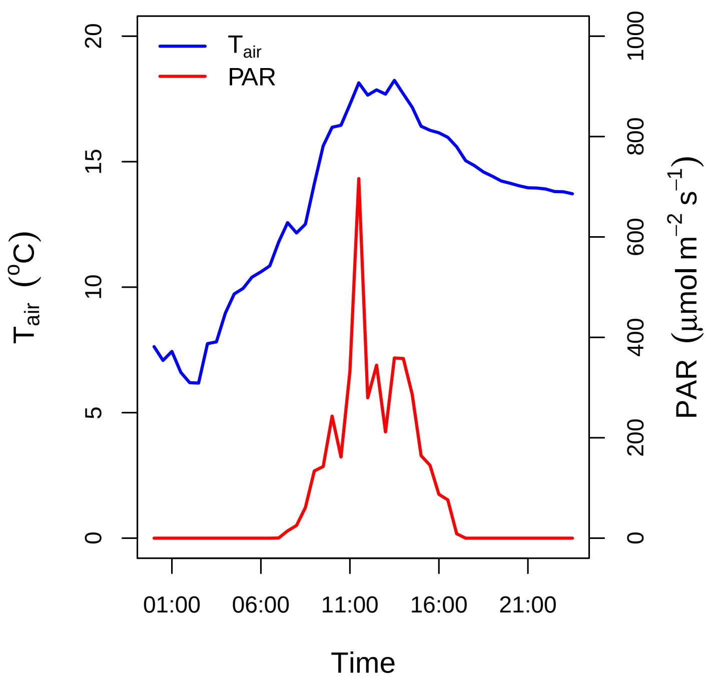


Figure 4.22: A prettified plot with two axes

Very advanced plots can be constructed with the `ggplot2` package. This package has its own steep learning curve (and its own book and website). For some complex graphs, though, it might be the easiest solution.

Further reading The `ggplot2` package offers an alternative way to produce elegant graphics in **R**, one that has been embraced by many **R** users (including those who developed RStudio.) It uses a totally different approach from the one presented here, however, so if you decide to investigate it, be prepared to spend some time learning new material. There are several good introductions available online: Edwin Chen's "Quick Introduction to ggplot2" at http://blog.echen.me/2012/01/17/quick-introduction-to-ggplot2/ and "Introduction to R Graphics with ggplot2," provided by the Harvard Institute for Quantitative Social Science at http://tutorials.iq.harvard.edu/R/Rgraphics/Rgraphics.html#orgheadline19. More detail is available in a book by Hadley Wickham, the author of `ggplot2`. The title is "ggplot2 : Elegant graphics for data analysis," and it is available online from the WSU Library. Finally, the creators of RStudio offer a handy cheat sheet at https://www.rstudio.com/wp-content/uploads/2015/12/ggplot2-cheatsheet-2.0.pdf.

4.6.1 Scatter plot with varying symbol sizes

The `symbols` function is an easy way to pack more information in to a scatter plot, by providing a way to scale the size of the plotting symbols to another variable in the dataframe. Consider the following example:

This code produces Fig. 4.23.

```
# Read data
cereal <- read.csv("Cereals.csv")

# Choose colours
# Find the order of factor levels, so that we can assign colours in the same order
levels(cereal$Cold.or.Hot)

## [1] "C" "H"

# We choose blue for cold, red for hot
palette(c("blue","red"))

# Make the plot
with(cereal, symbols(fiber, potass, circles=fat, inches=0.2, bg=as.factor(Cold.or.Hot),
                    xlab="Fiber content", ylab="Potassium content"))
```

4.6.2 Bar plots of means with confidence intervals

A very common figure in publications is to show group means with confidence intervals, an ideal companion to the output of an ANOVA analysis. Unfortunately, it is somewhat of a pain to produce these in **R**, but the `sciplot` package has made this very easy to do, with the `bargraph.CI` function.

Let's look at an example using the pupae data.

This code produces Fig. 4.24.

```
# Make sure to first install the sciplot package.
library(sciplot)

# Read the data, if you haven't already
```

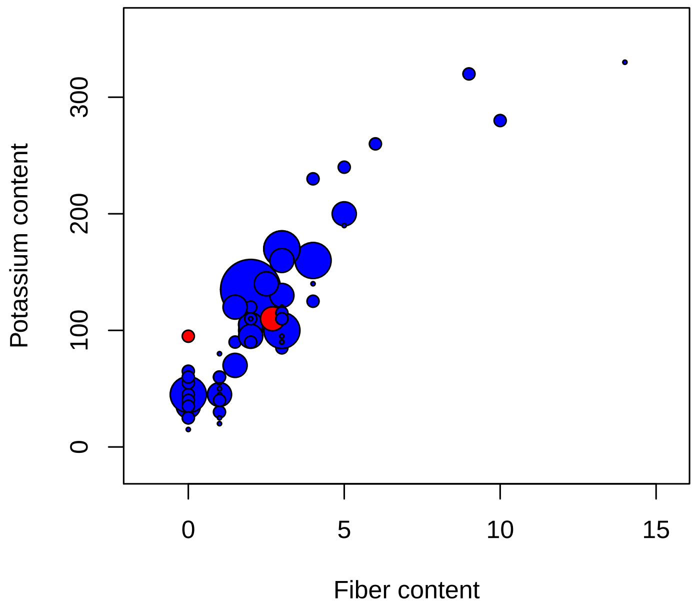


Figure 4.23: Cereal data with symbols size as a function of fat content and colour as function of whether the cereal is served hot (red) or cold (blue).

```
pupae <- read.csv("pupae.csv")

# A fairly standard plot. See ?bargraph.CI to customize many settings.
with(pupae,
     bargraph.CI(T_treatment, Frass, CO2_treatment, legend=TRUE, ylim=c(0,2.5)))
```

4.6.3 Log-log axes

When you make a plot on a logarithmic scale, **R** does not produce nice axis labels. One option is to use the `magicaxis` package, which magically makes pretty axes for log-log plots. Consider this example, which makes Fig. 4.25.

```
# Allometry data
allom <- read.csv("allometry.csv")

# Magic axis package
library(magicaxis)

# Set up some graphical parameters, like axis label size.
par(cex.lab=1.2)

# Log-log plot of branch mass versus diameter
# Here, we set axes=FALSE to suppress the axes
with(allom, plot(log10(diameter), log10(branchmass),
```

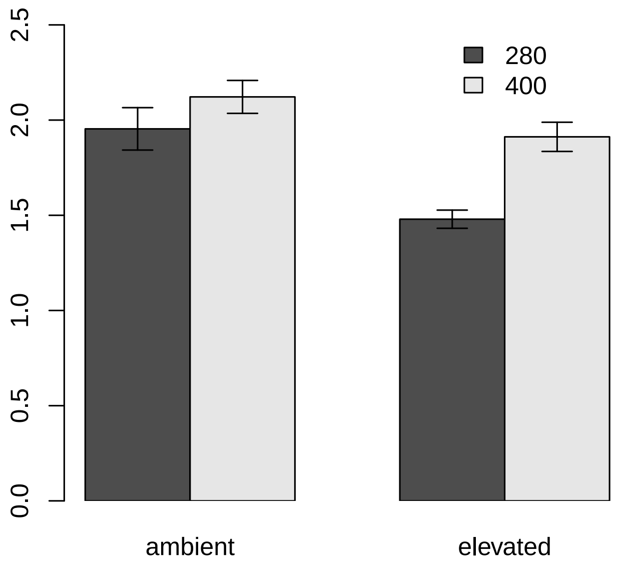


Figure 4.24: Mean frass by temperature and CO2 treatment made with bargraph.CI()

```
                 xlim=log10(c(1,100)),
                 ylim=log10(c(1,1100)),
                 pch=21, bg="lightgrey",
                 xlab="Diameter (cm)", ylab="Branch mass (kg)",
                 axes=FALSE))

# And then we add axes.
# unlog='xy' will make sure the labels are shown in the original scale,
# and we want axes on sides 1 (X) and 2 (Y)
magaxis(unlog='xy', side=c(1,2))

# To be complete, we will add a regression line,
# but only to cover the range of the data.
library(plotrix)
ablineclip(lm(log10(branchmass) ~ log10(diameter), data=allom),
           x1=min(log10(allom$diameter)),
           x2=max(log10(allom$diameter)))

# And add a box
box()
```

4.6.4 Trellis graphics

Trellis graphics are very useful for visualisation of hierarchically structured data. These include multi-factorial experimental designs and studies with multiple observations on the same experimental unit

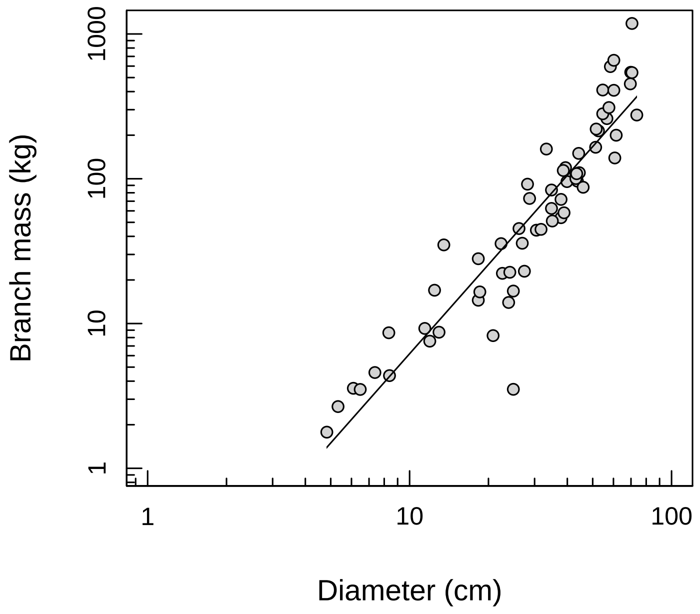


Figure 4.25: Branch mass versus tree diameter on a log-log scale. The axes were produced with the magicaxis package.

(multivariate data). Trellis graphics utilise the structure of a statistical model to easily generate a visual representation of the response according to this structure.

Let's use the data in `pupae` to generate plots of frass production by temperature, nested within CO_2 treatment (Fig. 4.26), using `bwplot` from the `lattice` package.

```
# Make sure to first install the 'lattice' package.
library(lattice)

# bwplot() works best with factors; running the function with numeric or
# integer variables produces odd plots.
CO2trt<-factor(pupae[['CO2_treatment']])

# Use the standard formula notation for specifying variables, using the '|' symbol
# to delineate the factor to split among subplots.
bwplot(Frass~T_treatment|CO2trt,data=pupae)
```

The dot represents the median value for the treatment, the box represents the 25 % and 75 % quantiles, and the hinges represent an estimated confidence interval. Outliers are plotted as circles outside of this range.

For comparisons of two numeric variables within levels of one or more factors, we use `xyplot`. The relationship between pupal weight and frass production can be seen using the following code, with the result in Fig. 4.27.

```
xyplot(Frass ~ PupalWeight|T_treatment:CO2trt, data=pupae)
```

The `par` settings will not help you here; trellis graphics use their own list of parameters. These can

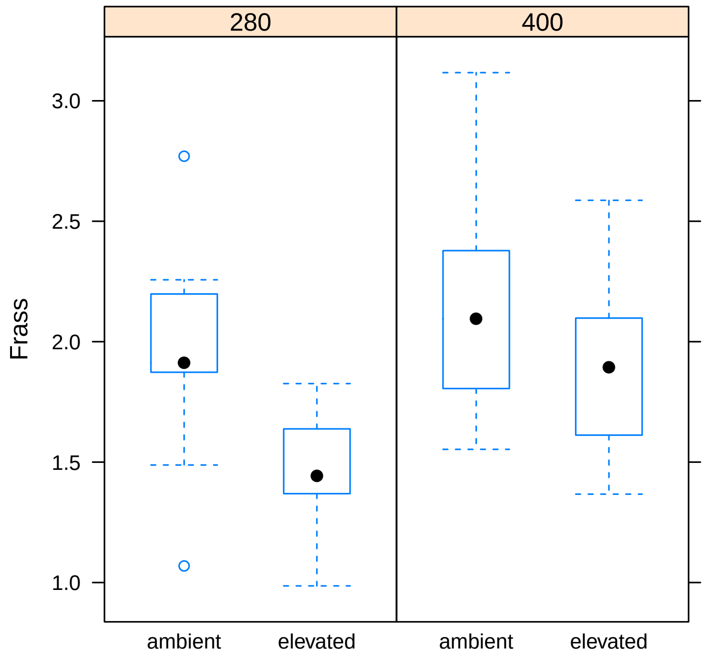


Figure 4.26: Box and whiskers plot of frass by temperature and CO2 treatment.

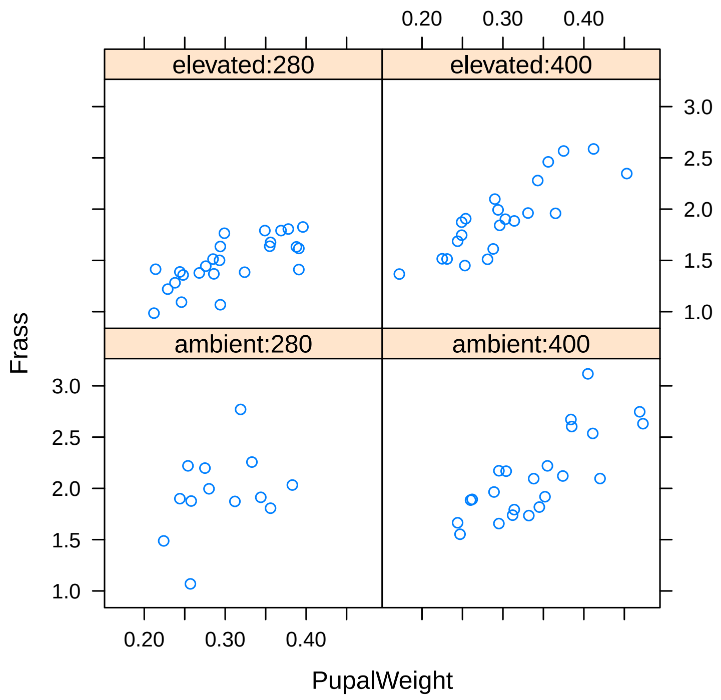


Figure 4.27: Relationship between pupal weight and frass production within each temperature and CO2 treatment combination.

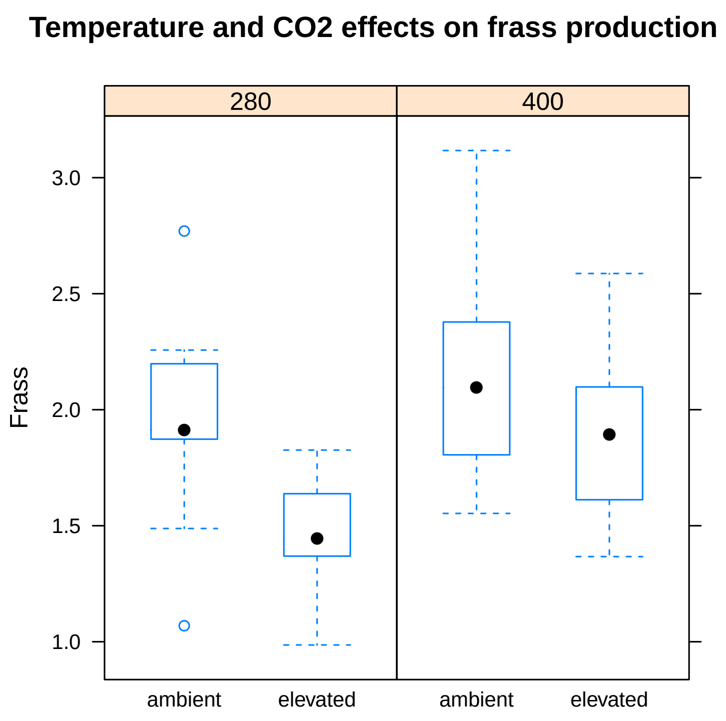


Figure 4.28: A bwplot with a main title added.

be found using `trellis.par.get` and changed using `trellis.par.set`. Plotting with `lattice` functions is different from generic plotting functions in that the call results in an object that contains all of the instructions for generating the plots. This way it is possible to make changes to the object without running the function again. First we run the function, saving it as an object.

```
frass.bwplot <- bwplot(Frass~T_treatment|CO2trt, data=pupae)
```

Typing `frass.bwplot` will results in Fig. 4.26. If you want to view or change any of the values stored in the object `frass.bwplot`, use the command: `str(frass.bwplot)`. For example, we could add a title to the plot (see Fig. 4.28) using the following:

```
frass.bwplot[['main']]<-'Temperature and CO2 effects on frass production'
frass.bwplot
```

The help page `?Lattice` has plenty of information on the options available for the different plot types.

Further reading The `lattice` package offers a comprehensive graphical environment. If you are interested in graphing multivariate data, it is an alternative to `ggplot2`. To find out more, start with the Quick-R website: `http://www.statmethods.net/advgraphs/trellis.html`. For a more detailed introduction see a lab written by Deepayan Sarkar, the package author, for his students: `http://www.isid.ac.in/~deepayan/R-tutorials/labs/04_lattice_lab.pdf`. Paul Murrell, the author of "R Graphics, Second Edition" (2011, Chapman Hall/CRC) has made his chapter on `lattice` graphics available online at `https://www.stat.auckland.ac.nz/~paul/RGraphics/chapter4.pdf`. Chapter 8, "An Introduction to the Lattice Package" from "A Beginner's Guide to R" by Zuur, Ieno and Meesters (2009, Springer) is available online through the WSU library.

4.7 Exporting figures

There is a confusing number of ways to export figures to other formats, to include them in Word documents, print them, or finalize formatting for submission to a journal. Unfortunately, there is no one perfect format for every application. One convenient workflow is to use R markdown, as discussed in Section 1.3.3. This allows you to embed figures directly into Word documents or PDF files. It also allows you to re-create the figures as needed when you update your analysis or data.

Saving figures

You can save figures using the 'Export' button (as pointed out in Section 4.2). A better way, though, is to make exporting your figures part of your script. For this purpose, you can use the `dev.copy2` type functions.

In this example, we make both a PDF and EPS of a figure. Here, we open up a plotting window (of a specified size, in this case 4 by 4 inches), make the plot, and save the output to an EPS and a PDF file.

```
windows(4,4)
par(mar=c(5,5,2,2))
plot(x,y)
dev.copy2pdf(file="Figure1.pdf")
dev.copy2eps(file="Figure1.eps")
```

Inserting figures in Word or Powerpoint

Copy and pasting figures into Word tends to result in a loss of quality. A better approach is to save files as PDFs and submit those with your manuscript. (And better yet, use markdown to incorporate your text, code and figures into a single PDF!)

If you do need to insert figures into Powerpoint, the easiest and safest way to do this is to copy-paste the figure straight into Powerpoint. In RStudio, follow these steps:

1. In the plotting window in RStudio, click 'Export'
2. Select 'Copy plot to clipboard...'
3. Select 'Metafile' just below the figure
4. Then click 'Copy plot'.

Then, in Powerpoint, paste (Ctrl-V).

4.7.1 Sharing figures

When sharing a figure with someone by email or another electronic method, PDF format is always preferred, because the quality of the figure in PDF is optimal. This is the case for viewing the plot on screen, as well as printed. See the previous section on how to generate a PDF of a figure.

4.7.2 Plots with many points or lines

When the above two options give you excessively large file sizes (for example, maps, or figures with tens of thousands of symbols or line pieces), consider converting the figure to a 'bitmap' type, such as `jpeg`, `png` or `tiff`. Care must be taken that you use a high enough resolution to allow a decent print quality.

To make a fairly large `.png`, do:

```r
# First make a plot..
plot(1)

# Example from ?dev.print
# Make a large PNG file of the current plot.
dev.print(png, file = "myplot.png", width = 1024, height = 768)
```

The main drawback of this type of plot is that the character sizes, symbol sizes and so on do not necessarily look like those on your screen. You will have to experiment, usually by making the figure a few different times, to find the right `par` settings for each type of output.

In practice, it may be quicker and safer to make a PDF first, and then convert the PDF to a tiff file with another software package (for instance, Adobe Illustrator).

4.8 Functions used in this chapter

For functions not listed here, please refer to the index at the end of this book.

Function	What it does	Example use
`abline`	Adds a straight line to an existing plot. For a vertical line, use `v` and an x value. For a horizontal line, use `h` and a y value. To add a sloped line, give slope (`a`) and intercept (`b`). Can also be passed a linear model object, see Section 5.5.	`abline(v=5)` `abline(a=0, b=1)`
`ablineclip`	From the `plotrix` package. Like `abline`, but adds the ability to specify coordinates where the line should start and stop.	`ablineclip(a=0, b=1, x1=1, y1=1, x2=3, y2=3)`
`bargraph.CI`	From the `sciplot` package. Makes a bargraph of means and confidence intervals (+/- one standard error.) Arguments include `x.factor` (for the x axis) and `response` (a numeric variable for which means will be calculated). For multifactorial experiments, use `group` for a second factor. Other statistics can be used instead of the mean and standard error; see `?bargraph.CI`.	
`barplot`	Used to create a bar plot. Takes a vector of y values. Names for groups can be supplied using `names.arg`. For bars beside one another, use `beside=TRUE`; for stacked bars use `beside=FALSE`. For a horizontal bar chart, use `horiz=TRUE`.	
`barplot2`	From the `gplots` package. Similar to `barplot`, but adds the ability to use different colours for the plot area, logarithmic axes, a background grid, and confidence intervals.	
`boxplot`	Creates quick box-and-whisker plots. Takes a formula: `y ~ factor`. Midline shows the median; lower and upper box edges show 1st and 3rd quartiles respectively. The `range` of the whiskers is a multiple of the difference between the 1st and 3rd quantiles. The default is 1.5; to extend to the furthest outlier use `range=0`.	`boxplot(ncases ~ agegp, data=esoph)`
`bwplot`	From the `lattice` package. Makes pretty box-and-whisker plots. Takes a formula: `response ~ factor`. For two grouping factors use `response ~ first_group|second_group`.	

Function	**What it does**	**Example use**
`colorRampPalette`	Can be a bit confusing, but very useful. When given two or more colours, it outputs a new, custom function. This new function can then be used to generate colour pallets with any number of steps from those colours. See Section 4.4.2 for more explanation.	`greypal <- colorRampPalette(gray(0:1))` `greypal(20)`
`curve`	Can be used to plot curves, or to add curves to an existing plot. The first argument can be either a function or an expression that evaluates to a curve. To specify the range of the curve, use the arguments `from` and `to`. Use `add=TRUE` to add a curve to an existing plot.	`curve(cos(x), from=0, to=2*pi)`
`cut`	Can be used to divide a numerical variable into bins and turn it into a factor variable. You must provide data to be broken and a number of `breaks`. Optionally, you can also provide `labels`.	`grades <- cut(50:100, 4, labels=c("P","C", "D","HD"))`
`demo`	A function which runs demos of different things **R** can do. In this chapter we looked at `demo(colors)`. For other available demos, type `demo()`.	`demo(colors)` `demo()`
`dev.copy2eps`	Copies a graphic you have constructed in **R** to an EPS file. Supply a `file` name.	`plot(mtcars)` `dev.copy2eps( file="Cars.eps")`
`dev.copy2pdf`	Copies a graphic you have constructed in **R** to a PDF file. Supply a `file` name.	`plot(iris)` `dev.copy2pdf( file="Iris.pdf")`
`dev.off`	Closes the current graphical device. Useful when you want to reset graphical parameters (`par`).	`dev.off()`
`dnorm`	Gives a density function for the normal distribution. Can be used to add a normal curve to a density histogram. You can supply a mean.	`dnorm(x, mean=5)`
`expression`	Used to add mathematical expressions to plot axes and annotations. These include subscripts, superscripts, Greek letters, other symbols, and Unicode characters. See `?plotmath` and Section 4.4.4 for examples.	`expression(Area~(m^2))` `expression(CO[2])` `expression( Delta*degree*C)`
`gray`, `grey`	Returns codes for levels of grey from 0 (black) to 1 (white). Can be given either a single number or a vector. Both spellings work.	`grey(0.5)` `grey(seq(0.1,0.9, by=0.1))`

Function	**What it does**	**Example use**
`heat.colors`	Creates a vector of continuous shades from red to yellow. Supply the number of intermediate shades.	`heat.colors(5)` `plot(1:10, col=heat.colors(10)`
`hist`	Plots a histogram. Supply a vector of data. You can set how to determine `breaks` in the data; see `?hist` for details. The argument `freq` specifies whether to plot frequency (`freq=TRUE`) or probability density (`freq=FALSE`).	
`layout`	Allows the design of complex layouts for graphics with multiple plots. The underlying idea is to divide the plot area into a grid (represented by a matrix) and use numbers to indicate which cells should be filled with 1...*n* plots. When more than one cell is filled with the same number, that plot will take up all of those cells. Once the layout has been designed, it will be filled, in order, with *n* plots.	`l<-layout( matrix(c(1,1,2,3), nrow=2, ncol=2))` `layout.show(l)`
`legend`	Takes care of most of the hard work involved in adding legends to plots. As a result, there are many, many possible arguments (see `?legend`). The simplest possible use is to give a location for the legend (possibilities include `"topright"`,`"topleft"`,`"bottomright"`, `"bottomleft"` and `"center"` as well as xy coordinates), text for the labels (`legend`) and corresponding colours (`fill`).	`legend("topright", legend=c(1,2), fill=rainbow(2))`
`magaxis`	From the `magicaxis` package. Adds nicely arranged log-scale axes with appropriately-spaced major and minor tick marks.	`magaxis(unlog="xy", side=c(1,2))`
`mtext`	Adds text to the margins of a plot. Specify which `side` the text should appear on using 1=bottom, 2=left, 3=top, 4=right. To control how far out text should be, use `line`, starting from 0 (right next to the axis) and counting up as you move away.	`mtext("Extra text", side=2, line=2)`
`palette`	A way of storing colours in **R**. Once you add colours to a new `palette`, they become available for use in future plots. See Section 4.4.2 for ways to take advantage of this. To create a new `palette`, supply a vector of colours. See also `rainbow`, `heat.colors`, `grey`, and `colorRampPalette`.	`palette(rainbow(7))` `palette(c("maroon1", "olivedrab4", "royalblue1"))`

Function	What it does	Example use
`pie`	Used to plot pie charts. Takes a vector of numbers. You may also supply `labels` and colors (using `col`).	`pie(c(49,51),labels= c("Half empty", "Half full"))`
`plot`	Used to produce scatterplots and line plots. Most of this chapter focuses on explaining ways to use this function. To change the appearance of a single plot, rather than all future plots, most graphical parameters for `par` can be used directly in the plot function.	`plot(1)`
`plotCI`	From the `plotrix` package. Give `x` and `y` coordinates for the plots, and an upper interval width (`uiw`). For non-symmetrical confidence intervals, provide a lower interval width (`liw`).	`See Section 4.3.2`
`points`	Adds points to an existing graph.	`See Section 4.4.7.`
`rainbow`	Produces a vector of colors from red through orange, yellow, green, blue, purple, and magenta back to red. Supply the number of colors to be returned.	`plot(1:20, col=rainbow(20), pch=15)`
`symbols`	Creates plots using different types of symbols for each data point. Can be set to `add` to an existing plot.	`with(trees, symbols(Height, Volume, circles = Girth/24, inches = FALSE))`
`tapply`	Used to summarize data in dataframes. Will be discussed further in Chapter 6.	`See Section 6.2.1`
`text`	Adds text to a plot. Takes `x` and `y` coordinates as locations for the text; text is supplied to `labels`. The `pos` parameter offsets the text. Values 1, 2, 3 and 4 place text below, left, above, and right.	`plot(1)` `text(1,1,"A point", pos=4)`
`trellis.par.get`	Used to get the current graphical parameters for `trellis` plots. These are independent of the `par` parameters.	
`trellis.par.set`	Used to set the current graphical parameters for `trellis` plots. These are independent of the `par` parameters.	
`windowsFonts`	For Windows devices only. Allows you to define and use system fonts in **R** plots.	`See Section 4.4.6`
`xyplot`	From the `trellis` package. Can be used to produce multivariate scatterplots or time series plots.	`See Section 4.6.4 and Fig. 4.27.`

4.9 Exercises

In these exercises, we use the following colour codes:

🟩 **Easy**: make sure you complete some of these before moving on. These exercises will follow examples in the text very closely.

⧫ **Intermediate**: a bit harder. You will often have to combine functions to solve the exercise in two steps.

▲ **Hard**: difficult exercises! These exercises will require multiple steps, and significant departure from examples in the text.

We suggest you complete these exercises in an **R** markdown file. This will allow you to combine code chunks, graphical output, and written answers in a single, easy-to-read file.

4.9.1 Scatter plot with the pupae data

1. 🟩 Read the pupae data (see Section A.6, p. 248). Convert 'CO2_treatment' to a factor. Inspect the levels of this factor variable.
2. 🟩 Make a scatter plot of `Frass` vs. `PupalWeight`, with blue solid circles for a CO_2 concentration of 280ppm and red for 400ppm. Also add a legend.
3. 🟩 The problem with the above figure is that data for both temperature treatments is combined. Make two plots (either in a PDF, or two plots side by side), one with the 'ambient' temperature treatment, one with 'elevated'.
4. In the above plot, make sure that the X and Y axis ranges are the same for both plots. *Hint*: use `xlim` and `ylim`.
5. 🟩 Instead of making two separate plots, make one plot that uses different colors for the $CO_(2)$ treatments and different symbols for the 'ambient' and 'elevated' temperature treatments. Choose some nice symbols from the help page of the `points` function.
6. ⧫ Add two legends to the above plot, one for the temperature treatment (showing different plotting symbols), and one for the CO_2 treatments (showing different colours).
7. ▲ Generate the same plot as above but this time add a single legend that contains symbols and colours for each treatment combination (CO_2 : T).
8. ⧫ In Fig. 4.4, figure out why no error bar was plotted for the first bar.

4.9.2 Flux data

Use the Eddy flux data for this exercise (Section A.8, p. 249).

1. ⧫ Produce a line plot for `FCO2` for *one day* out of the dataset (recall Section 3.6.2, p. 73).
2. ⧫ Adjust the X and Y axis ranges, add pretty labels on the axes, increase the thickness of the line (see `lwd` in `?par`).
3. ⧫ Now add points to the figure (using `points`, see Section 4.4.7 on p. 108), with different colours for when the variable `ustar` is less than 0.15 ('bad data' in red). *Hint:* recall Section 3.2 on p. 59 on how to split a numeric vector into a factor.

4.9.3 Hydro dam

Use the hydro dam data as described in Section A.3.

1. ■ Read the hydro data, make sure to first convert the `Date` column to a proper `Date` class.
2. ■ Make a line plot of `storage` versus `Date`
3. ■ Make the line thicker, and a dot-dashed style (see `?par`). Use the search box in the RStudio help pane to find this option (top-right of help pane).
4. ▲ Next, make the same plot with points (not lines), and change the colour of the points in the following way: `forestgreen` if storage is over 500, `orange` if storage is between 235 and 500, and `red` if storage is below 235. (*Hint*: use `cut`, as described in Section 3.2, p. 59).

4.9.4 Coloured scatter plot

Use the Coweeta tree data (see Section A.2, p. 247).

1. ■ Read the data, count the number of observations per species.
2. ▲ Take a subset of the data including only those species with at least 10 observations. *Hint:* simply look at `table(coweeta$species)`, make a note of which species have more than 10 observations, and take a subset of the dataset with those species (recall the examples in Section 2.3.2, p. 47).
3. ♦ Make a scatter plot of `biomass` versus `height`, with the symbol colour varying by `species`. Choose some nice colours, and use filled squares for the symbols. Also add a title to the plot, in italics.
4. ■ Log-transform `biomass`, and redraw the plot.

4.9.5 Superimposed histograms

First inspect the example for the vessel data in Section 4.5.1 (p. 112).

1. ▲ Use a function from the `epade` package to produce a single plot with both histograms (so that they are superimposed). You need to figure out this function yourself, it is not described in this book

4.9.6 Trellis graphics

1. ♦ Change the labels in the box and whisker plot (Fig. 4.26) to indicate the units of CO_2 concentration (ppm) and add a label to the x-axis indicating the factor (temperature).

Chapter 5

Basic statistics

This book is not an *Introduction to statistics*. There are many manuals (either printed or on the web) that document the vast array of statistical analyses that can be done with **R**. To get you started, though, we will show a few very common analyses that are easy to do in **R**.

In all of the following, we assume you have a basic understanding of linear regression, Student's t-tests, ANOVA, and confidence intervals for the mean.

5.1 Probability Distributions

As we assume you have completed an *Introduction to statistics* course, you have already come across several probability distributions. For example, the *Binomial* distribution is a model for the distribution of the number of *successes* in a sequence of independent trials, for example, the number of heads in a coin tossing experiment. Another commonly used discrete distribution is the *Poisson*, which is a useful model for many kinds of count data. Of course, the most important distribution of all is the *Normal* or Gaussian distribution.

R provides sets of functions to find densities, cumulative probabilities, quantiles, and to draw random numbers from many important distributions. The names of the functions all consist of a one letter prefix that specifies the type of function and a stem which specifies the distribution. Look at the examples in the table below.

prefix	function
`d`	density
`p`	cumulative probabilities
`q`	quantiles
`r`	simulate

stem	distribution
`binom`	Binomial
`pois`	Poisson
`norm`	Normal
`t`	Student's t
`chisq`	χ^2
`f`	F

So for example,

```
# Calculate the probability of 3 heads out of 10 tosses of a fair coin.
# This is a (d)ensity of a (binom)ial distribution.
dbinom(3, 10, 0.5)

## [1] 0.1171875

# Calculate the probability that a normal random variable (with
# mean of 3 and standard deviation of 2) is less than or equal to 4.
# This is a cumulative (p)robability of a (norm)al variable.
pnorm(4, 3, 2)

## [1] 0.6914625

# Find the t-value that corresponds to a 2.5% right-hand tail probability
# with 5 degrees of freedom.
# This is a (q)uantile of a (t)distribution.
qt(0.975, 5)

## [1] 2.570582

# Simulate 5 Poisson random variables with a mean of 3.
# This is a set of (r)andom numbers from a (pois)son distribution.
rpois(5, 3)

## [1] 1 3 3 3 5
```

See the help page `?Distributions` for more details.

To make a quick plot of a distribution, we already saw the use of the density function in combination with `curve` in Section 4.3.3. Here is another example (this makes Fig. 5.1).

```
# A standard normal distribution
curve(dnorm(x, sd=1, mean=0), from=-3, to=3,
      ylab="Density", col="blue")

# Add a t-distribution with 3 degrees of freedom.
curve(dt(x, df=3), from =-3, to=3, add=TRUE, col="red")

# Add a legend (with a few options, see ?legend)
legend("topleft", c("Standard normal","t-distribution, df=3"), lty=1, col=c("blue","red"),
       bty='n', cex=0.8)
```

Try this yourself Make a histogram (recall Section 4.3.3) of a sample of random numbers from a distribution of your choice.

5.2 Descriptive Statistics

Descriptive statistics summarise some of the properties of a given data set. Generally, we are interested in measures of location (central tendency, such as mean and median) and scale (variance or standard deviation). Other descriptions can include the sample size, the range, and so on. We already encountered a number of functions that can be used to summarize a vector.

Let's look at some examples for the Pupae dataset (described in Section A.6).

```
# Read data
pupae <- read.csv("pupae.csv")

# Extract the weights (for convenience)
weight <- pupae$PupalWeight

# Find the number of observations
length(weight)
```

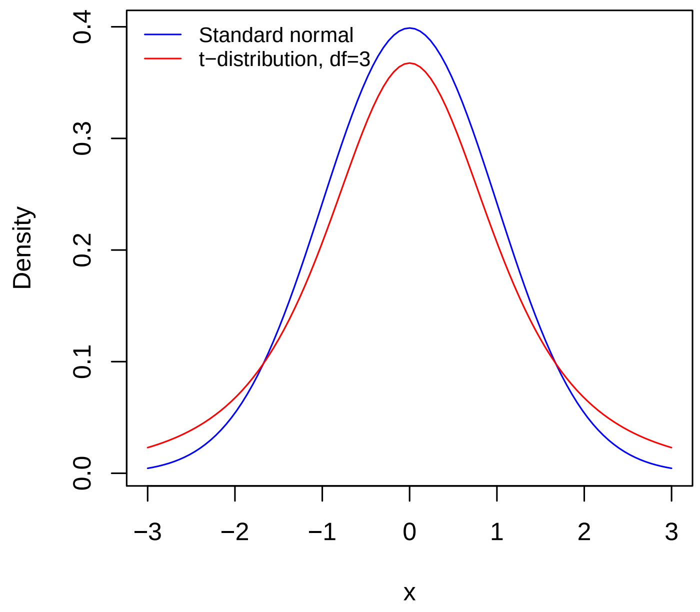


Figure 5.1: Two univariate distributions plotted with curve()

```
## [1] 84

# Find the average (mean) weight
mean(weight)

## [1] 0.3110238

# Find the Variance
var(weight)

## [1] 0.004113951
```

Note that **R** will compute the sample variance (not the population variance). The standard deviation can be calculated as the square root of the variance, or use the `sd` function directly.

```
# Standard Deviation
var.wgt <- var(weight)
sqrt(var.wgt)

## [1] 0.06414009

# Standard Deviation
sd(weight)

## [1] 0.06414009
```

Robust measures of the location and scale are the median and inter-quartile range; **R** has functions for these.

```
# median and inter-quartile range
median(weight)
```

```
## [1] 0.2975
IQR(weight)
## [1] 0.09975
```

The median is the 50th percentile or the second quartile. The `quantile` function can compute quartiles as well as arbitrary percentiles/quantiles.

```
quantile(weight)
##      0%     25%     50%     75%    100%
## 0.17200 0.25625 0.29750 0.35600 0.47300
quantile(weight, probs=seq(0,1,0.1))
##     0%    10%    20%    30%    40%    50%    60%    70%    80%    90%
## 0.1720 0.2398 0.2490 0.2674 0.2892 0.2975 0.3230 0.3493 0.3710 0.3910
##   100%
## 0.4730
```

Missing Values: All of the above functions will return `NA` if the data contains *any* missing values. However, they also provide an option to remove missing values (`NA`s) before their computations (see also Section 3.4).

```
weightNA <- weight
weightNA[40] <- NA
mean(weightNA)
## [1] NA
mean(weightNA, na.rm=TRUE)
## [1] 0.3113373
```

The `summary` function provides a lot of the above information in a single command:

```
summary(weight)
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.1720  0.2562  0.2975  0.3110  0.3560  0.4730
```

The `moments` package provides 'higher moments' if required, for example, the `skewness` and `kurtosis`.

```
# load the moments package
library(moments)
skewness(weight)
## [1] 0.3851656
kurtosis(weight)
## [1] 2.579144
```

The `pastecs` package includes a useful function that calculates many descriptive statistics for numeric vectors, including the standard error for the mean (for which **R** has no built-in function).

```
library(pastecs)

# see ?stat.desc for description of the abbreviations
stat.desc(weight)
##      nbr.val     nbr.null       nbr.na          min          max
## 84.000000000  0.000000000  0.000000000  0.172000000  0.473000000
##        range          sum       median         mean      SE.mean
##   0.301000000 26.126000000  0.297500000  0.311023810  0.006998258
## CI.mean.0.95          var      std.dev     coef.var
##   0.013919253  0.004113951  0.064140091  0.206222446
```

```
# conveniently, the output is a character vector which we can index by name,
# for example extracting the standard error for the mean
stat.desc(weight)["SE.mean"]
```

```
##     SE.mean
## 0.006998258
```

Sometimes you may wish to calculate descriptive statistics for subgroups in the data. We will come back to this extensively in Section 6.2, but here is a quick introduction.

The function `tapply` allows you to apply any function to subgroups defined by a second (grouping) variable, as we will see in Chapter 6.

```
# tapply
with(pupae, tapply(PupalWeight, Gender, mean))
```

```
##         0         1
## 0.2724884 0.3512857
```

5.3 Inference for a single population

Inference is answering questions about population parameters based on a sample. The mean of a random sample from a population is an estimate of the population mean. Since it is a single number it is called a point estimate. It is often desirable to estimate a range within which the population parameter lies with high probability. This is called a confidence interval.

One way to get confidence intervals in **R** is to use the quantile functions for the relevant distribution. Remember from your introductory statistics course that a $100(1-\alpha)\%$ confidence interval for the mean on normal population is given by,

$$\bar{x} \pm t_{\alpha/2,n-1} \frac{s}{\sqrt{n}}$$

where $\bar{x}$ is the sample mean, s the sample standard deviation and n is the sample size. $t_{\alpha/2,n-1}$ is the $\alpha/2$ tail point of a t-distribution on $n-1$ degrees of freedom. That is, if T has a t-distribution on $n-1$ degrees of freedom.

$$P(T \leq t_{\alpha/2,n-1}) = 1 - \alpha/2$$

The **R** code for this confidence interval can be written as,

```
alpha <- 0.05 # 95% confidence interval
xbar <- mean(weight)
s <- sd(weight)
n <- length(weight)
half.width <- qt(1-alpha/2, n-1)*s/sqrt(n)

# Confidence Interval
c(xbar - half.width, xbar + half.width)
```

```
## [1] 0.2971046 0.3249431
```

Here, we assumed a normal distribution for the population. You may have been taught that if n is *large*, say $n > 30$, then you can use a normal approximation. That is, replace `qt(1-alpha/2, n-1)` with `qnorm(1-alpha/2)`, but there is no need, **R** can use the t-distribution for any n (and the results will be the same, as the t-distribution converges to a normal distribution when the df is large).

Try this yourself Confirm that the t-distribution converges to a normal distribution when n is large (using `qt` and `qnorm`).

Hypothesis testing

There may be a reason to ask whether a dataset is consistent with a certain mean. For example, are the pupae weights consistent with a population mean of 0.29? For normal populations, we can use Student's t-test, available in **R** as the `t.test` function. Let's test the null hypothesis that the population mean is 0.29:

```
t.test(weight, mu=0.29)

##
##  One Sample t-test
##
## data:  weight
## t = 3.0041, df = 83, p-value = 0.00352
## alternative hypothesis: true mean is not equal to 0.29
## 95 percent confidence interval:
##  0.2971046 0.3249431
## sample estimates:
## mean of x
## 0.3110238
```

Note that we get the t-statistic, degrees of freedom ($n - 1$) and a p-value for the test, with the specified alternative hypothesis (not equal, i.e. two-sided). In addition, `t.test` gives us a 95% confidence interval (compare to the above), and the estimated mean, $\bar{x}$.

We can use `t.test` to get any confidence interval, and/or to do one-sided tests,

```
t.test(weight, mu=0.29, conf.level=0.90)

##
##  One Sample t-test
##
## data:  weight
## t = 3.0041, df = 83, p-value = 0.00352
## alternative hypothesis: true mean is not equal to 0.29
## 90 percent confidence interval:
##  0.2993828 0.3226649
## sample estimates:
## mean of x
## 0.3110238

t.test(weight, mu=0.29, alternative="greater", conf.level=0.90)

##
##  One Sample t-test
##
## data:  weight
## t = 3.0041, df = 83, p-value = 0.00176
## alternative hypothesis: true mean is greater than 0.29
## 90 percent confidence interval:
##  0.3019832       Inf
## sample estimates:
## mean of x
## 0.3110238
```

Note that the confidence interval is one-sided when the test is one-sided.

The `t.test` is appropriate for data that is approximately normally distributed. You can check this using a histogram or a QQ-plot (see Sections 4.3.3 and **??**). If the data is not very close to a normal distribution then the `t.test` is often still appropriate, as long as the sample is large.

If the data is not normal and the sample size is small, there are a couple of alternatives: transform the data (often a log transform is enough) or use a *nonparametric* test, in this case the Wilcoxon signed rank test. We can use the `wilcox.test` function for the latter, its interface is similar to `t.test` and it tests the hypothesis that the data is symmetric about the hypothesized population mean. For example,

```
wilcox.test(weight, mu=0.29)

##
##  Wilcoxon signed rank test with continuity correction
##
## data:  weight
## V = 2316.5, p-value = 0.009279
## alternative hypothesis: true location is not equal to 0.29

wilcox.test(weight, mu=0.29, alternative="greater")

##
##  Wilcoxon signed rank test with continuity correction
##
## data:  weight
## V = 2316.5, p-value = 0.004639
## alternative hypothesis: true location is greater than 0.29
```

Test for proportions

Sometimes you want to test whether observed proportions are consistent with a hypothesized population proportion. For example, consider a coin tossing experiment where you want to test the hypothesis that you have a fair coin (one with an equal probability of landing heads or tails). In your experiment, you get 60 heads out of 100 coin tosses. Do you have a fair coin? We can use the `prop.test` function:

```
# 60 'successes' out of a 100 trials, the hypothesized probability is 0.5.
prop.test(x=60, n=100, p=0.5)

##
##  1-sample proportions test with continuity correction
##
## data:  60 out of 100, null probability 0.5
## X-squared = 3.61, df = 1, p-value = 0.05743
## alternative hypothesis: true p is not equal to 0.5
## 95 percent confidence interval:
##  0.4970036 0.6952199
## sample estimates:
##   p
## 0.6
```

```
# Same as above, but for a one-sided test.
prop.test(60, 100, p=0.5, alternative="greater")
```

```
##
##  1-sample proportions test with continuity correction
##
## data:  60 out of 100, null probability 0.5
## X-squared = 3.61, df = 1, p-value = 0.02872
## alternative hypothesis: true p is greater than 0.5
## 95 percent confidence interval:
##  0.5127842 1.0000000
## sample estimates:
##   p
## 0.6
```

Note: You might think that I chose to do a one-sided test *greater* because 60 is greater than the expected number of 50 for a fair coin – I didn't! Don't use your data to decide on the null or alternative hypothesis – it renders the p-values meaningless.

5.4 Inference for two populations

Commonly, we wish to compare two (or more) populations. For example, the `pupae` dataset has pupal weights for female and male pupae. We may wish to compare the weights of males (gender=0) and females (gender=1).

There are two ways to use `t.test` to compare the pupal weights of males and females. In the first method, we make two vectors,

```
pupae <- read.csv("pupae.csv")
weight <- pupae$PupalWeight
gender <- pupae$Gender
weight.male <- weight[gender==0]
weight.female <- weight[gender==1]

# We will assume equal variance for male and female pupae (see Unequal variances, below):
t.test(weight.male, weight.female, var.equal=TRUE)
```

```
##
##  Two Sample t-test
##
## data:  weight.male and weight.female
## t = -7.3571, df = 76, p-value = 1.854e-10
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.10012896 -0.05746573
## sample estimates:
## mean of x mean of y
## 0.2724884 0.3512857
```

Try this yourself Confirm that there are missing data in both variables in the example above. The default action is to omit all missing values (see description under `na.action` in the help file `?t.test`).

There is also a *formula* interface for `t.test`. The formula interface is important because we will use it in many other functions, like linear regression and linear modelling. For the `t.test` we can use the formula interface on the extracted variables, or without extracting the variables.

```
# Using the vectors we constructed in the previous example
t.test(weight ~ gender,  var.equal=TRUE)

##
##  Two Sample t-test
##
## data:  weight by gender
## t = -7.3571, df = 76, p-value = 1.854e-10
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.10012896 -0.05746573
## sample estimates:
## mean in group 0 mean in group 1
##       0.2724884       0.3512857

# Or by specifying the data= argument.
t.test(PupalWeight~Gender,  data=pupae, var.equal=TRUE)

##
##  Two Sample t-test
##
## data:  PupalWeight by Gender
## t = -7.3571, df = 76, p-value = 1.854e-10
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.10012896 -0.05746573
## sample estimates:
## mean in group 0 mean in group 1
##       0.2724884       0.3512857
```

Paired data

The `t.test` can also be used when the data are paired, for example, measurements taken before and after some treatment on the same subjects. The pulse dataset is an example of paired data (see Section A.9). We will compare pulse rates before and after exercise, including only those subjects that exercised (`Ran=1`),

```
pulse <- read.table("ms212.txt", header=TRUE)
pulse.before <- with(pulse, Pulse1[Ran==1])
pulse.after <- with(pulse, Pulse2[Ran==1])
t.test(pulse.after, pulse.before, paired=TRUE)

##
##  Paired t-test
##
## data:  pulse.after and pulse.before
## t = 16.527, df = 45, p-value < 2.2e-16
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  45.12843 57.65418
## sample estimates:
## mean of the differences
##                 51.3913
```

Unequal variances

The default for the two-sample `t.test` is actually to *not* assume equal variances. The theory for this kind of test is quite complex, and the resulting t-test is now only approximate, with an adjustment called the 'Satterthwaite' or 'Welch' approximation made to the degrees of freedom.

```
t.test(PupalWeight ~ Gender,  data=pupae)

##
##  Welch Two Sample t-test
##
## data:  PupalWeight by Gender
## t = -7.4133, df = 74.628, p-value = 1.587e-10
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.09997364 -0.05762104
## sample estimates:
## mean in group 0 mean in group 1
##       0.2724884       0.3512857
```

Since this modified t-test makes fewer assumptions, you could ask why we ever use the equal variances form. If the assumption is reasonable, then this (equal variances) form will have more power, i.e. will reject the null hypothesis more often when it is actually false.

Assumed normality

The two-sample t-test assumes normality of the data (which you can check using a histogram or a QQ-plot) or that the sample sizes are large enough that the *central limit theorem* applies. Note that the paired t-test assumes only that the differences are normal. The `wilcox.test` can be used when any of these assumptions are suspect. In the case of two samples (unpaired), this test used is called the Wilcoxon rank sum test (also known as the Mann-Whitney test).

```
wilcox.test(pulse.after, pulse.before, paired=TRUE, exact=FALSE)

##
##  Wilcoxon signed rank test with continuity correction
##
## data:  pulse.after and pulse.before
## V = 1081, p-value = 3.624e-09
## alternative hypothesis: true location shift is not equal to 0

wilcox.test(PupalWeight ~ Gender,  data=pupae, exact=FALSE)

##
##  Wilcoxon rank sum test with continuity correction
##
## data:  PupalWeight by Gender
## W = 152.5, p-value = 1.704e-09
## alternative hypothesis: true location shift is not equal to 0
```

5.4.1 Power

When testing a hypothesis, remember that there are two types of possible errors, due to the random nature of sampling data. These are the "Type 1 error" (rejecting the null hypothesis when it is actually true), and the "Type 2 error" (failing to reject the null when it is actually false). The probability of a Type 1 error is controlled by α, the threshold on the p-value. The p-value is the probability of observing the test statistic, if the null hypothesis is actually true. So by keeping α small (for some reason, 0.05 is most commonly used), we control the chance of a Type 1 error.

Statistical power is defined as 1 - the probability of a Type 2 error. Or in other words, the probability that we reject the null hypothesis when it is actually false. Consider the situation where we compare the means of two samples. It is easy to see that our power depends not only on α, but also on the actual difference in means of the populations that the samples were drawn from. If they are very different, it will be easier to find a significant difference. So, to calculate the power we must specify how different the means are under the alternative hypothesis.

For a t-test, we can use the `power.t.test` function to calculate the power. To approximate the power for the pupal weight t-test (as we saw in the previous section), we can use the following,

```
power.t.test(n=35, delta=0.08, sd=0.05, sig.level=0.05)

##
##      Two-sample t test power calculation
##
##               n = 35
##           delta = 0.08
##              sd = 0.05
##       sig.level = 0.05
##           power = 0.9999982
##     alternative = two.sided
##
## NOTE: n is number in *each* group
```

Here we have assumed equal groups of size 35 for each gender (although this is not exactly correct), a true difference in mean weights of 0.08, and a standard deviation of 0.05. The power is over 99%, meaning that, with these conditions, we will be able to reject the null hypothesis 99% of the time.

We can also calculate the required sample size, if we wish to attain a certain power. For example, suppose we want to detect a difference of 0.02 with 75% power. What sample size do we need?

```
power.t.test(delta=0.02, sd=0.05, sig.level=0.05, power=0.75)

##
##      Two-sample t test power calculation
##
##               n = 87.7248
##           delta = 0.02
##              sd = 0.05
##       sig.level = 0.05
##           power = 0.75
##     alternative = two.sided
##
## NOTE: n is number in *each* group
```

We would need 88 observations for each gender.

Try this yourself Using `power.t.test` as in the examples above, see what happens when you set α (`sig.level`) to 0.01 or 0.1. Decide for yourself if the result makes sense.

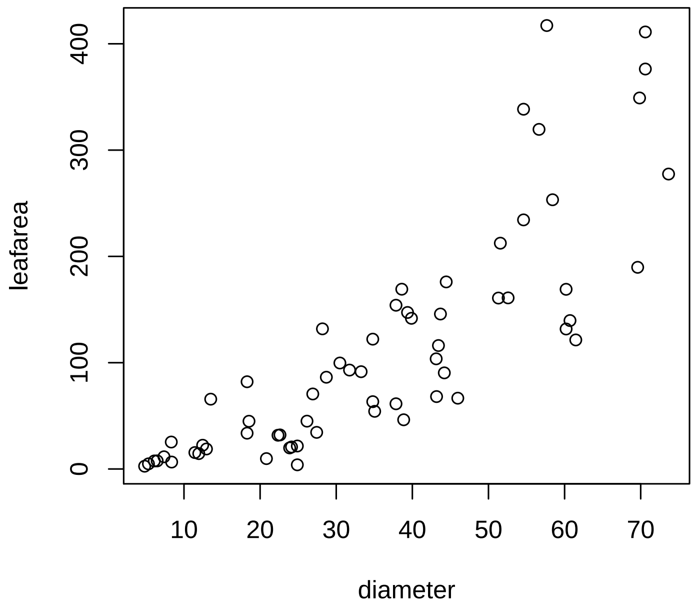


Figure 5.2: Quick inspection of the allometry data, before we perform a linear regression.

5.5 Simple linear regression

To fit linear models of varying complexity, we can use the `lm` function. The simplest model is a straight-line relationship between an x and a y variable. In this situation, the assumption is that the y-variable (the response) is a linear function of the x-variable (the independent variable), plus some random noise or measurement error. For the simplest case, both x and y are assumed to be continuous variables. In statistical notation we write this as,

$$y = \alpha + \beta x + \varepsilon \tag{5.1}$$

Here α and β are (population) parameters that need to be estimated from the data. The error (ϵ) is assumed to follow a normal distribution with a mean of zero, and a standard deviation of σ. It is also assumed that σ is constant and does not depend on x.

Let's look at an example using the allometry data (see Fig. 5.2),

```
# Read data
allom <- read.csv("Allometry.csv")
plot(leafarea~diameter, data=allom)
```

We can see from this plot that leaf area generally increases with tree diameter. So we can use `lm` to estimate the parameters in equation 5.1, or in other words to 'fit the model'.

```
# Fit linear regression of 'leafarea' on 'diameter',
# Results are stored in an object called model
model <- lm(leafarea~diameter, data=allom)

# Print a summary of the regression:
summary(model)
```

```
##
## Call:
## lm(formula = leafarea ~ diameter, data = allom)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -112.057  -40.519    4.699   30.344  201.377
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -51.3245    15.9746  -3.213   0.0021 **
## diameter      4.6333     0.3955  11.716   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 60.16 on 61 degrees of freedom
## Multiple R-squared:  0.6923,Adjusted R-squared:  0.6873
## F-statistic: 137.3 on 1 and 61 DF,  p-value: < 2.2e-16
```

```
# Or just the coefficients (intercept and slope):
coef(model)
```

```
## (Intercept)    diameter
##  -51.324479    4.633321
```

As you can see, `lm` uses the formula interface that we discussed earlier (it always has the form y ~ x).

The `summary` function prints a lot of information about the fit. In this case, it shows that the intercept is -51.324, which is the predicted leaf area for a tree with diameter of zero (not very useful in this case).

It also shows a standard error and a t-statistic for this intercept, along with a p-value which shows that the intercept is significantly different from zero. The second line in the coefficients table shows the slope is 4.633, and that this is highly significantly different from zero.

In addition, we have a `Residual standard error` of 60.16, which is an estimate of σ, and an `R-squared` of 0.69 (which is the squared correlation coefficient). Finally, the `F-statistic` says whether the overall fit is significant, which in this case, is the same as the test for β (because in this situation, the F-statistic is simply the square of the t-statistic).

It is straightforward to add the regression line to an existing plot (Fig. 5.3). Simply use `abline` and the `model` object we created previously:

```
plot(leafarea~diameter, data=allom)
abline(model)
```

5.5.1 Diagnostic plots

There are many ways to examine how well a model fits the data, and this step is important in deciding whether the model is appropriate. Most diagnostics are based on the residuals, the difference between the $\hat{y} = \hat{\alpha} + \hat{\beta}x$ fitted values and the actual data points.

If needed, the fitted values and residuals can be extracted using `fitted(model)` and `residuals(model)` respectively.

The two simplest and most useful diagnostic plots are the scale-location plot and a QQ-plot of the residuals. These can be produced with `plot`, but we much prefer two functions from the `car` package, as shown by the following example (Fig. 5.4):

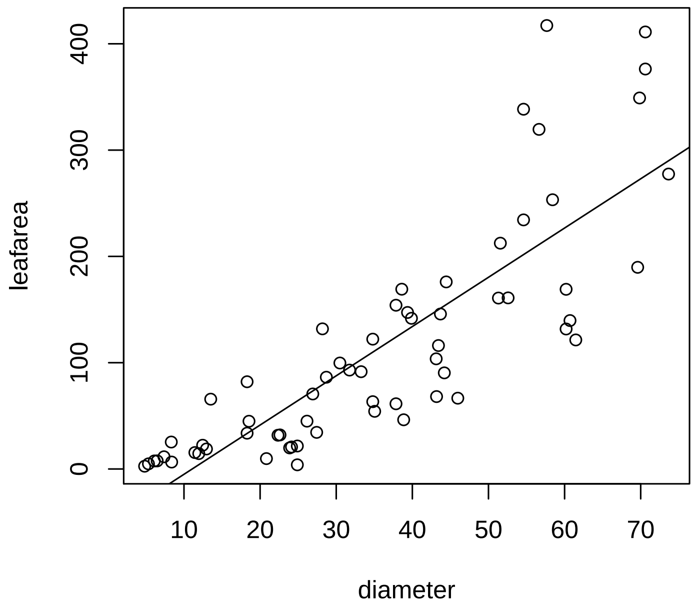


Figure 5.3: The allometry data, with an added regression line.

```
model <- lm(leafarea ~ diameter, data=allom)

library(car)

## Loading required package:  carData

residualPlot(model)
qqPlot(model)
```

```
## [1] 41 47
```

The scale-location plot shows the square root of the *standardized* residuals against the fitted values. In an ideal situation, there should be no structure in this plot. Any curvature indicates that the model is under- or over-fitting, and a general spread-out (or contracting) from left to right indicates non-constant variance ('heteroscedasticity'). The QQ-plot enables us to check for departures from normality. Ideally, the standardized residuals should lie on a straight line.

Some departure from the straight line is to be expected though, even when the underlying distribution is really normal. The `qqPlot` function from the `car` package enhances the standard QQ-plot, by including a confidence interval (Fig 5.5). In this case, there is some evidence of heteroscedasticity, and possibly curvature.

The following code makes the QQ-plot and a plot of the data on a log-log scale (Fig. 5.5).

```
library(car)
qqPlot(model)

## [1] 41 47

plot(leafarea ~ diameter, data=allom, log="xy")
```

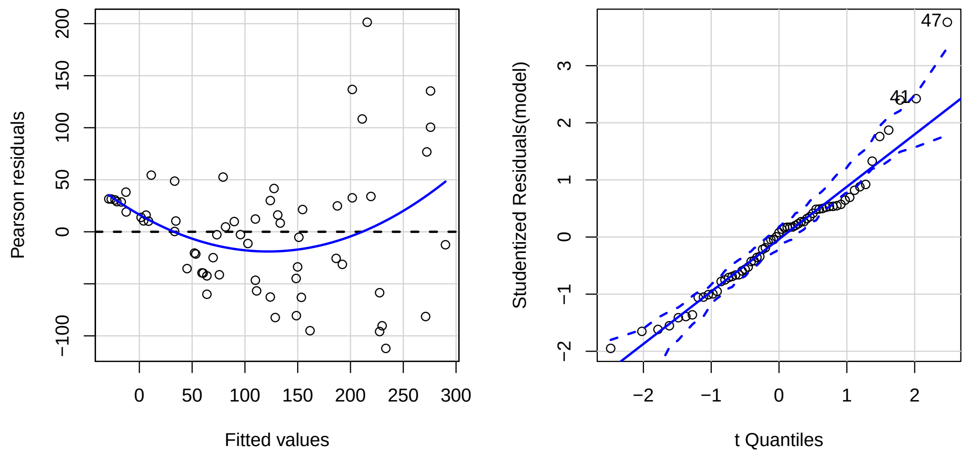


Figure 5.4: Two standard diagnostic plots for a fitted lm object.

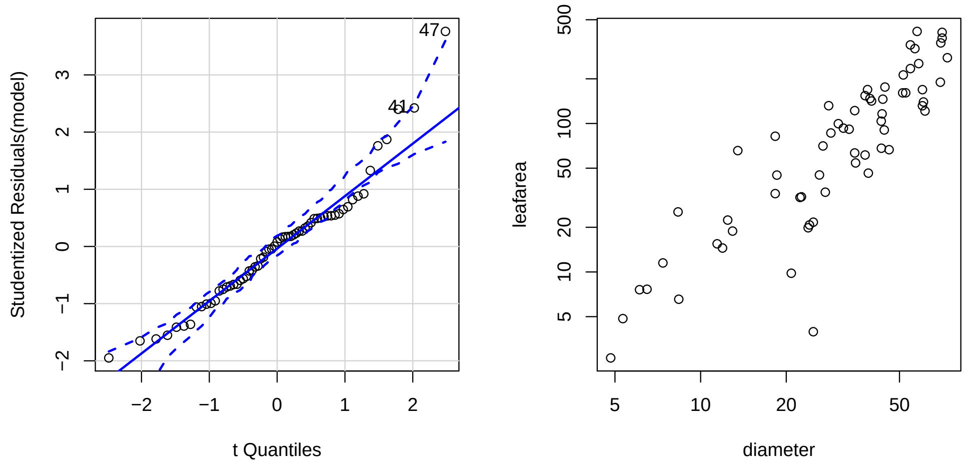


Figure 5.5: A plot of the Allometry data on a log-log scale.

On a log-log scale, it looks like the variance is much more constant, and the relationship is more linear. So, we go ahead and refit the model to log-transformed variables.

As we can see in Fig. 5.5, the diagnostic plots look much better, except for a couple of points at the lower left corner of the QQ-plot. Notice that these outliers have been marked with their row number from the dataframe.

The following code produces Fig. 5.6, including diagnostic plots and a plot of the data with a regression line added. Note that the `abline` function will only work as intended (shown before) on the log-log plot if we use log to the base 10 (`log10`), in the model fit.

```
model_log <- lm(log10(leafarea)~log10(diameter), data=allom)
summary(model_log)

##
## Call:
## lm(formula = log10(leafarea) ~ log10(diameter), data = allom)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.10384 -0.14155  0.02161  0.15434  0.52455
##
## Coefficients:
##                 Estimate Std. Error t value Pr(>|t|)
## (Intercept)      -0.4468     0.1600  -2.793  0.00697 **
## log10(diameter)   1.5387     0.1070  14.385  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.2611 on 61 degrees of freedom
## Multiple R-squared:  0.7723,Adjusted R-squared:  0.7686
## F-statistic: 206.9 on 1 and 61 DF,  p-value: < 2.2e-16

residualPlot(model_log)
qqPlot(model_log)

## [1]  3 32

plot(leafarea~diameter, data=allom, log="xy")
abline(model_log)
```

Try this yourself The residuals of a linear model fit can be extracted with the `residuals` function. For one of the linear models from the above examples, extract the residuals, and make a histogram to help inspect normality of the residuals.

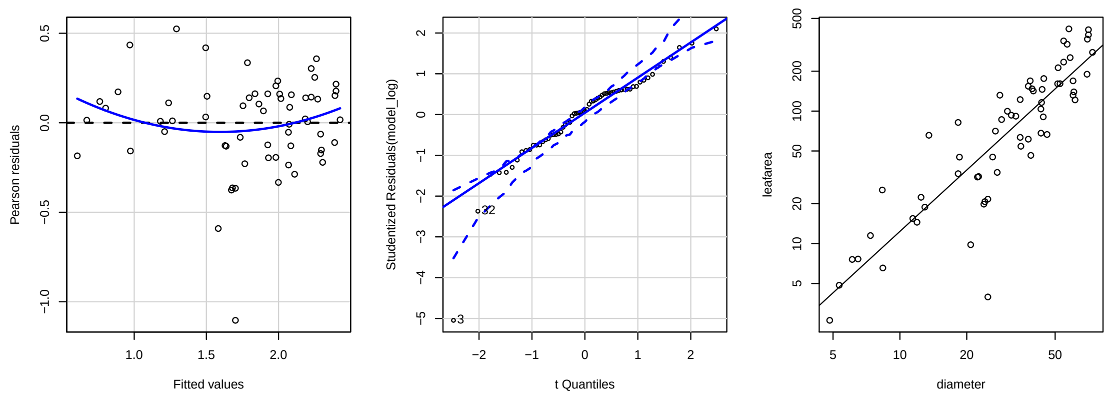


Figure 5.6: Diagnostic plots for the allometry data, refitted on a log-log scale (left panels). The allometry data fitted on a log-log scale, with the regression line (right panel).

5.6 Functions used in this chapter

For functions not listed here, please refer to the index at the end of this book.

Function	What it does	Example use
`abline`	Adds a straight line to an existing plot. For a vertical line, use `v` and an x value. For a horizontal line, use `h` and a y value. To add a sloped line, give a slope (`a`), and an intercept (`b`). Can also be passed a linear model object, see Section 5.5.	`abline(v=5)`
`curve`	Can be used to plot curves, or to add a curve to an existing plot. The first argument can be either a function or an expression that evaluates to a curve. To specify the range of the curve, use `from` and `to`. Use `add=TRUE` to add a curve to an existing plot.	`curve(cos(x), from=0, to=2*pi)`
`kurtosis`	From the `moments` package. Computes Pearson's measure of kurtosis.	
`length`	Returns the total number of elements in its input.	`length(LETTERS)`
`lm`	Fits linear models to data. Requires a formula in the form y ~ x. Also possible to specify `data` to be used and an `na.action`. Returns a linear model object, which can be used for further analysis.	`See Section 5.5`
`log`	Computes natural logarithms (base e) of its input. For common (base 10) logarithms, use `log10`.	`log(3)`
`mean`	Returns the arithmetic average of its input.	`mean(c(1,2,3,10))`
`power.t.test`	Computes the probability that a given t-test will accept the null hypothesis even though it is false. See Section 5.4.1.	`power.t.test(n=35, delta=0.08, sd=0.05, sig.level=0.05)`
`prop.test`	Tests whether an experimental proportion is equal to a hypothesized proportion. Takes `x` (number of successes), `n` (total number of trials) and `p` (the hypothesized proportion.) Specify what kind of test to use (`"two.sided", "less", "greater"`) using `alternative`.	`prop.test(x=60, n=100, p=0.5)`
`qqPlot`	From the `car` package. Given a linear model object, creates a prettified quantile-quantile plot.	`See Section 5.5.1`
`quantile`	Computes the quantiles of given data. Defaults are quartiles, but other sequences can be supplied using `probs`.	`See Section 5.2`
`residuals`	Extracts residuals from a linear model object.	
`sd`	Returns the standard deviation of its input.	`sd(c(99,85,50,87,89))`
`skewness`	From the `moments` package. Computes the skewness of a sample.	

Function	**What it does**	**Example use**
`summary`	Provides a summary of the variables in a dataframe. Statistics are given for numerical variables, and counts are given for factor variables.	`summary(warpbreaks)`
`t.test`	Computes Student's t-test for a given sample of data. It is possible to compare two samples, or a single sample to a hypothesized mean (`mu`).	`See Section 5.3`
`tapply`	Used to summarize data in dataframes. Will be discussed further in Chapter 6.	`See Section 6.2.1`
`var`	Returns the variance of its input.	`var(c(99,85,50,87,89))`
`wilcox.test`	Computes the Wilcoxon test, a nonparametric alternative to t-tests for small or non-normal samples. Can be used to compare two samples, or a single sample to a hypothesized mean (`mu`).	`See Section 5.3`

5.7 Exercises

In these exercises, we use the following colour codes:

■ **Easy**: make sure you complete some of these before moving on. These exercises will follow examples in the text very closely.

◆ **Intermediate**: a bit harder. You will often have to combine functions to solve the exercise in two steps.

▲ **Hard**: difficult exercises! These exercises will require multiple steps, and significant departure from examples in the text.

We suggest you complete these exercises in an **R** markdown file. This will allow you to combine code chunks, graphical output, and written answers in a single, easy-to-read file.

5.7.1 Probabilities

For this exercise, refer to the tables and examples in Section 5.1 (p. 130).

1. ■ For a normal random variable X with mean 5.0, and standard deviation 2.0, find the probability that X is less than 3.0.
2. ■ Find the probability that X is *greater than* 4.5.
3. ■ Find the value K so that $P(X > K) = 0.05$.
4. ■ When tossing a fair coin 10 times, find the probability of seeing no heads (*Hint:* this is a binomial distribution.)
5. ■ Find the probability of seeing exactly 5 heads.
6. ■ Find the probability of seeing more than 7 heads.

5.7.2 Univariate distributions

1. ■ Simulate a sample of 100 random data points from a normal distribution with mean 100 and standard deviation 5, and store the result in a vector.
2. ■ Plot a histogram and a boxplot of the vector you just created (see Section 4.3.3 on p. 92 and Section 4.3.5 on p. 95).
3. ■ Calculate the sample mean and standard deviation.
4. ■ Calculate the median and interquartile range.
5. ■ Using the data above, test the hypothesis that the mean equals 100 (using `t.test`).
6. ■ Test the hypothesis that mean equals 90.
7. ■ Repeat the above two tests using a Wilcoxon signed rank test. Compare the p-values with those from the t-tests you just did.

5.7.3 More t-tests

For this question, use the `pupae` data (see Section A.6 on p. 248).

1. ■ Use the `t.test` function to compare `PupalWeight` by `T_treatment`.
2. ■ Repeat above using a Wilcoxon rank sum test.
3. ■ Run the following code to generate some data:

```
base <- rnorm(20, 20, 5)
x <- base + rnorm(20,0,0.5)
y <- base + rnorm(20,1,0.5)
```

4. ■ Using a two-sample t-test compare the means of `x` and `y`, assume that the variance is equal for the two samples.
5. ■ Repeat the above using a paired t-test. How has the p-value changed?
6. ♦ Which test is most appropriate?

5.7.4 Simple linear regression

For this question, use the `pupae` data (see Section A.6, p. 248). Perform a simple linear regression of `Frass` on `PupalWeight`. Produce and inspect the following:

1. ■ Plots of the data.
2. ■ Summary of the model.
3. ■ Diagnostic plots.
4. ♦ All of the above for a subset of the data, where `Gender` is 0, and `CO2_treatment` is 400.

5.7.5 Quantile Quest

You have already used quantile-quantile (QQ) plots many times, but in this exercise you will get to the bottom of the idea of comparing quantiles of distributions.

As in the previous exercises, we will use the `pupae` data.

1. ■ From the pupae data, extract the PupalWeight and store it as a vector called 'pupweight'. Make a histogram of this vector, noticing that the distribution seems perhaps quite like the normal distribution.
2. ♦ When we say 'quite like the normal distribution', we mean that the overall shape seems similar. Now simulate a histogram like the one above, using `rnorm` with the mean and standard deviation of the pupal weights (i.e. `pupweight`), and the same sample size as well. Plot it repeatedly to get an idea of whether the simulated histogram looks similar often enough.
3. ♦ Of course a visual comparison like that is not good enough, but it is a useful place to start. We can also compare the quantiles as follows. If we calculate the 25% and 75% quantiles of `pupweight`, we are looking for the values below which 25% or 75% of all observations occur. Clearly if two distributions have the same *shape*, their quantiles should be roughly similar. Calculate the 25, 50 and 75% quantiles for `pupweight`, and also calculate them for the normal distribution using `qnorm`. Are they similar?
4. ▲ Now repeat the above exercise, but calculate many quantiles (e.g. from 2.5% to 97.5% with steps of 2.5% or whatever you choose) for both the measured data, and the standard normal distribution. Compare the two with a simple scatter plot, and add a 1:1 line. If you are able to do this, you just made your own QQ-plot (and if not, I suggest you inspect the solutions to this Exercise). *Hint:* use `seq` to make the vector of quantiles, and use it both in `quantile` and `qnorm`.

Save the results of both those as vectors, and plot. As a comparison, use `qqPlot(pupweight, distribution="norm")` (`car` package), make sure to plot the normal quantiles on the X-axis.

Chapter 6

Summarizing, tabulating and merging data

6.1 Summarizing dataframes

There are a few useful functions to print general summaries of a dataframe, to see which variables are included, what types of data they contain, and so on. We already looked at some of these in Section 2.2.

The most basic function is `summary`, which works on many types of objects.

Let's look at the output for the `allom` dataset.

```
summary(allom)

##  species      diameter          height          leafarea
##  PIMO:19   Min.   : 4.83   Min.   : 3.57   Min.   :  2.636
##  PIPO:22   1st Qu.:21.59   1st Qu.:21.26   1st Qu.: 28.581
##  PSME:22   Median :34.80   Median :28.40   Median : 86.351
##            Mean   :35.56   Mean   :26.01   Mean   :113.425
##            3rd Qu.:51.44   3rd Qu.:33.93   3rd Qu.:157.459
##            Max.   :73.66   Max.   :44.99   Max.   :417.209
##    branchmass
##  Min.   :   1.778
##  1st Qu.:  16.878
##  Median :  72.029
##  Mean   : 145.011
##  3rd Qu.: 162.750
##  Max.   :1182.422
```

For each factor variable, the levels are printed (the `species` variable, levels `PIMO`, `PIPO` and `PSME`. For all numeric variables, the minimum, first quantile, median, mean, third quantile, and the maximum values are shown.

To simply see what types of variables your dataframe contains (or, for objects other than dataframes, to summarize sort of object you have), use the `str` function (short for 'structure').

```
str(allom)

## 'data.frame': 63 obs. of  5 variables:
##  $ species   : Factor w/ 3 levels "PIMO","PIPO",..: 3 3 3 3 3 3 3 3 3 3 ...
##  $ diameter  : num  54.6 34.8 24.9 28.7 34.8 ...
##  $ height    : num  27 27.4 21.2 25 30 ...
##  $ leafarea  : num  338.49 122.16 3.96 86.35 63.35 ...
##  $ branchmass: num  410.25 83.65 3.51 73.13 62.39 ...
```

Finally, there are two very useful functions in the `Hmisc` package (recall how to install and load packages from Section 1.10). The first, `describe`, is much like `summary`, but offers slightly more sophisticated statistics.

The second, `contents`, is similar to `str`, but does a very nice job of summarizing the `factor` variables in your dataframe, prints the number of missing variables, the number of rows, and so on.

```
# read data
pupae <- read.csv("pupae.csv")

# Make sure CO2_treatment is a factor (it will be read as a number)
pupae$CO2_treatment <- as.factor(pupae$CO2_treatment)

# Show contents:
library(Hmisc)
contents(pupae)

##
## Data frame:pupae 84 observations and 5 variables    Maximum # NAs:6
##
##
##                Levels Storage NAs
## T_treatment         2 integer   0
## CO2_treatment       2 integer   0
## Gender                integer   6
## PupalWeight            double   0
## Frass                  double   1
##
## +-------------+----------------+
## |Variable     |Levels          |
## +-------------+----------------+
## |T_treatment  |ambient,elevated|
## +-------------+----------------+
## |CO2_treatment|280,400         |
## +-------------+----------------+
```

Here, `storage` refers to the internal storage type of the variable: note that the factor variables are stored as 'integer', and other numbers as 'double' (this refers to the precision of the number).

Try this yourself Use the `describe` function from the `Hmisc` package on the allometry data, and compare the output to `summary`.

6.2 Making summary tables

6.2.1 Summarizing vectors with `tapply()`

If we have the following dataset called `plantdat`,

PlantID	Treatment	Plantbiomass
1	Control	2.0
2	Control	2.2
3	Fertilized	3.2
4	Fertilized	3.6
5	Irrigated	2.8
6	Irrigated	3.0

and execute the command

```
with(plantdat, tapply(Plantbiomass, Treatment, mean))
```

we get the result

Control	Fertilized	Irrigated
2.1	3.4	2.9

Note that the result is a `vector` (elements of a vector can have names, like columns of a dataframe).

If we have the following dataset called `plantdat2`,

Treatment	Species	Plantbiomass
Control	A	2.0
Control	A	2.2
Control	B	2.3
Control	B	2.1
Fertilized	A	3.2
Fertilized	A	3.6
Fertilized	B	3.8
Fertilized	B	4.0
Irrigated	A	2.8
Irrigated	A	3.0
Irrigated	B	2.9
Irrigated	B	3.6

and execute the command

```
with(plantdat2, tapply(Plantbiomass, list(Species, Treatment), mean))
```

we get the result

	Control	Fertilized	Irrigated
A	2.1	3.4	2.90
B	2.2	3.9	3.25

Note that the result here is a `matrix`, where `A` and `B`, the species codes, are the rownames of this matrix.

Often, you want to summarize a variable by the levels of another variable. For example, in the `rain` data (see Section A.4), the `Rain` variable gives daily values, but we might want to calculate annual sums,

```
# Read data
rain <- read.csv("Rain.csv")
```

```
# Annual rain totals.
with(rain, tapply(Rain, Year, FUN=sum))

##   1996   1997   1998   1999   2000   2001   2002   2003   2004   2005
##  717.2  640.4  905.4 1021.3  693.5  791.5  645.9  691.8  709.5  678.2
```

The `tapply` function applies a function (`sum`) to a vector (`Rain`), that is split into chunks depending on another variable (`Year`).

We can also use the `tapply` function on more than one variable at a time. Consider these examples on the `pupae` data.

```
# Read data
pupae <- read.csv("pupae.csv")

# Average pupal weight by CO2 and T treatment:
with(pupae, tapply(PupalWeight, list(CO2_treatment, T_treatment), FUN=mean))

##       ambient elevated
## 280 0.2900000  0.30492
## 400 0.3419565  0.29900

# Further split the averages, by gender of the pupae.
with(pupae, tapply(PupalWeight, list(CO2_treatment, T_treatment, Gender), FUN=mean))

## , , 0
##
##      ambient  elevated
## 280 0.251625 0.2700000
## 400 0.304000 0.2687143
##
## , , 1
##
##       ambient  elevated
## 280 0.3406000 0.3386364
## 400 0.3568333 0.3692857
```

As the examples show, the `tapply` function produces summary tables by one or more factors. The resulting object is either a vector (when using one factor), or a matrix (as in the examples using the pupae data).

The limitations of `tapply` are that you can only summarize one variable at a time, and that the result is not a dataframe.

The main advantage of `tapply` is that we can use it as input to `barplot`, as the following example demonstrates (Fig. 6.1)

```
# Pupal weight by CO2 and Gender. Result is a matrix.
pupm <- with(pupae, tapply(PupalWeight, list(CO2_treatment,Gender),
                           mean, na.rm=TRUE))

# When barplot is provided a matrix, it makes a grouped barplot.
# We specify xlim to make some room for the legend.
barplot(pupm, beside=TRUE, legend.text=TRUE, xlim=c(0,8),
        xlab="Gender", ylab="Pupal weight")
```

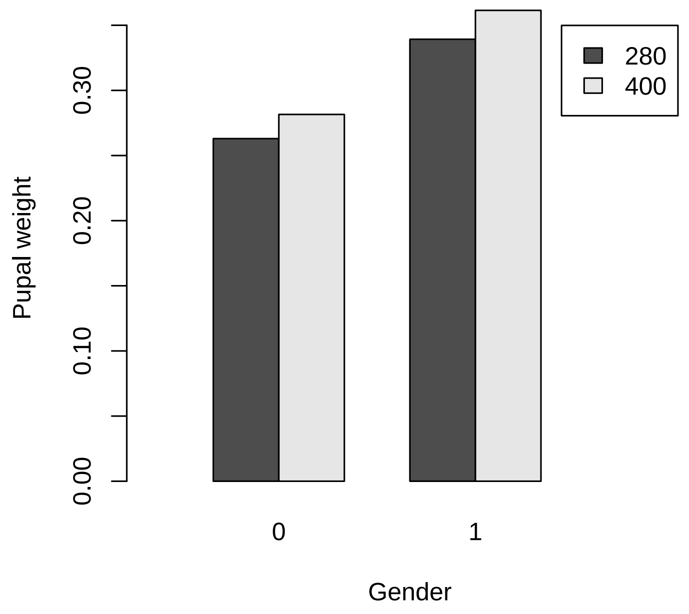


Figure 6.1: A grouped barplot of average pupal weight by CO2 and Gender for the pupae dataset. This is easily achieved via the use of tapply.

6.2.2 Summarizing dataframes with `summaryBy`

If we have the following dataset called `plantdat`,

PlantID	Treatment	Plantbiomass
1	Control	2.0
2	Control	2.2
3	Fertilized	3.2
4	Fertilized	3.6
5	Irrigated	2.8
6	Irrigated	3.0

and execute the command

```
library(doBy)
summaryBy(Plantbiomass ~ treatment, FUN=mean, data=plantdat)
```

we get the result

Treatment	Plantbiomass.mean
Control	2.1
Fertilized	3.4
irrigated	2.9

Note that the result here is a `dataframe`.

If we have the following dataset called `plantdat2`,

Treatment	Species	Plantbiomass
Control	A	2.0
Control	A	2.2
Control	B	2.3
Control	B	2.1
Fertilized	A	3.2
Fertilized	A	3.6
Fertilized	B	3.8
Fertilized	B	4.0
Irrigated	A	2.8
Irrigated	A	3.0
Irrigated	B	2.9
Irrigated	B	3.6

and execute the command

```
summaryBy(Plantbiomass ~ Species + Treatment, FUN=mean, data=dfr)
```

we get the result

Species	Treatment	Plantbiomass.mean
A	Control	2.1
A	Fertilized	3.4
A	Irrigated	2.9
B	Control	2.2
B	Fertilized	3.9
B	Irrigated	3.25

Note that the result here is a `dataframe`.

In practice, it is often useful to make summary tables of multiple variables at once, and to end up with a dataframe. In this book we use `summaryBy`, from the `doBy` package, to achieve this. (We ignore the `aggregate` function in base **R**, because `summaryBy` is much easier to use).

With `summaryBy`, we can generate multiple summaries (mean, standard deviation, etc.) on more than one variable in a dataframe at once. We can use a convenient formula interface for this. It is of the form,

```
summaryBy(Yvar1 + Yvar2 ~ Groupvar1 + Groupvar2, FUN=c(mean,sd), data=mydata)
```

where we summarize the (numeric) variables `Yvar1` and `Yvar2` by all combinations of the (factor) variables `Groupvar1` and `Groupvar2`.

```
# Load the doBy package
library(doBy)

# read pupae data if you have not already
pupae <- read.csv("pupae.csv")

# Get mean and standard deviation of Frass by CO2 and T treatments
summaryBy(Frass ~ CO2_treatment + T_treatment,
          data=pupae, FUN=c(mean,sd))

##   CO2_treatment T_treatment Frass.mean  Frass.sd
## 1           280     ambient         NA        NA
## 2           280    elevated   1.479520 0.2387150
## 3           400     ambient   2.121783 0.4145402
## 4           400    elevated   1.912045 0.3597471
```

```
# Note that there is a missing value. We can specify na.rm=TRUE,
# which will be passed to both mean() and sd(). It works because those
# functions recognize that argument (i.e. na.rm is NOT an argument of
# summaryBy itself!)
summaryBy(Frass ~ CO2_treatment + T_treatment,
          data=pupae, FUN=c(mean,sd), na.rm=TRUE)
```

```
##   CO2_treatment T_treatment Frass.mean  Frass.sd
## 1           280     ambient   1.953923 0.4015635
## 2           280    elevated   1.479520 0.2387150
## 3           400     ambient   2.121783 0.4145402
## 4           400    elevated   1.912045 0.3597471
```

```
# However, if we use a function that does not recognize it, we first have to
# exclude all missing values before making a summary table, like this:
pupae_nona <- pupae[complete.cases(pupae),]

# Get mean and standard deviation for
# the pupae data (Pupal weight and Frass), by CO2 and T treatment.
# Note that length() does not recognize na.rm (see ?length), which is
# why we have excluded any NA from pupae first.
summaryBy(PupalWeight+Frass ~ CO2_treatment + T_treatment,
          data=pupae_nona,
          FUN=c(mean,sd,length))
```

```
##   CO2_treatment T_treatment PupalWeight.mean Frass.mean PupalWeight.sd
## 1           280     ambient        0.2912500   1.957333     0.04895847
## 2           280    elevated        0.3014583   1.473167     0.05921000
## 3           400     ambient        0.3357000   2.103250     0.05886479
## 4           400    elevated        0.3022381   1.931000     0.06602189
##    Frass.sd PupalWeight.length Frass.length
## 1 0.4192227                 12           12
## 2 0.2416805                 24           24
## 3 0.4186310                 20           20
## 4 0.3571969                 21           21
```

You can also use any function that returns a vector of results. In the following example we calculate the 5% and 95% quantiles of all numeric variables in the allometry dataset. To do this, use `.` for the left-hand side of the formula.

```
# . ~ species means 'all numeric variables by species'.
# Extra arguments to the function used (in this case quantile) can be set here as well,
# they will be passed to that function (see ?quantile).
summaryBy(. ~ species, data=allom, FUN=quantile, probs=c(0.05, 0.95))
```

```
##   species diameter.5% diameter.95% height.5% height.95% leafarea.5%
## 1    PIMO      8.1900      70.9150  5.373000     44.675    7.540916
## 2    PIPO     11.4555      69.1300  5.903499     39.192   10.040843
## 3    PSME      6.1635      56.5385  5.276000     32.602    4.988747
##   leafarea.95% branchmass.5% branchmass.95%
## 1     380.3984      4.283342       333.6408
## 2     250.1295      7.591966       655.1097
## 3     337.5367      3.515638       403.7902
```

Working example : Calculate daily means and totals

Let's look at a more advanced example using weather data collected at the Hawkesbury Forest Experiment in 2008 (see Section A.10). The data given are in half-hourly time steps. It is a reasonable request to provide data as daily averages (for temperature) and daily sums (for precipitation).

The following code produces a daily weather dataset, and Fig. 6.2.

```
# Read data
hfemet <- read.csv("HFEmet2008.csv")

# This is a fairly large dataset:
nrow(hfemet)

## [1] 17568

# So let's only look at the first few rows:
head(hfemet)

##         DateTime  Tair AirPress   RH        VPD PAR Rain wind winddirection
## 1 1/1/2008 0:00 16.54 101.7967 93.3 0.12656434   0    0    0         152.6
## 2 1/1/2008 0:30 16.45 101.7700 93.9 0.11457229   0    0    0         166.7
## 3 1/1/2008 1:00 16.44 101.7300 94.2 0.10886828   0    0    0         180.2
## 4 1/1/2008 1:30 16.41 101.7000 94.5 0.10304019   0    0    0         180.2
## 5 1/1/2008 2:00 16.38 101.7000 94.7 0.09910379   0    0    0         180.2
## 6 1/1/2008 2:30 16.23 101.7000 94.7 0.09816111   0    0    0         180.2

# Read the date-time
library(lubridate)
hfemet$DateTime <- mdy_hm(hfemet$DateTime)

# Add a new variable 'Date', a daily date variable
hfemet$Date <- as.Date(hfemet$DateTime)

# First aggregate some of the variables into daily means:
library(doBy)
hfemetAgg <- summaryBy(PAR + VPD + Tair ~ Date, data=hfemet, FUN=mean)
# (look at head(hfemetAgg) to check this went OK!)

#--- Now get daily total Rainfall:
hfemetSums <- summaryBy(Rain ~ Date, data=hfemet, FUN=sum)

# To finish things off, let's make a plot of daily total rainfall:
# (type='h' makes a sort of narrow bar plot)
plot(Rain.sum ~ Date, data=hfemetSums, type='h', ylab=expression(Rain~(mm~day^-1)))
```

6.2.3 Tables of counts

It is often useful to count the number of observations by one or more multiple factors. One option is to use `tapply` or `summaryBy` in combination with the `length` function. A much better alternative is to use the `xtabs` and `ftable` functions, in addition to the simple use of `table`.

Consider these examples using the Titanic data (see Section A.12).

```
# Read titanic data
titanic <- read.table("titanic.txt", header=TRUE)
```

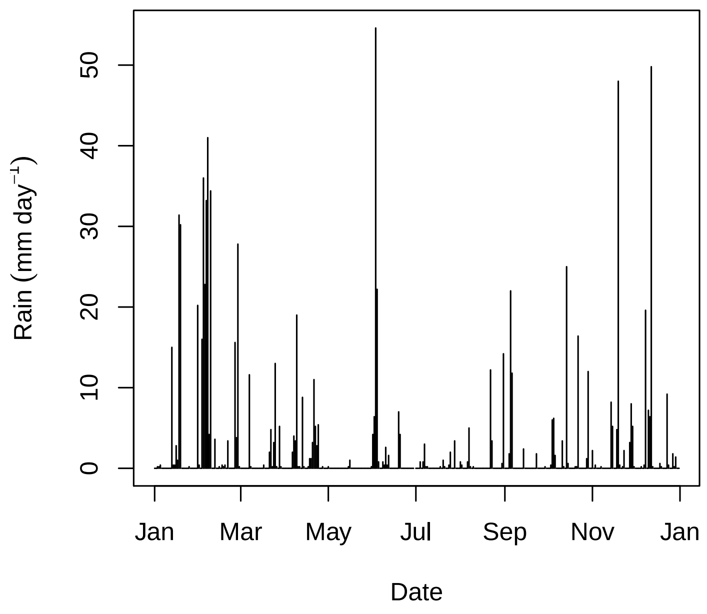


Figure 6.2: Daily rainfall at the HFE in 2008

```
# Count observations by passenger class
table(titanic$PClass)

##
## 1st 2nd 3rd
## 322 280 711

# With more grouping variables, it is more convenient to use xtabs.
# Count observations by combinations of passenger class, sex, and whether they survived:
xtabs( ~ PClass + Sex + Survived, data=titanic)

## , , Survived = 0
##
##       Sex
## PClass female male
##    1st      9  120
##    2nd     13  148
##    3rd    132  441
##
## , , Survived = 1
##
##       Sex
## PClass female male
##    1st    134   59
##    2nd     94   25
##    3rd     80   58

# The previous output is hard to read, consider using ftable on the result:
```

```
ftable(xtabs( ~ PClass + Sex + Survived, data=titanic))

##                Survived   0   1
## PClass Sex
## 1st    female             9 134
##        male             120  59
## 2nd    female            13  94
##        male             148  25
## 3rd    female           132  80
##        male             441  58
```

6.2.4 Adding simple summary variables to dataframes

We saw how `tapply` can make simple tables of averages (or totals, or other functions) of some variable by the levels of one or more factor variables. The result of `tapply` is typically a vector with a length equal to the number of levels of the factor you summarized by (see examples in Section 6.2.1).

What if you want the result of a summary that is the length of the original vector? One option is to aggregate the dataframe with `summaryBy`, and `merge` it back to the original dataframe (see Section 6.3.1). But we can use a shortcut.

Consider the `allometry` dataset, which includes tree height for three species. Suppose you want to add a new variable 'MaxHeight', that is the maximum tree height observed per species. We can use `ave` to achieve this:

```
# Read data
allom <- read.csv("Allometry.csv")

# Maximum tree height by species:
allom$MaxHeight <- ave(allom$height, allom$species, FUN=max)

# Look at first few rows (or just type allom to see whole dataset)
head(allom)

##   species diameter height   leafarea branchmass MaxHeight
## 1    PSME    54.61  27.04 338.485622  410.24638      33.3
## 2    PSME    34.80  27.42 122.157864   83.65030      33.3
## 3    PSME    24.89  21.23   3.958274    3.51270      33.3
## 4    PSME    28.70  24.96  86.350653   73.13027      33.3
## 5    PSME    34.80  29.99  63.350906   62.39044      33.3
## 6    PSME    37.85  28.07  61.372765   53.86594      33.3
```

Note that you can use any function in place of `max`, as long as that function can take a vector as an argument, and returns a single number.

Try this yourself If you want results similar to `ave`, you can use `summaryBy` with the argument `full.dimension=TRUE`. Try `summaryBy` on the `pupae` dataset with that argument set, and compare the result to `full.dimension=FALSE`, which is the default.

6.2.5 Reordering factor levels based on a summary variable

It is often useful to tabulate your data in a meaningful order. We saw that, when using `summaryBy`, `tapply` or similar functions, that the results are always in the order of your factor levels. Recall that the default order is alphabetical. This is rarely what you want.

You can reorder the factor levels by some summary variable. For example,

```
# Reorder factor levels for 'Manufacturer' in the cereal data
# by the mean amount of sodium.

# Read data, show default (alphabetical) levels:
cereal <- read.csv("cereals.csv")
levels(cereal$Manufacturer)

## [1] "A" "G" "K" "N" "P" "Q" "R"

# Now reorder:
cereal$Manufacturer <- with(cereal, reorder(Manufacturer, sodium, median, na.rm=TRUE))
levels(cereal$Manufacturer)

## [1] "A" "N" "Q" "P" "K" "G" "R"

# And tables are now printed in order:
with(cereal, tapply(sodium, Manufacturer, median))

##     A     N     Q     P     K     G     R
##   0.0   7.5  75.0 160.0 170.0 200.0 200.0
```

This trick comes in handy when making barplots; it is customary to plot them in ascending order if there is no specific order to the factor levels, as in this example.

The following code produces Fig. 6.3.

```
coweeta <- read.csv("coweeta.csv")
coweeta$species <- with(coweeta, reorder(species, height, mean, na.rm=TRUE))

library(doBy)
coweeta_agg <- summaryBy(height ~ species, data=coweeta, FUN=c(mean,sd))

library(gplots)

# This par setting makes the x-axis labels vertical, so they don't overlap.
par(las=2)
with(coweeta_agg, barplot2(height.mean, names.arg=species,
                           space=0.3, col="red",plot.grid=TRUE,
                           ylab="Height (m)",
                           plot.ci=TRUE,
                           ci.l=height.mean - height.sd,
                           ci.u=height.mean + height.sd))
```

Try this yourself The above example orders the factor levels by increasing median sodium levels. Try reversing the factor levels, using the following code after `reorder`.

`coweeta$species <- factor(coweeta$species, levels=rev(levels(coweeta$species)))`

Here we used `rev` to reverse the levels.

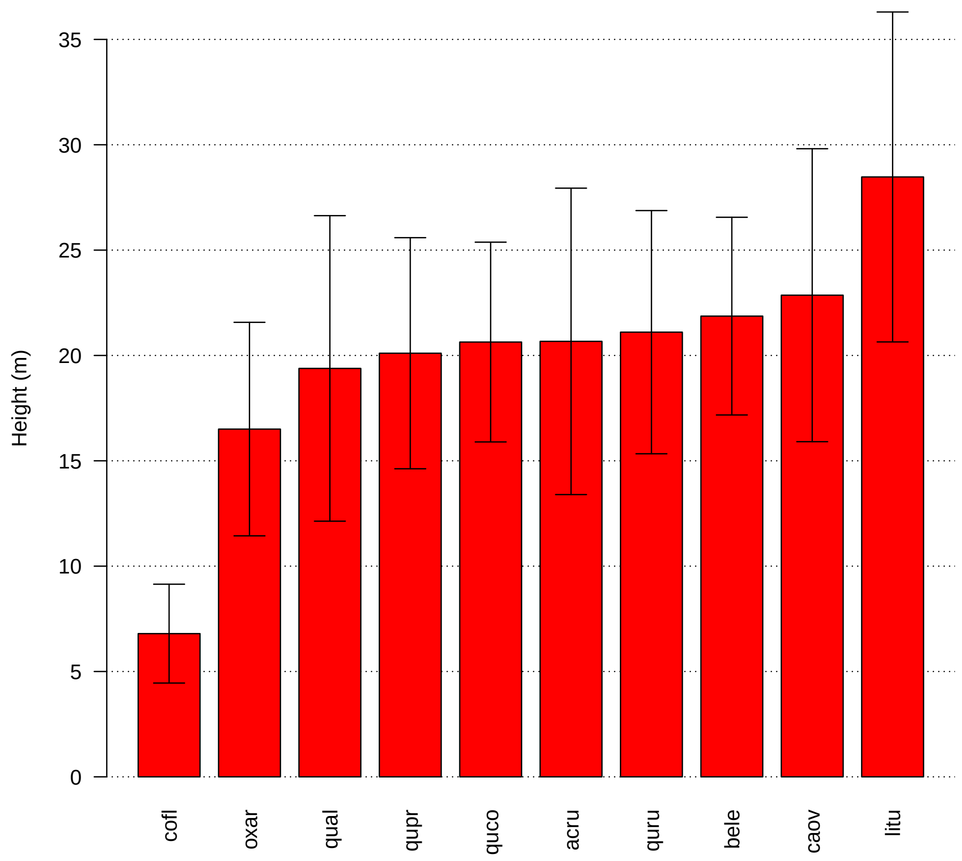


Figure 6.3: An ordered barplot for the coweeta tree data (error bars are 1 SD).

6.3 Combining dataframes

6.3.1 Merging dataframes

If we have the following dataset called `plantdat`,

PlantID	Treatment	Plantbiomass
1	Control	2.0
2	Control	2.2
3	Fertilized	3.2
4	Fertilized	3.6
5	Irrigated	2.8
6	Irrigated	3.0

and we have another dataset, that includes the same `PlantID` variable (but is not necessarily ordered, nor does it have to include values for every plant):

PlantID	Leafnitrogen
1	1.6
5	1.8
4	2.4
3	2.8
2	1.8

and execute the command

```
merge(plantdat, leafnitrogendata, by="PlantID")
```

we get the result

PlantID	Treatment	Plantbiomass	Leafnitrogen
1	Control	2.0	1.6
2	Control	2.2	1.9
3	Fertilized	3.2	2.8
4	Fertilized	3.6	2.4
5	Irrigated	2.8	1.8
6	Irrigated	3.0	NA

Note the missing value (`NA`) for the plant for which no leaf nitrogen data was available.

In many problems, you do not have a single dataset that contains all the measurements you are interested in – unlike most of the example datasets in this tutorial. Suppose you have two datasets that you would like to combine, or `merge`. This is straightforward in **R**, but there are some pitfalls.

Let's start with a common situation when you need to combine two datasets that have a different number of rows.

```
# Two dataframes
data1 <- data.frame(unit=c("x","x","x","y","z","z"),Time=c(1,2,3,1,1,2))
data2 <- data.frame(unit=c("y","z","x"), height=c(3.4,5.6,1.2))

# Look at the dataframes
data1

##   unit Time
## 1    x    1
## 2    x    2
## 3    x    3
## 4    y    1
## 5    z    1
## 6    z    2

data2

##   unit height
## 1    y    3.4
## 2    z    5.6
## 3    x    1.2

# Merge dataframes:
combdata <- merge(data1, data2, by="unit")

# Combined data
combdata

##   unit Time height
## 1    x    1    1.2
## 2    x    2    1.2
## 3    x    3    1.2
## 4    y    1    3.4
## 5    z    1    5.6
## 6    z    2    5.6
```

Sometimes, the variable you are merging with has a different name in either dataframe. In that case, you can either rename the variable before merging, or use the following option:

```
merge(data1, data2, by.x="unit", by.y="item")
```

Where `data1` has a variable called 'unit', and `data2` has a variable called 'item'.

Other times you need to merge two dataframes with multiple key variables. Consider this example, where two dataframes have measurements on the same units at some of the the same times, but on different variables:

```
# Two dataframes
data1 <- data.frame(unit=c("x","x","x","y","y","y","z","z","z"),
Time=c(1,2,3,1,2,3,1,2,3),
Weight=c(3.1,5.2,6.9,2.2,5.1,7.5,3.5,6.1,8.0))
data2 <- data.frame(unit=c("x","x","y","y","z","z"),
Time=c(1,2,2,3,1,3),
Height=c(12.1,24.4,18.0,30.8,10.4,32.9))

# Look at the dataframes
data1

##   unit Time Weight
## 1    x    1    3.1
## 2    x    2    5.2
## 3    x    3    6.9
## 4    y    1    2.2
## 5    y    2    5.1
## 6    y    3    7.5
## 7    z    1    3.5
## 8    z    2    6.1
## 9    z    3    8.0

data2

##   unit Time Height
## 1    x    1   12.1
## 2    x    2   24.4
## 3    y    2   18.0
## 4    y    3   30.8
## 5    z    1   10.4
## 6    z    3   32.9

# Merge dataframes:
combdata <- merge(data1, data2, by=c("unit","Time"))

# By default, only those times appear in the dataset that have measurements
# for both Weight (data1) and Height (data2)
combdata

##   unit Time Weight Height
## 1    x    1    3.1   12.1
## 2    x    2    5.2   24.4
## 3    y    2    5.1   18.0
## 4    y    3    7.5   30.8
## 5    z    1    3.5   10.4
## 6    z    3    8.0   32.9

# To include all data, use this command. This produces missing values for some times:
merge(data1, data2, by=c("unit","Time"), all=TRUE)

##   unit Time Weight Height
## 1    x    1    3.1   12.1
## 2    x    2    5.2   24.4
## 3    x    3    6.9     NA
## 4    y    1    2.2     NA
## 5    y    2    5.1   18.0
## 6    y    3    7.5   30.8
## 7    z    1    3.5   10.4
## 8    z    2    6.1     NA
## 9    z    3    8.0   32.9

# Compare this result with 'combdata' above!
```

Merging multiple datasets

Consider the cereal dataset (Section A.7), which gives measurements of all sorts of contents of cereals. Suppose the measurements for 'protein', 'vitamins' and 'sugars' were all produced by different labs, and each lab sends you a separate dataset. To make things worse, some measurements for sugars and vitamins are missing, because samples were lost in those labs.

How to put things together?

```
# Read the three datasets given to you from the three different labs:
cereal1 <- read.csv("cereal1.csv")
cereal2 <- read.csv("cereal2.csv")
cereal3 <- read.csv("cereal3.csv")

# Look at the datasets:
cereal1

##                    Cereal.name protein
## 1             Frosted_Flakes       1
## 2                 Product_19       3
## 3              Count_Chocula       1
## 4                 Wheat_Chex       3
## 5                 Honey-comb       1
## 6  Shredded_Wheat_spoon_size       3
## 7        Mueslix_Crispy_Blend       3
## 8           Grape_Nuts_Flakes       3
## 9    Strawberry_Fruit_Wheats       2
## 10                   Cheerios       6
```

```
cereal2
```

```
##                cerealbrand vitamins
## 1                Product_19      100
## 2             Count_Chocula       25
## 3                Wheat_Chex       25
## 4                Honey-comb       25
## 5 Shredded_Wheat_spoon_size        0
## 6      Mueslix_Crispy_Blend       25
## 7         Grape_Nuts_Flakes       25
## 8                  Cheerios       25
```

```
cereal3
```

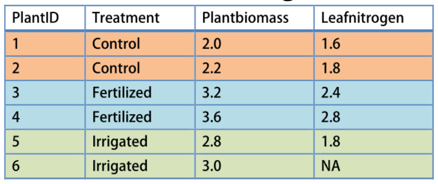
```
##               cerealname sugars
## 1          Frosted_Flakes     11
## 2              Product_19      3
## 3    Mueslix_Crispy_Blend     13
## 4       Grape_Nuts_Flakes      5
## 5 Strawberry_Fruit_Wheats      5
## 6                Cheerios      1
```

```
# Note that the number of rows is different between the datasets,
# and even the index name ('Cereal.name') differs between the datasets.

# To merge them all together, use merge() twice, like this.
cerealdata <- merge(cereal1, cereal2,
                    by.x="Cereal.name",
                    by.y="cerealbrand", all.x=TRUE)
# NOTE: all.x=TRUE specifies to keep all rows in cereal1 that do not exist in cereal2.

# Then merge again:
cerealdata <- merge(cerealdata, cereal3,
                    by.x="Cereal.name",
                    by.y="cerealname", all.x=TRUE)

# And double check the final result
cerealdata
```

```
##                Cereal.name protein vitamins sugars
## 1                 Cheerios       6       25      1
## 2            Count_Chocula       1       25     NA
## 3           Frosted_Flakes       1       NA     11
## 4        Grape_Nuts_Flakes       3       25      5
## 5               Honey-comb       1       25     NA
## 6     Mueslix_Crispy_Blend       3       25     13
## 7                    Product_19       3      100      3
## 8  Shredded_Wheat_spoon_size       3        0     NA
## 9    Strawberry_Fruit_Wheats       2       NA      5
## 10                    Wheat_Chex       3       25     NA
# Note that missing values (NA) have been inserted where some data was not available.
```

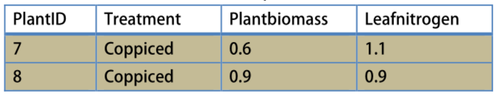
6.3.2 Row-binding dataframes

If we have the following dataset called `plantdat`,



PlantID	Treatment	Plantbiomass	Leafnitrogen
1	Control	2.0	1.6
2	Control	2.2	1.8
3	Fertilized	3.2	2.4
4	Fertilized	3.6	2.8
5	Irrigated	2.8	1.8
6	Irrigated	3.0	NA

and we have another dataset (`plantdatmore`), *with exactly the same columns* (including the names and order of the columns),

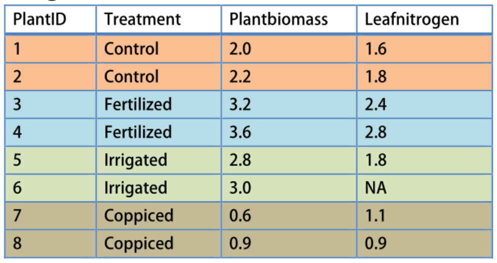


PlantID	Treatment	Plantbiomass	Leafnitrogen
7	Coppiced	0.6	1.1
8	Coppiced	0.9	0.9

and execute the command

```
rbind(plantdat, plantdatmore)
```

we get the result



PlantID	Treatment	Plantbiomass	Leafnitrogen
1	Control	2.0	1.6
2	Control	2.2	1.8
3	Fertilized	3.2	2.4
4	Fertilized	3.6	2.8
5	Irrigated	2.8	1.8
6	Irrigated	3.0	NA
7	Coppiced	0.6	1.1
8	Coppiced	0.9	0.9

Using `merge`, we are able to glue dataframes together side-by-side based on one or more 'index' variables. Sometimes you have multiple datasets that can be glued together top-to-bottom, for example when you have multiple very similar dataframes. We can use the `rbind` function, like so:

```
# Some fake data
mydata1 <- data.frame(var1=1:3, var2=5:7)
mydata2 <- data.frame(var1=4:6, var2=8:10)

# The dataframes have the same column names, in the same order:
mydata1
##   var1 var2
## 1    1    5
## 2    2    6
## 3    3    7
mydata2
##   var1 var2
## 1    4    8
## 2    5    9
## 3    6   10

# So we can use rbind to row-bind them together:
rbind(mydata1, mydata2)

##   var1 var2
## 1    1    5
## 2    2    6
## 3    3    7
## 4    4    8
## 5    5    9
## 6    6   10
```

Let's look at the above `rbind` example again but with a modification where some observations are duplicated between dataframes. This might happen, for example, when working with files containing time-series data and where there is some overlap between the two datasets. The `union` function from the `dplyr` package only returns unique observations:

```
# Some fake data
mydata1 <- data.frame(var1=1:3, var2=5:7)
mydata2 <- data.frame(var1=2:4, var2=6:8)

# The dataframes have the same column names, in the same order:
mydata1

##   var1 var2
## 1    1    5
## 2    2    6
## 3    3    7

mydata2

##   var1 var2
## 1    2    6
## 2    3    7
## 3    4    8

# 'rbind' leads to duplicate observations, 'union' removes these:
library(dplyr)
union(mydata1, mydata2)

##   var1 var2
## 1    1    5
## 2    2    6
## 3    3    7
## 4    4    8

rbind(mydata1, mydata2)

##   var1 var2
## 1    1    5
## 2    2    6
## 3    3    7
## 4    2    6
## 5    3    7
## 6    4    8
```

Sometimes, you want to `rbind` dataframes together but the column names do not exactly match. One option is to first process the dataframes so that they do match (using subscripting). Or, just use the `bind_rows` function from the `dplyr` package. Look at this example where we have two dataframes that have only one column in common, but we want to keep all the columns (and fill with NA where necessary),

```
# Some fake data
mydata1 <- data.frame(index=c("A","B","C"), var1=5:7)
mydata2 <- data.frame(var1=8:10, species=c("one","two","three"))

# smartbind the dataframes together
bind_rows(mydata1, mydata2)
```

```
##   index var1 species
## 1     A    5    <NA>
## 2     B    6    <NA>
## 3     C    7    <NA>
## 4  <NA>    8     one
## 5  <NA>    9     two
## 6  <NA>   10   three
```

Note: an equivalent function to bind dataframes side-by-side is `cbind`, which can be used instead of `merge` when no index variables are present. However, in this book, the use of `cbind` is discouraged for dataframes (and we don't discuss matrices), as it can lead to problems that are difficult to fix.

Try this yourself The `dplyr` package contains a number of functions for merging dataframes that may be more intuitive to you than `merge`. For example, `left_join(mydata1, mydata2)` keeps all rows from `mydata1` after merging the two dataframes, while `right_join(mydata1, mydata2)` keeps all rows from `mydata2`. Read more about the `dplyr` functions for joining tables at `?dplyr::join`, and try to recreate the examples above using these functions instead of `merge`.

6.4 Exporting summary tables

To export summary tables generated with `aggregate`, `tapply` or `table` to text files, you can use `write.csv` or `write.table` just like you do for exporting dataframes (see Section 2.4).

For example,

```
cereals <- read.csv("cereals.csv")

# Save a table as an object
mytable <- with(cereals, table(Manufacturer, Cold.or.Hot))

# Write to file
write.csv(mytable, "cerealtable.csv")
```

6.4.1 Inserting tables into documents using R markdown

How do we insert tables from **R** into Microsoft Word (for example, results from `tapply` or `aggregate`)? The best way is, of course, to use R markdown. There are now several packages that make it easy to produce tables in markdown, and from there, markdown easily converts them into Word. First, we'll need some suitable data. We'll start by making a summary table of the pupae dataset we loaded earlier in the chapter.

```
# Load the doBy package
library(doBy)

# read pupae data if you have not already
pupae <- read.csv("pupae.csv")

# Make a table of means and SD of the pupae data
puptab <- summaryBy(Frass + PupalWeight ~ CO2_treatment + T_treatment,
                         FUN=c(mean,sd), data=pupae, na.rm=TRUE)

# It is more convenient to reorder the dataframe to have sd and mean
# together.
puptab <- puptab[,c("CO2_treatment","T_treatment",
                    "Frass.mean","Frass.sd",
                    "PupalWeight.mean","PupalWeight.sd")]

# Give the columns short, easy to type names
names(puptab) <- c("CO2","T","Frass","SD.1","PupalWeight","SD.2")

# Convert temperature, which is a factor, to a character variable
# (Factors don't work well with the data-reshaping that takes place in pixiedust)
puptab$T <- as.character(puptab$T)
```

To insert any of the following code into a markdown document, use the following around each code chunk:

~~~
```{r echo=TRUE, results="asis"}
```
~~~

Tables with kable

We'll start with the `kable` function from the `knitr` package. The `knitr` package provides the functionality behind RStudio's markdown interface, so, if you have been using markdown, you don't need to install anything extra to use `kable`. This function is simple, robust and offers the ability to easily add captions in Word.

```
library(knitr)
kable(puptab, caption="Table 1. Summary stats for the pupae data.")
```

If you run this in the console, you'll see a markdown table. Knitting your markdown document as "Word" will result in a Word document with a table and a nice caption. You can rename the table columns using the `col.names` argument (just be sure to get them in the same order!)

As you'll see in the next example, you can also use markdown formatting within the table – note the addition of a subscript to CO_2.

```
kable(puptab, caption="Table 1. Summary stats for the pupae data.",
      col.names=c("CO~ 2~ Treatment","Temperature","Frass","SD","Pupal Weight", "SD"))
```

Other things that are easily controlled using `kable` are the number of digits shown in numeric columns (`digits`), row names (`row.names`), and the alignment of each column (`align`).

```
kable(puptab, caption="Table 1. Summary stats for the pupae data.",
      col.names=c("CO~ 2~ Treatment","Temperature","Frass","SD","Pupal Weight", "SD"),
      digits=1,
      align=c('l','c','r')) # Values will be recycled to match number of columns
```

Tables with pander

Other options for creating tables in markdown include the function `pander` from the `pander` package. This function is designed to translate a wide range of input into markdown format. When you give it input it can recognize as a table, it will output a markdown table. Like `kable`, it can take a caption argument.

```r
library(pander)
pander(puptab, caption = "Table 1. Summary stats for the pupae data")
```

One of the advantages of `pander` is that it can accept a broader range of input than `kable`. For instance, it can make tables out of the results produced by linear models (something we'll discuss in Chapter 7.) Formatting options available in `pander` include options for rounding, cell alignment (including dynamic alignments based on cell values), and adding emphasis, such as italic or bold formatting (which can also be set dynamically).

```r
library(pander)

# Cell and row emphasis are set before calling pander,
# using functions that start with 'emphasize.':
# emphasize.strong.rows,  emphasize.strong.cols,  emphasize.strong.cells,
# emphasize.italics.rows, emphasize.italics.cols, emphasize.italics.cells
emphasize.strong.cols(1)
emphasize.italics.cells(which(puptab == "elevated", arr.ind = TRUE))
pander(puptab,
       caption = "Table 1. Summary stats for the pupae data, with cell formatting",
       # for <justify>, length must match number of columns
       justify = c('left', 'center', 'right','right','right','right'),
       round=3)
```

There are also options to set what type of table you would like (e.g., `grid`, `simple`, or `multiline`), cell width, line breaking within cells, hyphenation, what kind of decimal markers to use, and how replace missing values. For more details on these, see the pander vignettes, or `?pandoc.table.return`.

Tables with pixiedust

A third option for formatting tables is the imaginatively-named `pixiedust` package. The syntax used by `pixiedust` is a bit different the other packages we've discussed so far. It starts with the function `dust`, then sends the results of addition options (called 'sprinkles') to `dust` using the symbols `%>%`, which are known as the pipe operator. To make a markdown table, use the 'markdown' sprinkle:

```r
# pixiedust requires broom for table formatting
library(broom)
library(pixiedust)

dust(puptab) %>%
  sprinkle_print_method("markdown")
```

There are a range of formatting options similar to those available in `pander`, including bold, italic, a missing data string, and rounding. Each of these is added as a new sprinkle. Cells are specified by row and column:

```r
dust(puptab) %>%
  sprinkle_print_method("markdown")  %>%
  sprinkle(rows = c(2, 4), bold = TRUE) %>%
  sprinkle(rows = c(3, 4), cols=c(1, 1), italic = TRUE) %>%
  sprinkle(round = 1)
```

It is also possible to add captions, change column names, and replace the contents of a given cell during formatting.

```
# Captions are added to the dust() function
dust(puptab, caption="Table 1. Summary stats for the pupae data,
     formatted with pixiedust") %>%
  # Note that identical column names are not allowed
  sprinkle_colnames("CO2 Treatment","Temperature","Frass","Frass SD",
                    "Pupal Weight", "Weight SD") %>%
  # Replacement must have the same length as what it replaces
  sprinkle(cols = 1, replace =
             c("ambient", "ambient", "elevated", "elevated")) %>%
  sprinkle_print_method("markdown")
```

If you use latex or html, you will want to look into this package further, as it has a number of special 'sprinkles' for these formats.

Further reading If you want to know more about `kable`, the help page has a thorough explanation. Type `?kable` at the command line. There is a good introduction to `pander` called "Rendering tables with pandoc.table" at https://cran.r-project.org/web/packages/pander/vignettes/pandoc_table.html. (Don't be confused by the name; `pandoc.table` is the table function hidden under the hood of `pander`.) Likewise, `pixiedust` has a good introductory vignette called "Creating magic with pixiedust" at https://cran.r-project.org/web/packages/pixiedust/vignettes/pixiedust.html. For a list of all of the `pixiedust` 'sprinkles' available, see https://cran.r-project.org/web/packages/pixiedust/vignettes/sprinkles.html.

6.5 Exercises

In these exercises, we use the following colour codes:

🟩 **Easy**: make sure you complete some of these before moving on. These exercises will follow examples in the text very closely.

♦ **Intermediate**: a bit harder. You will often have to combine functions to solve the exercise in two steps.

▲ **Hard**: difficult exercises! These exercises will require multiple steps, and significant departure from examples in the text.

We suggest you complete these exercises in an **R** markdown file. This will allow you to combine code chunks, graphical output, and written answers in a single, easy-to-read file.

6.5.1 Summarizing the cereal data

1. 🟩 Read the cereal data, and produce quick summaries using `str`, `summary`, `contents` and `describe` (recall that the last two are in the `Hmisc` package). Interpret the results.
2. 🟩 Find the average sodium, fiber and carbohydrate contents by `Manufacturer`.
3. 🟩 Add a new variable 'SodiumClass', which is 'high' when sodium >150 and 'low' otherwise. Make sure the new variable is a factor. Look at the examples in Section 3.2 to recall how to do this. Now, find the average, minimum and maximum sugar content for 'low' and 'high' sodium. *Hint:* make sure to use `na.rm=TRUE`, because the dataset contains missing values.
4. ♦ Find the maximum sugar content by Manufacturer and sodiumClass, using `tapply`. Inspect the result and notice there are missing values. Try to use `na.rm=TRUE` as an additional argument to `tapply`, only to find out that the values are still missing. Finally, use `xtabs` (see Section 6.2.3, p. 159) to count the number of observations by the two factors to find out we have missing values in the `tapply` result.
5. 🟩 Repeat the previous question with `summaryBy`. Compare the results.
6. 🟩 Count the number of observations by Manufacturer and whether the cereal is 'hot' or 'cold', using `xtabs` (see Section 6.2.3, p. 159).

6.5.2 Words and the weather

1. 🟩 Using the 'Age and memory' dataset (first read Section A.11 on p. 251 how to read this dataset), find the mean and maximum number of words recalled by 'Older' and 'Younger' age classes.
2. ♦ Using the HFE weather dataset (see Section A.10, p. 251), find the mean air temperature by month. To do this, first add the month variable as shown in Section 3.6.2 (p. 73).

6.5.3 Merge new data onto the pupae data

1. 🟩 First read the pupae data (see Section A.6, p. 248). Also read this short dataset, which gives a label 'roomnumber' for each CO_2 treatment.

```
|CO2_treatment |Roomnumber   |
|-------------:|-----------:|
|280            |1            |
|400            |2            |
```

To read this dataset, consider the `data.frame` function described in Section 2.1.2 (p. 40).

2. ■ Merge the short dataset onto the pupae data. Check the result.

6.5.4 Merging multiple datasets

Read Section 6.3.1 (p. 166) before trying this exercise.

First, run the following code to construct three dataframes that we will attempt to merge together.

```
dataset1 <- data.frame(unit=letters[1:9], treatment=rep(LETTERS[1:3],each=3),
                       Damage=runif(9,50,100))
unitweight <- data.frame(unit=letters[c(1,2,4,6,8,9)], Weight = rnorm(6,100,0.3))
treatlocation <- data.frame(treatment=LETTERS[1:3], Glasshouse=c("G1","G2","G3"))
```

1. ⧫ Merge the three datasets together, to end up with one dataframe that has the columns 'unit', 'treatment', 'Glasshouse', 'Damage' and 'Weight'. Some units do not have measurements of `Weight`. Merge the datasets in two ways to either include or exclude the units without `Weight` measurements.

6.5.5 Ordered boxplot

1. First recall Section 4.3.5 (p. 95), and produce Fig. 4.8 (p. 96).
2. ⧫ Now, redraw the plot with Manufacturer in order of increasing mean sodium content (use `reorder`, see Section 6.2.5 on p. 161).
3. ⧫ Inspect the help page `?boxplot`, and change the boxplots so that the width varies with the number of observations per manufacturer (*Hint:* find the `varwidth` argument).

6.5.6 Variances in the I x F

Here, we use the tree inventory data from the irrigation by fertilization (I x F) experiment in the Hawkesbury Forest Experiment (HFE) (see Section A.15, p. 252).

1. ⧫ Use only data from 2012 for this exercise. You can use the file 'HFEIFplotmeans2012.csv' if you want to skip this step.
2. ⧫ There are four treatments in the dataframe. Calculate the variance of diameter for each of the treatments (this should give four values). These are the *within-treatment* variances. Also calculate the variance of tree diameter across all plots (this is one number). This is the *plot-to-plot variance*.
3. ⧫ In 2, also calculate the mean within-treatment variance. Compare the value to the plot-to-plot variance. What can you tentatively conclude about the treatment effect?

6.5.7 Weight loss

This exercise brings together many of the skills from the previous chapters.

Consider a dataset of sequential measurements of a person's weight while on a diet (the 'weightloss' dataset, see Section A.5 on p. 248).

1. ■ Read the dataset ('weightloss.csv'), and convert the 'Date' variable to the `Date` class. See Section 3.6.1 for converting the date, and note the example in Section 3.6.2.1 (p. 75).
2. ■ Add a new variable to the dataset, with the subjects's weight in kilograms (kg) (1 kg = 2.204 pounds).
3. ■ Produce a line plot that shows weight (in kg) versus time.
4. ▲ The problem with the plot you just produced is that all measurements are connected by a line, although we would like to have line breaks for the days where the weight was not measured. To do this, construct a dataframe based on the `weightloss` dataset that has daily values. Hints:
    - Make an entirely new dataframe, with a Date variable, ranging from the first to last days in the weightloss dataset, with a step of one day (see Section 3.6.2.1, p. 75).
    - Using `merge`, paste the Weight data onto this new dataframe. Check for missing values. Use the new dataframe to make the plot.
5. ▲ Based on the new dataframe you just produced, graph the daily change in weight versus time. Also add a dashed horizontal line at `y=0`.

Chapter 7

Linear modelling

7.1 One-way ANOVA

Remember that in Chapter 5 we learned how to compare two means using a two sample t-test. In the simplest case, we assumed that the samples came from populations with the same variance. One-way ANOVA (ANalysis Of VAriance) can be used to compare means across more than two populations. We will not go into the theory here, but the foundation of ANOVA is comparing the variation *between* the group means to the variation *within* the groups (using an F-statistic).

We can use either the `aov` function or `lm` to perform ANOVAs. We will focus exclusively on the latter as it can be generalized more easily to other models. The use of `aov` is only appropriate when you have a balanced design (i.e., the same sample sizes in each of your groups).

To use `lm` for an ANOVA, we need a dataframe containing a (continuous) response variable and a factor variable that defines the groups. For example, in the Coweeta dataset, the species variable is a factor that defines groups by species. We can compute (for example) the mean height by species. Let's look at an example using the Coweeta data, but with only four species to simplify the output.

```
coweeta <- read.csv("coweeta.csv")

# Take a subset and drop empty levels with droplevels.
cowsub <- droplevels(subset(coweeta, species %in% c("cofl","bele","oxar","quru")))

# Quick summary table
with(cowsub, tapply(height, species, mean))
```

```
##     bele     cofl     oxar     quru
## 21.86700  6.79750 16.50500 21.10556
```

We might want to ask, does the mean height vary by species? Before you do any test for significance, a graphical summary of the data is always useful. For this type of data, box plots are preferred since they visualize not just the means but also the spread of the data (see also Section 4.3.5) (Fig. 7.1).

```
boxplot(height~species, data=cowsub)
```

It seems like some of the species differences are quite large. We can fit a one-way ANOVA with `lm`, like so:

```
fit1 <- lm(height ~ species, data=cowsub)
fit1
```

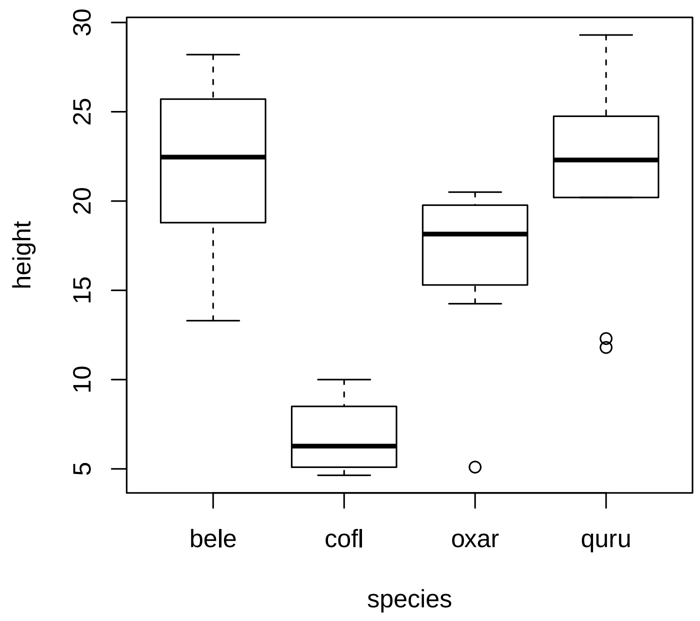


Figure 7.1: Simple box plot for the Coweeta data.

```
##
## Call:
## lm(formula = height ~ species, data = cowsub)
##
## Coefficients:
## (Intercept)  speciescofl  speciesoxar  speciesquru
##     21.8670     -15.0695      -5.3620      -0.7614
```

Notice the four estimated `Coefficients`, these represent the so-called *contrasts*. In this case, `Intercept` represents the mean of the *first* species, `bele`. The next three coefficients are the differences between each species and the first (e.g., species `cofl` has a mean that is -15.07 lower than `bele`).

We can get more details of the fit using `summary`.

```
summary(fit1)

##
## Call:
## lm(formula = height ~ species, data = cowsub)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -11.405  -2.062   0.695   3.299   8.194
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  21.8670     1.5645  13.977 7.02e-14 ***
## speciescofl -15.0695     2.9268  -5.149 2.04e-05 ***
## speciesoxar  -5.3620     2.3467  -2.285   0.0304 *
## speciesquru  -0.7614     2.2731  -0.335   0.7402
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 4.947 on 27 degrees of freedom
## Multiple R-squared:  0.5326,Adjusted R-squared:  0.4807
## F-statistic: 10.26 on 3 and 27 DF,  p-value: 0.0001111
```

This gives us a t-statistic (and p-value) for each coefficient, for a test where the value is compared to zero. Not surprisingly the `Intercept` (i.e., the mean for the first species) is significantly different from zero (as indicated by the very small p-value). Two of the next three coefficients are also significantly different from zero.

At the end of the summary, the F-statistic and corresponding p-value are shown. This p-value tells us whether the whole model is significant. In this case, it is comparing a model with four coefficients (one for each species) to a model that just has the same mean for all groups. In this case, the model is highly significant – i.e. there is evidence of different means for each group. In other words, a model where the mean varies between the four species performs much better than a model with a single grand mean.

7.1.1 Multiple comparisons

The ANOVA, as used in the previous section, gives us a single p-value for the overall 'species effect'. The summary statement further shows whether individual species are different from the first level in the model, which is not always useful. If we want to know whether the four species were all different from each other, we can use a multiple comparison test.

We will use the `emmeans` function from the `emmeans` package (as a side note, base **R** includes the `TukeyHSD` function, but that does not work with `lm`, only with `aov`, and so we do not show you how to use it).

```r
# First fit the linear model again (a one-way ANOVA,
# because species is a factor)
lmSpec <- lm(height ~ species, data=cowsub)

# Load package
library(emmeans)

# Estimate marginal means and confidence interval for each
# level of 'species'. These estimates can be seen if you
# view the result, but we won't do this.
tukey_Spec <- emmeans(lmSpec, 'species')

# Print results of contrasts. This shows p-values for the null hypotheses
# that species A is no different from species B, and so on. By default,
# Tukey contrasts are calculated.
pairs(tukey_Spec)

##  contrast    estimate   SE df t.ratio p.value
##  bele - cofl   15.069 2.93 27   5.149  0.0001
##  bele - oxar    5.362 2.35 27   2.285  0.1267
##  bele - quru    0.761 2.27 27   0.335  0.9868
##  cofl - oxar   -9.707 3.03 27  -3.204  0.0171
##  cofl - quru  -14.308 2.97 27  -4.813  0.0003
##  oxar - quru   -4.601 2.40 27  -1.914  0.2462
```

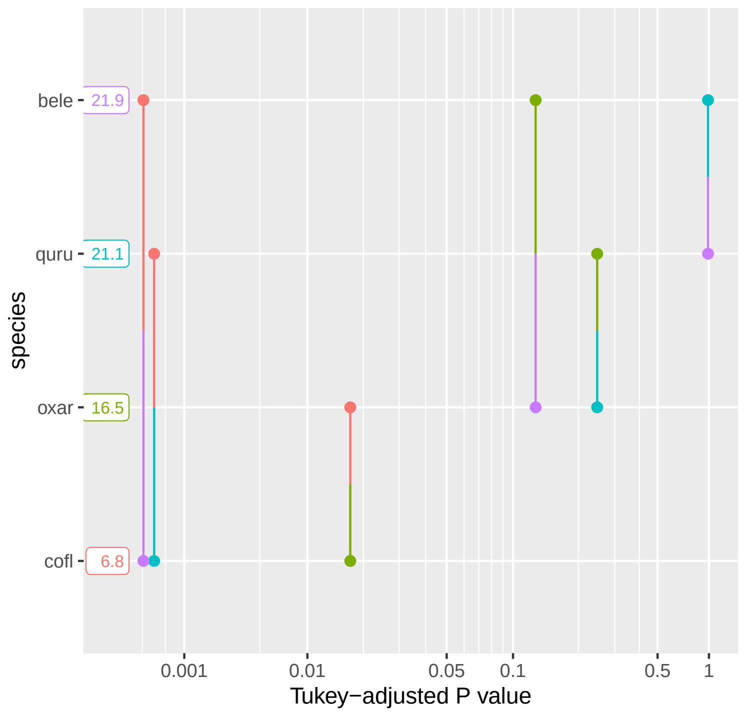


Figure 7.2: A plot of a pairwise p-values for multiple comparisons.

```
##
## P value adjustment: tukey method for comparing a family of 4 estimates
# Some of the species are different from each other, but not all.
```

Try this yourself In the above summary of the multiple comparison (`summary(tukey_Spec)`), the p-values are adjusted for multiple comparisons with the so-called 'single-step method'. To use a different method for the correction (there are many), try the following example:
`pairs(tukey_Spec, adjust="bonferroni")`
Also look at the other options in the help page for `?summary.emmGrid`, in the 'P-value adjustments' section.

We can also produce a pairwise p-value plot of the multiple comparisons, which shows the level of support for each null hypothesis of a pair having the same mean.

This code produces Fig. 7.2.

```
# A plot of a fitted 'emmeans' object (multiple comparison)
pwpp(tukey_Spec)
```

There are several vignettes for `emmeans` at `https://cran.r-project.org/web/packages/emmeans/`, these are very helpful for finding out what is possible with this package.

7.2 Two-way ANOVA

Sometimes there are two (or more) *treatment* factors. The 'age and memory' dataset (see Section A.11) includes the number of words remembered from a list for two age groups and five memory techniques.

This dataset is balanced, as shown below. in a table of counts for each of the combinations. First we fit a linear model of the *main effects*.

```
# Read tab-delimited data
memory <- read.table("eysenck.txt", header=TRUE)

# To make the later results easier to interpret, reorder the Process
# factor by the average number of words remembered.
memory$Process <- with(memory, reorder(Process, Words, mean))

# Count nr of observations
xtabs( ~ Age + Process, data=memory)

##           Process
## Age        Counting Rhyming Adjective Imagery Intentional
##   Older          10      10        10      10          10
##   Younger        10      10        10      10          10

# Fit linear model
fit2 <- lm(Words ~ Age + Process, data=memory)
summary(fit2)

##
## Call:
## lm(formula = Words ~ Age + Process, data = memory)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -9.100 -2.225 -0.250  1.800  9.050
##
## Coefficients:
##                    Estimate Std. Error t value Pr(>|t|)
## (Intercept)          5.2000     0.7632   6.813 8.99e-10 ***
## AgeYounger           3.1000     0.6232   4.975 2.94e-06 ***
## ProcessRhyming       0.5000     0.9853   0.507    0.613
## ProcessAdjective     6.1500     0.9853   6.242 1.24e-08 ***
## ProcessImagery       8.7500     0.9853   8.880 4.41e-14 ***
## ProcessIntentional   8.9000     0.9853   9.033 2.10e-14 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 3.116 on 94 degrees of freedom
## Multiple R-squared:  0.6579,Adjusted R-squared:  0.6397
## F-statistic: 36.16 on 5 and 94 DF,  p-value: < 2.2e-16
```

The `summary` of the fitted model displays the individual t-statistics for each estimated coefficient. As with the one-way ANOVA, the significance tests for each coefficient are performed relative to the base level (by default, the first level of the factor). In this case, for the `Age` factor, the `Older` is the first level, and for the `Process` factor, `Adjective` is the first level. Thus all other coefficients are tested relative to the "Older/Adjective" group. The F-statistic at the end is for the overall model, it tests whether the model is significantly better than a model that includes only a mean count.

If we want to see whether `Age` and/or `Process` have an effect, we need F-statistics for these terms. Throughout this book, to compute p-values for terms in linear models, we use the `Anova` function from the `car` package.

```
# Perform an ANOVA on a fitted model, giving F-statistics
library(car)
Anova(fit2)
```

```
## Anova Table (Type II tests)
##
## Response: Words
##            Sum Sq Df F value    Pr(>F)
## Age        240.25  1  24.746 2.943e-06 ***
## Process   1514.94  4  39.011 < 2.2e-16 ***
## Residuals  912.60 94
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

In this form, the F-statistic is formed by comparing models that do not include the term, but include all others. For example, `Age` is tested by comparing the full model against a model that includes all other terms (in this case, just `Process`).

7.2.1 Interactions

An important question when we have more than one factor in an experiment is whether there are any interactions. For example, do `Process` effects differ for the two `Age` groups, or are they simply additive? We can add interactions to a model by modifying the formula. An interaction is indicated using a "`:`". We can also include all *main effects and interactions* using the `*` operator.

```
# Two equivalent ways of specifying a linear model that includes all main effects
# and interactions:
fit3 <- lm(Words ~ Age + Process + Age:Process, data=memory)

# Is the same as:
fit3.2 <- lm(Words ~ Age * Process, data=memory)
Anova(fit3.2)
```

```
## Anova Table (Type II tests)
##
## Response: Words
##              Sum Sq Df F value    Pr(>F)
## Age          240.25  1 29.9356 3.981e-07 ***
## Process     1514.94  4 47.1911 < 2.2e-16 ***
## Age:Process  190.30  4  5.9279 0.0002793 ***
## Residuals    722.30 90
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The `Anova` table shows that the interaction is significant. When an interaction is significant, this tells you nothing about the direction or magnitude of the interaction term. You can inspect the estimated coefficients in the `summary` output, but we recommend to first visualize the interaction with simple plots, as the coefficients can be easily misinterpreted. One way to visualize the interaction is to use the `interaction.plot` function, as in the following example.

This code produces Fig. 7.3. If there were no interaction between the two factor variables, you would expect to see a series of parallel lines (because the effects of `Process` and `Age` would simply be additive).

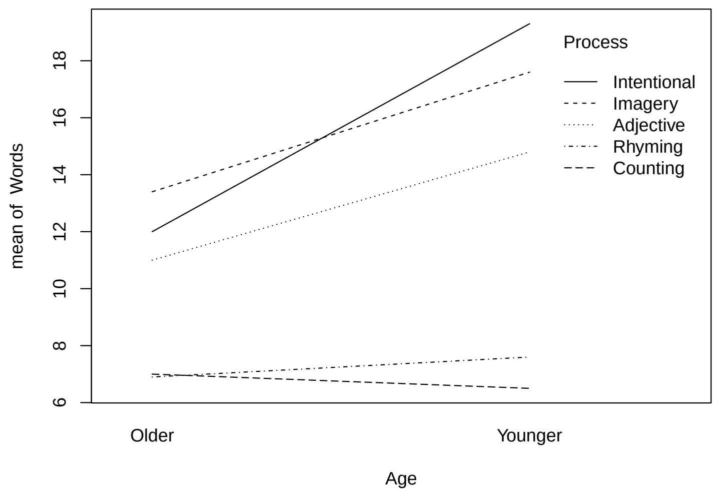


Figure 7.3: An interaction plot for the memory data, indicating a strong interaction (because the lines are not parallel).

```
# Plot the number of words rememberd by Age and Process
# This standard plot can be customized in various ways, see ?interaction.plot
with(memory, interaction.plot(Age, Process, Words))
```

7.2.1.1 More multiple comparisons

When there is a signficant interaction term, we might want to know under what levels of one factor is the effect of another factor signficant. This can be done easily using functions in the `emmeans` package.

```
# Load package
library(emmeans)

# Estimate marginal means associated with 'Age' for each level of 'Process'.
# Notice here that we use the formula interface to specify contrasts, whereas
# we specified contrasts using a character string in the one-way ANOVA example.
fit3.emm <- emmeans(fit3, ~ Age | Process)

# Calculate p-values for each pairwise contrast
pairs(fit3.emm)
```

```
## Process = Counting:
##  contrast        estimate   SE df t.ratio p.value
##  Older - Younger      0.5 1.27 90   0.395  0.6940
##
## Process = Rhyming:
##  contrast        estimate   SE df t.ratio p.value
##  Older - Younger     -0.7 1.27 90  -0.553  0.5820
##
## Process = Adjective:
##  contrast        estimate   SE df t.ratio p.value
##  Older - Younger     -3.8 1.27 90 -2.999  0.0035
##
## Process = Imagery:
##  contrast        estimate   SE df t.ratio p.value
##  Older - Younger     -4.2 1.27 90 -3.315  0.0013
##
## Process = Intentional:
##  contrast        estimate   SE df t.ratio p.value
##  Older - Younger     -7.3 1.27 90 -5.762  <.0001
```

Using the `pairs` function on an `emmGrid` object returns effect sizes and significance tests for each pair-wise contrast. In this example, we can see that older subjects remember signficantly fewer words than younger ones when using the Adjective, Imagery and Intentional processes but not the Counting or Rhyming processes. Alternatively, we may want to contrast different processes within each level of Age. Here it is helpful to use the `cld` function from the `multcomp` package to identify groups of levels that are not significantly different.

```
# Estimate marginal means associated with 'Process' for each level of 'Age'
fit3.emm.2 <- emmeans(fit3, ~ Process | Age)

# Identify groups of processes for which the processes are similar.
# (use multcomp:: to access the function without loading the library)
multcomp::cld(fit3.emm.2)
```

```
## Age = Older:
##  Process     emmean    SE df lower.CL upper.CL .group
##  Rhyming        6.9 0.896 90     5.12     8.68  1
##  Counting       7.0 0.896 90     5.22     8.78  1
##  Adjective     11.0 0.896 90     9.22    12.78   2
##  Intentional   12.0 0.896 90    10.22    13.78   2
##  Imagery       13.4 0.896 90    11.62    15.18   2
##
## Age = Younger:
##  Process     emmean    SE df lower.CL upper.CL .group
##  Counting       6.5 0.896 90     4.72     8.28  1
##  Rhyming        7.6 0.896 90     5.82     9.38  1
##  Adjective     14.8 0.896 90    13.02    16.58   2
##  Imagery       17.6 0.896 90    15.82    19.38   23
##  Intentional   19.3 0.896 90    17.52    21.08    3
##
## Confidence level used: 0.95
## P value adjustment: tukey method for comparing a family of 5 estimates
## significance level used: alpha = 0.05
```

The resulting groups can then be used, for example, to determine which letter(s) should be placed above each treatment in a barplot.

7.2.2 Comparing models

In the above example, we fitted two models for the Memory dataset: one without, and one with the interaction between `Process` and `Age`. We assessed the significance of the interaction by inspecting the p-value for the `Age:Process` term in the `Anova` statement. Another possibility is to perform a likelihood ratio test on two 'nested' models, the model that includes the term, and a model that excludes it. We can perform a likelihood ratio test with the `anova` function, not to be confused with `Anova` from the `car` package![1]

```
# We can perform an anova on two models to compare the fit.
# Note that one model must be a subset of the other model.
# In this case, the second model has the same predictors as the first plus
# the interaction, and the likelihood ratio test thus tests the interaction.
anova(fit2,fit3)
```

```
## Analysis of Variance Table
##
## Model 1: Words ~ Age + Process
## Model 2: Words ~ Age + Process + Age:Process
##   Res.Df   RSS Df Sum of Sq      F    Pr(>F)
## 1     94 912.6
## 2     90 722.3  4     190.3 5.9279 0.0002793 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

A second common way to compare different models is by the AIC (*Akaike's Information Criterion*), which is calculated based on the likelihood (a sort of goodness of fit), and penalized for the number of parameters (i.e. number of predictors in the model). The model with the lowest AIC is the preferred model. Calculating the `AIC` is simple,

```
AIC(fit2, fit3)
```

```
##      df      AIC
## fit2  7 518.9005
## fit3 11 503.5147
```

We once again conclude that the interaction improved the model fit substantially.

7.2.3 Diagnostics

The standard ANOVA assumes normality of the residuals, and we should always check this assumption with some diagnostic plots (Fig. 7.4). Although **R** has built-in diagnostic plots, we prefer the use of `qqPlot` and `residualPlot`, both from the `car` package.

```
par(mfrow=c(1,2))

library(car)

# Residuals vs. fitted
residualPlot(fit3)

# QQ-plot of the residuals
qqPlot(fit3)
```

```
## [1] 26 86
```

The QQ-plot shows some slight non-normality in the upper right. The non-normality probably stems from the fact that the `Words` variable is a 'count' variable. We will return to this in Section 7.5.

Try this yourself Check whether a log-transformation of `Words` makes the residuals closer to normally distributed.

[1]You may have seen the use of `anova` on a single model, which also gives an ANOVA table but is confusing because a sequential (Type-I) test is performed, which we strongly advise against as it is never the most intuitive test of effects.

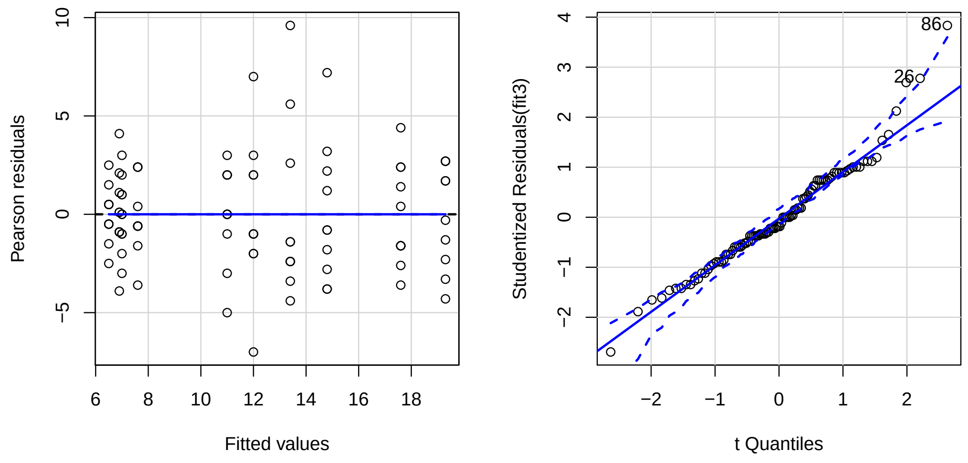


Figure 7.4: Simple diagnostic plots for the Memory ANOVA.

7.3 Multiple linear regression

In Chapter 5, we introduced simple linear regression. We use this method to study the relationship between two continuous variables: a *response* ('y variable') and a *predictor* (or independent variable) ('x variable'). We can use multiple regression to study the relationship between a response and more than one predictor. For example, in the Allometry dataset (see Section A.1), we might expect that leaf area depends on both tree diameter and height. We test this idea in the following example.

The first step should always be to inspect the data visually. In this case, let's make two simple scatter plots (Fig. 7.5).

```
# Read the data, if you haven't already
allom <- read.csv("allometry.csv")

# Two simple scatter plots on a log-log scale
par(mfrow=c(1,2))
plot(leafarea~diameter, data=allom, log="xy")
plot(leafarea~height, data=allom, log="xy")
```

These plots are shown on a log-log scale, and it certainly looks like there is a relationship between leaf area and diameter as well as height. However, since diameter and height are probably correlated we need to take a close look at a linear model to understand what is going on.

We are going to fit a model that looks like, using statistical notation,

$$y = \alpha + \beta_1 x_1 + \beta_2 x_2 + \varepsilon \tag{7.1}$$

where α is the intercept, x_1 and x_2 are the two predictors (height and diameter), andthe *two* slopes are β_1 and β_2. We are particularly interested in testing for significant effect of both predictors, which is akin to saying that we are testing the values of the two slopes against zero. We will first use the original scale (i.e. not log-transformed) and perform some diagnostic plots (shown in Fig. 7.6).

```
# Fit a multiple regression without interactions, and inspect the summary
fit4 <- lm(leafarea~diameter+height, data=allom)
summary(fit4)
```

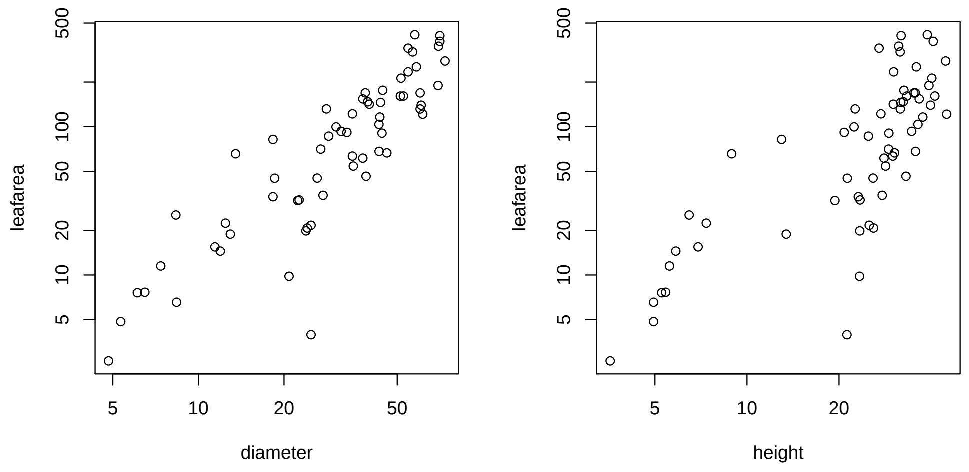


Figure 7.5: Leaf area as a function of height and diameter (note the log-log scale).

```
##
## Call:
## lm(formula = leafarea ~ diameter + height, data = allom)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -126.892  -24.429    1.775   27.676  207.676
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -17.8759    17.6390  -1.013  0.31492
## diameter      6.9137     0.7562   9.143 5.68e-13 ***
## height       -4.4030     1.2794  -3.441  0.00106 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 55.43 on 60 degrees of freedom
## Multiple R-squared:  0.743,Adjusted R-squared:  0.7345
## F-statistic: 86.75 on 2 and 60 DF,  p-value: < 2.2e-16
```

```
# Basic diagnostic plots.
par(mfrow=c(1,2))
residualPlot(fit4)
qqPlot(fit4)
```

```
## [1] 38 47
```

The summary shows that both coefficients (βs) are significantly different from zero, and the p-value for `diameter` is much smaller. However, the diagnostic plots indicate some problems. The scale-location plot shows a somewhat increasing trend – i.e. the variance is not constant, and the QQ-plot shows some departures from normality.

Let's refit the model using log-transformed variables, and check the diagnostic plots (Fig. 7.7).

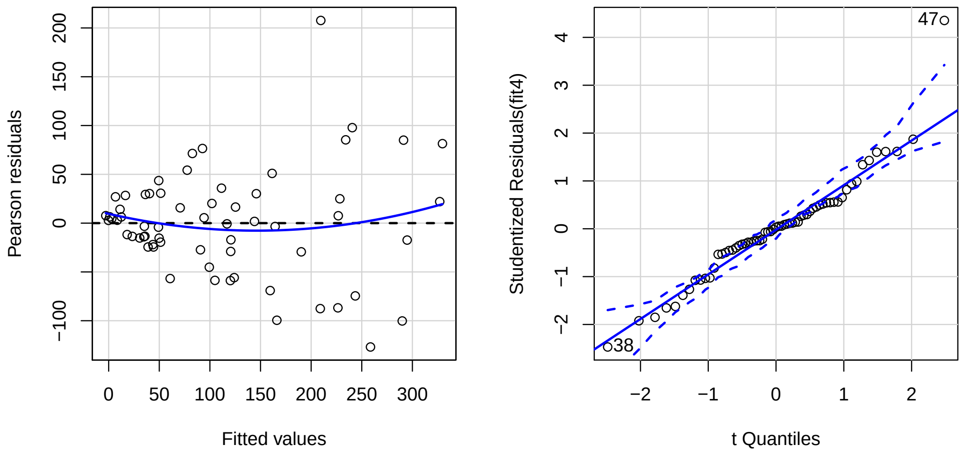


Figure 7.6: Diagnostic plots for the multiple regression of the Allometry data.

```
# For convenience, add log-transformed variables to the dataframe:
allom$logLA <- log10(allom$leafarea)
allom$logD <- log10(allom$diameter)
allom$logH <- log10(allom$height)

# Refit model, using log-transformed variables.
fit5 <- lm(logLA ~ logD + logH, data=allom)
summary(fit5)

##
## Call:
## lm(formula = logLA ~ logD + logH, data = allom)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.07231 -0.17010  0.01759  0.17347  0.45205
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  -0.4133     0.1554  -2.659   0.0100 *
## logD          2.1901     0.3040   7.205 1.12e-09 ***
## logH         -0.7343     0.3222  -2.279   0.0262 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.2525 on 60 degrees of freedom
## Multiple R-squared:  0.7905,Adjusted R-squared:  0.7835
## F-statistic: 113.2 on 2 and 60 DF,  p-value: < 2.2e-16

# And diagnostic plots.
par(mfrow=c(1,2))
residualPlot(fit5)
qqPlot(fit5)
```

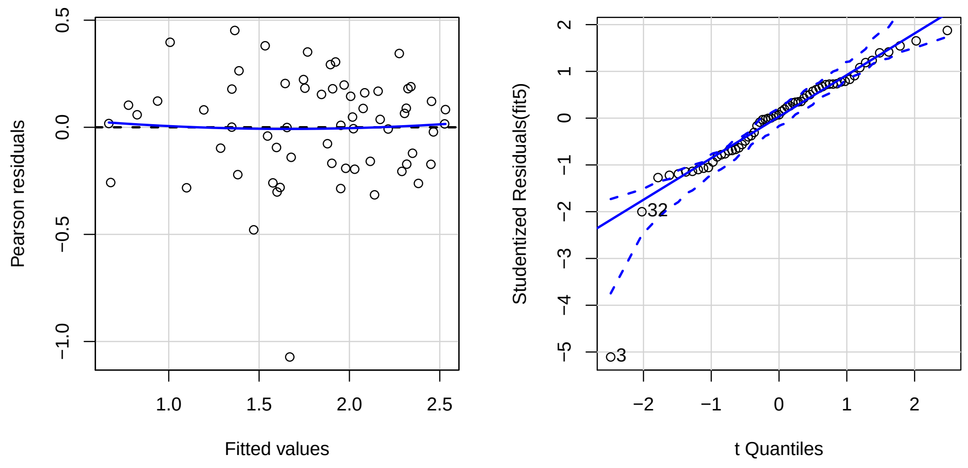


Figure 7.7: Diagnostic plots for the multiple regression of the Allometry data, using log-transformed variables.

```
## [1]  3 32
```

The coefficients (βs) are still significant (as shown by small p-values) and the diagnostic plots are better, although the QQ-plot is affected by some outliers.

Of course, we are not restricted to just two predictors, and we can include interactions as well. For example,

```
# A multiple regression that includes all main effects as wel as interactions
fit6 <- lm(logLA ~ logD * logH, data=allom)
summary(fit6)
```

```
##
## Call:
## lm(formula = logLA ~ logD * logH, data = allom)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.07242 -0.17007  0.01782  0.17350  0.45190
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -0.41519    0.64371  -0.645  0.52143
## logD         2.19222    0.75013   2.922  0.00492 **
## logH        -0.73309    0.51042  -1.436  0.15621
## logD:logH   -0.00135    0.44214  -0.003  0.99757
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.2547 on 59 degrees of freedom
## Multiple R-squared:  0.7905,Adjusted R-squared:  0.7798
## F-statistic: 74.19 on 3 and 59 DF,  p-value: < 2.2e-16
```

In this case, the interaction is not significant.

7.4 Linear models with factors and continuous variables

So far we have looked at ANOVA, including two-way ANOVA, where a response variable is modelled as dependent on two treatments or factors, and *regression* including multiple linear regression where a response variable is modelled as dependent on two continuous variables. These are just special cases of the *linear model*, and we can extend these simple models by including a mix of factors and continuous predictors. The situation where we have one continuous variable and one factor variable is also known as ANCOVA (analysis of covariance), but using the `lm` function we can specify any model with a combination of predictors.

Returning to the allometry data set once more, we might expect that leaf area depends also on `species`, which is a factor with three levels:

```
# A linear model combining factors and continuous variables
fit7 <- lm(logLA ~ species + logD + logH, data=allom)
summary(fit7)
```

```
##
## Call:
## lm(formula = logLA ~ species + logD + logH, data = allom)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.10273 -0.09629 -0.00009  0.13811  0.38500
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -0.28002    0.14949  -1.873  0.06609 .
## speciesPIPO -0.29221    0.07299  -4.003  0.00018 ***
## speciesPSME -0.09095    0.07254  -1.254  0.21496
## logD         2.44336    0.27986   8.731  3.7e-12 ***
## logH        -1.00967    0.29870  -3.380  0.00130 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.2252 on 58 degrees of freedom
## Multiple R-squared:  0.839,Adjusted R-squared:  0.8279
## F-statistic: 75.54 on 4 and 58 DF,  p-value: < 2.2e-16
```

The summary of this fit shows that `PIPO` differs markedly from the base level species, in this case `PIMO`. However, `PSME` does not differ from `PIMO`. The linear effects in log diameter and log height remain significant.

An ANOVA table will tell us whether adding species improves the model overall (as a reminder, you need the `car` package loaded for this function).

```
Anova(fit7)
```

```
## Anova Table (Type II tests)
##
## Response: logLA
##           Sum Sq Df F value    Pr(>F)
## species   0.8855  2   8.731 0.0004845 ***
## logD      3.8652  1  76.222 3.701e-12 ***
## logH      0.5794  1  11.426 0.0013009 **
## Residuals 2.9412 58
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

We see that adding species to the model did improve the model.

Perhaps the slopes on log diameter (and other measures) vary by species. This is an example of an interaction between a factor and a continuous variable. We can fit this as:

```
# A linear model that includes an interaction between a factor and
# a continuous variable
fit8 <- lm(logLA ~ species * logD + species * logH, data=allom)
summary(fit8)
```

```
##
## Call:
## lm(formula = logLA ~ species * logD + species * logH, data = allom)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.05543 -0.08806 -0.00750  0.11481  0.34124
##
## Coefficients:
##                   Estimate Std. Error t value Pr(>|t|)
## (Intercept)        -0.3703     0.2534  -1.461   0.1498
## speciesPIPO        -0.4074     0.3483  -1.169   0.2473
## speciesPSME         0.2249     0.3309   0.680   0.4997
## logD                1.3977     0.4956   2.820   0.0067 **
## logH                0.1610     0.5255   0.306   0.7606
## speciesPIPO:logD    1.5029     0.6499   2.313   0.0246 *
## speciesPSME:logD    1.7231     0.7185   2.398   0.0200 *
## speciesPIPO:logH   -1.5238     0.6741  -2.261   0.0278 *
## speciesPSME:logH   -2.0890     0.7898  -2.645   0.0107 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.2155 on 54 degrees of freedom
## Multiple R-squared:  0.8626,Adjusted R-squared:  0.8423
## F-statistic: 42.39 on 8 and 54 DF,  p-value: < 2.2e-16
```

Note that the species effect is still fitted, because we used the `*` operator (where `species*logD` is equivalent to `species + logD + species:logD`).

From the summary, the `logD` (log diameter) coefficient appears to be significant. In this example, this coefficient represents the slope for the base species, `PIMO`. Other terms, including `speciesPIPO:logD`, are also significant. This means that their slopes differ from that of the base species (`PIMO`).

We can also look at the `Anova` to decide whether adding these slope terms improved the model.

```
Anova(fit8)
```

```
## Anova Table (Type II tests)
##
## Response: logLA
##              Sum Sq Df F value    Pr(>F)
## species      0.8855  2  9.5294 0.0002854 ***
## logD         3.9165  1 84.2945 1.286e-12 ***
## logH         0.6102  1 13.1325 0.0006425 ***
## species:logD 0.3382  2  3.6394 0.0329039 *
```

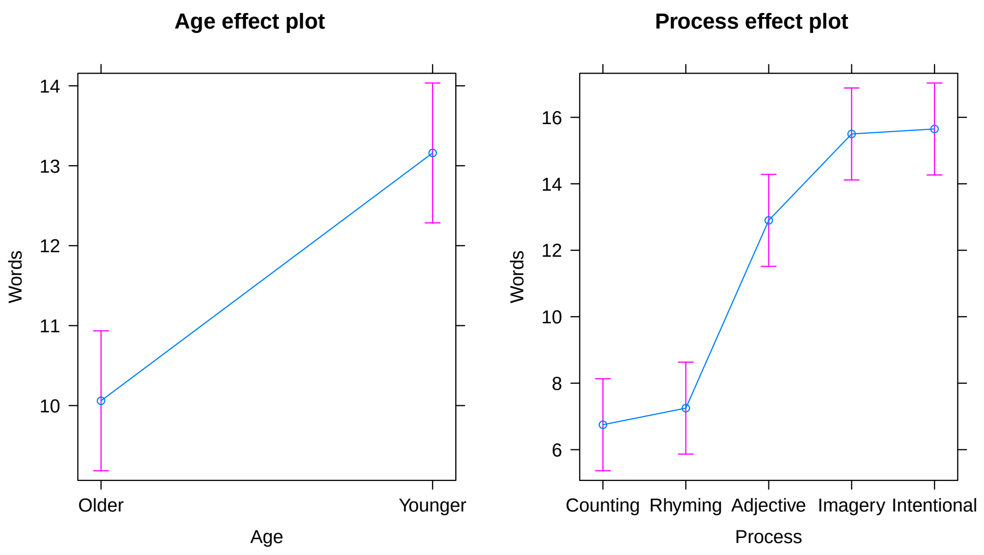


Figure 7.8: Effects plot of a linear model using the Memory data, including main effects only. Effects of Age (left panel) and Process (right panel) are assumed to be additive.

```
## species:logH 0.3752  2  4.0378 0.0232150 *
## Residuals    2.5089 54
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

7.4.1 Predicted effects

The coefficients in a linear model are usually contrasts (i.e. differences between factor levels), slopes or intercepts. While this is useful for comparisons of treatments, it is often more instructive to visualize the predictions at various combinations of factor levels.

For balanced designs where the model contains all interactions, the predicted means from a linear model will be the same as the group means calculated from the data.

However, if the data is not balanced and/or some of the interactions are left out the group mean and the predicted mean will be different. The package `effects` contains a very useful function to calculate and visualise these effects.

For example, consider this example using the memory data (Fig. 7.8) with main effects only.

```
# Load the effects package
library(effects)

# Fit linear model, main effects only
fit9 <- lm(Words ~ Age + Process, data=memory)
plot(allEffects(fit9))
```

And compare the output when we add all interactions (Fig. 7.9),

```
fit10 <- lm(Words ~ Age * Process, data=memory)
plot(allEffects(fit10))
```

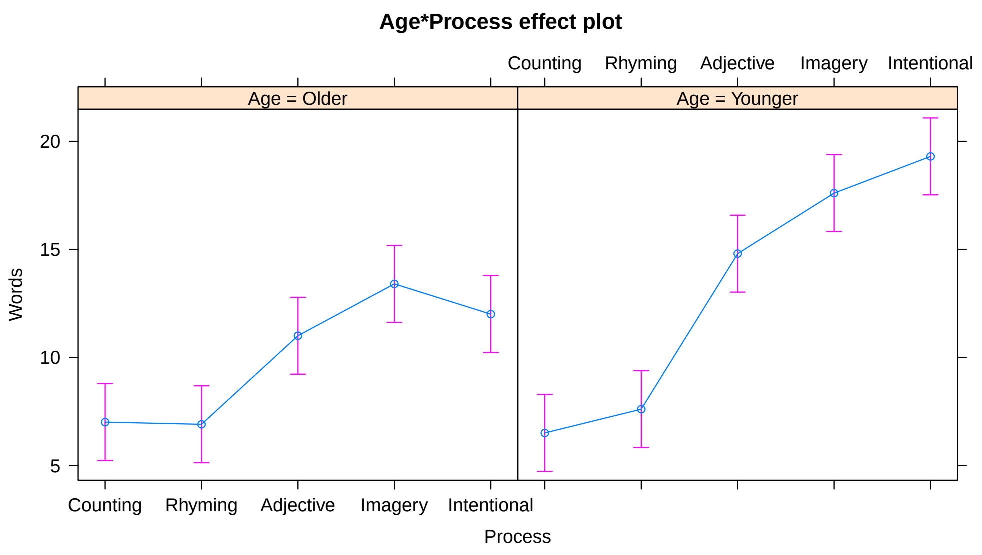


Figure 7.9: Effects plot a linear model of the Memory data, including main effects and interactions. The effects of Age and Process are not simply additive.

Try this yourself Use the command `plot(allEffects(...))` on the two model fits from Section 7.4, and compare the results. Also, use these plots to double check your interpretation of the model coefficients, as shown in the output of `summary`.

Another option is to use the `visreg` package, which can be used to make attractive plots of the predictions of a linear model. In the following example we redo Fig. 7.9 using the `visreg` package.

The following example makes Fig. 7.10.

```r
library(visreg)

# Read tab-delimited data
memory <- read.table("eysenck.txt", header=TRUE)

# To make the plots easier to interpret, reorder the Process
# factor by the average number of words remembered (we did this earlier in
# the chapter already for this dataset, it is repeated here).
memory$Process <- with(memory, reorder(Process, Words, mean))

# Refit the linear model as above.
fit10 <- lm(Words ~ Age*Process, data=memory)

# Here we specify which variable should be on the X axis (Process),
# and which variable should be added with different colours (Age).
visreg(fit10, "Process", by="Age", overlay=TRUE)
```

Here is another example. This time, we include a continuous and a factor variable in the model (Fig. 7.11).

```r
# Read data and make sure CO2 treatment is a factor!
pupae <- read.csv("pupae.csv")
pupae$CO2_treatment <- as.factor(pupae$CO2_treatment)
```

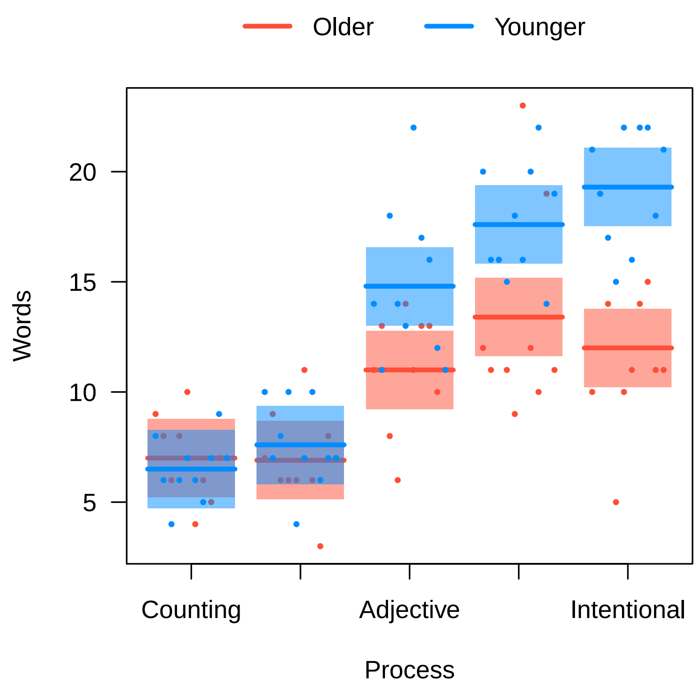


Figure 7.10: Visualization of a fitted linear model with the visreg package.

```
# A linear model with a factor and a numeric variable
fit_b <- lm(Frass ~ PupalWeight * CO2_treatment, data=pupae)

# When plotting a model that has interactions, make sure to specify the variable
# that should appear on the X axis (the first argument), and the factor
# variable (the 2nd argument).
visreg(fit_b, "PupalWeight", by="CO2_treatment", overlay=TRUE)
```

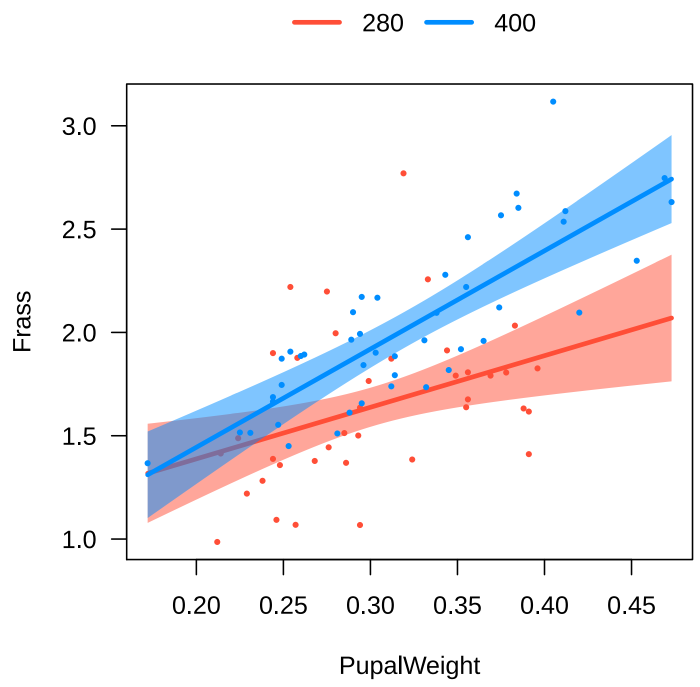


Figure 7.11: Visualization of a fitted linear model with a continuous and factor variable using the `visreg` package.

7.4.2 Using predicted effects to make sense of model output

Next we show an example of a model where using predicted effects as introduced in Section 7.4.1 is very helpful in understanding model output. For this example, we use measurements of photosynthesis and transpiration of tree leaves in the EucFACE experiment (see Section A.23 for a description of the data). We are interested in the relationship between photosynthesis and transpiration (the ratio of which is known as the water-use efficiency), and whether this relationship differs with CO_2 treatment. The data are shown in Fig. 7.12.

```
eucgas <- read.csv("eucface_gasexchange.csv")

palette(c("blue","red"))
with(eucgas, plot(Trmmol, Photo, pch=19, col=CO2))
legend("topleft", levels(eucgas$CO2), pch=19, col=palette())

boxplot(Photo ~ CO2, data=eucgas, col=palette(), ylab="Photo")
```

It seems quite clear from Fig. 7.12 that, at a given `Trmmol`, `Photo` is higher in the elevated (`Ele`) CO_2 treatment. From the boxplot on the right, it also seems more than reasonable to expect that overall, `Photo` is higher in `Ele` (when not accounting for the `Trmmol` covariate).

Let's fit a linear model to confirm this effect.

```
# A linear model with a continuous and a factor predictor, including the interaction.
lmfit <- lm(Photo ~ CO2*Trmmol, data=eucgas)

# Significance of overall model terms (sequential anova)
anova(lmfit)

## Analysis of Variance Table
##
## Response: Photo
##            Df Sum Sq Mean Sq F value    Pr(>F)
## CO2         1 322.96  322.96 31.1659 3.142e-07 ***
## Trmmol      1 668.13  668.13 64.4758 7.074e-12 ***
## CO2:Trmmol  1   0.49    0.49  0.0471    0.8288
## Residuals  80 829.00   10.36
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

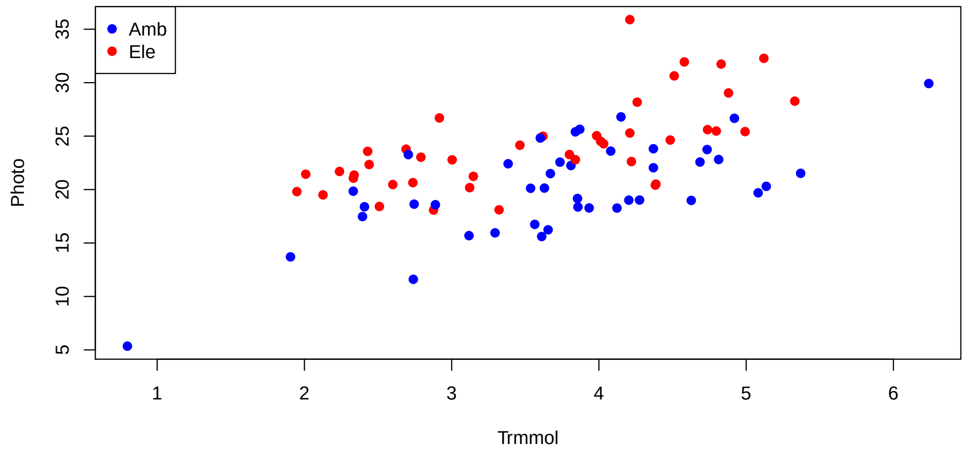
And the coefficients table from `summary`:

```
##                 Estimate Std. Error    t value Pr(>|t|)
## (Intercept)    9.1590768  1.9364911  4.7297284    1e-05 ***
## CO2Ele         5.0146388  2.6955506  1.8603393  0.06651   .
## Trmmol         2.9221253  0.4973512  5.8753756    0e+00 ***
## CO2Ele:Trmmol -0.1538359  0.7090711 -0.2169542  0.82880
```

```
# Significance of the individual coefficients:
summary(lmfit)
```

Look at the 'coefficients' table in the `summary` statement. Four parameters are shown, they can be interpreted as, 1) the intercept for 'Amb', 2) the slope for 'Amb', 3) the *difference* in the intercept for 'Ele', compared to 'Amb', 4) the *difference* in the slope for 'Ele', compared to 'Amb'.

It seems that neither the intercept or slope effect of `CO2` is significant here, which is surprising. Also confusing is the fact that the `anova` statement showed a clear significant effect of `CO2`, so what is going on here?



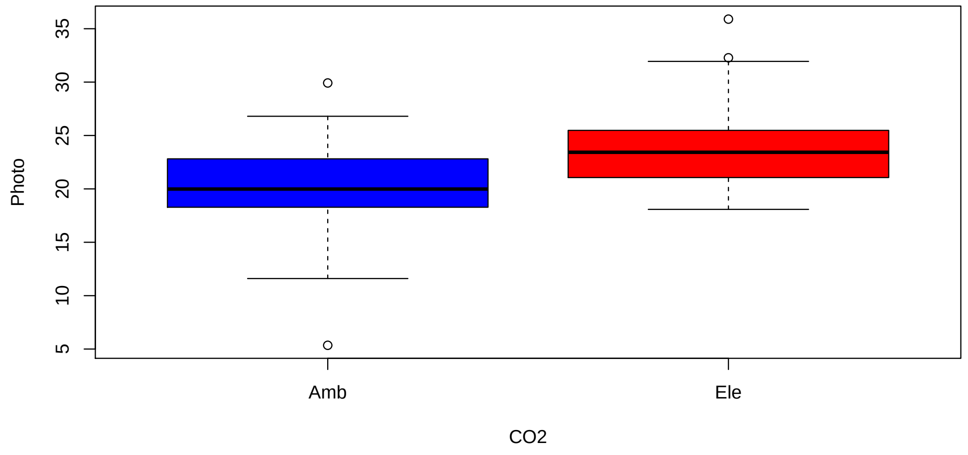


Figure 7.12: Photosynthesis and leaf transpiration rate (Trmmol) for leaves in elevated (red) and ambient (blue) CO2 concentration.

First recall that the sequential anova tests each term against a model that includes *only the terms preceding it*. So, since we added CO2 as the first predictor, its test in the anova is tested against a model that has no predictors. This is similar in approach to simply performing a t-test on Photo vs. CO2 (which also shows a significant effect), in other words testing for separation in the right-hand panel of Fig. 7.12. It is clearly a different test from those shown in the summary statement.

To understand the tests of the coefficients, we will plot predictions of the model, together with confidence intervals. The following code makes Fig. 7.13, and we introduce the use of the predict function to estimate fitted values, and confidence intervals, from a fitted model.

```r
# Set up a regular sequence of numbers, for which 'Photo' is to be predicted from
xval <- seq(0, max(eucgas$Trmmol), length=101)

# Two separate dataframes, one for each treatment/
amb_dfr <- data.frame(Trmmol=xval, CO2="Amb")
ele_dfr <- data.frame(Trmmol=xval, CO2="Ele")

# Predictions of the model using 'predict.lm'
# The first argument is the fitted model, the second argument a dataframe
# containing values for the predictor variables.
predamb <- as.data.frame(predict(lmfit, amb_dfr, interval="confidence"))
predele <- as.data.frame(predict(lmfit, ele_dfr, interval="confidence"))

# Plot. Set up the axis limits so that they start at 0, and go to the maximum.
palette(c("blue","red"))
with(eucgas, plot(Trmmol, Photo, pch=19, col=CO2,
                  xlim=c(0, max(Trmmol)),
                  ylim=c(0, max(Photo))))

# Add the lines; the fit and lower and upper confidence intervals.
with(predamb, {
  lines(xval, fit, col="blue", lwd=2)
  lines(xval, lwr, col="blue", lwd=1, lty=2)
  lines(xval, upr, col="blue", lwd=1, lty=2)
})

with(predele, {
  lines(xval, fit, col="red", lwd=2)
  lines(xval, lwr, col="red", lwd=1, lty=2)
  lines(xval, upr, col="red", lwd=1, lty=2)
})
```

Try this yourself The above plot can also be easily made with the visreg package, as we have seen already. Use the code visreg(lmfit, "Trmmol", by="CO2", overlay=TRUE) to make a similar plot. Set the x-axis limit with xlim to include the intercept.

The intercept in a linear model is of course the value of the Y variable where X is zero. As we can see in Fig. 7.13, the confidence intervals for the regression lines overlap when Trmmol is zero - which is the comparison made in the summary statement for the intercept. We now see why the intercept was not significant, but it says very little about the treatment difference *in the range of the data*.

Perhaps it is more meaningful to test for treatment differences at a mean value of Trmmol. There are four ways to do this.

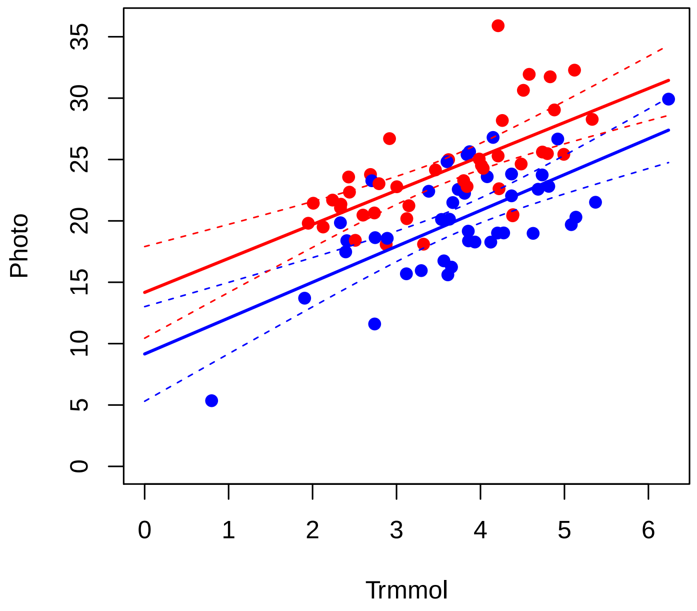


Figure 7.13: Predicted relationship between Photo and Trmmol for ambient and elevated CO2 concentrations in the EucFACE leaf gas exchange dataset. Dashed lines are confidence intervals for the regression line.

Centering the predictor

The first approach is to recenter the predictor so that the intercept can be interpreted as the value where the predictor (in our case, `Trmmol`) is at its mean value.

```
# Rescaled transpiration rate
# This is equivalent to Trmmol - mean(Trmmol)
eucgas$Trmmol_center <- scale(eucgas$Trmmol, center=TRUE, scale=FALSE)

# Refit using centered predictor
lmfit2 <- lm(Photo ~ Trmmol_center*CO2, data=eucgas)
```

The coefficients table in the `summary` statement now shows a highly significant effect for `CO2Ele`, a difference of about 4.22 units. It is also possible to compute confidence intervals on the coefficients via `confint(lmfit2)`, try this yourself.

```
# Summary of the fit:
summary(lmfit2)
```

```
##                          Estimate Std. Error    t value Pr(>|t|)
## (Intercept)            19.8846914  0.4988568 39.8605236   <2e-16 ***
## Trmmol_center           2.9221253  0.4973512  5.8753756        0 ***
## CO2Ele                  4.4499864  0.7055393  6.3072129        0 ***
## Trmmol_center:CO2Ele -0.1538359  0.7090711 -0.2169542   0.8288
```

Using the effects package

Another way is to compute the `CO2` effect at a mean value of `Trmmol`. This avoids having to refit the model with centered data, and is more flexible.

```
# The effects package calculates effects for a variable by averaging over all other
# terms in the model
library(effects)
Effect("CO2", lmfit)

##
##  CO2 effect
## CO2
##      Amb      Ele
## 19.88469 24.33468

# confidence intervals can be obtained via
summary(Effect("CO2", lmfit))

##
##  CO2 effect
## CO2
##      Amb      Ele
## 19.88469 24.33468
##
##  Lower 95 Percent Confidence Limits
## CO2
##      Amb      Ele
## 18.89193 23.34178
##
##  Upper 95 Percent Confidence Limits
## CO2
##      Amb      Ele
## 20.87745 25.32757
```

The `effects` package is quite flexible. For example, we can calculate the predicted effects at any specified value of the predictors, like so (output not shown):

```
# For example, what is the CO2 effect when Trmmol was 3?
summary(Effect("CO2", lmfit, given.values=c(Trmmol=3)))
```

Least-square means

The effect size while holding other predictors constant at their mean value is also known as the 'least-square mean' (or even 'estimated marginal means'), and is implemented as such in the `emmeans` package. It is a powerful package, also to make sense of models that are far more complex than the one in this example, as seen in section 7.2.1.1.

```
library(emmeans)
summary(emmeans(lmfit, "CO2"))

## NOTE: Results may be misleading due to involvement in interactions

##  CO2 emmean    SE df lower.CL upper.CL
##  Amb   19.9 0.499 80     18.9     20.9
##  Ele   24.3 0.499 80     23.3     25.3
##
## Confidence level used: 0.95

# emmeans warns that perhaps the results are misleading - this is true for more
# complex models but not a simple one as shown here.
```

Using the `predict` function

Finally, we show that the effects can also be obtained via the use of `predict`, as we already saw in the code to produce Fig. 7.13.

```
# Predict fitted Photo at the mean of Trmmol, for both CO2 treatments
predict(lmfit, data.frame(Trmmol=mean(eucgas$Trmmol),
                          CO2=levels(eucgas$CO2)),
        interval="confidence")

##        fit      lwr      upr
## 1 19.88469 18.89193 20.87745
## 2 24.33468 23.34178 25.32757
```

Further reading This example shows that interpretation of main effects (in this case, `CO2`) is not at all straightforward when the model also includes an interaction term (`CO2:Trmmol`). A readable review of this problem is Engqvist 2005, *Animal Behaviour* 70(4):967-971. There, it is shown that many studies misinterpret the main effects in cases like this.

7.4.3 Quadratic and polynomial terms

So far we have seen models with just linear terms, but it is straightforward and often necessary to add quadratic (x^2) or higher-order terms (e.g. x^3) when the response variable is far from linear in the predictors. You can add any transformation of the predictors in an `lm` model by nesting the transformation inside the `I()` function, like so:

```
lmq1 <- lm(height ~ diameter + I(diameter^2), data=allom)
```

This model fits both the linear (`diameter`) and squared terms (`I(diameter2̂)`) of the predictor, as well as the usual intercept term. If you want to fit all polynomial terms up to some order, you can use the `poly` function like so,

```
lmq1 <- lm(height ~ poly(diameter, 2), data=allom)
```

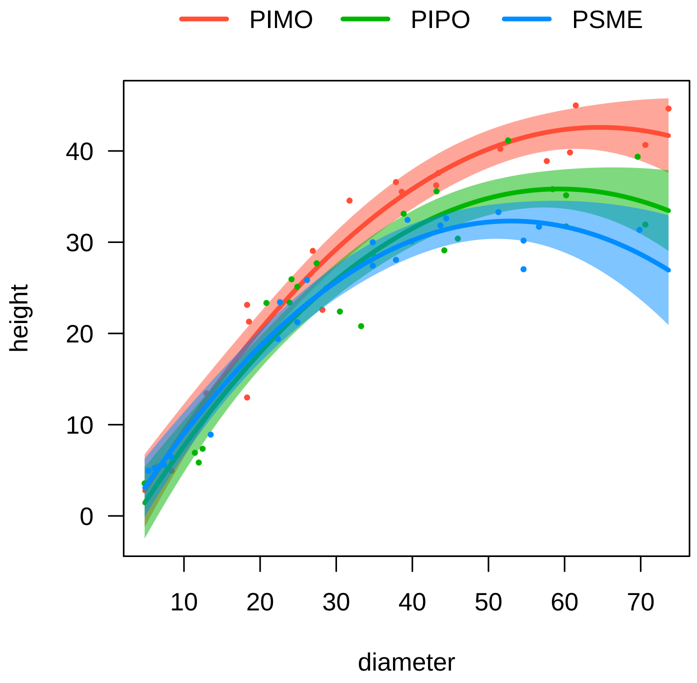
This model specification is exactly equivalent to the above, and is more convenient when you have multiple quadratic / polynomial terms and interactions with factor variables.

The following example quickly tests whether a quadratic term improves the model fit of `height` vs. `diameter` for the species `PIPO` in the allometry dataset.

```
pipo <- subset(allom, species == "PIPO")

# Fit model with the quadratic term:
lmq2 <- lm(height ~ diameter + I(diameter^2), data=pipo)

# The very small p-value for the quadratic terms shows that the
# relationship is clearly not linear, but better described with a
# quadratic model.
Anova(lmq2)

## Anova Table (Type II tests)
##
## Response: height
##               Sum Sq Df F value    Pr(>F)
## diameter      821.25  1  53.838 5.905e-07 ***
## I(diameter^2) 365.78  1  23.980 0.0001001 ***
## Residuals     289.83 19
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

When fitting a quadratic model, it is again very useful to use `visreg` to inspect the model that is estimated, since it becomes even more difficult to make sense of the various linear, quadratic, and intercept terms, especially when interactions with factor variables are added. Consider this example for the allometry dataset, which makes Fig. **??**.

```
# Fit a linear model with linear, quadratic terms, and all species interactions.
allomquadfit <- lm(height ~ poly(diameter, 2)*species, data=allom)

# Inspect summary(allomquadfit) to confirm you cannot possibly make sense of this
# many coefficients.

# But plotting the predictions is much easier to understand:
visreg(allomquadfit, "diameter", by="species", overlay=TRUE)
```


PIMO
PIPO
PSME
height
diameter
0
10
20
30
40
10
20
30
40
50
60
70

7.5 Generalized Linear Models

So far we have looked at modelling a continuous response to one or more factor variables (ANOVA), one or more continuous variables (multiple regression), and combinations of factors and continuous variables. We have also assumed that the predictors are normally distributed, and that, as a result, the response will be, too. We used a log-transformation in one of the examples to meet this assumption.

In some cases, there is no obvious way to transform the response or predictors, and in other cases it is nearly impossible. Examples of difficult situations are when the response represents a count or when it is binary (i.e., has only two possible outcomes).

Generalized linear models (GLMs) [2] extend linear models by allowing non-normally distributed errors. The basic idea is as follows. Suppose you have a response variable y and one or more predictors (independent variables) that can be transformed into a linear predictor in the same way as in linear models, which we call η. Then y is modelled as some distribution with mean μ, which itself is related to the predictors through the linear predictor and a *link*-function, g. Formally,

$$g(\mu) = \eta \tag{7.2}$$

In practice, the distribution for y and the link-function are chosen depending on the problem. Logistic regression is an example of a GLM, with a binomial distribution and the link-function

$$\log\left(\frac{\mu}{1-\mu}\right) = \eta$$

Another common GLM uses the Poisson distribution, in which case the most commonly used link-function is $\log$. This is also called Poisson regression, and it is often used when the response represents counts of items. In the following, we will demonstrate two common GLMs: logistic regression and Poisson regression.

7.5.1 Logistic Regression

The Titanic dataset (see Section A.12) contains information on the survival of passengers on the Titanic, including ages, gender and passenger class. Many ages are missing, and for this example we will work with only the non-missing data (but note that this is not always the best choice). Refer to Section 3.4 (p. 63) for working with and removing missing values.

```
# Read tab-delimited data
titanic <- read.table("titanic.txt", header=TRUE)

# Complete cases only (drops any row that has a missing value anywhere)
titanic <- titanic[complete.cases(titanic),]

# Construct a factor variable based on 'Survived'
titanic$Survived <- factor(ifelse(titanic$Survived==1, "yes", "no"))

# Look at a standard summary
summary(titanic)
```

[2]The term *General Linear Model*, which you may see used sometimes, is not the same as a GLM, although some statistics packages use GLM to mean general linear model, and use another term for generalized linear model.

```
##                              Name      PClass         Age             Sex
##  Carlsson, Mr Frans Olof      :  2   1st:226   Min.   : 0.17   female:288
##  Connolly, Miss Kate          :  2   2nd:212   1st Qu.:21.00   male  :468
##  Kelly, Mr James              :  2   3rd:318   Median :28.00
##  Abbing, Mr Anthony           :  1             Mean   :30.40
##  Abbott, Master Eugene Joseph:  1             3rd Qu.:39.00
##  Abbott, Mr Rossmore Edward   :  1             Max.   :71.00
##  (Other)                      :747
##  Survived
##  no :443
##  yes:313
##
##
##
##
##
```

The idea here is to find out whether the probability of survival depends on the passenger's `Age`, `Sex` and `PClass` (passenger class). Before we proceed, it is always a good idea to start by visualizing the data to find out what we are dealing with (and to make sure we will interpret the model output correctly). If we plot two factor variables against each other, **R** produces quite a useful plot, as the following example demonstrates (Fig. 7.14).

```
par(mfrow=c(1,2), mgp=c(2,1,0))
with(titanic, plot(Sex, Survived, ylab="Survived", xlab="Sex"))
with(titanic, plot(PClass, Survived, ylab="Survived", xlab="Passenger class"))
```

In logistic regression we model the probability of the "1" response (in this case the probability of survival). Since probabilities are between 0 and 1, we use a logistic transform of the linear predictor, where the linear predictor is of the form we would like to use in the linear models above. If η is the linear predictor and Y is the binary response, the logistic model takes the form,

$$P(Y = 1) = \frac{1}{1 + e^{-\eta}} \tag{7.3}$$

These models are easily fit with `glm`. It has a similar interface to `lm` with a couple of additional features. To fit a logistic regression to the (modified) titanic data we use,

```
# Fit a logistic regression
fit11 <- glm(Survived~Age+Sex+PClass, data=titanic, family=binomial)
summary(fit11)

##
## Call:
## glm(formula = Survived ~ Age + Sex + PClass, family = binomial,
##     data = titanic)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.7226  -0.7065  -0.3917   0.6495   2.5289
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept)  3.759662   0.397567   9.457  < 2e-16 ***
## Age         -0.039177   0.007616  -5.144 2.69e-07 ***
## Sexmale     -2.631357   0.201505 -13.058  < 2e-16 ***
```

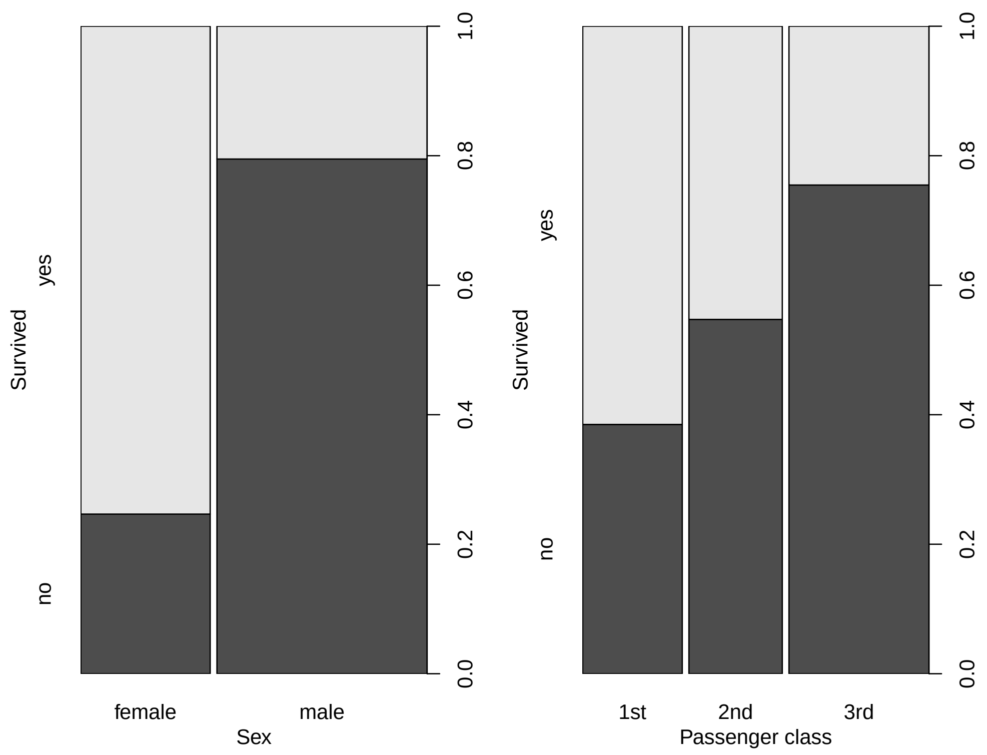


Figure 7.14: Probability of survival versus passenger class and sex for the titanic data.

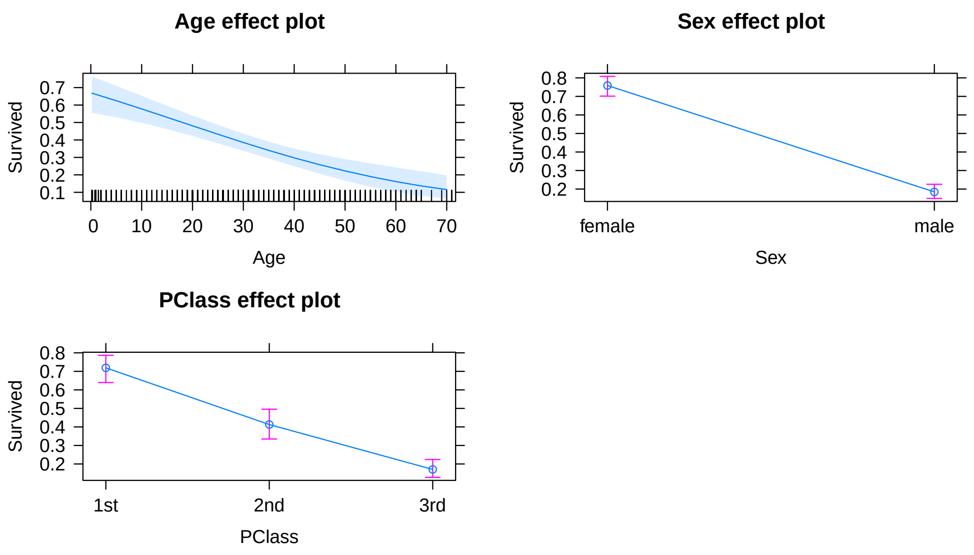


Figure 7.15: Fitted effects for the Titanic logistic regression example.

```
## PClass2nd   -1.291962   0.260076  -4.968 6.78e-07 ***
## PClass3rd   -2.521419   0.276657  -9.114  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 1025.57  on 755  degrees of freedom
## Residual deviance:  695.14  on 751  degrees of freedom
## AIC: 705.14
##
## Number of Fisher Scoring iterations: 5
```

The text `family=binomial` specifies that we wish to perform a *logistic regression*. The `summary` shows that all terms are significant. Interpreting the coefficients has to be done with care. For binary factors (such as `Sex`) they can be interpreted as a log-odds ratio, but this is beyond the scope of this text. The signs of the coefficients tell us about how the factor variable affects the probability of survival. In this example, all of them are negative so we can conclude: 1) an increase in age decreases the chance of survival, 2) being male decreases survival, 3) being in 2nd or 3rd class decrease survival with 3rd being worse than 2nd.

The function `allEffects` from the `effects` package can be used here to visualize the effects (Fig. 7.15).

```
# The 'type' argument is used to back-transform the probability
# (Try this plot without that argument for comparison)
plot(allEffects(fit11), type="response")
```

7.5.1.1 Tabulated data

Sometimes the available data are not at an individual level (as in the Titanic example above), but counts have already been aggregated into various factors.

We will first aggregate the Titanic data into a table of counts, and then show how we can still fit a `glm` for a logistic regression.

Suppose instead of age we only know the adult status of passengers.

```
titanic$AgeGrp <- factor(ifelse(titanic$Age>18, "Adult", "Child"))
```

We can then count the numbers of survivors and non-survivors,

```
# Count the number of survivors and non-survivors by various factor combinations
titanic2 <- aggregate(cbind(Survived=="yes",Survived=="no") ~ AgeGrp+Sex+PClass,
                      data=titanic, sum)
titanic2

##    AgeGrp    Sex PClass V1  V2
## 1   Adult female    1st 87   4
## 2   Child female    1st  9   1
## 3   Adult   male    1st 37  81
## 4   Child   male    1st  6   1
## 5   Adult female    2nd 58   9
## 6   Child female    2nd 17   1
## 7   Adult   male    2nd 10  99
## 8   Child   male    2nd 11   7
## 9   Adult female    3rd 27  36
## 10  Child female    3rd 19  20
## 11  Adult   male    3rd 25 157
## 12  Child   male    3rd  7  27

# Tidy up the names
names(titanic2)[4:5] <- c("survivors", "nonsurvivors")
```

Use the following example to fit a logistic regression when your data looks like the above. We will again plot the fitted effects (Fig. 7.16). As you can see, the conclusions are the same as before.

```
fit12 <- glm(cbind(survivors, nonsurvivors)~AgeGrp+Sex+PClass,
             data=titanic2, family=binomial)
summary(fit12)

##
## Call:
## glm(formula = cbind(survivors, nonsurvivors) ~ AgeGrp + Sex +
##     PClass, family = binomial, data = titanic2)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -3.1339  -1.2940   0.8257   2.0386   2.9973
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)   2.0682     0.2196   9.419  < 2e-16 ***
## AgeGrpChild   0.8478     0.2519   3.365 0.000765 ***
## Sexmale      -2.5658     0.1980 -12.960  < 2e-16 ***
## PClass2nd    -0.8771     0.2353  -3.728 0.000193 ***
## PClass3rd    -2.0428     0.2425  -8.423  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
```

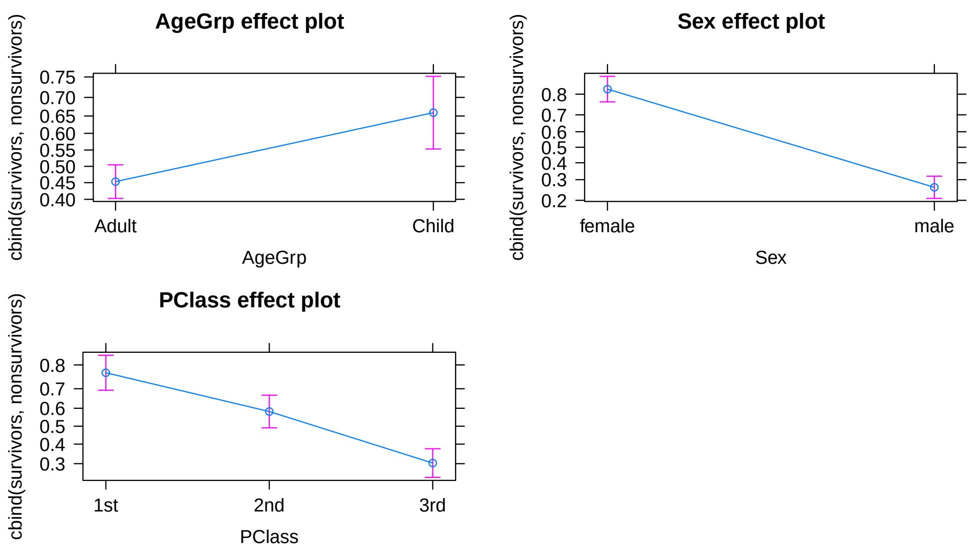


Figure 7.16: Fitted effects for the Titanic logistic regression example, when fitted using tabulated data.

```
##     Null deviance: 361.947  on 11  degrees of freedom
## Residual deviance:  48.528  on  7  degrees of freedom
## AIC: 101.01
##
## Number of Fisher Scoring iterations: 5

# Effects plot
plot(allEffects(fit12))
```

7.5.2 Poisson regression

For this example we return to the Age and Memory dataset. In this dataset, the response variable is a count of the number of words recalled by subjects using one of four different methods for memorization. When the response variable represents a count, a Poisson regression is often appropriate.

The following code performs a Poisson regression of the Age and Memory data. See Fig. 7.17 for the fitted effects.

```
# Fit a generalized linear model
fit13 <- glm(Words~Age*Process, data=memory, family=poisson)
summary(fit13)

##
## Call:
## glm(formula = Words ~ Age * Process, family = poisson, data = memory)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.2903  -0.4920  -0.1987   0.5623   2.3772
##
## Coefficients:
##                                 Estimate Std. Error z value Pr(>|z|)
## (Intercept)                      1.94591    0.11952  16.281  < 2e-16 ***
## AgeYounger                    -0.07411    0.17225  -0.430 0.667026
## ProcessRhyming                -0.01439    0.16964  -0.085 0.932406
## ProcessAdjective               0.45199    0.15289   2.956 0.003115 **
## ProcessImagery                 0.64934    0.14747   4.403 1.07e-05 ***
## ProcessIntentional             0.53900    0.15040   3.584 0.000339 ***
## AgeYounger:ProcessRhyming      0.17073    0.23942   0.713 0.475769
## AgeYounger:ProcessAdjective    0.37084    0.21335   1.738 0.082179 .
## AgeYounger:ProcessImagery      0.34675    0.20692   1.676 0.093777 .
## AgeYounger:ProcessIntentional  0.54931    0.20781   2.643 0.008210 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for poisson family taken to be 1)
##
##     Null deviance: 227.503  on 99  degrees of freedom
## Residual deviance:  60.994  on 90  degrees of freedom
## AIC: 501.32
##
## Number of Fisher Scoring iterations: 4
```

```
# Look at an ANOVA of the fitted model, and provide likelihood-ratio tests.
Anova(fit13)
```

```
## Analysis of Deviance Table (Type II tests)
##
## Response: Words
##             LR Chisq Df Pr(>Chisq)
## Age           20.755  1  5.219e-06 ***
## Process      137.477  4  < 2.2e-16 ***
## Age:Process    8.277  4    0.08196 .
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
# Plot fitted effects
plot(allEffects(fit13))
```

Remember that when we fit this as a linear model (using `lm`), we found that the interaction term was highly significant. This time, when we used Poisson regression, the interaction is no longer significant. However, the default link for Poisson regression is $\log$. This means that the mean μ in this case depends on the factors `Age` and `Process` in a multiplicative fashion, whereas in the linear model (using `lm`) it was additive without the interaction. In other words, we have now modelled the interaction implicitly by using the $\log$ link function.

Note that in example above we use the `anova` function with the option `test="LRT"`, which allows us to perform a *Likelihood Ratio Test* (LRT). This is appropriate for GLMs because the usual F-tests may not be inaccurate when the distribution is not normal.

There are diagnostic plots for GLMs just like linear models, but these are beyond the scope of this text.

Further reading An excellent book on linear modelling, including many tools that we did not cover in this chapter is: Fox, John, and Sanford Weisberg. *An R companion to applied regression*. Sage, 2010. This book describes the `car` package.

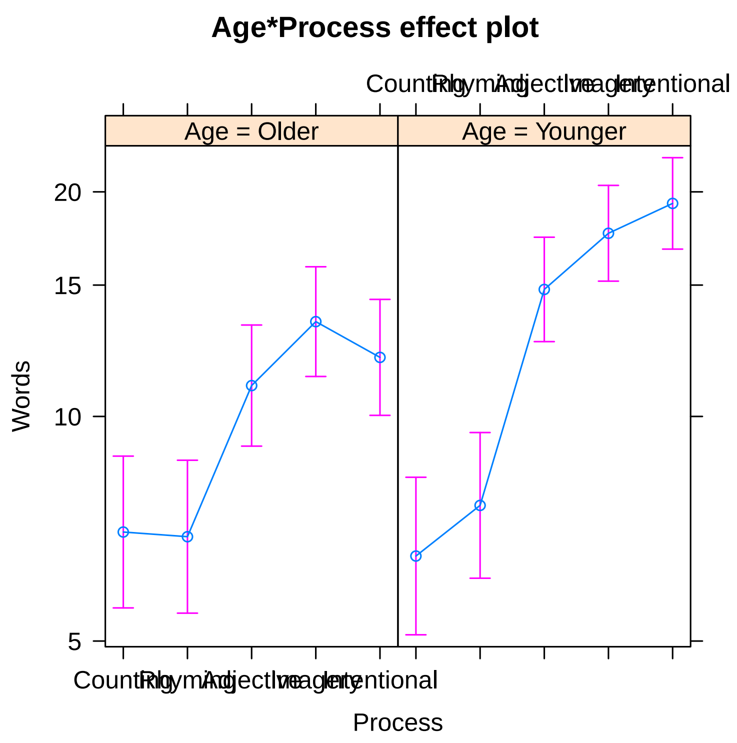


Figure 7.17: Fitted effects for a Poisson regression of the Age and Memory data. Note the log scale of the y-axis.

7.6 Functions used in this chapter

For functions not listed here, please refer to the index at the end of this book.

Function	What it does	Example use
`lm`	Linear models, including regression models, ANOVAs, and ANCOVAs.	`Throughout this chapter`
`glm`	Generalized linear models, similar to `lm` but allows various non-normally distributed errors.	`Section 7.5, p. 204`
`emmeans`	From the `emmeans` package. Very useful for multiple comparisons. Interpret output with the `summary`, `pairs` and `pwpp` functions, plus `cld` from the `multcomp` package.	`Section 7.2.1.1, p. 179, p. 183, p. 201`
`anova`	Using `anova` on two models performs a likelihood-ratio test. We do **not** recommend its use on a single model, use `Anova` instead	`Throughout this chapter`

Function	**What it does**	**Example use**
`Anova`	From the `car` package. Analysis of variance table for fitted models, showing marginal tests for main effects and interactions (order of variables does not matter.)	`Anova(model1)`
`drop1`	Tests of significance for main effects and interactions in a fitted model, dropping one term at a time. Equivalent to `Anova` with default settings.	`drop1(model1, test="F")`
`AIC`	Akaike's Information Criterion – lower is better. Can also be used on many fitted models at once for comparison.	`AIC(model1, model2)`
`allEffects`	From the `effects` package. Convenient function to estimate and plot 'effects' from a fitted model – highly recommended to avoid misinterpretation of fitted models.	`plot(allEffects(model1))`
`visreg`	From the `visreg` package. Similar to `allEffects`, makes more attractive plots but slightly less flexible overall.	`visreg(model1, "xvar", by="facvar")`

7.7 Exercises

In these exercises, we use the following colour codes:

> 🟩 **Easy**: make sure you complete some of these before moving on. These exercises will follow examples in the text very closely.
>
> ♦ **Intermediate**: a bit harder. You will often have to combine functions to solve the exercise in two steps.
>
> ▲ **Hard**: difficult exercises! These exercises will require multiple steps, and significant departure from examples in the text.

We suggest you complete these exercises in an **R** markdown file. This will allow you to combine code chunks, graphical output, and written answers in a single, easy-to-read file.

7.7.1 One-way ANOVA

1. 🟩 For the Titanic data (see Section A.12, p. 251), use a one-way ANOVA to compare the average passenger age by passenger class. (Note: by default, `lm` will delete all observations where `Age` is missing.)
2. 🟩 For the Age and Memory data (Section A.11, p. 251), make a subset of the `Older` subjects, and conduct a one-way ANOVA to compare words remembered by memory technique.

7.7.2 Two-way ANOVA

1. 🟩 Using the pupae dataset, fit a two-way ANOVA to compare `PupalWeight` to `Gender` and `CO2_treatment`. Which main effects are significant? After reading in the pupae data, make sure to convert `Gender` and `CO2_treatment` to a factor first (see Section 3.2, p. 59).
2. 🟩 Is there an interaction between `Gender` and `CO2_treatment`?
3. 🟩 Repeat the above using `T_treatment` instead of `CO2_treatment`.
4. ♦ Recall Exercise 6.5.6 (p. 175). Perform a two-way ANOVA to test irrigation and fertilisation effects on tree diameter. Compare the results. *Hint*: the treatments are not coded by 'irrigation' or 'fertilization', you must first make two new variables based on the `treat` variable in the dataframe.

7.7.3 Multiple regression

1. ♦ Using the Pulse data (see Section A.9, p. 250) fit a multiple linear regression of `Pulse1` against `Age`, `Weight` and `Height` (add the variables to the model in that order).
2. 🟩 Are any terms significant at the 5% level?
3. 🟩 Generate some diagnostic plots of the fitted models.

7.7.4 Linear Model with factor and numeric variables

1. 🟩 Repeat exercise 7.7.3, and also include the factor `Exercise`. You will need to first convert `Exercise` to a factor as it is stored numerically in the CSV file. Does adding `Exercise` improve the model?

2. ♦ Using the same data, fit a model of `Pulse2` as a function of `Pulse1` and `Ran` as main effects only (Note: convert `Ran` to a factor first). Use the `effects` package to inspect the model fit, and help understand the estimated coefficients in the model.
3. ♦ Now add the interaction between `Pulse1` and `Ran`. Is it significant? Also look at the effects plot, how is it different from the model without an interaction?
4. ■ Add (factor versions of) `Alcohol`, `Smokes` and `Exercise` to this model, and inspect whether the model improved.

7.7.5 Logistic regression

1. ■ Using the Pulse data once more, build a model to see if `Pulse2` can predict whether people were in the `Ran` group. Make sure that `Ran` is coded as a factor.
2. ■ The `visreg` package is very helpful in visualizing the fitted model. For the logistic regression you just fit , run the following code and make sure you understand the output. (This code assumes you called the object `fit6`, if not change `fit6` to the name you used.)

```
library(visreg)
visreg(fit6, scale="response")
```

7.7.6 Generalized linear model (GLM)

1. First run the following code to generate some data,

```
len <- 21
x <- seq(0,1,length=len)
y <- rpois(len,exp(x-0.5))
```

2. ■ Fit a Poisson GLM to model `y` on `x`. Is `x` significant in the model?
3. ■ Repeat above with a larger sample size (e.g., `len <- 101`). Compare the results.
4. ♦ The `memory` data were analysed assuming a normal error distribution in Section 7.2 and using a Poisson error distribution in Section 7.5, and each approach resulted in a different outcome for the significance of the interaction term. The participants in the study were asked to remember up to 27 words, not an unlimited number, and some of the participants were remembering close to this upper limit. Therefore, it may make sense to think of the response as consisting of a number of 'successes' and a number of 'failures', as we do in a logistic regression. Use `glm` to model the response using a binomial error distribution. Refer to Section 7.5.1.1 (p. 207) for a similar example.

Chapter 8

Functions, lists and loops

8.1 Writing simple functions

We have already used many built-in functions throughout this tutorial, but you can become very efficient at complex data tasks when you write your own simple functions. Writing your own functions can help with tasks that are carried out many times, which would otherwise result in a lot of code.

For example, suppose you frequently convert units from pounds to kilograms. It would be useful to have a function that does this, so you don't have to type the conversion factor every time. This is also good practice, as it reduces the probability of making typos.

```
# This function takes a 'weight' argument and multiplies it with some number
# to return kilograms.
poundsToKg <- function(weight){
  weight * 0.453592
}
```

We can use this `function` just like any other in **R**, for example, let's convert 'weight' to kilograms in the weightloss data (Section A.5).

```
# Read data
weightloss <- read.csv("weightloss.csv")

# Convert weight to kg.
weightloss$Weight <- poundsToKg(weightloss$Weight)
```

Let's write a function for the standard error of the mean, a function that is not built-in in **R**.

```
# Compute the standard error of the mean for a vector
SEmean <- function(x){
  se <- sd(x) / sqrt(length(x))
  return(se)
}
```

Here, the `function` SEmean takes one 'argument' called `x` (i.e., input), which is a numeric vector. The standard error for the mean is calculated in the first line, and stored in an object called `se`, which is then returned as output. We can now use the function on a numeric vector like this:

```
# A numeric vector
unifvec <- runif(10, 1,2)
```

```
# The sample mean
mean(unifvec)

## [1] 1.506451

# Standard error for the mean
SEmean(unifvec)

## [1] 0.09723349
```

Try this yourself You can use functions that you defined yourself just like any other function, for example in `summaryBy`. First read in the `SEmean` function defined in the example above, and then use the cereal data to calculate the mean and SE of `rating` by `Manufacturer` (or use data of your choosing).

Functions with multiple arguments

Functions can also have multiple arguments. The following very simple function takes two numbers, and finds the absolute difference between them, using `abs`.

```
# Define function
absDiff <- function(num1,num2)abs(num1 - num2)

# Test it with two numbers:
absDiff(5,1)

## [1] 4

# As in many functions in R, you can also give multiple values
# as an argument.
# The following returns the absolute difference between
# 1 and 3, then 5 and 6, and 9 and 0 (in that order).
absDiff(c(1,5,9), c(3,6,0))

## [1] 2 1 9
```

Functions can return many results

What if a function should return not just one result, as in the examples above, but many results?

For example, this function computes the standard deviation and standard error of a vector, and returns both stored in a vector. Note that we also use the `SEmean` function, which we defined above.

```
# An function that computes the SE and SD of a vector
seandsd <- function(x){

  seresult <- SEmean(x)
  sdresult <- sd(x)

  # Store results in a vector with names
  vec <- c(seresult, sdresult)
  names(vec) <- c("SE","SD")

return(vec)
}
```

```
# Test it:
x <- rnorm(100, mean=20, sd=4)
seandsd(x)

##        SE        SD
## 0.4343727 4.3437272
```

Functions without arguments

Sometimes, a function takes no arguments (input) at all. Consider this very helpful example.

```
sayhello <- function()message("Hello!")

sayhello()

## Hello!
```

We will return to defining our own functions when we look at applying functions many times to sections of a dataframe (Section 8.2.2).

8.2 Working with lists

Sofar, we have worked a lot with vectors, with are basically strings of numbers or bits of text. In a vector, each element has to be of the same data type. Lists are a more general and powerful type of vector, where each element of the `list` can be anything at all. This way, lists are a very flexible type of object to store a lot of information that may be in different formats.

Lists can be somewhat daunting for the beginning `R` user, which is why most introductory texts and tutorials skip them altogether. However, with some practice, lists can be mastered from the start. Mastering a few basic skills with lists can really help increase your efficiency in dealing with more complex data analysis tasks.

To make a list from scratch, you simply use the `list` function. Here is a list that contains a numeric vector, a character vector, and a dataframe:

```
mylist <- list(a=1:10, txt=c("hello","world"), dfr=data.frame(x=c(2,3,4),y=c(5,6,7)))
```

Indexing lists

To extract an element from this list, you may do this by its name ('a','txt' or 'dfr' in this case), or by the element number (1,2,3). For lists, we use a double square bracket for indexing. Consider these examples,

```
# Extract the dataframe:
mylist[["dfr"]]

##   x y
## 1 2 5
## 2 3 6
## 3 4 7

# Is the same as:
mylist$dfr
```

```
##   x y
## 1 2 5
## 2 3 6
## 3 4 7
# Extract the first element:
mylist[[1]]
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Note that in these examples, the contents of the elements of the list are returned (for 'dfr', a dataframe), but the result itself is not a list anymore. If we select multiple elements, the result should still be a list. To do this, use the single square bracket.

Look at these examples:

```
# Extract the 'a' vector, result is a vector:
mylist[['a']]
##  [1]  1  2  3  4  5  6  7  8  9 10
# Extract the 'a' vector, result is a list:
mylist['a']
## $a
##  [1]  1  2  3  4  5  6  7  8  9 10
# Extract multiple elements (result is still a list):
mylist[2:3]
## $txt
## [1] "hello" "world"
##
## $dfr
##   x y
## 1 2 5
## 2 3 6
## 3 4 7
```

Converting lists to dataframes or vectors

Although lists are the most flexible way to store data and other objects in larger, more complex, analyses, ultimately you would prefer to output as a dataframe or vector.

Let's look at some examples using `do.call(rbind,...)` and `unlist`.

```
# A list of dataframes:
dfrlis <- list(data1=data.frame(a=1:3,b=2:4), data2=data.frame(a=9:11,b=15:17))
dfrlis
## $data1
##   a b
## 1 1 2
## 2 2 3
## 3 3 4
##
## $data2
##    a  b
## 1  9 15
## 2 10 16
## 3 11 17

# Since both dataframes in the list have the same number of columns and names,
# we can 'successively row-bind' the list like this:
do.call(rbind, dfrlis)

##          a  b
## data1.1  1  2
## data1.2  2  3
## data1.3  3  4
## data2.1  9 15
## data2.2 10 16
## data2.3 11 17

# A list of vectors:
veclis <- list(a=1:3, b=2:4, f=9:11)

# In this case, we can use the 'unlist' function, which will
# successively combine the three vectors into one:
unlist(veclis)

## a1 a2 a3 b1 b2 b3 f1 f2 f3
##  1  2  3  2  3  4  9 10 11
```

In real-world applications, some trial-and-error will be necessary to convert lists to more pretty formats.

Combining lists

Combining two lists can be achieved using `c()`, like this:

```
veclis <- list(a=1:3, b=2:4, f=9:11)
qlis <- list(q=17:15)
c(veclis,qlis)

## $a
## [1] 1 2 3
##
## $b
## [1] 2 3 4
##
## $f
## [1]  9 10 11
##
## $q
## [1] 17 16 15

# But be careful when you like to quickly add a vector
# the 'veclis'. You must specify list() like this
veclis <- c(veclis, list(r=3:1))
```

Extracting output from built-in functions

One reason to gain a better understanding of lists is that many built-in functions return not just single numbers, but a diverse collection of outputs, organized in lists. Think of the linear model function (`lm`), it returns a lot of things at the same time (not just the p-value).

Let's take a closer look at the `lm` output to see if we can extract the adjusted R^2.

```
# Read data
allom <- read.csv("Allometry.csv")

# Fit a linear model
lmfit <- lm(height ~ diameter, data=allom)

# And save the summary statement of the model:
lmfit_summary <- summary(lmfit)

# We already know that simply typing 'summary(lmfit)' will give
# lots of text output. How to extract numbers from there?
# Let's look at the structure of lmfit:
str(lmfit_summary)

## List of 11
##  $ call         : language lm(formula = height ~ diameter, data = allom)
##  $ terms        :Classes 'terms', 'formula'  language height ~ diameter
##   .. ..- attr(*, "variables")= language list(height, diameter)
##   .. ..- attr(*, "factors")= int [1:2, 1] 0 1
##   .. .. ..- attr(*, "dimnames")=List of 2
##   .. .. .. ..$ : chr [1:2] "height" "diameter"
##   .. .. .. ..$ : chr "diameter"
##   .. ..- attr(*, "term.labels")= chr "diameter"
##   .. ..- attr(*, "order")= int 1
....

# The output of lm is a list, so we can look at the names of # that list as well:
names(lmfit_summary)

##  [1] "call"          "terms"          "residuals"     "coefficients"
##  [5] "aliased"       "sigma"          "df"            "r.squared"
##  [9] "adj.r.squared" "fstatistic"     "cov.unscaled"
```

So, now we can extract results from the summary of the fitted regression. Also look at the help file `?summary.lm`, in the section 'Values' for a description of the fields contained here.

To extract the adjusted R^2, for example:

```
lmfit_summary[["adj.r.squared"]]

## [1] 0.7639735

# Is the same as:
lmfit_summary$adj.r.squared

## [1] 0.7639735
```

This sort of analysis will be very useful when we do many regressions, and want to summarize the results in a table.

Try this yourself Run the code in the above examples, and practice extracting some other elements from the linear regression. Compare the output to the summary of the `lm` fit (that is, compare it to what `summary(lmfit)` shows on screen).

8.2.1 Creating lists from dataframes

For more advanced analyses, it is often necessary to repeat a particular analysis many times, for example for sections of a dataframe.

Using the `allom` data for example, we might want to split the dataframe into three dataframes (one for each species), and repeat some analysis for each of the species. One option is to make three subsets (using `subset`), and repeating the analysis for each of them. But what if we have hundreds of species?

A more efficient approach is to `split` the dataframe into a list, so that the first element of the list is the dataframe for species 1, the 2nd element species 2, and so on. In case of the allom dataset, the resulting list will have three components.

Let's look at an example on how to construct a list of dataframes from the allom dataset, one per species:

```
# Read allom data and make sure 'species' is a factor:
allom <- read.csv("Allometry.csv")
is.factor(allom$species)

## [1] TRUE

# The levels of the factor variable 'species'
levels(allom$species)

## [1] "PIMO" "PIPO" "PSME"

# Now use 'split' to construct a list:
allomsp <- split(allom, allom$species)

# The length of the list should be 3, with the names equal to the
# original factor levels:
length(allomsp)

## [1] 3

names(allomsp)

## [1] "PIMO" "PIPO" "PSME"
```

Try this yourself Run the code in the above example, and confirm that `allomsp[[2]]` is identical to taking a subset of `allom` of the second species in the dataset (where 'second' refers to the second level of the factor variable `species`, which you can find out with `levels`).

Let's look at an example using the `hydro` data. The data contains water levels of a hydrodam in Tasmania, from 2005 to 2011 (see Section A.3).

```
# Read hydro data, and convert Date to a proper date class.
hydro <- read.csv("hydro.csv")

library(lubridate)
hydro$Date <- dmy(as.character(hydro$Date))
hydro$year <- year(hydro$Date)

# Look at the Date range:
range(hydro$Date)

## [1] "2005-08-08" "2011-08-08"

# Let's get rid of the first and last years (2005 and 2011) since they are incomplete
```

```
hydro <- subset(hydro, !year %in% c(2005,2011))

# Now split the dataframe by year. This results in a list, where every
# element contains the data for one year:
hydrosp <- split(hydro, hydro$year)

# Properties of this list:
length(hydrosp)

## [1] 5

names(hydrosp)

## [1] "2006" "2007" "2008" "2009" "2010"
```

To extract one element of the two lists that we created (`allomsp` and `hydrosp`), recall the section on indexing lists.

8.2.2 Applying functions to lists

We will introduce two basic tools that we use to apply functions to each element of a list: `sapply` and `lapply`. The `lapply` function always returns a list, whereas `sapply` will attempt to simplify the result. When the function returns a single value, or a vector, `sapply` can often be used. In practice, try both and see what happens!

Using `sapply`

First let's look at some simple examples:

```
# Let's make a simple list with only numeric vectors (of varying length)
numlis <- list(x=1000, y=c(2.1,0.1,-19), z=c(100,200,100,100))

# For the numeric list, let's get the mean for every element, and count
# the length of the three vectors.
# Here, sapply takes a list and a function as its two arguments,
# and applies that function to each element of the list.
sapply(numlis, mean)

##      x      y      z
## 1000.0   -5.6  125.0

sapply(numlis, length)

## x y z
## 1 3 4
```

You can of course also define your own functions, and use them here. Let's look at another simple example using the `numlis` object defined above.

For example,

```
# Let's find out if any diameters are duplicated in the allom dataset.
# A function that does this would be the combination of 'any' and 'duplicated',
anydup <- function(vec)any(duplicated(vec))
# This function returns TRUE or FALSE

# Apply this function to numlis (see above):
```

```
sapply(numlis, anydup)

##     x     y     z
## FALSE FALSE  TRUE

# You can also define the function on the fly like this:
sapply(numlis, function(x)any(duplicated(x)))

##     x     y     z
## FALSE FALSE  TRUE
```

Now, you can use any function in `sapply` as long as it returns a single number based on the element of the list that you used it on. Consider this example with `strsplit`.

```
# Recall that the 'strsplit' (string split) function usually returns a list of values.
# Consider the following example, where the data provider has included the units in
# the measurements of fish lengths. How do we extract the number bits?
fishlength <- c("120 mm", "240 mm", "159 mm", "201 mm")

# Here is one solution, using strsplit
strsplit(fishlength," ")

## [[1]]
## [1] "120" "mm"
##
## [[2]]
## [1] "240" "mm"
##
## [[3]]
## [1] "159" "mm"
##
## [[4]]
## [1] "201" "mm"

# We see that strsplit returns a list, let's use sapply to extract only
# the first element (the number)
splitlen <- strsplit(fishlength," ")
sapply(splitlen, function(x)x[1])

## [1] "120" "240" "159" "201"

# Now all you need to do is use 'as.numeric' to convert these bits of text to numbers.
```

The main purpose of splitting dataframes into lists, as we have done above, is so that we can save time with analyses that have to be repeated many times. In the following examples, you must have already produced the objects `hydrosp` and `allomsp` (from examples in the previous section).Both those objects are *lists of dataframes*, that is, each element of the list is a dataframe in itself. Let's look at a few examples with `sapply` first.

```
# How many observations per species in the allom dataset?
sapply(allomsp, nrow)

## PIMO PIPO PSME
##   19   22   22

# Here, we applied the 'nrow' function to each separate dataframe.
# (note that there are easier ways to find the number of observations per species!,
# this is just illustrating sapply.)

# Things get more interesting when you define your own functions on the fly:
```

```
sapply(allomsp, function(x)range(x$diameter))

##       PIMO  PIPO  PSME
## [1,]  6.48  4.83  5.33
## [2,] 73.66 70.61 69.85

# Here, we define a function that takes 'x' as an argument:
# sapply will apply this function to each element of the list,
# one at a time. In this case, we get a matrix with ranges of the diameter per species.

# How about the correlation of two variables, separate by species:
sapply(allomsp, function(x)cor(x$diameter, x$height))

##      PIMO      PIPO      PSME
## 0.9140428 0.8594689 0.8782781

# For hydro, find the number of days that storage was below 235, for each year.
sapply(hydrosp, function(x)sum(x$storage < 235))

## 2006 2007 2008 2009 2010
##    0   18    6    0    0
```

Using `lapply`

The `lapply` function is much like `sapply`, except it always returns a list.

For example,

```
# Get a summary of the hydro dataset by year:
lapply(hydrosp, summary)

## $`2006`
##       Date               storage           year
##  Min.   :2006-01-02   Min.   :411.0   Min.   :2006
##  1st Qu.:2006-04-01   1st Qu.:449.5   1st Qu.:2006
##  Median :2006-06-29   Median :493.0   Median :2006
##  Mean   :2006-06-29   Mean   :514.3   Mean   :2006
##  3rd Qu.:2006-09-26   3rd Qu.:553.8   3rd Qu.:2006
##  Max.   :2006-12-25   Max.   :744.0   Max.   :2006
##
## $`2007`
....
```

Suppose you have multiple similar datasets in your working directory, and you want to read all of these into one list, use `lapply` like this (run this example yourself and inspect the results).

```
# Names of your datasets:
filenames <- c("pupae.csv","pupae.csv","pupae.csv")
# (This toy example will read the same file three times).

# Read all files into one list,
alldata <- lapply(filenames, read.csv)

# Then, if you are sure the datasets have the same number of columns and names,
# use do.call to collapse the list:
dfrall <- do.call(rbind, alldata)
```

Try this yourself Recall the use of `dir` to list files, and even to find files that match a specific pattern (see Section 1.8). Read all CSV files in your working directory (or elsewhere) into a single list, and count the number of rows for each dataframe.

Finally, we can use `lapply` to do all sorts of complex analyses that return any kind of object. The use of `lapply` with lists ensures that we can organize even large amounts of data in this way.

Let's do a simple linear regression of log(leafarea) on log(diameter) for the `allom` dataset, by species:

```
# Run the linear regression on each element of the list, store in a new object:
lmresults <- lapply(allomsp, function(x)lm(log10(leafarea) ~ log10(diameter), data=x))

# Now, lmresults is itself a list (where each element is an object as returned by lm)
# We can extract the coefficients like this:
sapply(lmresults, coef)
```

```
##                       PIMO       PIPO       PSME
## (Intercept)     -0.3570268 -0.7368336 -0.3135996
## log10(diameter)  1.5408859  1.6427773  1.4841361
```

```
# This shows the intercept and slope by species.
# Also look at (results not shown):
# lapply(lmresults, summary)

# Get R2 for each model. First write a function that extracts it.
getR2 <- function(x)summary(x)$adj.r.squared
sapply(lmresults, getR2)
```

```
##      PIMO      PIPO      PSME
## 0.8738252 0.8441844 0.6983126
```

Try this yourself Try to fully understand the difference between `sapply` and `lapply` by using `lapply` in some of the examples where we used `sapply` (and vice versa).

Try this yourself The above example shows a general way in which you can analyze many fitted models at once. A more convenient method to summarize many models is to us the `broom` package, and in particular the `glance` function. Try this code (install the `broom` package first):

```
library(broom)
sapply(lmresults, glance)
```

8.3 Loops

Loops can be useful when we need to repeat certain analyses many times, and it is difficult to achieve this with `lapply` or `sapply`. To understand how a `for` loop works, look at this example:

```
for(i in 1:5){
  message(i)
}
```

```
## 1

## 2

## 3
## 4

## 5
```

Here, the bit of code between is executed five times, and the object `i` has the values 1,2,3,4 and 5, in that order. Instead of just printing `i` as we have done above, we can also index a vector with this object:

```
# make a vector
myvec <- round(runif(5),1)

for(i in 1:length(myvec)){
  message("Element ", i, " of the vector is: ", myvec[i])
}

## Element 1 of the vector is:  0.8

## Element 2 of the vector is:  0

## Element 3 of the vector is:  0.8

## Element 4 of the vector is:  0.8

## Element 5 of the vector is:  0.8
```

Note that this is only a toy example: the same result can be achieved by simply typing `myvec`.

Now let's look at a useful application of a `for` loop: producing multiple plots in a `pdf`, using the `allomsp` object we created earlier.

This bit of code produces a `pdf` in your current working directory. If you can't find it, recall that you can use `getwd()` to get the current working directory.

```
# Open a pdf to send the plots to:
pdf("Allom plot by species.pdf", onefile=TRUE)
for(i in 1:3){
  with(allomsp[[i]],
       plot(diameter, leafarea, pch=15, xlim=c(0,80), ylim=c(0,450),
            main=levels(allom$species)[i]))
}
# Close the pdf (important!)
dev.off()
```

Here, we create three plots (`i` goes from 1 to 3), every time using a different element of the list `allomsp`. First, `i` will have the value 1, so that we end up using the dataframe `allomsp[[1]]`, the first element of the list. And so on. Take a look at the resulting PDF to understand how the code works.

Note: On windows (with Adobe reader) If the pdf ('Allom plot by species.pdf') is open, the above will fail. If you try this anyway, close the pdf and try again. You may have to run the command `dev.off()` another time to make sure the device is ready.

Another way to achieve the same result is to avoid splitting the dataframe into a list first, and simply take subsets on the fly. Consider this template (make your own working example based on any dataset).

We assume here you have a dataframe called 'dataset' with a factor 'species', for which you want to create separate plots of Y vs. X.

```
pdf("somefilename.pdf", onefile=TRUE)
for(lev in levels(dataset$species)){

  with(subset(dataset, species==lev),
       plot(X,Y,
            main=as.character(lev)))
}
dev.off()
```

Try this yourself Apply the above template to a dataset of your choice, to make a multi-page pdf with a scatter plot on each page.

An example line plot

Another important use of for-loops in plotting is when you want to plot multiple lines in one figure, based on levels of a factor. The default plot in **R** is not smart enough to do this correctly. In the following example, we use the hydro data to see if the pattern in storage is similar between years (producing Fig. 8.1).

```
# First make the hydrosp object from the example above.
# This is a list, where every element is a dataframe for a year of data.
# First we plot the first year, then add subsequent years with a for loop.
# We plot storage versus day of year
# (conveniently extracted with yday from the lubridate package)
# We also first setup a range of colours.
lineCols <- rainbow(5)
with(hydrosp[[1]], plot(yday(Date), storage, type='l', col=lineCols[1],
                        lwd=2, ylim=c(0,1000),
                        xlab="Day of year", ylab="Dam storage (kWh)"))
for(i in 2:5)with(hydrosp[[i]], lines(yday(Date), storage, col=lineCols[i], lwd=2))
legend("bottomleft", names(hydrosp), col=lineCols, lty=1, lwd=2, cex=0.6)
```

8.4 Advanced working example

In this example, we use lists, `lapply`, `sapply` as well as loops. We also introduce `fitdistr` from the `MASS` package, and use `curve` to add curves to a histogram.

Let's fit a weibull distribution to the vessel diameter data, separately for the base and apex positions (see Section A.13). We will also visualize the results, and print the coefficients of the fit.

Note that in the following example, you could also simply make two subsets and repeat the analysis for both. But this way of working also applied to datasets where you have hundreds of species.

```
# Read raw data
vessel <- read.csv("vessel.csv")

# Split the dataframe
vesselsp <- split(vessel,vessel$position)

# Load MASS package
library(MASS)

##
## Attaching package:  'MASS'

## The following object is masked from 'package:dplyr':
##
```

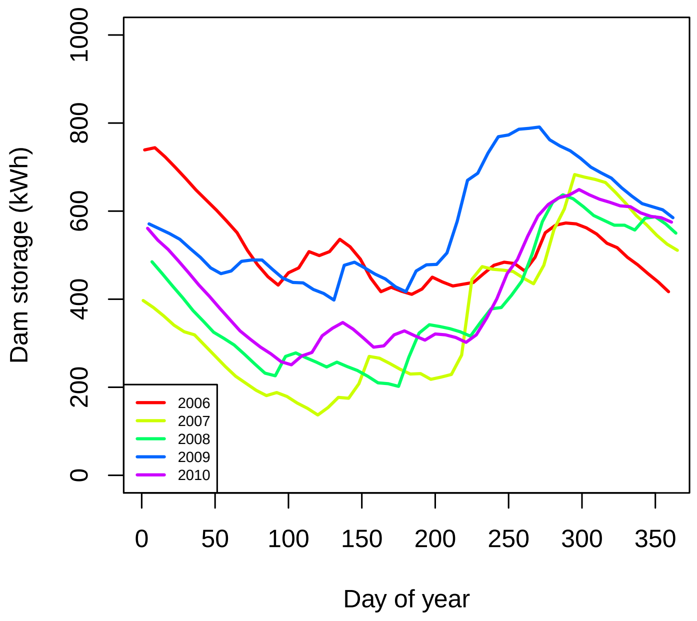


Figure 8.1: A line plot for the hydro data.

```
##    select

# Fit the weibull (results are stored in a list 'weibfits' containing all outputs).
weibfits <- lapply(vesselsp, function(x)fitdistr(x$vesseldiam, "weibull"))

# To look at the standard print output for the first one:
weibfits[[1]]




# But if you look at str(weibfits), you will notice the output is a list with many
# components. We will extract 'estimate' which gives the two parameters of the weibull:
weibcoefs <- sapply(weibfits, function(x)x$estimate)

# And look at the results:
weibcoefs
```

Next, we will plot the two distributions again (as we did in Section 4.5.1), together with a curve of the fitted weibull.

This code produces Fig. 8.2.

```
# First define a plotting function that makes a histogram of 'vesseldiam',
# and adds a curve with a weibull distribution, given the shape and scale parameters:
```

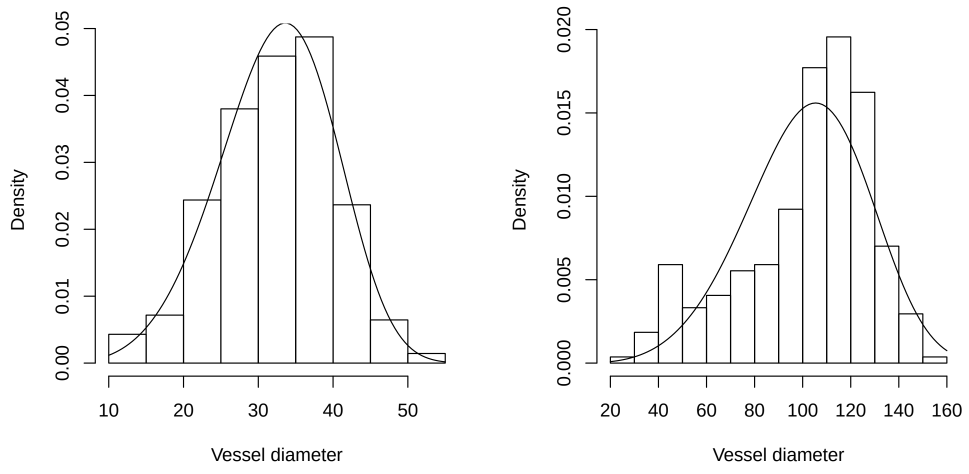


Figure 8.2: Two histograms of the vessel diameter data, with fitted Weibull curves

```
plotHistWeib <- function(dfr, shape, scale){
    hist(dfr$vesseldiam, freq=FALSE, main="", xlab="Vessel diameter")
    curve(dweibull(x, shape, scale), add=TRUE)
}

# Setup a plotting region with two plots:
par(mfrow=c(1,2))

# Loop through the vesselsp list, plot histogram and weibull curve for each:
for(i in 1:length(vesselsp)){
  plotHistWeib(vesselsp[[i]], shape=weibcoefs[1,i], scale=weibcoefs[2,i])
}
```

Note: we index the `weibcoefs` object as if it were a dataframe, with the order ROWS, COLUMNS between the square brackets. In fact, it is a matrix but it behaves as a dataframe.

8.5 Functions used in this chapter

For functions not listed here, please refer to the index at the end of this book.

Function	What it does	Example use
`function`	Define a function	`Section 8.1`
`return`	Inside a function definition, return an object	
`message`	Print a text message to the screen	`message("Hello!")`
`list`	Define a list	`list(a=1, b=rnorm(100))`
`unlist`	For a list of vectors, turn the list into a single long vector	`unlist(list(a=1:3, b=4:6))`
`do.call`	Apply a function given a list of arguments, most often used to successively row-bind dataframes in a list	`Section 8.2`
`split`	Turn a dataframe into a list, split by levels of a factor variable	`split(allom, allom$species)`
`sapply`	Apply a function to each element of a list, try to simplify the result (into a vector or matrix)	
`lapply`	Apply a function to each element of a list, but do not try to simplify the result – always return a list	
`strsplit`	Split a character vector into pieces separated by some specified character	`strsplit("a/b/c", "/")`

8.6 Exercises

In these exercises, we use the following colour codes:

> 🟩 **Easy**: make sure you complete some of these before moving on. These exercises will follow examples in the text very closely.
>
> ♦ **Intermediate**: a bit harder. You will often have to combine functions to solve the exercise in two steps.
>
> ▲ **Hard**: difficult exercises! These exercises will require multiple steps, and significant departure from examples in the text.

We suggest you complete these exercises in an **R** markdown file. This will allow you to combine code chunks, graphical output, and written answers in a single, easy-to-read file.

8.6.1 Writing functions

1. 🟩 Write a function that adds two numbers, and divides the result by 2.
2. 🟩 You learned in Section 3.5 that you can take subset of a string using the `substr` function. First, using that function to extract the first 2 characters of a bit of text. Then, write a function called `firstTwoChars` that extracts the first two characters of any bit of text.
3. 🟩 Write a function that checks if there are any missing values in a vector (using `is.na` and `any`). The function should return `TRUE` if there are missing values, and `FALSE` if not.
4. ♦ Improve the function so that it tells you which of the values are missing, if any (*Hint:*use the `which` function).
5. ♦ The function `readline` can be used to ask for data to be typed in. First, figure out how to use `readline` by reading the corresponding help file. Then, construct a function called `getAge` that asks the user to type his/her age. (*Hint:* check the examples in the `readline` help page).
6. ▲ Look at the calculations for a confidence interval of the mean in the example in Section 5.3 (p. 134). Write a function that returns the confidence interval for a vector. The function should have two inputs: the vector, and the desired 'alpha'.
7. ▲ Recall the functions `head` and `tail`. Write a function called `middle` that shows a few rows around (approx.) the 'middle' of the dataset. *Hint:* use `nrow`, `print`, and possibly `floor`.

8.6.2 Working with lists

First read the following list:

```
veclist <- list(x=1:5, y=2:6, z=3:7)
```

1. 🟩 Using `sapply`, check that all elements of the list are vectors of the same length. Also calculate the sum of each element.
2. ♦ Add an element to the list called 'norms' that is a vector of 10 numbers drawn from the standard normal distribution (recall Section 5, p. 130).
3. ♦ Using the pupae data (Section A.6, p. 248), use a t-test to find if PupalWeight varies with temperature treatment, separate for the two CO_2 treatments (so, do two t-tests). Use `split` and `lapply`.

4. 🟩 / ♦ Recall the exercise in Section 4.9.4 (p. 129). First read the data. Then, split the data by `species`, to produce a list called `coweeta_sp`. Keep only those species that have at least 10 observations. (*Hint:* first count the number of observations per species, save that as a vector, find which are at least 10, and use that to subscript the list.) If you don't know how to do this last step, skip it and continue to the next item.
5. 🟩 Using the split Coweeta data, perform a linear regression of `log10(biomass)` on `log10(height)`, separately by species. (*Hint:* recall section 8.2.2, p. 222).
6. ♦ Run this code to get two vectors:

   ```
   x <- rnorm(100)
   y <- x + rnorm(100)
   ```

   Run a linear regression y = f(x), save the resulting object. Look at the structure of this object, and note the names of the elements. Extract the residuals and make a histogram.
7. ▲ From question 6, write a function that takes an `lm` object as an argument, and plots a histogram of the residuals.

8.6.3 Using functions to make many plots

1. ♦ Read the cereal data. Create a subset of data where the `Manufacturer` has at least two observations (use `table` to find out which you want to keep first). Don't forget to drop the empty factor level you may have created!
2. ♦ Make a single PDF with six plots, with a scatter plot between potassium and fiber for each of the six (or seven?) Manufacturers. (*Hint:* ook at the template for producing a PDF with multiple pages at the bottom of Section 8.3, p. 225).
3. ▲ Recall that we can use `points` to add points or lines to a current plot. See Section 4.4.7 (p. 108) for an example using the Dutch election data. Read the data (and convert the Date variable!).
4. ▲ Write a function that adds lines for each of the parties in the election data to a plot. First set up an empty plot using,

   ```
   with(election, plot(Date, VVD, type='n', ylim=c(0,40)))
   ```

   Then carefully inspect the example in Section 4.4.7 (p. 108) to see how you can write a function that adds a line for one party (e.g. 'SP') to that plot.
5. ▲ Loop through all columns of the election data, add a line for each column to the plot.

8.6.4 Monthly weather plots

1. ♦ For the HFE weather dataset (Section A.10, p. 251), write a function that makes a scatter plot between PAR and VPD.
2. ♦ Then, split the dataset by month (recall Section 8.2.1, p. 221), and make twelve such scatter plots. Save the result in a single PDF, or on one page with 12 small figures.

8.6.5 The Central limit theorem

The 'central limit theorem' (CLT) forms the backbone of inferential statistics. This theorem states (informally) that if you draw samples (of *n* units) from a population, the mean of these samples follows a normal distribution. This is true regardless of the underlying distribution you sample from.

In this exercise, you will apply a simple simulation study to test the CLT, and to make histograms and quantile-quantile plots.

1. ♦ Draw 200 samples of size 10 from a uniform distribution. Use the `runif` function to sample from the uniform distribution, and the `replicate` function to repeat this many times.
2. ♦ Compute the sample mean for each of the 200 samples in 1. Use `apply` or `colMeans` to calculate column-wise means of a matrix (note: `replicate` will return a matrix, if used correctly).
3. ♦ Draw a histogram of the 200 sample means, using `hist`. Also draw a normal quantile-quantile plot, using `qqnorm`.
4. ♦ On the histogram, add a normal curve using the `dnorm` function. Note: to do this, plot the histogram with the argument `freq=FALSE`, so that the histogram draws the probability density, not the frequency.
5. ▲ Write a function that does all of the above, and call it `PlotCLT`.

Chapter 9

Project management and workflow

9.1 Tips on organizing your code

In this chapter, we present a few tips on how to improve your workflow and organization of scripts, functions, raw data, and outputs (figures, processed data, etc.). The structure that we present below is just an example, and will depend on the particular project, and your personal taste. We encourage you to experiment with different workflows and organization of your script(s) and outputs, to declutter your working directory and to be able to keep track of what you have accomplished in your analysis sofar.

Although much of this is subject to personal preference, we believe that if you follow these rules, management of your code will be much easier and more transparent:

1. Use 'projects' in Rstudio to manage your files and workspace
2. Keep raw (original) data in a separate folder, and *never modify raw data*
3. Outputs (figures, processed datasets) are disposable, your scripts can always re-produce the output
4. Keep functions separate from other code
5. Write functions as much as possible

Further reading Further reading is on this excellent blogpost, from which we borrowed some ideas: `http://nicercode.github.io/blog/2013-04-05-projects/`

If you follow (something like) the structure we show here, you have the added benefit that your directory is fully portable. That is, you can zip it, email it to someone, they can unzip it and run the entire analysis.

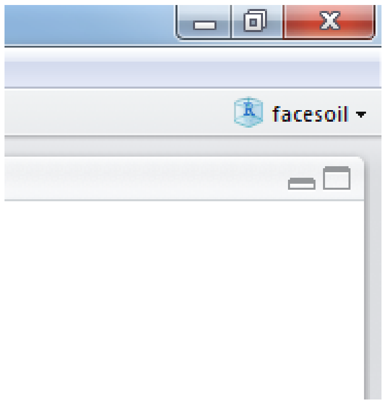
The most important tip is to *use projects in Rstudio*. Projects are an efficient method to keep your files organized, and to keep all your different projects separated. There is a natural tendency for analyses to grow over time, to the point where they become too large to manage properly. The way we usually deal with this is to try to split projects into smaller ones, even if there is some overlap between them.

9.2 Set up a project in Rstudio

In Rstudio, click on the menu item `File/New Project...`. If you already have a folder for the project, take the 2nd option (`Existing directory`), otherwise create a folder as well by choosing the 1st option (`New project`). We will not discuss "version control" in this chapter (the third option).

Browse for the directory you want to create a project in, and click `Choose`. This creates a file with extension `.Rproj`. Whenever you open this project, Rstudio will set the working directory to the location of the project file. If you use projects, you no longer need to set the working directory manually as we showed in Section 1.8.

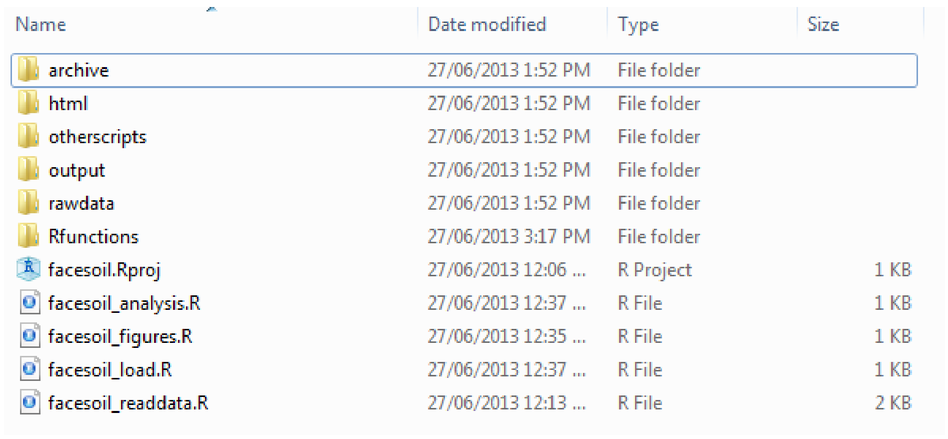
Rstudio has now switched to your new project. Notice in the top-right corner there is a button that shows the current project. For the example project 'facesoil', it looks like this:



By clicking on that button you can easily switch over to other projects. The working directory is automatically set to the right place, and all files you had open last time are remembered as well. As an additional bonus, the workspace is also cleared. This ensures that if you switch projects, you do not inadvertently load objects from another project.

9.3 Directory structure

For the 'facesoil' project, we came up with the following directory structure. Each item is described further below.



Name	Date modified	Type	Size
archive	27/06/2013 1:52 PM	File folder	
html	27/06/2013 1:52 PM	File folder	
otherscripts	27/06/2013 1:52 PM	File folder	
output	27/06/2013 1:52 PM	File folder	
rawdata	27/06/2013 1:52 PM	File folder	
Rfunctions	27/06/2013 3:17 PM	File folder	
facesoil.Rproj	27/06/2013 12:06 ...	R Project	1 KB
facesoil_analysis.R	27/06/2013 12:37 ...	R File	1 KB
facesoil_figures.R	27/06/2013 12:35 ...	R File	1 KB
facesoil_load.R	27/06/2013 12:37 ...	R File	1 KB
facesoil_readdata.R	27/06/2013 12:13 ...	R File	2 KB

`rawdata`

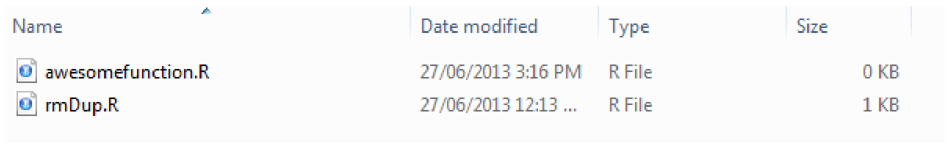
Most importantly, *keep your raw data separate from everything else*. Here we have placed our raw CSV files in the `rawdata` directory.

In some projects it makes sense to further keep raw data files separate from each other, for example you might have subfolders in the rawdata folder that contain different types of datasets (e.g. 'rawdata/leafdata', 'rawdata/isotopes'). Again, the actual solution will depend on your situation, but it is at least very good practice to store your raw data files in a separate folder.

`Rfunctions`

If you do not frequently write functions already, you should force yourself to do so. Particularly for tasks that you do more than once, functions can greatly improve the clarity of your scripts, helps you avoid mistakes, and makes it easier to reuse code in another project.

It is good practice to keep functions in a separate folder, for example `Rfunctions`, with each function in a separate file (with the extension `.R`). It may look like this,

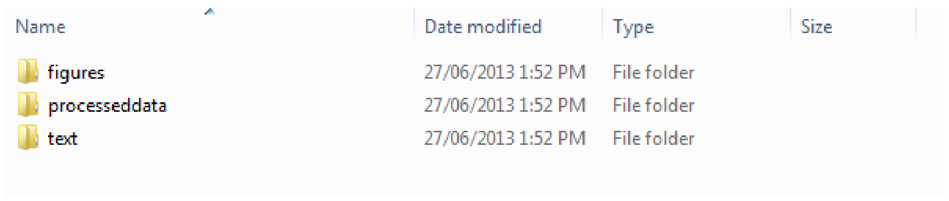


Name	Date modified	Type	Size
awesomefunction.R	27/06/2013 3:16 PM	R File	0 KB
rmDup.R	27/06/2013 12:13 ...	R File	1 KB

We will use `source()` to load these functions, see further below.

`output`

It is a good idea to send all output from your **R** scripts to a separate folder. This way, it is very clear what the *outputs* of the analysis are. It may also be useful to have subfolders specifying what type of output it is. Here we decided to split it into figures, processeddata, and text :



Name	Date modified	Type	Size
figures	27/06/2013 1:52 PM	File folder	
processeddata	27/06/2013 1:52 PM	File folder	
text	27/06/2013 1:52 PM	File folder	

9.4 The R scripts

A few example scripts are described in the following sections. Note that these are just examples, the actual setup will depend on your situation, and your personal preferences. The main point to make here is that it is tremendously useful to separate your code into a number of separate scripts. This makes it easier to maintain your code, and for an outsider to follow the logic of your workflow.

`facesoil_analysis.R`

This is our 'master' script of the project. It calls (i.e., executes) a couple of scripts using `source`. First, it 'sources' the `facesoil_load.R` script, which loads packages and functions, and reads raw data. Next, we do some analyses (here is a simple example where we calculate daily averages), and call a script that makes the figures (`facesoil_figures.R`).

Note how we direct all output to the 'output' folder, by specifying the *relative path*, that is, the path relative to the current working directory.

```
# Calls the load script.
source("facesoil_load.R")

# Export processed data
write.csv(allTheta, "output/processeddata/facesoil_allTheta.csv",
          row.names=FALSE)

## Aggregate by day

# Make daily data
allTheta$Date <- as.Date(allTheta$DateTime)
allTheta_agg <- summaryBy(. ~ Date + Ringnr, data=allTheta,
                          FUN=mean, keep.names=TRUE)

# Export daily data
write.csv(allTheta_agg, "output/processeddata/facesoil_alltheta_daily.csv",
          row.names=FALSE)

## make figures
source("figures.R")
```

`facesoil_figures.R`

In this example we make the figures in a separate script. If your project is quite small, perhaps this makes little sense. When projects grow in size, though, I have found that collecting the code that makes the figures in a separate script really helps to avoid clutter.

Also, you could have a number of different 'figure' scripts, one for each 'sub-analysis' of your project. These can then be sourced in the master script (here `facesoil_analysis.R`), for example, to maintain a transparent workflow.

Here is an example script that makes figures only. Note the use of `dev.copy2pdf`, which will produce a PDF and place it in the `output/figures` directory.

```
# Make a plot of soil water content over time
windows()
with(allTheta, plot(DateTime, R30.mean, pch=19, cex=0.2,
                    col=Ringnr))
dev.copy2pdf(file="./output/figures/facesoil_overtime.pdf")

# More figures go here!
```

facesoil_load

This script contains all the bits of code that are

1. Cleaning the workspace
2. Loading packages
3. Loading homemade functions
4. Reading and pre-processing the raw data

Remember that it is important to start with a clean workspace (see Section 1.9), as it guarantees that all your work is reproducible (and not dependent on objects that you created ages ago).

It is useful to load all packages in one location in your, which makes it easy to fix problems should they arise (i.e., some packages are not installed, or not available).

```
# Clean workspace
rm(list=ls())

# Load packages
library(doBy)
library(lubridate)

# Source functions (this loads functions but does no actual work)
source("Rfunctions/rmDup.R")

# Make the processed data (this runs a script)
source("facesoil_readdata.R")
```

facesoil_readdata.R

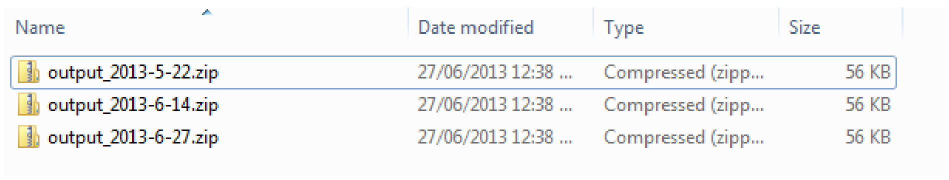
This script produces a dataframe based on the raw CSV files in the 'rawdata' folder. The example below just reads a dataset and changes the DateTime variable to a POSIXct class. In this script, I normally also do all the tedious things like deleting missing data, converting dates and times, merging, adding new variables, and so on. The advantage of placing all of this in a separate script is that you keep the boring bits separate from the code that generates results, such as figures, tables, and analyses.

```
# Read raw data from 'rawdata' subfolder
allTheta <- read.csv("rawdata/FACE_SOIL_theta_2013.csv")

# Convert DateTime
allTheta$DateTime <- ymd_hms(as.character(allTheta$DateTime))

# Add Date
allTheta$Date <- as.Date(allTheta$DateTime)

# Etc.
```

9.5 Archiving the output

In the example workflow we have set up in the previous sections, all items in the output folder will be automatically overwritten every time we run the master script `facesoil_analysis.R`. One simple way to back up your previous results is to create a zipfile of the entire output directory, place it in the `archive` folder, and rename it so it has the date as part of the filename.

After a while, that directory may look like this:



Name	Date modified	Type	Size
output_2013-5-22.zip	27/06/2013 12:38 ...	Compressed (zipp...	56 KB
output_2013-6-14.zip	27/06/2013 12:38 ...	Compressed (zipp...	56 KB
output_2013-6-27.zip	27/06/2013 12:38 ...	Compressed (zipp...	56 KB

If your `processData` folder is very large, this may not be the optimal solution. Perhaps the `processedData` can be in a separate output folder, for example.

Adding a Date stamp to output files

Another option is to use a slightly different output filename every time, most usefully with the current Date as part of the filename. The following example shows how you can achieve this with the `today` from the `lubridate` package, and `paste0` (which is the same as `paste`, with the argument `sep=""`).

```r
# For the following to work, load lubridate
# Recall that in your workflow, it is best to load all packages in one place.
library(lubridate)

# Make a filename with the current Date:
fn <- paste0("output/figures/FACE_soilfigure1_",today(),".pdf")
fn

## [1] "output/figures/FACE_soilfigure1_2019-09-17.pdf"

# Also add the current time, make sure to reformat as ':' is not allowed!
fn <- paste0("output/figures/FACE_soilfigure1_",format(now(),"%Y-%m-%d_%H-%M"),".pdf")
fn

## [1] "output/figures/FACE_soilfigure1_2019-09-17_12-22.pdf"
```

With the above filename, we can now make a PDF, like so:

```r
windows(7,7)
# .... make plot here ....
dev.copy2pdf(file=fn)
```

Chapter 10

Hints

10.1 Identifying and managing different types of objects

10.1.1 Vectors

Dimensions: n elements in one dimension, all of one mode

- `length(vector)` gives the number of elements of `vector`

Generating:

- use `c()` with data
- use `vector(mode, length)` without data, where mode = 'numeric', 'integer', 'character', or 'logical'

Indexing:

- `vector[i]` takes the i^{th} element of `vector`
- `vector['name']` takes the element of `vector` that is named 'name'

```
# Example:
num1 <- c(1,3,5)
str(num1)

##  num [1:3] 1 3 5

length(num1)

## [1] 3

num1

## [1] 1 3 5

num1[2]

## [1] 3

names(num1) <- c('a', 'b', 'c')
num1

## a b c
## 1 3 5
```

```
num1['b']

## b
## 3
```

10.1.2 Matrices

Dimensions: n elements in two dimensions (i, j), all of one mode

- `length(matrix)` gives the number of elements in `matrix`
- `dim(matrix)` gives the number of rows and columns in `matrix`
- `nrow(matrix)` and `ncol(matrix)` give the number of rows and columns, respectively, in `matrix`

Generating:

- use `matrix(c(), nrow, ncol)` with data
- use `matrix(NA, nrow, ncol)` without data

Indexing:

- `matrix[i, j]` takes the element in the i^{th} row and j^{th} column of `vector`
- `matrix['rowname', 'colname']` takes the element of `vector` with the row named 'rowname' and the column named 'colname'

```
# Example:
mat1 <- matrix(c(1,3,5,7,9,11), nrow=3, ncol=2, byrow=T)
str(mat1)

##  num [1:3, 1:2] 1 5 9 3 7 11

length(mat1)

## [1] 6

dim(mat1)

## [1] 3 2

ncol(mat1); nrow(mat1)

## [1] 2
## [1] 3

mat1

##      [,1] [,2]
## [1,]    1    3
## [2,]    5    7
## [3,]    9   11

mat1[2,2]

## [1] 7

rownames(mat1) <- c('a', 'b', 'c')
colnames(mat1) <- c('A', 'B')
mat1
```

```
##   A  B
## a 1  3
## b 5  7
## c 9 11
```

```r
mat1['b', 'B']
```

```
## [1] 7
```

10.1.3 Lists

Dimensions: variable – each element is an object, possibly of any type

- `length(list[[i]])` gives the number of elements in the i^{th} element of `list`
- `length(list)` gives the number of elements at the highest level of `list` respectively

Generating:

- use `list(x, y, etc.)` with data, where each element could be any type of object
- use `vector(mode = 'list', length)` without data, then add data to each element via indexing

Indexing:

- `list[[i]]` takes the object in the i^{th} element of `list`
- `list[[c(i, j)]]` returns the j^{th} element of the i^{th} element `list`
- `list[['name']]` takes the object within `list` that is named 'name'
- `list[i]` returns a list containing the i^{th} element of `list`
- `list['name']` or `list$name` returns a list containing the element of `list` that is named 'name'

```r
# Example:
list1 <- list(1:10, c('a', 'to', 'z'), matrix(1:4, nrow=2, ncol=2))
str(list1)
```

```
## List of 3
##  $ : int [1:10] 1 2 3 4 5 6 7 8 9 10
##  $ : chr [1:3] "a" "to" "z"
##  $ : int [1:2, 1:2] 1 2 3 4
```

```r
list1
```

```
## [[1]]
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## [[2]]
## [1] "a"  "to" "z"
##
## [[3]]
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

```r
list1[[2]]
```

```
## [1] "a"  "to" "z"
```

```r
list1[[2]][2]
```

```
## [1] "to"
```

```
list1[[c(2,2)]]
```

```
## [1] "to"
```

```
list1[2]
```

```
## [[1]]
## [1] "a"  "to" "z"
```

```
list1[c(2,1)]
```

```
## [[1]]
## [1] "a"  "to" "z"
##
## [[2]]
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
names(list1) <- c('num', 'char', 'mat')
list1[['mat']]
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

```
list1$mat
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

10.1.4 Dataframes

Dimensions: elements in vectors, possibly of different modes, but each with n elements (a special type of list)

- `dim(df)`
- `length(dataframe)` gives the number of columns in `dataframe`
- `dim(dataframe)` gives the number of rows and columns in `matrix`
- `nrow(dataframe)` and `ncol(dataframe)` give the number of rows and columns,

Generating:

- `data.frame(vector1, vector2)` creates a dataframe with two columns from two vectors
- `data.frame(col1=vector1, col2=vector2)` creates a dataframe with two columns named 'col1' and 'col2' from two vectors

Indexing:

- `dataframe[i, j]` takes the element in the i^th row and j^th column of `dataframe`
- `dataframe[['colname']]` or `dataframe$colname` takes the column of `dataframe` that is named 'colname'

```
df <- data.frame(vec1=11:20, vec2=rnorm(n=10))
str(df)
```

```
## 'data.frame': 10 obs. of  2 variables:
##  $ vec1: int  11 12 13 14 15 16 17 18 19 20
##  $ vec2: num  -1.142 0.593 -0.75 1.707 0.563 ...
```

```
dim(df)
```

```
## [1] 10  2
```

```
length(df)
```

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

```
df
```

```
##    vec1       vec2
## 1    11 -1.1415432
## 2    12  0.5926600
## 3    13 -0.7500866
## 4    14  1.7069878
## 5    15  0.5625549
## 6    16  0.2677543
## 7    17  1.0087672
## 8    18 -1.6415777
## 9    19  0.9512273
## 10   20 -0.1265844
```

```
df[2, ]
```

```
##   vec1    vec2
## 2   12 0.59266
```

```
df[, 2]
```

```
##  [1] -1.1415432  0.5926600 -0.7500866  1.7069878  0.5625549  0.2677543
##  [7]  1.0087672 -1.6415777  0.9512273 -0.1265844
```

```
df[2,2]
```

```
## [1] 0.59266
```

```
df[['vec2']]
```

```
##  [1] -1.1415432  0.5926600 -0.7500866  1.7069878  0.5625549  0.2677543
##  [7]  1.0087672 -1.6415777  0.9512273 -0.1265844
```

```
df$vec2
```

```
##  [1] -1.1415432  0.5926600 -0.7500866  1.7069878  0.5625549  0.2677543
##  [7]  1.0087672 -1.6415777  0.9512273 -0.1265844
```

```
names(df) <- c('var1', 'var2')
str(df)
```

```
## 'data.frame': 10 obs. of  2 variables:
##  $ var1: int  11 12 13 14 15 16 17 18 19 20
##  $ var2: num  -1.142 0.593 -0.75 1.707 0.563 ...
```

Chapter 11

Getting Help

11.1 Web

That's right, Google should be your number one choice. Try typing a question in full, for example, "How to read tab-delimited data in R".
`www.google.com`

Search engine for **R**-related topics (including packages, blogs, email lists, etc.):
`www.rseek.org`

Excellent resource for graphical parameters, like par() (and more resources on **R**):
`http://www.statmethods.net/advgraphs/parameters.html`

Archive (and search) of the R-help mailing list can be found on Nabble:
`http://r.789695.n4.nabble.com/`

Search engine for the R-help mailing list:
`http://finzi.psych.upenn.edu/search.html`

11.2 Free online books

You can find a large number of manuals on the CRAN website. There, find the link 'Contributed' on the left:
`www.cran.r-project.org`

If you are interested in the inner workings of **R**, the 'R inferno' is not to be missed:
`http://www.burns-stat.com/pages/Tutor/R_inferno.pdf`

11.3 Blogs

A compendium of blogs about **R** can be found here. These blogs include many posts on frequently asked questions, as well as advanced topics:
`http://www.r-bloggers.com/`

Examples of various simple or complicated plots that have been produced using **R**, with code provided can be found here:
`http://rgraphgallery.blogspot.com.au/`

11.4 Finding packages

With >10000 packages available on CRAN alone (`http://cran.r-project.org/`), it can be difficult to find a package to perform a specific analysis. Often the easiest way to find a package is to use Google (for example, 'r package circular statistics'). CRAN has also categorized many packages in their 'Task Views':
`http://cran.r-project.org/web/views/`
For example, the 'Spatial' page lists over 130 packages to do with various aspects of spatial analysis and visualization.

11.5 Offline paper books

An exhaustive list of printed books can be found here:
`http://www.r-project.org/doc/bib/R-books.html`

Springer publishes many books about **R**, incluing a special series called "Use R!" If your library has a subscription, many of these books can be read online. For example, the classic *Introductory Statistics with R* by Peter Dalgaard:
`http://link.springer.com/book/10.1007/978-0-387-79054-1/page/1`

Appendix A

Description of datasets

A.1 Tree allometry

File: 'Allometry.csv'

This dataset contains measurements of tree dimensions and biomass. Data kindly provided by John Marshall, University of Idaho.

Variables:

- `species` - The tree species (PSME = Douglas fir, PIMO = Western white pine, PIPO = Ponderosa pine).
- `diameter` - Tree diameter at 1.3m above ground (cm).
- `height` - Tree height (m).
- `leafarea` - Total leaf area (m^2).
- `branchmass` - Total (oven-dry) mass of branches (kg).

A.2 Coweeta tree data

File: 'coweeta.csv'

Tree measurements in the Coweeta LTER (`http://coweeta.uga.edu/`). Data are from Martin et al. (1998; Can J For Res 28:1648-1659).

Variables:

- `species` One of 10 tree species
- `site` Site abbreviation
- `elev` Elevation (m asl)
- `age` Tree age (yr)
- `DBH` Diameter at breast height (cm)
- `height` Tree height (m)
- `folmass` Foliage mass (kg)

- `SLA` Specific leaf area (cmg^-1)
- `biomass` Total tree mass (kg)

A.3 Hydro dam

File: 'hydro.csv'

This dataset describes the storage of the hydrodam on the Derwent river in Tasmania (Lake King William & Lake St. Clair), in equivalent of energy stored. Data were downloaded from `http://www.hydro.com.au/water/energy-data`.

Variables:

- `Date` - The date of the bi-weekly reading
- `storage` - Total water stored, in energy equivalent (GWh).

A.4 Rain

File: 'Rain.csv'

This dataset contains ten years (1995-2006) of daily rainfall amounts as measured at the Richmond RAAF base. Source : BOM (`http://www.bom.gov.au/climate/data/`).

Variables:

- `Year`
- `DOY` - Day of year (1-366)
- `Rain` - Daily rainfall (mm)

A.5 Weight loss

File: 'Weightloss.csv'

This dataset contains measurements of a Jeremy Zawodny over a period of about 3 months while he was trying to lose weight (data obtained from `http://jeremy.zawodny.com/blog/archives/006851.html`). This is an example of an irregular timeseries dataset (intervals between measurements vary).

Variables:

- `Date` - The Date (format, dd/mm/yy)
- `Weight` - The person's weight (lbs)

A.6 Pupae

File: 'pupae.csv'

This dataset is from an experiment where larvae were left to feed on *Eucalyptus* leaves, in a glasshouse that was controlled at two different levels of temperature and CO_2 concentration. After the larvae pupated (that is, turned into pupae), the body weight was measured, as well as the cumulative 'frass' (larvae excrement) over the entire time it took to pupate.

Data courtesy of Tara Murray, and simplified for the purpose of this book.

Variables:

- `T_treatment` - Temperature treatments ('ambient' and 'elevated')
- `CO2_treatment` - CO_2 treatment (280 or 400 ppm).
- `Gender` - The gender of the pupae : 0 (male), 1 (female)
- `PupalWeight` - Weight of the pupae (g)
- `Frass` - Frass produced (g)

A.7 Cereals

File: 'Cereals.csv'

This dataset summarizes 77 different brands of breakfast cereals, including calories, proteins, fats, and so on, and gives a 'rating' that indicates the overall nutritional value of the cereal.

This dataset was downloaded from Statlib at CMU, and is frequently used as an example dataset.

Variables:

- `Cereal name` - Name of the cereal (text)
- `Manufacturer` - One of: "A","G","K","N","P","Q","R" (A = American Home Food Products; G = General Mills; K = Kelloggs; N = Nabisco; P = Post; Q = Quaker Oats; R = Ralston Purina)
- `Cold or Hot` - Either "C" (cold) or "H" (Hot).
- `calories` - number
- `protein` - g
- `fat` - g
- `sodium` - g
- `fiber` - g
- `carbo` - g
- `sugars` - g
- `potass` - g
- `vitamins` - 0,25 or 100
- `rating` - Nutritional rating (function of the above 8 variables).

NOTE: also included are the files 'cereal1.csv', 'cereal2.csv' and 'cereal3.csv', small subsets of the cereal data used in Section 6.3.1.

A.8 Flux tower data

File: 'Fluxtower.csv'

This dataset contains measurements of CO_2 and H_2O fluxes (and related variables) over a pine forest in Quintos de Mora, Spain. The site is a mixture of *Pinus pinaster* and *Pinus pinea*, and was planted in the 1960's.

Data courtesy of Victor Resco de Dios, and simplified for the purpose of this book.

Variables:

- `TIMESTAMP` - Character vector with date and time
- `FCO2` - Canopy CO2 flux (μ mol m^{-2} s^{-1})
- `FH2O` - Canopy H2O flux (mmol m^{-2} s^{-1})
- `ustar` - Roughness length (m s^{-1})
- `Tair` - Air temperature (degrees C)
- `RH` - Relative humidity (%)
- `Tsoil` - Soil temperature (degrees C)
- `Rain` - Rainfall (mm half hour^{-1})

A.9 Pulse rates before and after exercise

File: 'ms212.txt'

This data is in a TAB separated file; to read it in use `read.table("ms212.txt", header=TRUE)`.

A dataset on pulse rates before and after exercise. Taken from the OzDASL library (Smyth, GK (2011). Australasian Data and Story Library (OzDASL). http://www.statsci.org/data.)

More information at http://www.statsci.org/data/oz/ms212.html.

Variables:

- `Height` - Height (cm)
- `Weight` - Weight (kg)
- `Age` - Age (years)
- `Gender` - Sex (1 = male, 2 = female)
- `Smokes` - Regular smoker? (1 = yes, 2 = no)
- `Alcohol` - Regular drinker? (1 = yes, 2 = no)
- `Exercise` - Frequency of exercise (1 = high, 2 = moderate, 3 = low)
- `Ran` - Whether the student ran or sat between the first and second pulse measurements (1 = ran, 2 = sat)
- `Pulse1` - First pulse measurement (rate per minute)
- `Pulse2` - Second pulse measurement (rate per minute)
- `Year` - Year of class (93 - 98)

A.10 Weather data at the HFE

File: 'HFEmet2008.csv'

Data for the weather station at the Hawkesbury Forest Experiment for the year 2008.

Data courtesy of Craig Barton.

Variables:

- `DateTime` - Date Time (half-hourly steps)
- `Tair` - Air temperature (degrees C)
- `AirPress` - Air pressure (kPa)
- `RH` - Relative humidity (%)
- `VPD` - Vapour pressure deficit (kPa)
- `PAR` - Photosynthetically active radiation (μ mol m^{-2} s^{-1})
- `Rain` - Precipitation (mm)
- `wind` - Wind speed (m s^{-1})
- `winddirection` - Wind direction (degrees)

A.11 Age and memory

File: 'eysenck.txt' This data is in a TAB separated file, so to read it in use `read.table("eysenck.txt", header=TRUE)`.

A dataset on the number of words remembered from list, for various learning techniques, and in two age groups. Taken from the OzDASL library (Smyth, GK (2011). Australasian Data and Story Library (OzDASL). http://www.statsci.org/data.)

More information at http://www.statsci.org/data/general/eysenck.html.

Variables:

- `Age` - Younger or Older
- `Process` - The level of processing: Counting, Rhyming, Adjective, Imagery or Intentional
- `Words` - Number of words recalled

A.12 Passengers on the Titanic

File: 'titanic.txt'

This data is in a TAB separated file, so to read it in use `read.table("titanic.txt", header=TRUE)`.

Survival status of passengers on the Titanic, together with their names, age, sex and passenger class. Taken from the OzDASL library (Smyth, GK (2011). Australasian Data and Story Library (OzDASL). http://www.statsci.org/data.)

More information at http://www.statsci.org/data/general/titanic.html.

Variables:

- `Name` Recorded name of passenger
- `PClass` Passenger class: 1st, 2nd or 3rd
- `Age` Age in years (many missing)
- `Sex` male or female
- `Survived` 1 = Yes, 0 = No

A.13 Xylem vessel diameters

File: 'vessel.csv'

Measurements of diameters of xylem (wood) vessels on a single *Eucalyptus saligna* tree grown at the Hawkesbury Forest Experiment.

Data courtesy of Sebastian Pfautsch.

Variables:

- `position` - Either 'base' or 'apex' : the tree was sampled at stem base and near the top of the tree.
- `imagenr` - At the stem base, six images were analyzed (and all vessels measured in that image). At apex, three images.
- `vesselnr` - Sequential number
- `vesseldiam` - Diameter of individual vessels (μ m).

A.14 Eucalyptus leaf endophytes

File: 'endophytes_env.csv' and 'endophytes.csv'

Community fingerprints from fungi colonising living leaves and litter from nine Eucalyptus spp. in the HFE common garden and surrounding area. 'endophytes.csv' contains a 'species-sample' matrix, with 98 samples in rows and 874 operational taxonomic units (OTUs) in columns.

The variables below refer to the data in 'endophytes_env.csv'. The rows are matched across the two tables.

Variables:

- `species` - Tree species
- `type` - Whether leaves came from canopy (fresh) or ground (litter)
- `percentC` - Leaf carbon content, per gram dry mass
- `percentN` - Leaf nitrogen content, per gram dry mass
- `CNratio` - Ratio of C to N in leaves

A.15 I x F at the HFE - plot averages

File: 'HFEIFplotmeans.csv'

Tree inventory data from the irrigation by fertilization (I x F) experiment in the Hawkesbury Forest Experiment (HFE). This dataset includes the plot means, see I x F tree data for the tree-level observations.

Data courtesy of Craig Barton and Burhan Amiji.

Variables:

- `plotnr` - A total of sixteen plots (four treatments).
- `Date` - The date of measurement.
- `totalvolume` - Total estimated woody volume (m^3 ha^{-1}).
- `meandiameter` - Mean diameter for the sample trees (cm).
- `meanheight` - Mean height for the sample trees (m).
- `treat` - One of four treatments (I - irrigated, F - dry fertilized, IL - Liquid fertilizer plus irrigation, C - control).

A.16 I x F at the HFE - tree observations

File: 'HFEIFbytree.csv'

Tree inventory data from the irrigation by fertilization (I x F) experiment in the Hawkesbury Forest Experiment (HFE). This dataset includes the tree-level observations.

Data courtesy of Craig Barton and Burhan Amiji.

Variables:

- `ID` A unique identifier for each tree.
- `plotnr` - A total of sixteen plots (four treatments).
- `treat` - One of four treatments (I - irrigated, F - dry fertilized, IL - Liquid fertilizer plus irrigation, C - control)
- `Date` - The date of measurement (YYYY-MM-DD)
- `height` - Mean height for the sample trees (m).
- `diameter` - Mean diameter for the sample trees (cm).

A.17 Dutch election polls

File: 'dutchelection.csv'

Polls for the 12 leading political parties in the Netherlands, leading up to the general election on 12 Sept. 2012. Data are in 'wide' format, with a column for each party. Values are in percentages.

Data taken from `http://en.wikipedia.org/wiki/Dutch_general_election,_2012`, based on polls by Ipsos NL.

A.18 Tibetan Plateau plant community data

File: 'tibplat.csv'

Plant community data collected from 0.5 m by 0.5 m plots in an alpine grassland in the Tibetan Plateau. The experiment included two factors: fertilisation (none, 30 grams nitrogen per square metre) and grazing (enclosed to prevent grazing, not enclosed). This dataset is a subset of the whole dataset, including only the eight most abundant species of the 48 species measured withing the plots. The full dataset was analysed in Yang et al., 2012, Ecology 93:2321 (doi:10.1890/11-2212.1).

Variables:

- `fertilization` - 1:yes, 0:no
- `enclosure` - 1:yes, 0:no
- columns 3:50 - aboveground biomass per plot; column heading indicates species identity

A.19 Genetically modified soybean litter decomposition

File: 'masslost.csv'

Soybean litter decomposition as a function of time (`date`), type of litter (`variety`), herbicides applied (`herbicide`), and where in the soil profile it is placed (`profile`). `masslost` refers to the proportion of the litter that was lost from the bag (decomposed) relative to the start of the experiment. Herbicide treatments were applied at the level of whole plots, with both treatments represented within each of four blocks. Both levels of variety and profile were each represented within each plot, with six replicates of each treatment added to each plot.

Variables:

- `plot` - A total of eight plots.
- `block` - A total of four blocks.
- `variety` - Soybean variety is genetically modified ('gm') or not ('nongm'); manipulated at the subplot level.
- `herbicide` - Herbicide applied is glyphosate ('gly') or conventional program ('conv'); manipulated at plot level.
- `profile` - Whether litter was 'buried' in the soil or placed at the soil 'surface'; manipulated at the subplot level.
- `date` - Date at which litter bags were recovered.
- `sample` - Factor representing timing of sampling ('incrop1', 'incrop2', 'postharvest').
- `masslost` - The proportion of the initial mass that was lost from each litter bag during field incubation. Some values are lower than zero due to insufficient washing of dirt and biota from litter prior to weighing.

A.20 EucFACE ground cover data

File: 'eucfaceGC.csv'

This file contains estimates of plant and litter cover within the rings of the EucFACE experiment, evaluating forest ecosystem responses to elevated CO_2, on two dates. Within each ring are four plots and within each plot are four 1m by 1m subplots. Values represent counts along a grid of 16 points within each subplot.

Variables:

- `Date` - Date at which measurements took place.
- `Ring` - The identity of the EucFACE Ring, the level at which the experimental treatment is applied.
- `Plot` - A total of four plots, nested within each level of Ring.
- `Sub` - A total of four subplots, nested within each level of Plot.
- `Forbes` - Number of points where dicot plants are observed.
- `Grass` - Number of points where grass is observed.
- `Litter` - Number of points where leaf litter is observed.
- `Trt` - The experimental treatment: `ctrl` for ambient levels of atmospheric carbon dioxide, `elev` for ambient plus 150ppm.

A.21 Tree canopy gradients in the Priest River Experimental Forest (PREF)

File: 'prefdata.csv'

The dataset contains measurements of leaf mass per area (LMA), and distance from the top of the tree (dfromtop) on 35 trees of two species.

Data courtesy of John Marshall (Marshall, J.D., Monserud, R.A. 2003. Foliage height influences specific leaf area of three conifer species. Can J For Res 33:164-170), and simplified for the purpose of this book.

Variables:

- `ID` - ID of the individual tree
- `species` - Pinus ponderosa or Pinus monticola
- `dfromtop` - Distance from top of tree (where leaf sample was taken) (m)
- `totheight` - Total height of the tree (m)
- `height` - Height from the ground (where sample was taken) (m)
- `LMA` - Leaf mass per area (g m^{-2})
- `narea` - Nitrogen per area (gN m^{-2})

A.22 Seed germination

File: 'germination_fire.csv' and 'germination_water.csv'

Two datasets on the germination success of seeds of four *Melaleuca* species, when subjected to temperature, fire cue, and dehydration treatments. Seeds were collected from a number of sites and subjected to 6 temperature treatments and fire cues (in the fire germination data), or two a range of dehydration levels (in the water germination data).

Data are from Hewitt et al. 2015 (Austral Ecology 40(6):661-671), shared by Charles Morris, and simplified for the purpose of this book.

Variables:

'germination_fire.csv' :

- `species` - One of four Melaleuca species
- `temp` - Temperature treatment (C)
- `fire.cues` - Fire cue treatment (yes or no)
- `site` - Coding for the site where the seed was collected
- `cabinet` - ID for the cabinet where seeds were treated
- `germ` - Number of germinated seeds
- `n` - Number of seeds tested (20 for all rows)

'germination_water.csv' :

- `species` - One of four Melaleuca species
- `site` - Coding for the site where the seed was collected
- `water.potential` - Water potential of the seed (Mpa) after incubation (low values is drier)
- `germ` - Number of germinated seeds
- `n` - Number of seeds tested (25 for all rows)

A.23 Leaf gas exchange at the EucFACE

File: 'eucface_gasexchange.csv'

Measurements of leaf net photosynthesis at the EucFACE experiment, on leaves of different trees growing in ambient and elevated CO_2 concentrations. Measurements were repeated four times during 2013 (labelled as Date=A,B,C,D).

Data are from Gimeno et al. 2015 (Functional Ecology, doi: 10.1111/1365-2435.12532), and simplified for the purpose of this book.

Variables:

- `Date` - One of four campaigns (A,B,C,D)
- `CO2` - CO2 treatment (either ambient - Amb, or elevated - Ele)
- `Ring`- One of six 'rings' (plots) at the EucFACE
- `Tree` - Unique identifier for the tree number
- `Photo` - Net leaf photosynthesis (μ mol m^{-2} s^{-1})
- `Trmmol` - Leaf transpiration rate (mmol m^{-2} s^{-1})
- `VpdL` - Vapour pressure deficit (kPa)

A.24 Howell height, age and weight data

File: 'howell.csv'

Downloaded from `https://tspace.library.utoronto.ca/handle/1807/17996`, subsetted for non-missing data and one outlier removed. These data were also used by McElreath (2016, "Statistical Rethinking", CRC Press). Data include measurements of height, age and weight on Khosan people.

Variables:

- `sex` - male or female
- `age` - Age (years)
- `weight` - Body weight)kg)
- `height` - Total height (cm)

A.25 Wild mouse metabolism

File: 'wildmousemetabolism.csv'

Data courtesy of Chris Turbill.

rmr resting metabolic rate minimum of a running average over 12min (kC hour-1) each run is six days bm body mass in grams food

Variables:

- `id` - Individual number.
- `run` - The experiment was repeated three times (run = 1,2,3)
- `day` - Day of experiment (1-6)
- `temp` - Temperature (deg C)
- `food` - Whether food was provided ('Yes') or not ('No')
- `bm` - Body mass (g)
- `wheel` - Whether the mouse could use an exercise wheel ('Yes') or not ('No')
- `rmr` - Resting metabolic rate (minimum rate of a running average over 12min) (kC hour-1)
- `sex` - Male or Female

A.26 Plant drought tolerance

File: 'Choat_precipP50.csv'

Data are from Choat et al. 2012 (Nature 491: 752âĂŞ755), and were simplified for the purpose of this book.

Data include a measure of plant drought tolerance (P50, more negative values indicate plant stems can tolerate lower water contents), and mean annual precipitation of the location where the sample was taken. Data are for 115 individual species (species name not included).

Variables:

- `annualprecip` - Mean annual precipitation (mm)
- `P50` - Measure of plant drought tolerance (the water potential at which 50% of plant hydraulic conductivity is lost) (MPa)

A.27 Child anthropometry

File: 'anthropometry.csv'

Data were downloaded from `http://mreed.umtri.umich.edu/mreed/downloads.html`. Data include measurements of age, foot length, and height for 3898 children. These data are a small subset of many dozens of measurements on the same children, described in detail by Snyder (1977) (see above link for more information).

Variables:

- `age` - Age (years, converted from months in original dataset)
- `gender` - Female or male
- `foot_length` - Total foot length (mm)
- `height` - Total height (cm)

A.28 Alphabet

File: 'alphabet.txt'

Lyrics for the song 'Alphabet Aerobics' by Blackalicious (3-2-1 Records, 1999). The rendition by Daniel Radcliffe is worth a watch (`https://www.youtube.com/watch?v=aKdV5FvXLuI`).

Technical information

This book was compiled with the following R version and packages.

- R version 3.6.1 (2019-07-05), `x86_64-apple-darwin15.6.0`
- Locale: `en_AU.UTF-8/en_AU.UTF-8/en_AU.UTF-8/C/en_AU.UTF-8/en_AU.UTF-8`
- Running under: `macOS Sierra 10.12.6`
- Matrix products: default
- BLAS: `/Library/Frameworks/R.framework/Versions/3.6/Resources/lib/libRblas.0.dylib`
- LAPACK: `/Library/Frameworks/R.framework/Versions/3.6/Resources/lib/libRlapack.dylib`
- Base packages: base, datasets, graphics, grDevices, methods, stats, utils
- Other packages: car 3.0-3, carData 3.0-2, doBy 4.6-2, dplyr 0.8.3, effects 4.1-2, emmeans 1.4, epade 0.3.8, Formula 1.2-3, ggplot2 3.2.1, gplots 3.0.1.1, Hmisc 4.2-0, knitr 1.24, lattice 0.20-38, lubridate 1.7.4, magicaxis 2.0.10, MASS 7.3-51.4, moments 0.14, pastecs 1.3.21, plotrix 3.7-6, RColorBrewer 1.1-2, scales 1.0.0, sciplot 1.1-1, survival 2.44-1.1, visreg 2.5-1
- Loaded via a namespace (and not attached): abind 1.4-5, acepack 1.4.1, assertthat 0.2.1, backports 1.1.4, base64enc 0.1-3, bitops 1.0-6, boot 1.3-23, caTools 1.17.1.2, celestial 1.4.6, cellranger 1.1.0, checkmate 1.9.4, cluster 2.1.0, coda 0.19-3, codetools 0.2-16, colorspace 1.4-1, compiler 3.6.1, crayon 1.3.4, curl 4.0, data.table 1.12.2, DBI 1.0.0, digest 0.6.20, estimability 1.3, evaluate 0.14, forcats 0.4.0, foreign 0.8-72, formatR 1.7, gdata 2.18.0, glue 1.3.1, grid 3.6.1, gridExtra 2.3, gtable 0.3.0, gtools 3.8.1, haven 2.1.1, highr 0.8, hms 0.5.1, htmlTable 1.13.1, htmltools 0.3.6, htmlwidgets 1.3, KernSmooth 2.23-15, latticeExtra 0.6-28, lazyeval 0.2.2, lme4 1.1-21, magrittr 1.5, mapproj 1.2.6, maps 3.3.0, Matrix 1.2-17, minqa 1.2.4, mitools 2.4, multcomp 1.4-10, multcompView 0.1-7, munsell 0.5.0, mvtnorm 1.0-11, NISTunits 1.0.1, nlme 3.1-141, nloptr 1.2.1, nnet 7.3-12, openxlsx 4.1.0.1, pillar 1.4.2, pkgconfig 2.0.2, plyr 1.8.4, pracma 2.2.5, purrr 0.3.2, R6 2.4.0, RANN 2.6.1, Rcpp 1.0.2, readxl 1.3.1, rio 0.5.16, rlang 0.4.0, rpart 4.1-15, rstudioapi 0.10, sandwich 2.5-1, sm 2.2-5.6, splines 3.6.1, stringi 1.4.3, stringr 1.4.0, survey 3.36, tcltk 3.6.1, TH.data 1.0-10, tibble 2.1.3, tidyselect 0.2.5, tools 3.6.1, vctrs 0.2.0, withr 2.1.2, xfun 0.9, xtable 1.8-4, zeallot 0.1.0, zip 2.0.4, zoo 1.8-6

Code is available on `www.bitbucket.org/remkoduursma/hiermanual`.

Index

List of nearly all functions used in this book. Packages are in **bold**. Bold page numbers refer to the key description of the function.