
Data analysis and Visualization with R

AFP Edition

Remko Duursma¹, Jeff Powell¹, Glenn Stone²

¹Hawkesbury Institute for the Environment

²School of Computing, Engineering and Mathematics
Western Sydney University

WESTERN SYDNEY
UNIVERSITY



Hawkesbury Institute
for the Environment

May 16, 2017

Contents

1	Basics of R	4
1.1	Installing R and so on	4
1.1.1	R installation	4
1.1.2	Using and installing RStudio	4
1.2	Basic operations	5
1.2.1	R is a calculator	5
1.3	Working with scripts and markdown files	6
1.3.1	R scripts	6
1.3.2	Using scripts to program with objects	7
1.3.3	Working with markdown files	8
1.3.4	R code in markdown	10
1.3.5	Body text in markdown	11
1.4	Working with vectors	12
1.5	Generating data	16
1.5.1	Sequences of numbers	16
1.5.2	Random numbers	17
1.6	Objects in the workspace	17
1.7	Files in the working directory	18
1.8	Keep a clean memory	18
1.9	Packages	19
1.9.1	Using packages in markdown files	19
1.10	Updating R and packages	21
1.11	Accessing the help files	21
1.12	Exercises	22
1.12.1	Calculating	22
1.12.2	Simple objects	22
1.12.3	Working with a single vector	23
1.12.4	Scripts	23
1.12.5	To quote or not to quote	23
1.12.6	Working with two vectors	24
1.12.7	Alphabet aerobics 1	24
1.12.8	Comparing and combining vectors	25
1.12.9	Packages	25
1.12.10	Save your work	25
2	Reading and subsetting data	26
2.1	Reading data	26
2.1.1	Reading CSV files	26
2.1.2	Reading other data	27
2.2	Working with dataframes	29
2.2.1	Variables in the dataframe	29

2.2.2	Changing column names in dataframes	31
2.3	Extracting data from vectors and dataframes	31
2.3.1	Vectors	31
2.3.2	Subsetting dataframes	34
2.3.3	Difference between <code>[]</code> and <code>subset()</code>	38
2.3.4	Deleting columns from a dataframe	38
2.4	Exporting data	39
2.5	Exercises	40
2.5.1	Working with a single vector 2	40
2.5.2	Alphabet aerobics 2	40
2.5.3	Basic operations with the Cereal data	40
2.5.4	A short dataset	41
2.5.5	Titanic	41
2.5.6	Managing your workspace	41
3	Special data types	43
3.1	Types of data	43
3.2	Working with factors	44
3.2.1	Changing the levels of a factor	47
3.3	Working with logical data	47
3.4	Working with missing values	48
3.4.1	Basics	48
3.4.2	Missing values in dataframes	50
3.4.3	Subsetting when there are missing values	51
3.5	Working with text	51
3.5.1	Basics	51
3.5.2	Column and row names	52
3.5.3	Text in dataframes and <code>grep</code>	53
3.6	Working with dates and times	56
3.6.1	Reading dates	56
3.6.2	Date-Time combinations	58
3.7	Converting between data types	61
3.8	Exercises	64
3.8.1	Titanic	64
3.8.2	Hydro dam	64
3.8.3	HFE tree measurements	65
3.8.4	Flux data	65
3.8.5	Alphabet Aerobics 3	65
3.8.6	DNA Aerobics	66
4	Visualizing data	67
4.1	The R graphics system	67
4.2	Plotting in RStudio	67
4.3	Choosing a plot type	67
4.3.1	Using the <code>plot</code> function	68
4.3.2	Bar plots	68
4.3.3	Histograms and curves	70
4.3.4	Pie charts	71
4.3.5	Box plots	73
4.4	Fine-tuning the formatting of plots	75
4.4.1	A quick example	75
4.4.2	Customizing and choosing colours	76
4.4.3	Customizing symbols and lines	80
4.4.4	Formatting units, equations and special symbols	83

4.4.5	Resetting the graphical parameters	84
4.4.6	Changing the font	84
4.4.7	Adding to a current plot	85
4.4.8	Changing the layout	87
4.4.9	Finding out about more options	88
4.5	Formatting examples	89
4.5.1	Vessel data	89
4.5.2	Weather data	90
4.6	Special plots	93
4.6.1	Scatter plot with varying symbol sizes	93
4.6.2	Bar plots of means with confidence intervals	93
4.6.3	Log-log axes	94
4.7	Exporting figures	96
4.8	Exercises	98
4.8.1	Scatter plot with the pupae data	98
4.8.2	Flux data	98
4.8.3	Hydro dam	99
4.8.4	Coloured scatter plot	99
4.8.5	Superimposed histograms	99
4.8.6	Trellis graphics	99
5	Summarizing, tabulating and merging data	100
5.1	Summarizing dataframes	100
5.2	Making summary tables	102
5.2.1	Summarizing vectors with <code>tapply()</code>	102
5.2.2	Summarizing dataframes with <code>summaryBy</code>	104
5.2.3	Tables of counts	108
5.2.4	Adding simple summary variables to dataframes	109
5.2.5	Reordering factor levels based on a summary variable	109
5.3	Combining dataframes	112
5.3.1	Merging dataframes	112
5.3.2	Row-binding dataframes	116
5.4	Exporting summary tables	118
5.4.1	Inserting tables into documents using R markdown	118
5.5	Exercises	122
5.5.1	Summarizing the cereal data	122
5.5.2	Words and the weather	122
5.5.3	Merge new data onto the pupae data	122
5.5.4	Merging multiple datasets	123
5.5.5	Ordered boxplot	123
5.5.6	Variances in the I x F	123
5.5.7	Weight loss	123

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).


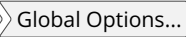

To install **R** on your system, go here : <http://cran.r-project.org>, click on the download link at the top, and then 'base', and finally download the installer.

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 1.10.

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. For now, make sure you go to   and on the  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).

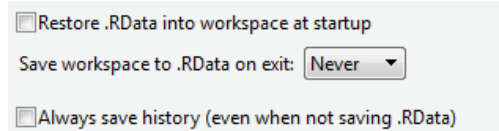


Figure 1.1: Settings in Tools/Global Options/General, to prevent automatically loading objects from a previous session. This contaminates your workspace, causing problems that are difficult to spot.

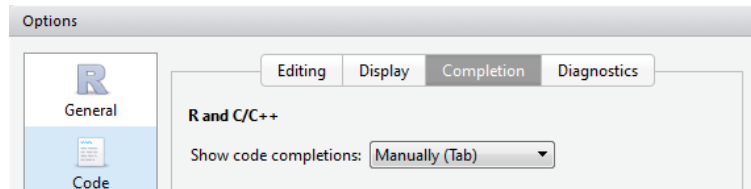


Figure 1.2: Settings in Tools > Global Options > Code > Completion to avoid automatic completion of your code, which opens hint windows as you type. Some people find this helpful, others find it annoying and distracting.

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 lose 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 a 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 known 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 1.9. 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, `rmarkdown` and `pandoc`, are now 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:

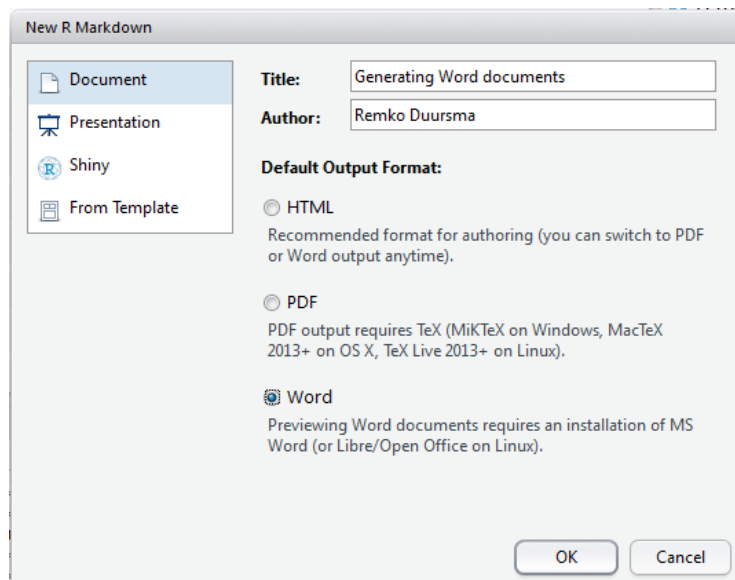


Figure 1.3: Go to **File** > **New File** > **R Markdown**, enter a title for your document and select Word document.

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
<code>echo</code>	Should the code be shown in the final document?	TRUE, FALSE
<code>eval</code>	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 are 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	**	<b>**bold text**</b>
Italic	*	<i>*italic text*</i>
Strike through	~~	~~crossed-out text~~
Superscript	^	x <sup>2</sup>
Subscript	~	CO <sub>2</sub>
Bulleted lists	*	<ul style="list-style-type: none"> <li>* A list item</li> <li>* Another list item</li> <li>* Yet another list item</li> </ul>
Numbered lists	1.	<ol style="list-style-type: none"> <li>1. First list item</li> <li>2. Second list item</li> <li>3. Third list item</li> </ol>
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 marker1
2. Strike boldly
3. Watch out for the people with the erasers!
```

```
###1Suggested 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<sup>1</sup>
2. Strike **boldly**
3. Watch out for the people with the erasers!

#### <sup>1</sup>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.

## 1.4 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

Minimum 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 [1.12.5](#)).

```
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 Generating data

### 1.5.1 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" "E. saligna"
```

## 1.5.2 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] 0.8815171 0.2138320 0.9304782 0.2642452 0.7464531 0.5761449 0.8444270
[8] 0.6054497 0.8271040 0.6100517

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

[1] 148.6012 524.7988 316.0703 708.5905 347.6635
```

**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] 8 4 11 6 2 4 8 5 11 4 15 7 8 5 10 5 14 14 7 4
```

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

## 1.6 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())
```



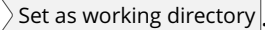
Finally, when you are ending your **R** session, but you want to continue exactly at this point the next time, make sure to save the current workspace. In RStudio, find the menu Session (at the top). Here

you can save the current workspace, and also load a previously saved one.

## 1.7 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   .

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  .

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

```
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`).

```
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`).

## 1.8 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 1.7), to make sure all the relevant files can be accessed.

```
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

## 1.9 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.

### 1.9.1 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...
```

## 1.10 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 (available for Windows only; Mac users have to download and install **R** manually):

```
library(installr)

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

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.

## 1.11 Accessing the help files

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

```
?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
```

## 1.12 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.

### 1.12.1 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  $\pi$ . *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`).

### 1.12.2 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.

### 1.12.3 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. 12).

```
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 13, 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)?

### 1.12.4 Scripts

This exercise will make sure you are able to make a 'reproducible 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 1.1.2 on page 4), 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).

### 1.12.5 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
```

### 1.12.6 Working with two vectors

- 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).
- Calculate the correlation between lengths and diameters (use the `cor` function).
- Calculate the volume of each cylinder ( $V = \text{length} * \pi * (\text{diameter} / 2)^2$ ).
- Calculate the mean, standard deviation, and coefficient of variation of the volumes.
- ◆ 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.
- ◆ 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?

### 1.12.7 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.

- ◆ 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"
- ◆ 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.

## 1.12.8 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)`.

## 1.12.9 Packages

This exercise makes sure you know how to install packages, and load them. First, read Section 1.9 (p. 19).

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 1.11 on p. 21)
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.

## 1.12.10 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 ?? on p. ??.

Before you start this exercise, first make sure you have a reproducible 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 1.7) 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.

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 ?? on page ??.

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  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. To use the `readxl` package, do:

```
library(readxl)
mydata <- read_excel("myspreadsheet.xls", sheet=2)
```

In this case we would read the data stored on the second sheet. It is also possible to read specified ranges of cells, skip rows of data, etc.

## 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:

```
mydata <- read.delim("clipboard", 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
PIM0:19 Min. : 4.83 Min. : 3.57 Min. : 2.636
PIP0:22 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 1.5.1.

```
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] 8.1 0.9 13.8 2.5 21.2

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 ?? 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 PIP0 69.60 39.37 189.7330 452.4246
38 PIP0 60.20 31.73 131.7273 408.3383
41 PIP0 70.61 31.93 411.1604 1182.4222
44 PIP0 60.20 35.14 169.0536 658.2397
45 PIM0 60.71 39.84 139.6512 139.2559
46 PIM0 70.61 40.66 376.3083 541.3062
58 PIM0 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] 417.20944 338.48562 103.67263 99.73679 21.64960 33.71358 25.38065
[8] 11.50285 15.47602 66.63367

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 PIP0 69.60 39.369999 189.733007 452.42455
24 PIP0 58.42 35.810000 253.308265 595.64015
25 PIP0 33.27 20.800001 91.538428 160.44416
26 PIP0 44.20 29.110001 90.453658 149.72883
27 PIP0 30.48 22.399999 99.736790 44.13532
28 PIP0 27.43 27.690001 34.464685 22.98360
29 PIP0 43.18 35.580000 68.150309 106.40410
30 PIP0 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.

```
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. 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 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.

### 2.5.1 Working with a single vector 2

Recall Exercise 1.12.3 on p. 23. Read in the `rainfall` data once more. We now practice subsetting a vector (see Section 2.3.1, p. 31).

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.

### 2.5.2 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?

### 2.5.3 Basic operations with the Cereal data

For this exercise, we will use the Cereal data, see Section ?? (p. ??) 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. 29).
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. 39).
9. ■ Rename the column 'Manufacturer' to 'Producer' (see Section 2.2.2, p. 31).

## 2.5.4 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. 27) (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 1.5.1, p. 16).
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.

## 2.5.5 Titanic

1. ■ Read the data described in Section ?? (p. ??)
2. ■ Make two new dataframes : a subset of male survivors, and a subset of female survivors. Recall Section 2.3.2 (p. 34) (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`.

## 2.5.6 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              | <code>c(1, 12.3491, 10/2, 10*6)</code>                       |                       |
| character | Character strings       | <code>c("E. saligna", "HFE", "a b c")</code>                 | <a href="#">3.5</a>   |
| factor    | Categorical variable    | <code>factor(c("Control", "Fertilized", "Irrigated"))</code> | <a href="#">3.2</a>   |
| logical   | Either TRUE or FALSE    | <code>10 == 100/10</code>                                    | <a href="#">3.3</a>   |
| Date      | Special Date class      | <code>as.Date("2010-6-21")</code>                            | <a href="#">3.6.1</a> |
| POSIXct   | Special Date-time class | <code>Sys.time()</code>                                      | <a href="#">3.6.2</a> |

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](#)).

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 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 ...
```

```
$ C02_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$C02_treatment <- as.factor(pupae$C02_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 ...
$ C02_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 5.2.5](#).

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.

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

Look at a summary to see if there are missing values:
summary(pupae)

T_treatment C02_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_sub1 <- subset(pupae, !is.na(Gender))

Option 2: take subset of data where Frass AND Gender are not missing
pupae_sub2 <- 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$T_C02 <- with(pupae, paste(T_treatment, C02_treatment, sep="-"))
head(pupae$T_C02)

[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_C02` (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 and row 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.

Dataframes can also have row names (while `names` refers to column names). In some cases, row names

can be useful to simplify indexing. For example,

```
Make up some data
dfr <- data.frame(weight=runif(3,20,30), size=runif(3,100,150))

Print the row names (the default ones are not very useful):
rownames(dfr)

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

Rename:
rownames(dfr) <- c("bear","panther","gorilla")

Use rownames to index, much like column names:
dfr["panther",]

weight size
panther 21.3319 118.1305
```

### 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.

```
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[grepl("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':
grepl("^Raisin",cerealnames)

[1] 59 60 61

Or end with 'Bran'
The $ symbol is part of a 'regular expression', and indicates 'ends with':
grepl("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:

```
grepl("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           |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------|---------------------------------|-------------------------|
| <code>^abc</code>   | Matches any string that starts with <i>abc</i>                                                                                                      | <code>^eco</code>          | economics,<br>ecology           | evo-eco                 |
| <code>abc\$</code>  | Matches any string that ends with <i>abc</i>                                                                                                        | <code>ology\$</code>       | biology, ecology                | ologies                 |
| <code>.</code>      | Matches any character except newline.                                                                                                               | <code>b.t</code>           | bat, bet, bit                   | beet                    |
| <code>a b</code>    | Matches string <i>a</i> or string <i>b</i> .                                                                                                        | <code>green blue</code>    | green, blue                     | red                     |
| <code>*</code>      | Matches a given string 0 or more times.                                                                                                             | <code>ab*a</code>          | aa, abba,<br>abbba, abbbba      | ba, ab                  |
| <code>?</code>      | Matches a given string 0 times or 1 time – no more.                                                                                                 | <code>ab?a</code>          | aa, aba                         | abba, abbbba            |
| <code>(abc)</code>  | Groups a string to be matched as part of an expression.                                                                                             | <code>(4570)</code>        | 4570 1125,<br>4570 1617         | 9772 6585,<br>9772 6100 |
| <code>[abc]</code>  | Matches any of the characters inside the brackets.                                                                                                  | <code>gr[ae]y</code>       | gray, grey                      | greey, graey            |
| <code>[^abc]</code> | Matches any characters <i>not</i> inside the brackets.                                                                                              | <code>b[^e]</code>         | bid, ban                        | bees, bed               |
| <code>-</code>      | Within brackets, gives a range of characters to be matched.                                                                                         | <code>[a-c]t</code>        | at, bt, ct                      | dt, et                  |
| <code>{n}</code>    | Used to give the number of times to match the preceding group.                                                                                      | <code>ab{2}a</code>        | abba                            | aba, abbba              |
| <code>\\</code>     | Used to "escape" a special character so it can be matched as part of an expression. Note that in <b>R</b> , you must use a double <code>\\</code> . | <code>why\\?</code>        | why?,<br>why ask why?           | why not                 |
| <code>\\s</code>    | Matches any whitespace character.                                                                                                                   | <code>day:\\s[A-z]</code>  | day: monday,<br>day: Fri        | day:Fri                 |
| <code>\\w</code>    | Matches any whole word.                                                                                                                             | <code>my \\w</code>        | my hobby,<br>my name            | his program             |
| <code>\\d</code>    | Matches any digit.                                                                                                                                  | <code>\\d years old</code> | 5 years old,<br>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](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)

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] "2017-05-16"

We can now simply subtract your birthday from today's date.
today() - as.Date("1976-5-22")
Time difference of 14969 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 ??), 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'))
```

## 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)
```

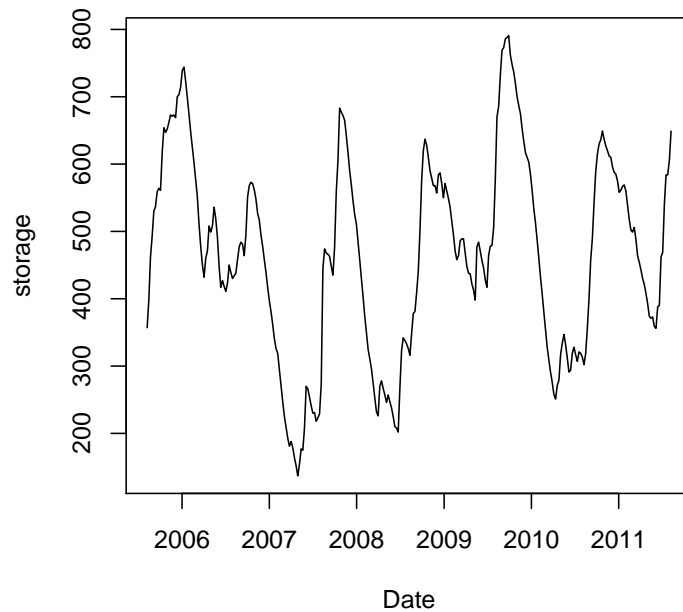


Figure 3.1: A simple plot of the hydro data.

```
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")

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] "2017-05-16 20:04:30 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 1.5.1).

```
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"
```

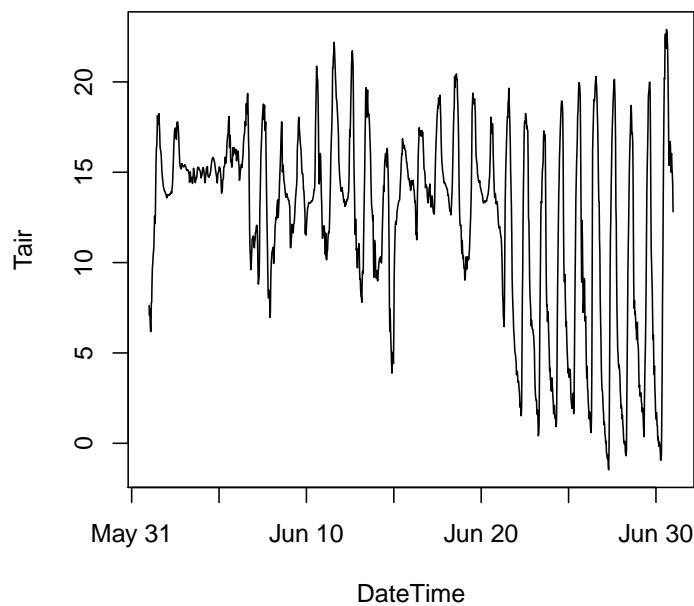


Figure 3.2: Air temperature for June at the HFE

```
[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"
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("2012-6-1")
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.

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 `C02_treatment` variable in the

pupae dataset a factor:

```
pupae <- read.csv("pupae.csv")
pupae$C02_treatment <- as.factor(pupae$C02_treatment)
```



## 3.8 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.

### 3.8.1 Titanic

For this section, read the data described in Section ?? (p. ??). *Note:* the data are TAB-delimited, use `read.table` as shown on p. ??.

1. ■ Convert the 'Name' (passenger name) variable to a 'character' variable, and store it in the dataframe. See Section 3.5.3 (p. 53).
2. ■ How many observations of 'Age' are missing from the dataframe? See examples in Section 3.4 (p. 48).
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. 44).
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. 47).
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. 50), and then use `table`, as well as `nrow`.

### 3.8.2 Hydro dam

Use the hydro dam data as described in Section ??.

1. ■ Start by reading in the data. Change the first variable to a `Date` class (see Section 3.6.1, p. 56).
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`.

### 3.8.3 HFE tree measurements

Use the data for the HFE irrigation x fertilisation experiment (see Section ??, p. ??).

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. 48).
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. 47).
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`.

### 3.8.4 Flux data

In this exercise, you will practice useful skills with the flux tower dataset. See Section ?? (p. ??) for a description of the dataset.

1. ■ Read the dataframe. Rename the first column to 'DateTime' (recall Section 3.5.2 on p. 52).
2. ■ Convert DateTime to a `POSIXct` class. Beware of the formatting (recall Section 3.6.2 on p. 58).
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. 44).
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`.

### 3.8.5 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. 51), 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 ?? *Hint* : use a combination of `paste`, `strsplit`, `gsub`, `table` and `sort`.

### 3.8.6 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 1.5.2, p. 17 with `replacement`, and use `paste` as shown in Section 3.5, p. 51). 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.

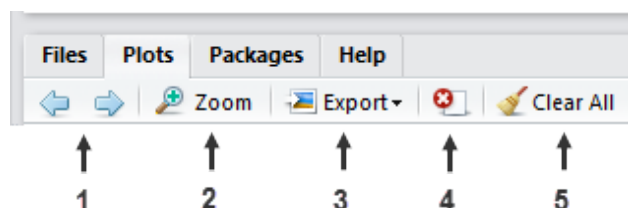


Figure 4.1: The plotting window in RStudio. 1 : cycle through plotting history, 2 : displays the plot in a large window, 3 : Export the plot to an image or PDF, 4 : delete the current plot from the plotting history, 5 : clear all plotting history

### 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                                                                   |
|----------------------|------------------------------------------------------------------------------|
| <code>plot</code>    | Scatter plots and various others (Section 4.3.1)                             |
| <code>barplot</code> | Bar plot (including stacked and grouped bar plots) (Section 4.3.2)           |
| <code>hist</code>    | Histograms and (relative) frequency diagrams (Section 4.3.3)                 |
| <code>curve</code>   | Curves of mathematical expressions (Section 4.3.3)                           |
| <code>pie</code>     | Pie charts (for less scientific uses) (Section 4.3.4)                        |
| <code>boxplot</code> | Box-and-whisker plots (Section 4.3.5)                                        |
| <code>symbols</code> | 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.

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 5.2.1.

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")
```

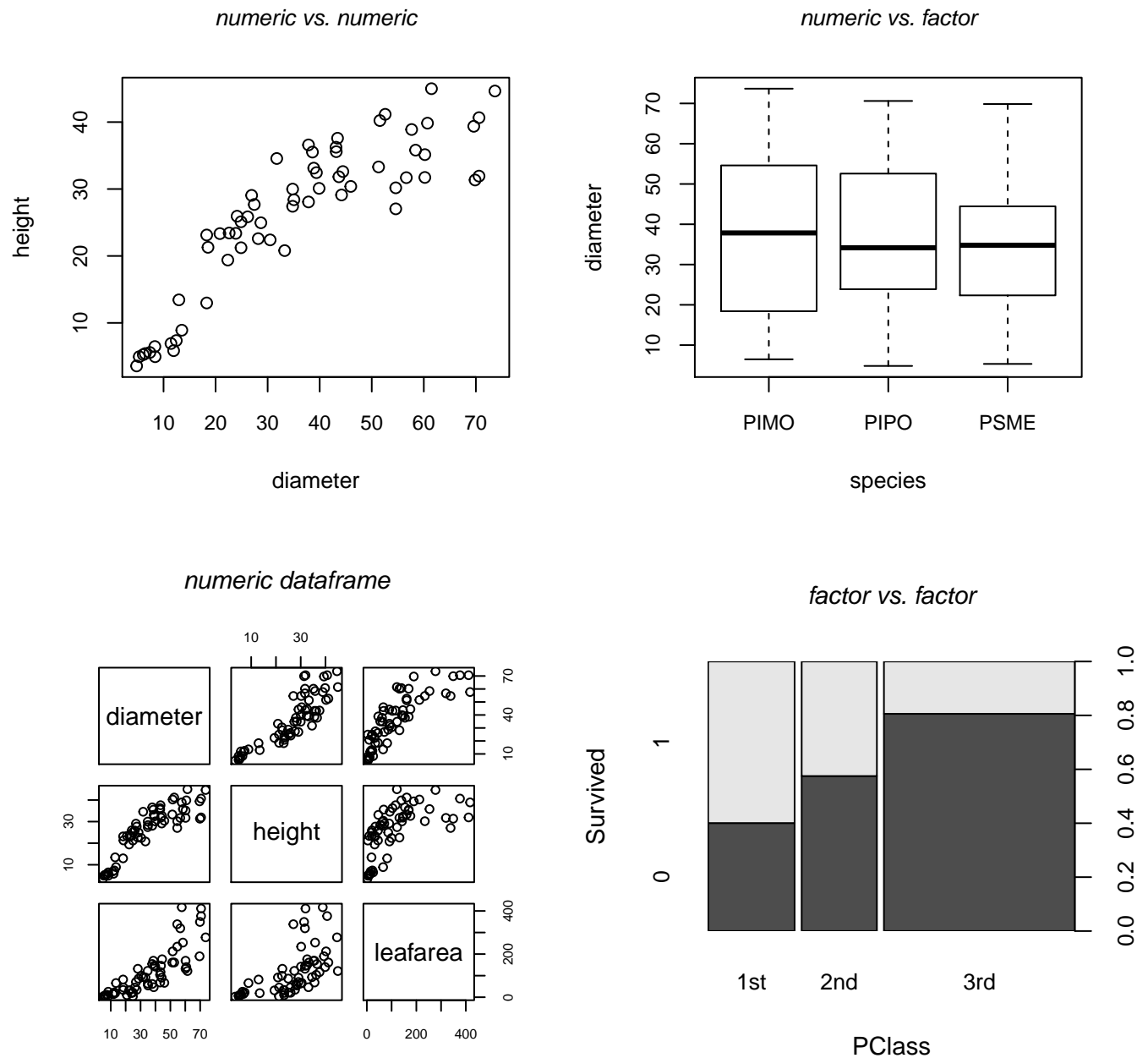


Figure 4.2: Four possible outcomes of a basic call to `plot`, depending on whether the Y and X variables are numeric or factor variables. The term 'numeric dataframe' means a dataframe where all columns are numeric.

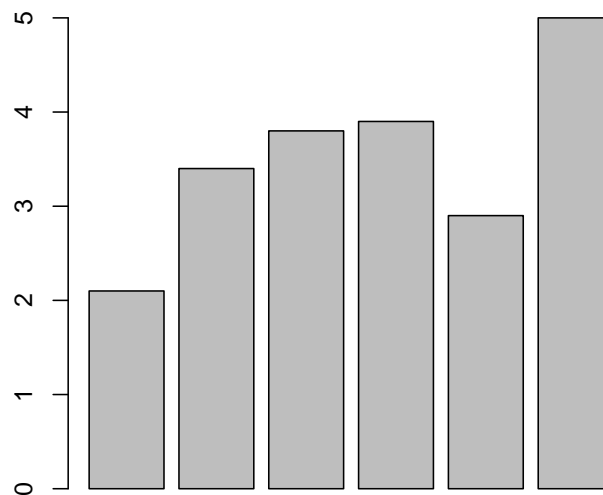


Figure 4.3: Simple bar plot with default settings

```
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, ratingManufacturerSD, 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.

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.

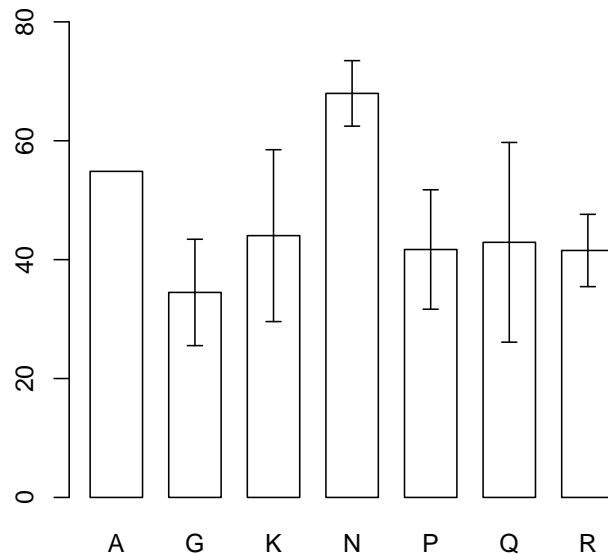


Figure 4.4: Simple bar plot with error bars (shown are means  $\pm$  one standard deviation).

```
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.



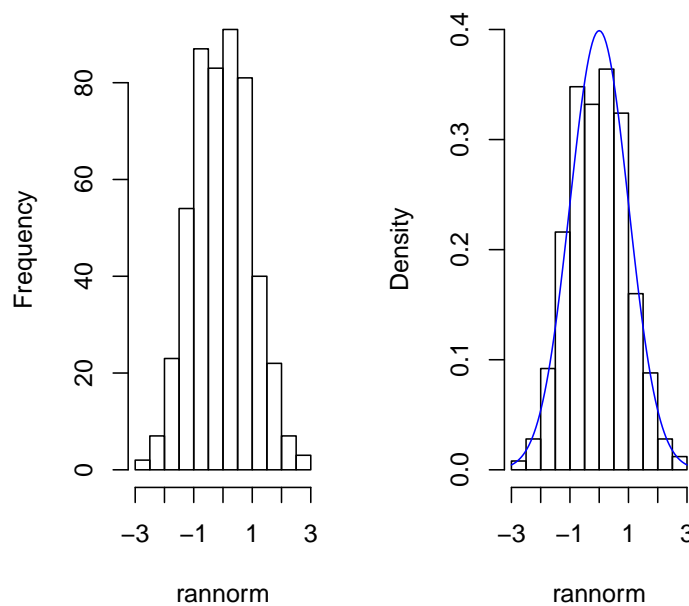


Figure 4.5: Two histogram examples for some normally distributed data.

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.

```
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

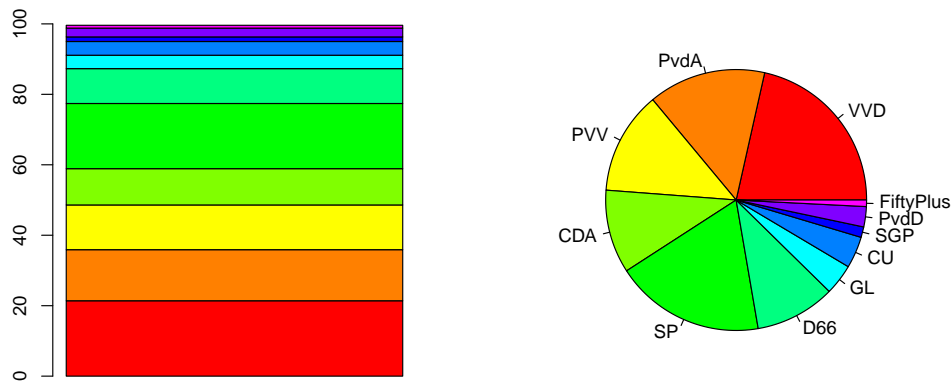


Figure 4.6: A stacked bar and pie chart for the election poll data.

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.

```
A subset for the first and last dates
percentages2 <- election[c(1, nrow(election)), -1]

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")
```

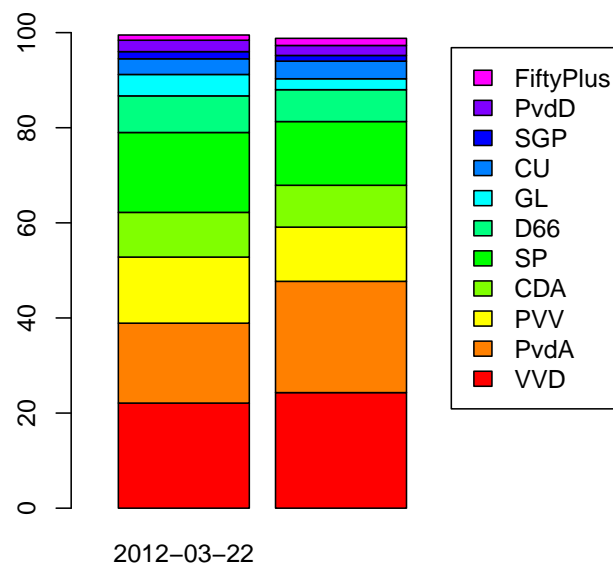


Figure 4.7: A stacked bar chart for the election poll data at two timepoints.

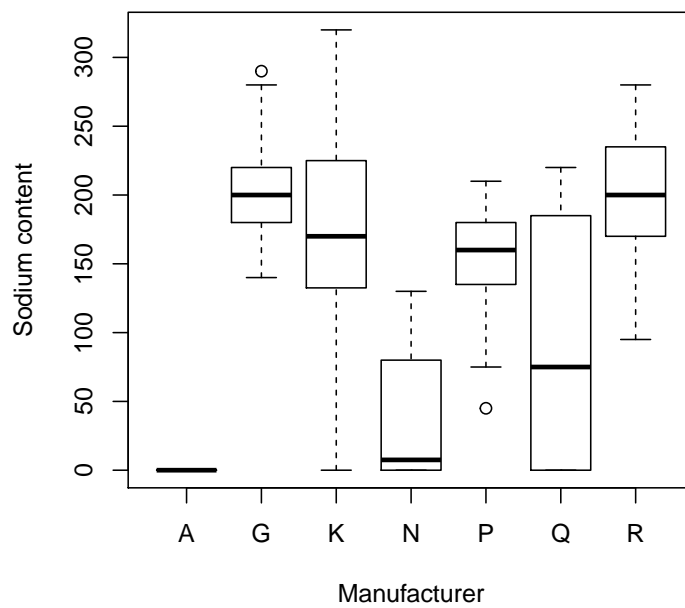


Figure 4.8: A simple box plot for the cereal data.

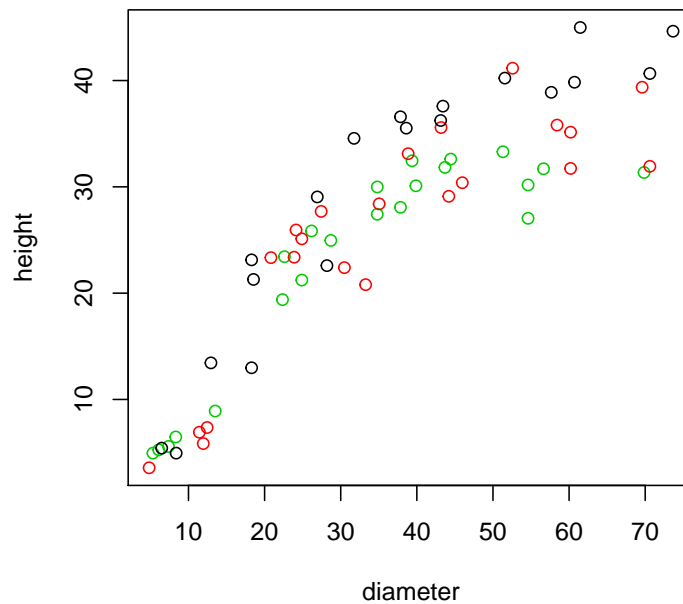


Figure 4.9: Simple scatter plot with default settings

## 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

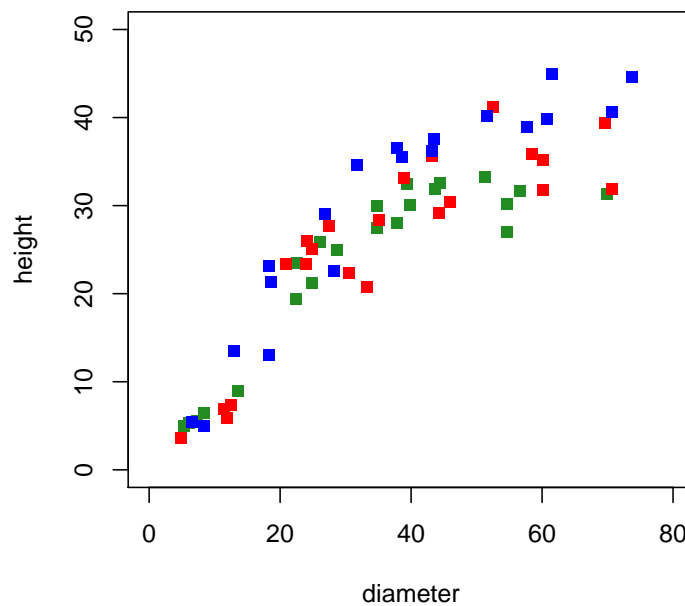


Figure 4.10: Simple scatter plot : changed plotting symbol and X and Y axis ranges.

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).

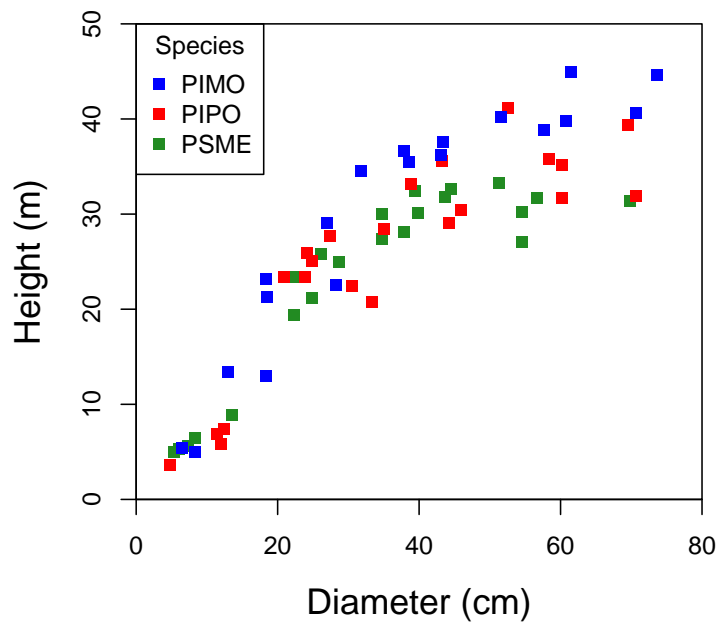


Figure 4.11: Simple scatter plot : many settings customized.

```
plot(x,y, col=3)
```

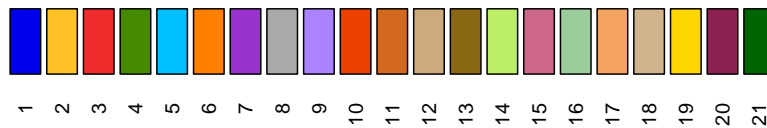
To pick one or more of the 657 built-in colours in **R**, this website is very useful : <http://research.stowers-institute.org/efg/R/colour/Chart/>, especially the link to a PDF at the top, which lists the colours by name. You can also use the built-in `demo` function to show the colours,

```
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.

```
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).

**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 ??). 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)
```

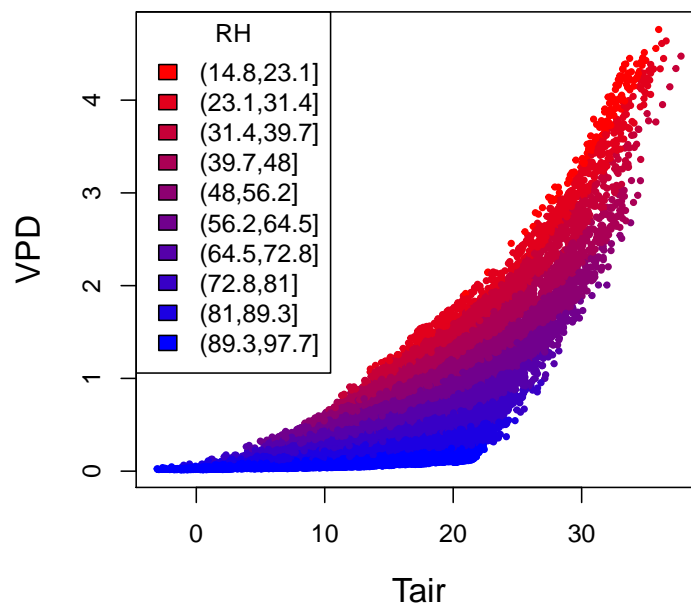


Figure 4.12: Relationship between air temperature, vapour pressure deficit and relative humidity at the HFE.

```
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")
```

## 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.

```
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))
```



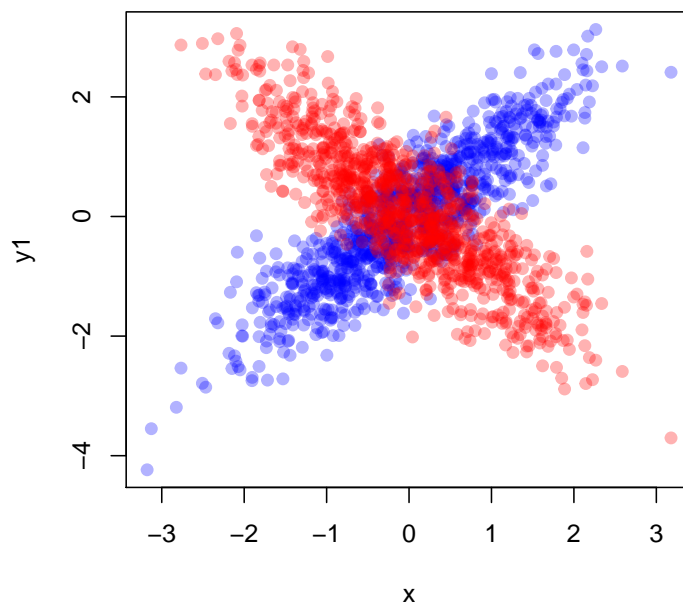


Figure 4.13: A plot of some random numbers, using a semi-transparent colour.

### 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.

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"))
```

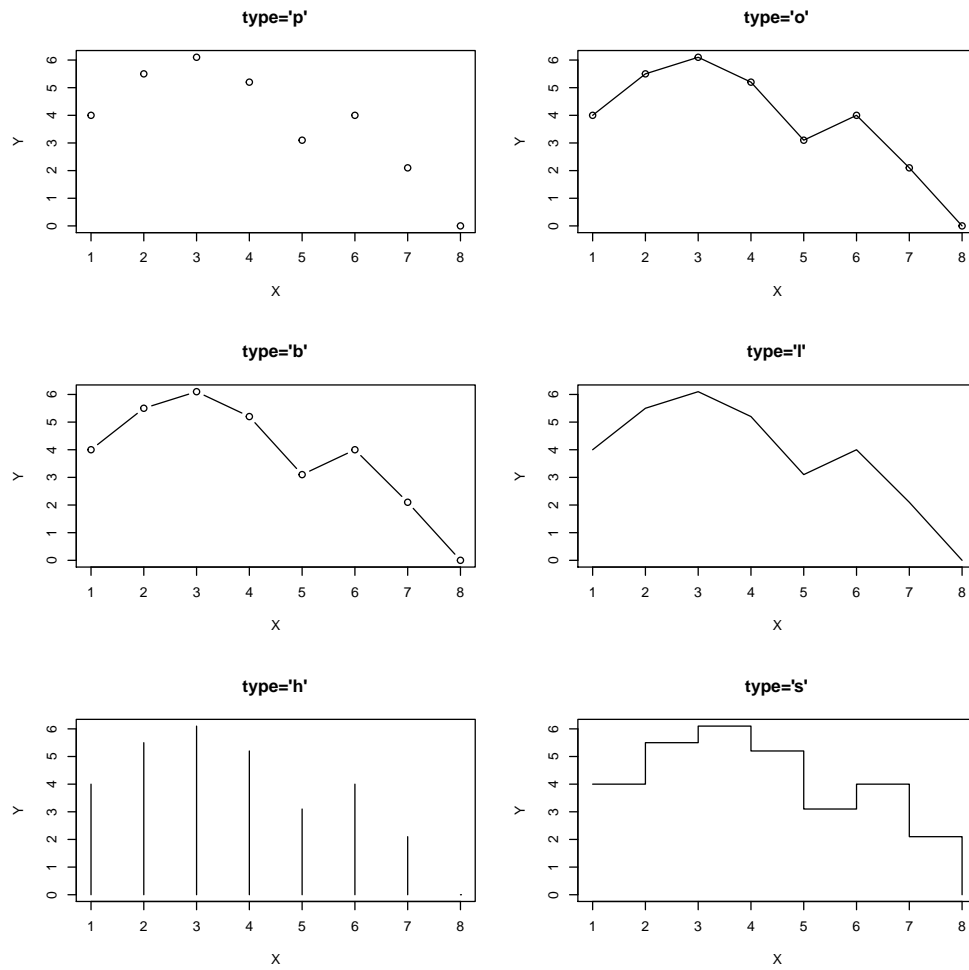


Figure 4.14: Four options for the plotting type (with type=).

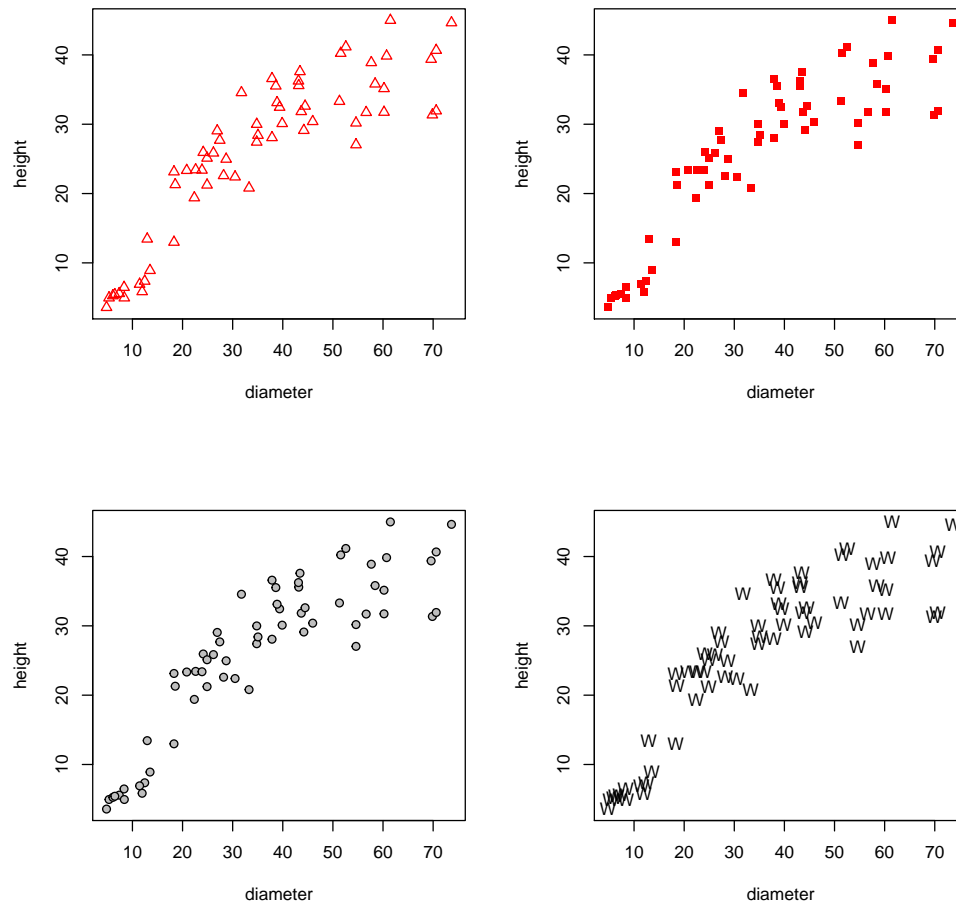


Figure 4.15: Four options for the plotting symbols (with pch).

```
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)
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.

```
palette(c("blue", "red", "forestgreen"))
with(allom, plot(diameter, height,
 col=species,
 pch=c(1,2,15)[species],
```

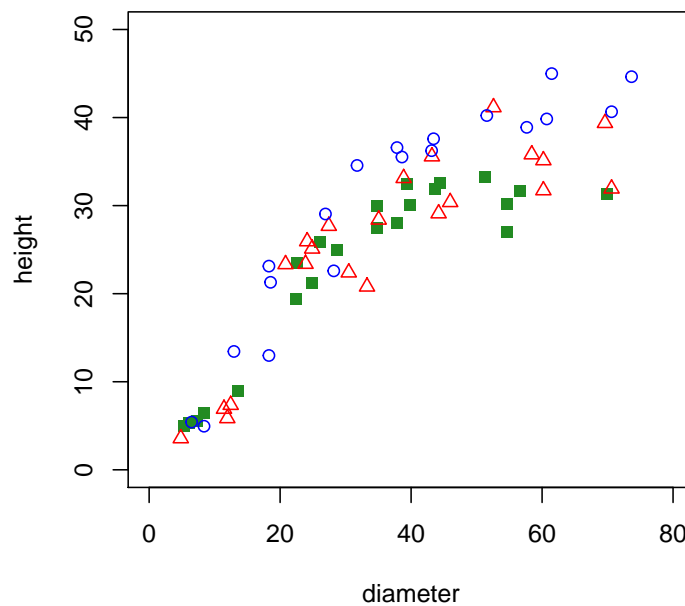


Figure 4.16: Scatter plot : vary colour and symbol by species

```
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.

#### 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 ( $\text{cm}^2$ )

```
expression(Photosynthesis~ ~(mu*mol~m^-2~s^-1))
```

Photosynthesis ( $\mu\text{mol m}^{-2} \text{s}^{-1}$ )

```
expression(Temperature~ ~(degree*C))
```

Temperature (°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](http://en.wikipedia.org/wiki/Unicode_symbols)). You can plot those with this short example (results not shown):

```
expression(Per mille~"\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")
```

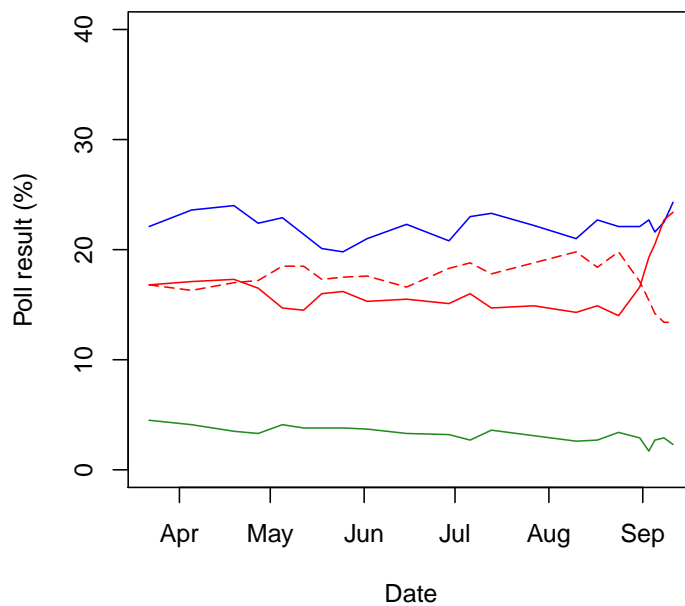


Figure 4.17: Adding lines to an existing plot.

## 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 ??.

```
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:

```
Add a bold label 'A' to an existing plot:
legend("topleft", expression(bold(A)), bty='n', inset=0.01)
```

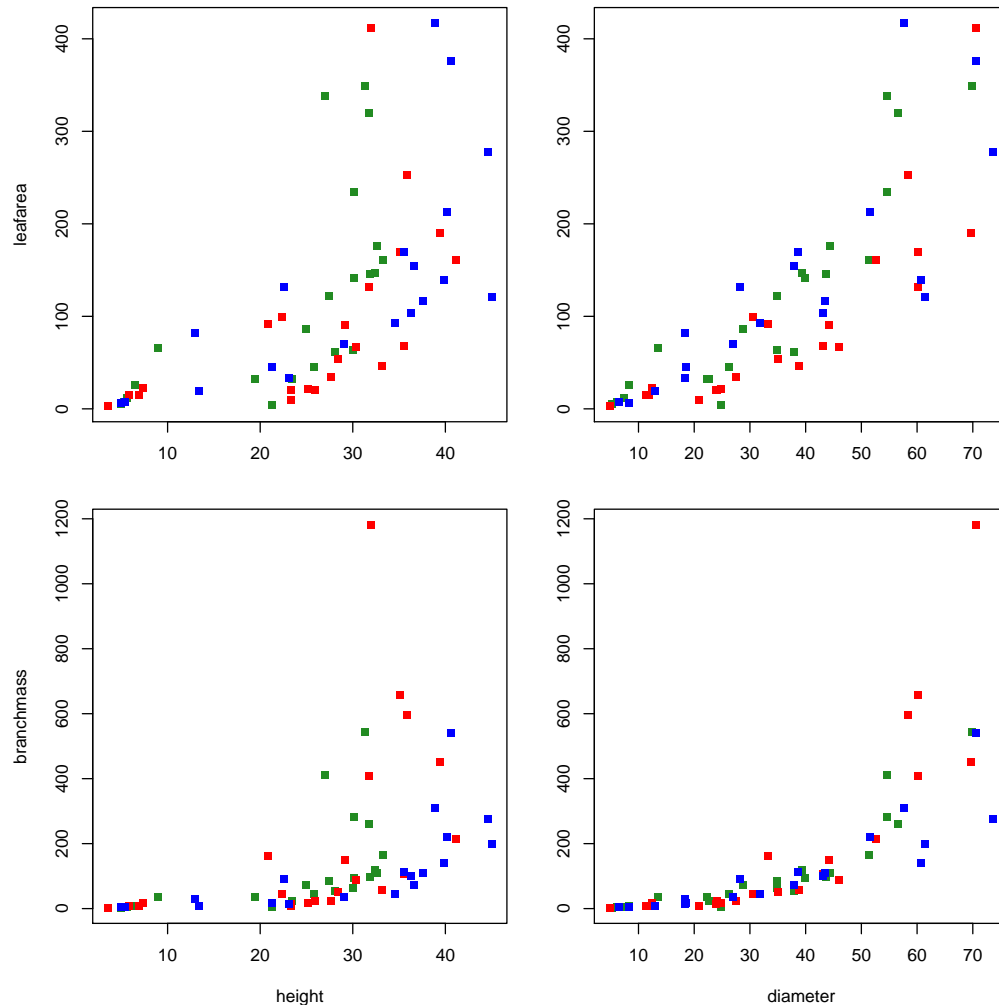


Figure 4.18: Multiple plots within a single plot window.

**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 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                                                                                                                           |
|----------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------|
| <code>pch</code>                             | Sets the type of symbols used in the plot; see <code>points()</code> for a list of options.                                           |
| <code>type</code>                            | Sets whether to plot points, lines, both, or something else (see <code>?plot</code> .)                                                |
| <code>col</code>                             | Sets the colour of plotting symbols and lines.                                                                                        |
| <code>lty</code>                             | Sets the line type (1=solid, 2=dashed, etc.)                                                                                          |
| <code>lwd</code>                             | Sets the line width                                                                                                                   |
| <code>cex</code>                             | Controls the size of text and points in the plot area. Short for 'character expansion', it acts as a multiplier of the default value. |
| <code>cex.axis</code> , <code>cex.lab</code> | Character expansion of axes and the labels.                                                                                           |
| <code>cex.main</code>                        | Character expansion of the title of the plot.                                                                                         |
| <code>family</code>                          | Sets the font for labels and titles. Varies by system, but 'serif', 'sans' and 'mono' should always work.                             |
| <code>bty</code>                             | Sets the type of box, if any, to be drawn around the plot. Use <code>bty='n'</code> for none.                                         |
| <code>las</code>                             | Sets the orientation of the text labels relative to the axis                                                                          |
| <code>mar</code>                             | Sets the number of lines in each margin, in the order bottom, left, top, right.                                                       |
| <code>xaxs</code> , <code>yaxs</code>        | Preset functions for calculating axis intervals.                                                                                      |
| <code>xaxp</code> , <code>yaxp</code>        | Sets the coordinates for tick marks on each axis.                                                                                     |
| <code>xaxt</code> , <code>yaxt</code>        | 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).

```
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 ??). 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",
```

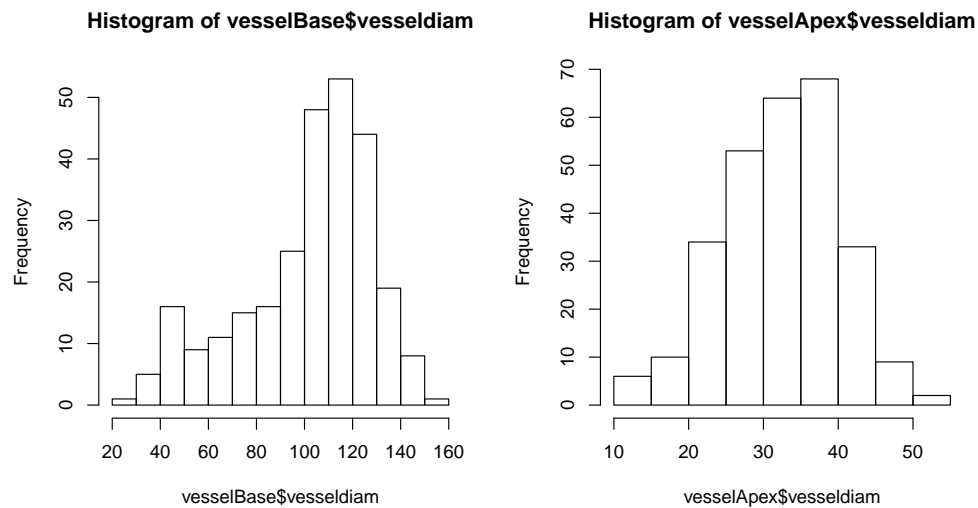


Figure 4.19: Two simple default histograms

```
xlim=c(0,160), breaks=seq(0,160,by=10),
xlab=expression(Vessel~diameter~ ~(mu*m)),
ylab="Number of vessels")
```

```
Second panel
```

```
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"))
```

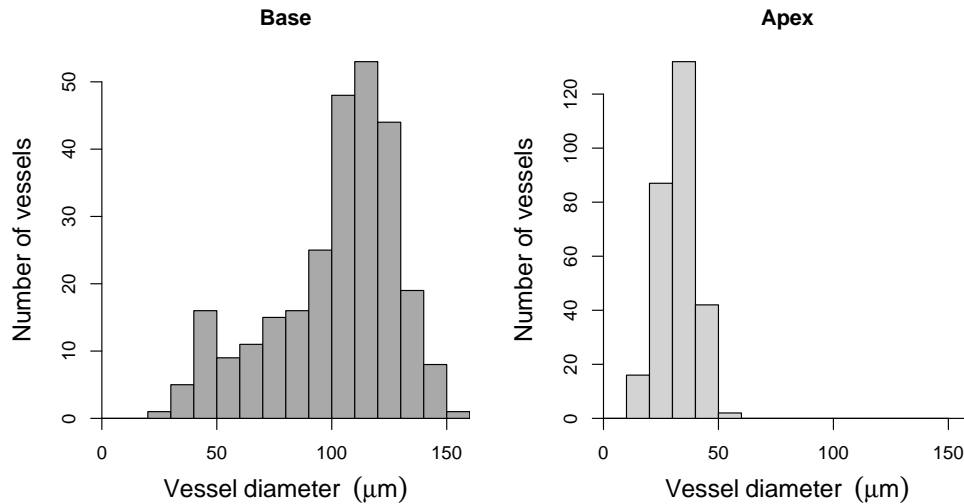


Figure 4.20: Two customized histograms

```
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)
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] ~ ~("°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')
```

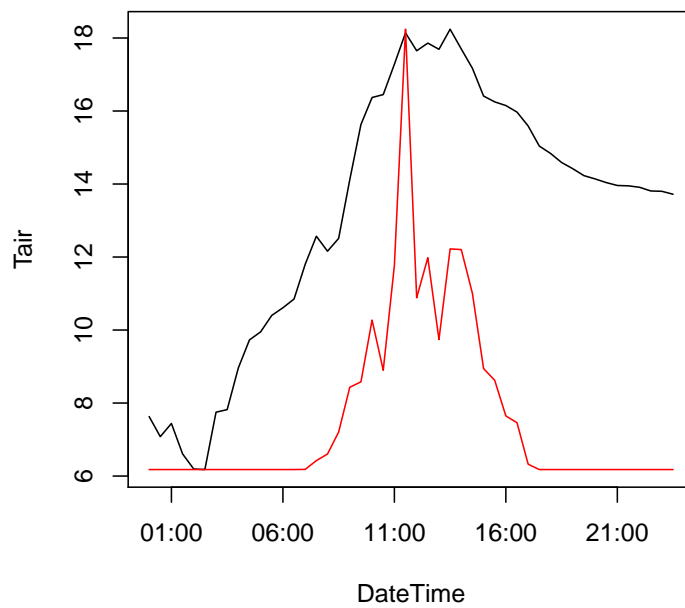


Figure 4.21: A default plot with two axes

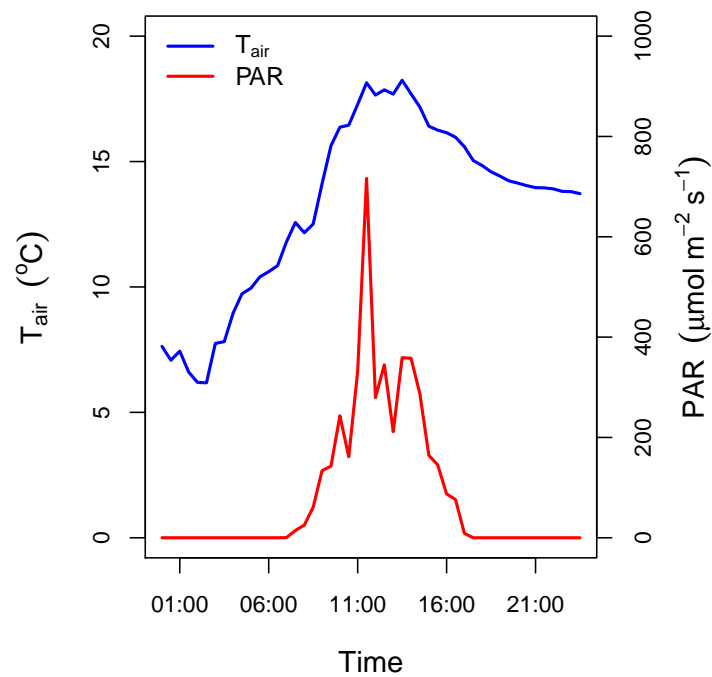


Figure 4.22: A prettified plot with two axes

## 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.

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 `ggplots2` 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.

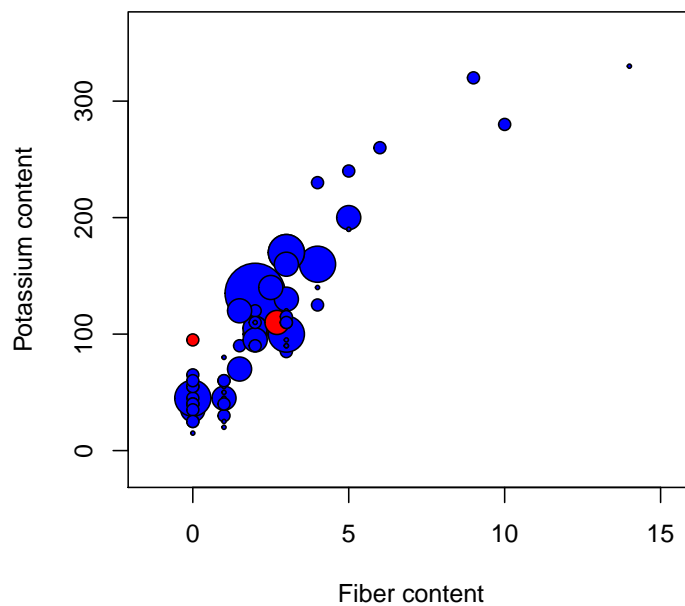


Figure 4.23: Cereal data with symbols size as a function of fat content and colour as function of whether the cereal is served hot (red) or cold (blue).

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
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))
```

### 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)
```

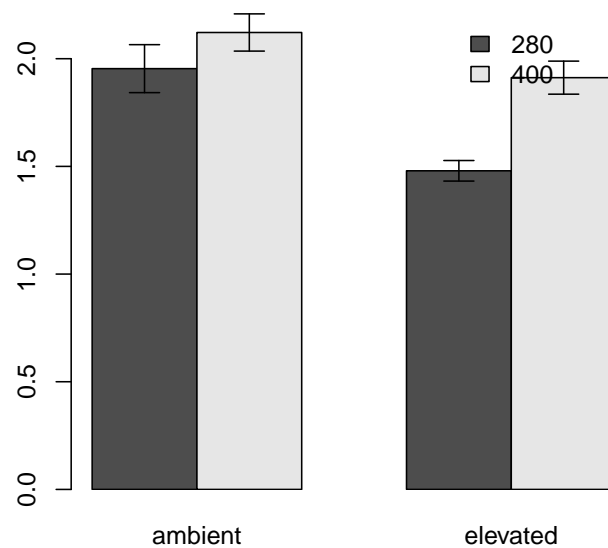


Figure 4.24: Mean frass by temperature and CO2 treatment made with bargraph.CI()

```
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),
 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()
```



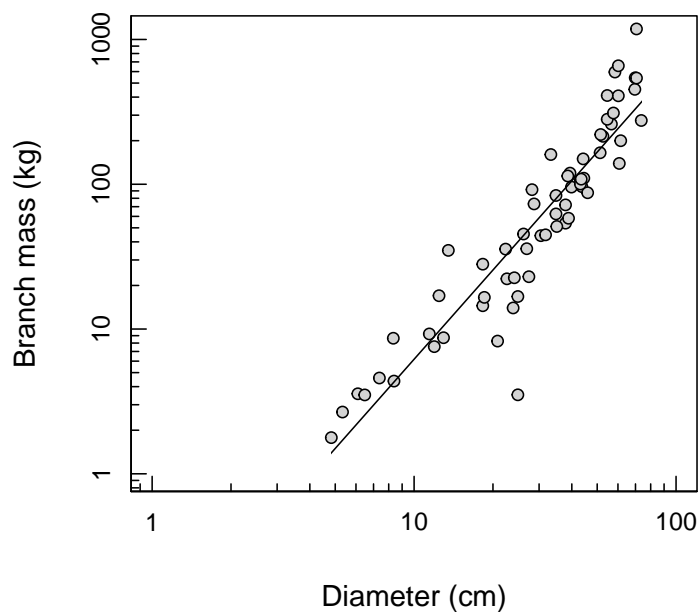


Figure 4.25: Branch mass versus tree diameter on a log-log scale. The axes were produced with the `magicaxis` package.

## 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")
```

## 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.

## 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:

```
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 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.

### 4.8.1 Scatter plot with the pupae data

1. ■ Read the pupae data (see Section ??, p. ??). Convert 'CO<sub>2</sub>\_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<sub>2</sub> 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<sub>2</sub> 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<sub>2</sub> 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<sub>2</sub> : T).
8. ◆ In Fig. 4.4, figure out why no error bar was plotted for the first bar.

### 4.8.2 Flux data

Use the Eddy flux data for this exercise (Section ??, p. ??).

1. ◆ Produce a line plot for FC02 for *one day* out of the dataset (recall Section 3.6.2, p. 58).
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. 85), with different colours for when the variable `ustar` is less than 0.15 ('bad data' in red). *Hint:* recall Section 3.2 on p. 44 on how to split a numeric vector into a factor.

### 4.8.3 Hydro dam

Use the hydro dam data as described in Section ??.

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. 44).

### 4.8.4 Coloured scatter plot

Use the Coweeta tree data (see Section ??, p. ??).

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. 34).
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.

### 4.8.5 Superimposed histograms

First inspect the example for the vessel data in Section 4.5.1 (p. 89).

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

### 4.8.6 Trellis graphics

1. ◆ Change the labels in the box and whisker plot (Fig. ??) to indicate the units of  $\text{CO}_2$  concentration (ppm) and add a label to the x-axis indicating the factor (temperature).

## Chapter 5

# Summarizing, tabulating and merging data

### 5.1 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
PIM0:19 Min. : 4.83 Min. : 3.57 Min. : 2.636
PIP0:22 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 PIM0, PIP0 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 "PIM0","PIP0",...: 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 1.9). 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 C02_treatment is a factor (it will be read as a number)
pupae$C02_treatment <- as.factor(pupae$C02_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
C02_treatment 2 integer 0
Gender integer 6
PupalWeight double 0
Frass double 1
##
+-----+-----+
|Variable|Levels|
+-----+-----+
|T_treatment|ambient,elevated|
+-----+-----+
|C02_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`.

## 5.2 Making summary tables

### 5.2.1 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 ??), 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. 5.1)

```
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")
```



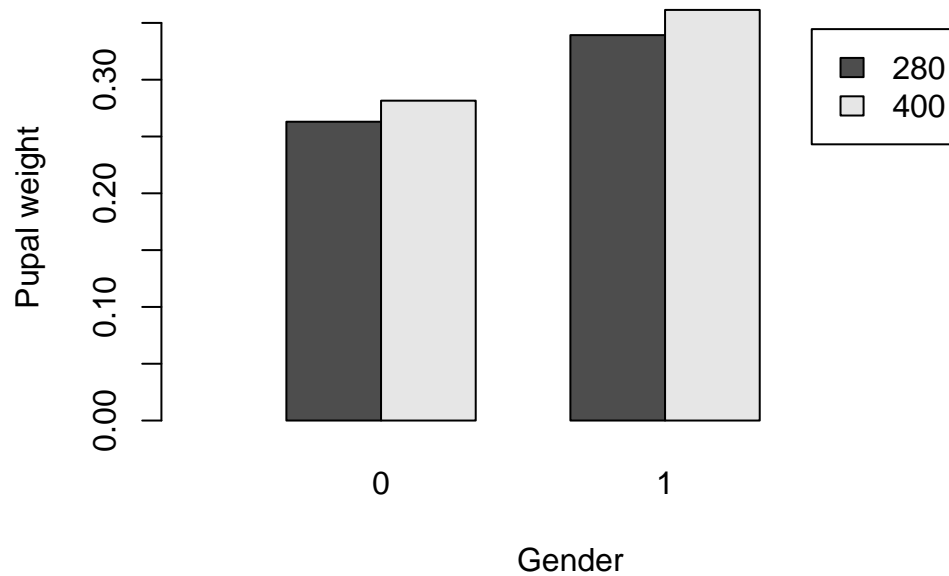


Figure 5.1: A grouped barplot of average pupal weight by CO2 and Gender for the pupae dataset. This is easily achieved via the use of `tapply`.

### 5.2.2 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(dplyr)
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 ~ C02_treatment + T_treatment,
 data=pupae, FUN=c(mean,sd), na.rm=TRUE)

C02_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 C02 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 ~ C02_treatment + T_treatment,
 data=pupae_nona,
 FUN=c(mean,sd,length))

C02_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

```

## 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 ??). 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. 5.2.

```

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

```

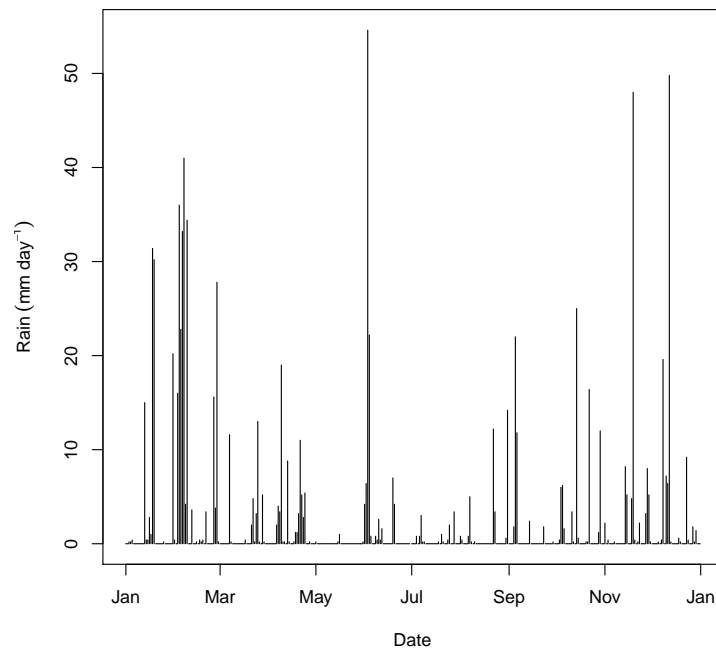


Figure 5.2: Daily rainfall at the HFE in 2008

```
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)))
```

### 5.2.3 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.

Consider these two examples, using the `xylem` vessel data (Section ??), and the `titanic` data (see Section ??).

```
Read vessel data
vessel <- read.csv("vessel.csv")

Count observations by position
xtabs(~ position, data=vessel)

position
apex base
279 271

Read titanic data
titanic <- read.table("titanic.txt", header=TRUE)

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
```

## 5.2.4 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 5.2.1).

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 5.3.1). 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.

## 5.2.5 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. 5.3.

```
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.

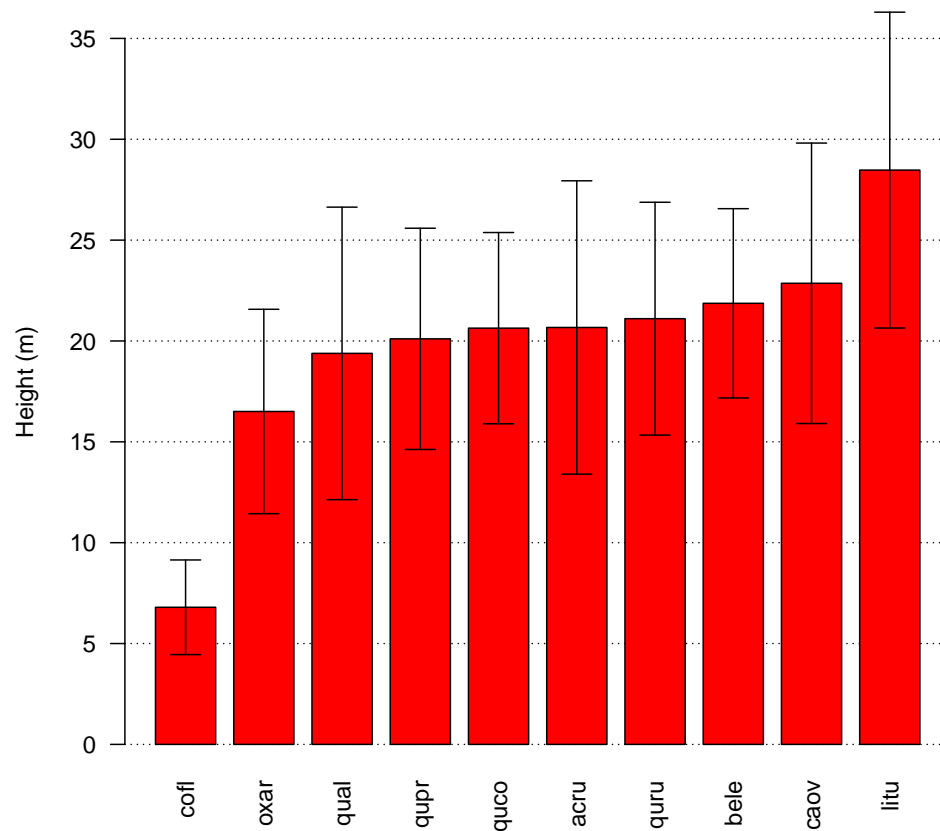


Figure 5.3: An ordered barplot for the coweeta tree data (error bars are 1 SD).



## 5.3 Combining dataframes

### 5.3.1 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:
combddata <- merge(data1, data2, by="unit")

Combined data
combddata
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:
combddata <- 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)
combddata

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 'combddata' above!
```

## Merging multiple datasets

Consider the cereal dataset (Section ??), 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.
```

### 5.3.2 Row-binding dataframes

If we have the following dataset called `plantdat`,

| 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 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          |

If you have dataframes that *don't* have the same names and/or order of the columns, consider the `rbind_all` function in the `dplyr` package.

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
```

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 `rbind.fill` function from the `plyr` 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
library(plyr)

##
Attaching package: 'plyr'
##
The following objects are masked from 'package:Hmisc':
##
is.discrete, summarize
##
The following object is masked from 'package:maps':
##
ozone
##
The following object is masked from 'package:lubridate':
##
here

rbind.fill(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.

## 5.4 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")
```

### 5.4.1 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<sub>2</sub>.

```
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.

```
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 ??.) 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).

```
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 `?pander.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:

```
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:

```
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 (but poorly proof-read) introduction to the things you can do with `pander` called "Rendering tables with `pandoc.table`" at [https://cran.r-project.org/web/packages/pander/vignettes/pandoc\\_table.html](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>.

## 5.5 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.

### 5.5.1 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 5.2.3, p. 108) 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 5.2.3, p. 108).

### 5.5.2 Words and the weather

1. ■ Using the 'Age and memory' dataset (first read Section ?? on p. ?? 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 ??, p. ??), find the mean air temperature by month. To do this, first add the month variable as shown in Section 3.6.2 (p. 58).

### 5.5.3 Merge new data onto the pupae data

1. ■ First read the pupae data (see Section ??, p. ??). Also read this short dataset, which gives a label 'roomnumber' for each CO<sub>2</sub> treatment.

```
CO2_treatment	Roomnumber
```

|     |   |  |
|-----|---|--|
| 280 | 1 |  |
| 400 | 2 |  |

To read this dataset, consider the `data.frame` function described in Section 2.1.2 (p. 27).

2. ■ Merge the short dataset onto the pupae data. Check the result.

## 5.5.4 Merging multiple datasets

Read Section 5.3.1 (p. 114) 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.

## 5.5.5 Ordered boxplot

1. First recall Section 4.3.5 (p. 73), and produce Fig. 4.8 (p. 74).
2. ♦ Now, redraw the plot with `Manufacturer` in order of increasing mean sodium content (use `reorder`, see Section 5.2.5 on p. 109).
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).

## 5.5.6 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 ??, p. ??).

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?

## 5.5.7 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 ?? on p. ??).

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. 60).
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. 60).
  - 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  $y=0$ .

## Technical information

This book was compiled with the following R version and packages.

- R version 3.4.0 (2017-04-21), x86\_64-w64-mingw32
- Locale: LC\_COLLATE=English\_Australia.1252, LC\_CTYPE=English\_Australia.1252, LC\_MONETARY=English\_Australia.1252, LC\_NUMERIC=C, LC\_TIME=English\_Australia.1252
- Running under: Windows 7 x64 (build 7601) Service Pack 1
- Matrix products: default
- Base packages: base, datasets, graphics, grDevices, methods, stats, utils
- Other packages: celestial 1.3, doBy 4.5-15, epade 0.3.8, Formula 1.2-1, ggplot2 2.2.1, gplots 3.0.1, gtools 3.5.0, Hmisc 4.0-2, knitr 1.15.20, lattice 0.20-35, lubridate 1.6.0, magicaxis 2.0.0, mapproj 1.2-4, maps 3.1.1, MASS 7.3-47, plotrix 3.6-4, plyr 1.8.4, RANN 2.5, scales 0.4.1, sciplot 1.1-0, sm 2.2-5.4, survival 2.41-3
- Loaded via a namespace (and not attached): acepack 1.4.1, backports 1.0.5, base64enc 0.1-3, bitops 1.0-6, caTools 1.17.1, checkmate 1.8.2, cluster 2.0.6, colorspace 1.3-2, compiler 3.4.0, data.table 1.10.4, digest 0.6.12, evaluate 0.10, foreign 0.8-68, formatR 1.5, gdata 2.17.0, grid 3.4.0, gridExtra 2.2.1, gtable 0.2.0, highr 0.6, htmlTable 1.9, htmltools 0.3.6, htmlwidgets 0.8, KernSmooth 2.23-15, latticeExtra 0.6-28, lazyeval 0.2.0, magrittr 1.5, Matrix 1.2-10, munsell 0.4.3, nnet 7.3-12, RColorBrewer 1.1-2, Rcpp 0.12.10, rlang 0.0.0.9018, rpart 4.1-11, splines 3.4.0, stringi 1.1.5, stringr 1.2.0, tibble 1.3.0.9002, tools 3.4.0

Code is available on [www.bitbucket.org/remkoduursma/hiermanual](http://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.

- !, 33, 47
- ;, 16
- ==, 33, 47
- ?, 21
- ??, 21
- \$, 30
- %in%, 33
- &, 33, 48
- |, 33
- |, 48
- >, 33
- >, 47
- <, 33
- <, 48
  
- abline, 86
- all, 64
- any, 40, 58
- as.Date, 56, 57
- as.factor, 44, 52
- as.numeric, 49
- ave, 109
  
- bargraph.CI, 93
- barplot, 68, 72, 77, 103
- barplot2, 68
- boxplot, 73, 123
  
- c, 16, 23
- cbind, 118
- ceiling, 6
- colorRampPalette, 78
- complete.cases, 50
- contents, 101, 122
- cor, 24
- cumsum, 13
- curve, 71
- cut, 46, 78, 99
  
- data.frame, 26, 29, 123
- demo, 77
- describe, 65, 101, 122
- dev.copy2eps, 96
- dev.copy2pdf, 96
- dev.off, 84
- diff, 13, 64, 64
- difftime, 56
- dir, 18
- doBy**, 105
- droplevels, 46
- duplicated, 40
- dust, 120, 120
  
- epade**, 99
- example, 25
- expression, 83, 86, 91
  
- factor, 45, 45
- floor, 6
- foreign**, 28
- ftable, 108
  
- ggplot2.**, 93
- ggplots2**, 93
- gplots**, 68, 93
- grep, 53, 53, 54
- grepl, 53, 54
- grey, 78
- gsub, 66
  
- head, 13, 27, 29, 40
- heat.colors, 78
- hist, 70, 89
- Hmisc**, 65, 101, 122
- hour, 60
  
- ifelse, 45, 46, 64
- install.packages, 19
  
- installr**, 21
- intersect, 25
- is.character, 44
- is.Date, 44
- is.factor, 44
- is.logical, 44
- is.na, 49, 65
- is.numeric, 44
- ISOdate, 57
  
- kable, 119, 119, 121
- knitr**, 8, 9, 119, 119
  
- layout, 87
- legend, 76, 91
- length, 13, 108
- LETTERS, 24, 40
- letters, 24, 40
- levels, 45, 45, 47, 65
- levels(), 47
- library(), 20
- list.files, 18
- ls, 17, 42
- lubridate**, 44, 56, 57, 58
  
- magicaxis**, 94
- match, 52
- max, 13, 58, 109
- mdy, 57
- mean, 13
- median, 13
- merge, 112, 116
- min, 13, 58
- minute, 60
- month, 60
- mtext, 86, 91
  
- names, 31, 52
- nchar, 15, 51, 65

- ncol, 31
- nlevels, 65
- now, 59
- nrow, 31
- order, 13, 41
- p\_load, 20, 20
- pacman**, 20, 20
- palette, 75–77
- pander, 119, 119–121
- pander**, 119
- pandoc**, 9
- pandoc.table, 121
- par, 84, 88, 88, 89, 97
- paste, 52, 66
- pie, 71
- pixiedust**, 120, 120, 121
- plot, 68, 80, 88
- plotCI, 68
- plotrix**, 68, 93
- plyr**, 117
- points, 85, 88, 98
- R markdown
  - code chunks in, 10
  - formatting body text, 11
  - introduction to, 8
  - kable function, 119
  - loading packages in, 19
  - pander package, 119
  - pixiedust package, 120
  - tables in, 118
- R scripts
  - introduction to, 6
- rainbow, 78
- rbind, 116, 117
- rbind.fill, 117
- read.csv, 26, 26, 27, 53, 61
- read.table, 28, 53, 61
- readLines, 65
- readxl**, 27
- reorder, 109, 123
- rep, 16, 16, 24
- rev, 13, 110
- rexp, 17
- rm, 17, 42
- rmarkdown**, 9
- rnorm, 17
- round, 6, 13, 22
- rownames, 27, 53
- runif, 17, 22
- sample, 17, 22, 24, 66
- scales**, 79
- sciplot**, 93
- sd, 13
- seq, 16, 60
- setdiff, 25
- setwd, 19
- showtext**, 84, 85, 85
- sort, 13, 15, 24, 66
- str, 29, 43, 52, 65, 100, 101, 122
- strsplit, 66
- subset, 34, 36, 36–38, 48, 50, 64
- substr, 51
- summary, 30, 65, 100, 101, 122
- summaryBy, 105, 105, 108, 109
- symbols, 93
- table, 45, 65, 66
- tail, 27
- tapply, 68, 103, 103, 108, 109
- text, 84, 86
- today, 57
- union, 25
- unique, 13, 65
- var, 13
- which, 49, 51
- which.max, 13, 41
- which.min, 13, 41
- windowsFonts, 84, 84
- with, 30
- write.csv, 39, 41, 118
- write.table, 39, 118
- xtabs, 108, 122
- yday, 60
- ymd, 57