

Learning statistics with R: A tutorial for psychology students and other beginners. (Version 0.6.1)

DJ Navarro

2018-12-29

Contents

Overview	7
Licensing	9
Dedication	11
Preface	13
0.1 Preface to Version 0.6.1	13
0.2 Preface to Version 0.6	13
0.3 Preface to Version 0.5	14
0.4 Preface to Version 0.4	14
0.5 Preface to Version 0.3	14
Part I. Background	17
1 Why do we learn statistics?	19
1.1 On the psychology of statistics	19
1.2 The cautionary tale of Simpson’s paradox	21
1.3 Statistics in psychology	24
1.4 Statistics in everyday life	25
1.5 There’s more to research methods than statistics	26
2 A brief introduction to research design	27
2.1 Introduction to psychological measurement	27
2.2 Scales of measurement	30
2.3 Assessing the reliability of a measurement	34
2.4 The “role” of variables: predictors and outcomes	35
2.5 Experimental and non-experimental research	35
2.6 Assessing the validity of a study	37
2.7 Confounds, artifacts and other threats to validity	40
2.8 Summary	47

Part II. An introduction to R	49
3 Getting started with R	51
3.1 Installing R	52
3.2 Typing commands at the R console	56
3.3 Doing simple calculations with R	61
3.4 Storing a number as a variable	63
3.5 Using functions to do calculations	66
3.6 Letting RStudio help you with your commands	70
3.7 Storing many numbers as a vector	74
3.8 Storing text data	76
3.9 Storing “true or false” data	78
3.10 Indexing vectors	84
3.11 Quitting R	87
3.12 Summary	88
4 Additional R concepts	91
4.1 Using comments	91
4.2 Installing and loading packages	92
4.3 Managing the workspace	100
4.4 Navigating the file system	103
4.5 Loading and saving data	108
4.6 Useful things to know about variables	115
4.7 Factors	119
4.8 Data frames	122
4.9 Lists	125
4.10 Formulas	126
4.11 Generic functions	127
4.12 Getting help	128
4.13 Summary	129
Part III. Working with data	131
5 Descriptive statistics	133
5.1 Measures of central tendency	134
5.2 Measures of variability	144
5.3 Skew and kurtosis{#skew}	153
5.4 Getting an overall summary of a variable	157

5.5	Descriptive statistics separately for each group	160
5.6	Standard scores	163
5.7	Correlations	164
5.8	Handling missing values	178
5.9	Summary	181
5.10	Epilogue: Good descriptive statistics are descriptive!	182
6	Drawing graphs	185
6.1	An overview of R graphics	187
6.2	An introduction to plotting	188
6.3	Histograms	200
6.4	Stem and leaf plots	206
6.5	Boxplots	208
6.6	Scatterplots	218
6.7	Bar graphs	225
6.8	Saving image files using R and Rstudio	229
6.9	Summary	232
7	Pragmatic matters	233
7.1	Tabulating and cross-tabulating data	234
7.2	Transforming and recoding a variable	238
7.3	A few more mathematical functions and operations	243
7.4	Extracting a subset of a vector	247
7.5	Extracting a subset of a data frame	252
7.6	Sorting, flipping and merging data	260
7.7	Reshaping a data frame	267
7.8	Working with text	275
7.9	Reading unusual data files	285
7.10	Coercing data from one class to another	289
7.11	Other useful data structures	290
7.12	Miscellaneous topics	295
7.13	Summary	300
8	Basic programming	301
8.1	Scripts	301
8.2	Loops	308
8.3	Conditional statements	318
8.4	Writing functions	320

8.5 Implicit loops	324
8.6 Summary	326
9 Introduction to probability	327
9.1 How are probability and statistics different?	328
9.2 What does probability mean?	329
9.3 Basic probability theory	333
9.4 The binomial distribution	335
9.5 The normal distribution	342
9.6 Other useful distributions	350
9.7 Summary	361
10 Estimating unknown quantities from a sample	363
10.1 Samples, populations and sampling	363
10.2 The law of large numbers	374
10.3 Sampling distributions and the central limit theorem	375
10.4 Estimating population parameters	387
10.5 Estimating a confidence interval	393
10.6 Summary	400
11 Hypothesis testing	401
11.1 A menagerie of hypotheses	401
11.2 Two types of errors	404
11.3 Test statistics and sampling distributions	405
11.4 Making decisions	407
11.5 The p value of a test	409
11.6 Reporting the results of a hypothesis test	412
11.7 Running the hypothesis test in practice	414
11.8 Effect size, sample size and power	414
11.9 Some issues to consider	421
11.10 Summary	423

Overview

Learning Statistics with R covers the contents of an introductory statistics class, as typically taught to undergraduate psychology students, focusing on the use of the R statistical software. The book discusses how to get started in R as well as giving an introduction to data manipulation and writing scripts. From a statistical perspective, the book discusses descriptive statistics and graphing first, followed by chapters on probability theory, sampling and estimation, and null hypothesis testing. After introducing the theory, the book covers the analysis of contingency tables, t-tests, ANOVAs and regression. Bayesian statistics are covered at the end of the book.

Licensing

This book is published under a Creative Commons BY-SA license (CC BY-SA) version 4.0. This means that this book can be reused, remixed, retained, revised and redistributed (including commercially) as long as appropriate credit is given to the authors. If you remix, or modify the original version of this open textbook, you must redistribute all versions of this open textbook under the same license - CC BY-SA.
<https://creativecommons.org/licenses/by-sa/4.0/>

Dedication

This book was brought to you today by the letter ‘R’.

Preface

0.1 Preface to Version 0.6.1

Hi! I'm not Danielle.

This version is a work in progress to transport the book from LaTeX to bookdown, this allows for the code chunks shown in the text to be interactively rendered and is the first step in updating sections to make use of the **tidyverse**.

Moving from LaTeX is a bit fiddly so there are broken pieces in this version that I am updating progressively. If you notice errors or have suggestions feel free to raise an issue (or pull request) at <http://github.com/ekothe/rbook>

Cheers Emily Kothe

0.2 Preface to Version 0.6

The book hasn't changed much since 2015 when I released Version 0.5 – it's probably fair to say that I've changed more than it has. I moved from Adelaide to Sydney in 2016 and my teaching profile at UNSW is different to what it was at Adelaide, and I haven't really had a chance to work on it since arriving here! It's a little strange looking back at this actually. A few quick comments...

- Weirdly, the book *consistently* misgenders me, but I suppose I have only myself to blame for that one :-) There's now a brief footnote on page 12 that mentions this issue; in real life I've been working through a gender affirmation process for the last two years and mostly go by she/her pronouns. I am, however, just as lazy as I ever was so I haven't bothered updating the text in the book.
- For Version 0.6 I haven't changed much I've made a few minor changes when people have pointed out typos or other errors. In particular it's worth noting the issue associated with the etaSquared function in the **lsr** package (which isn't really being maintained any more) in Section 14.4. The function works fine for the simple examples in the book, but there are definitely bugs in there that I haven't found time to check! So please take care with that one.
- The biggest change really is the licensing! I've released it under a Creative Commons licence (CC BY-SA 4.0, specifically), and placed all the source files to the associated GitHub repository, if anyone wants to adapt it.

Maybe someone would like to write a version that makes use of the **tidyverse**... I hear that's become rather important to R these days :-)

Best, Danielle Navarro

0.3 Preface to Version 0.5

Another year, another update. This time around, the update has focused almost entirely on the theory sections of the book. Chapters 9, 10 and 11 have been rewritten, hopefully for the better. Along the same lines, Chapter 17 is entirely new, and focuses on Bayesian statistics. I think the changes have improved the book a great deal. I've always felt uncomfortable about the fact that all the inferential statistics in the book are presented from an orthodox perspective, even though I almost always present Bayesian data analyses in my own work. Now that I've managed to squeeze Bayesian methods into the book somewhere, I'm starting to feel better about the book as a whole. I wanted to get a few other things done in this update, but as usual I'm running into teaching deadlines, so the update has to go out the way it is!

Dan Navarro February 16, 2015

0.4 Preface to Version 0.4

A year has gone by since I wrote the last preface. The book has changed in a few important ways: Chapters 3 and 4 do a better job of documenting some of the time saving features of Rstudio, Chapters 12 and 13 now make use of new functions in the lsr package for running chi-square tests and t tests, and the discussion of correlations has been adapted to refer to the new functions in the lsr package. The soft copy of 0.4 now has better internal referencing (i.e., actual hyperlinks between sections), though that was introduced in 0.3.1. There's a few tweaks here and there, and many typo corrections (thank you to everyone who pointed out typos!), but overall 0.4 isn't massively different from 0.3.

I wish I'd had more time over the last 12 months to add more content. The absence of any discussion of repeated measures ANOVA and mixed models more generally really does annoy me. My excuse for this lack of progress is that my second child was born at the start of 2013, and so I spent most of last year just trying to keep my head above water. As a consequence, unpaid side projects like this book got sidelined in favour of things that actually pay my salary! Things are a little calmer now, so with any luck version 0.5 will be a bigger step forward.

One thing that has surprised me is the number of downloads the book gets. I finally got some basic tracking information from the website a couple of months ago, and (after excluding obvious robots) the book has been averaging about 90 downloads per day. That's encouraging: there's at least a few people who find the book useful!

Dan Navarro February 4, 2014

0.5 Preface to Version 0.3

There's a part of me that really **doesn't** want to publish this book. It's not finished.

And when I say that, I mean it. The referencing is spotty at best, the chapter summaries are just lists of section titles, there's no index, there are no exercises for the reader, the organisation is suboptimal, and the coverage of topics is just not comprehensive enough for my liking. Additionally, there are sections with content that I'm not happy with, figures that really need to be redrawn, and I've had almost no time to hunt down inconsistencies, typos, or errors. In other words, *this book is not finished*. If I didn't have a looming teaching deadline and a baby due in a few weeks, I really wouldn't be making this available at all.

What this means is that if you are an academic looking for teaching materials, a Ph.D. student looking to learn R, or just a member of the general public interested in statistics, I would advise you to be cautious. What you're looking at is a first draft, and it may not serve your purposes. If we were living in the days when publishing was expensive and the internet wasn't around, I would never consider releasing a book in this form. The thought of someone shelling out \$80 for this (which is what a commercial publisher told me it would retail for when they offered to distribute it) makes me feel more than a little uncomfortable.

However, it's the 21st century, so I can post the pdf on my website for free, and I can distribute hard copies via a print-on-demand service for less than half what a textbook publisher would charge. And so my guilt is assuaged, and I'm willing to share! With that in mind, you can obtain free soft copies and cheap hard copies online, from the following webpages:

- <http://www.compcogscisydney.com/learning-statistics-with-r.html>
- <http://www.lulu.com/content/13570633>

Even so, the warning still stands: what you are looking at is Version 0.3 of a work in progress. If and when it hits Version 1.0, I would be willing to stand behind the work and say, yes, this is a textbook that I would encourage other people to use. At that point, I'll probably start shamelessly flogging the thing on the internet and generally acting like a tool. But until that day comes, I'd like it to be made clear that I'm really ambivalent about the work as it stands.

All of the above being said, there is one group of people that I can enthusiastically endorse this book to: the psychology students taking our undergraduate research methods classes (DRIP and DRIP:A) in 2013. For you, this book is ideal, because it was written to accompany your stats lectures. If a problem arises due to a shortcoming of these notes, I can and will adapt content on the fly to fix that problem. Effectively, you've got a textbook written specifically for your classes, distributed for free (electronic copy) or at near-cost prices (hard copy). Better yet, the notes have been tested: Version 0.1 of these notes was used in the 2011 class, Version 0.2 was used in the 2012 class, and now you're looking at the new and improved Version 0.3. I'm not saying these notes are titanium plated awesomeness on a stick – though if *you* wanted to say so on the student evaluation forms, then you're totally welcome to – because they're not. But I am saying that they've been tried out in previous years and they seem to work okay. Besides, there's a group of us around to troubleshoot if any problems come up, and you can guarantee that at least *one* of your lecturers has read the whole thing cover to cover!

Okay, with all that out of the way, I should say something about what the book aims to be. At its core, it is an introductory statistics textbook pitched primarily at psychology students. As such, it covers the standard topics that you'd expect of such a book: study design, descriptive statistics, the theory of hypothesis testing, *t*-tests, χ^2 tests, ANOVA and regression. However, there are also several chapters devoted to the R statistical package, including a chapter on data manipulation and another one on scripts and programming. Moreover, when you look at the content presented in the book, you'll notice a lot of topics that are traditionally swept under the carpet when teaching statistics to psychology students. The Bayesian/frequentist divide is openly discussed in the probability chapter, and the disagreement between Neyman and Fisher about hypothesis testing makes an appearance. The difference between probability and density is discussed. A detailed treatment of Type I, II and III sums of squares for unbalanced factorial ANOVA is provided. And if you have a look in the Epilogue, it should be clear that my intention is to add a lot more advanced content.

My reasons for pursuing this approach are pretty simple: the students can handle it, and they even seem to enjoy it. Over the last few years I've been pleasantly surprised at just how little difficulty I've had in getting undergraduate psych students to learn R. It's certainly not easy for them, and I've found I need to be a little charitable in setting marking standards, but they do eventually get there. Similarly, they don't seem to have a lot of problems tolerating ambiguity and complexity in presentation of statistical ideas, as long as they are assured that the assessment standards will be set in a fashion that is appropriate for them. So if the students can handle it, why *not* teach it? The potential gains are pretty enticing. If they learn R, the students get access to CRAN, which is perhaps the largest and most comprehensive library of statistical tools in existence. And if they learn about probability theory in detail, it's easier for them to switch from orthodox null hypothesis testing to Bayesian methods if they want to. Better yet, they learn data analysis skills that they can take to an employer without being dependent on expensive and proprietary software.

Sadly, this book isn't the silver bullet that makes all this possible. It's a work in progress, and maybe when it is finished it will be a useful tool. One among many, I would think. There are a number of other books that try to provide a basic introduction to statistics using R, and I'm not arrogant enough to believe that mine is better. Still, I rather like the book, and maybe other people will find it useful, incomplete though it is.

Dan Navarro January 13, 2013

Part I. Background

Chapter 1

Why do we learn statistics?

*“Thou shalt not answer questionnaires Or quizzes upon World Affairs,
Nor with compliance Take any test. Thou shalt not sit
With statisticians nor commit“*

• W.H. Auden¹

1.1 On the psychology of statistics

To the surprise of many students, statistics is a fairly significant part of a psychological education. To the surprise of no-one, statistics is very rarely the *favourite* part of one’s psychological education. After all, if you really loved the idea of doing statistics, you’d probably be enrolled in a statistics class right now, not a psychology class. So, not surprisingly, there’s a pretty large proportion of the student base that isn’t happy about the fact that psychology has so much statistics in it. In view of this, I thought that the right place to start might be to answer some of the more common questions that people have about stats...

A big part of this issue at hand relates to the very idea of statistics. What is it? What’s it there for? And why are scientists so bloody obsessed with it? These are all good questions, when you think about it. So let’s start with the last one. As a group, scientists seem to be bizarrely fixated on running statistical tests on everything. In fact, we use statistics so often that we sometimes forget to explain to people why we do. It’s a kind of article of faith among scientists – and especially social scientists – that your findings can’t be trusted until you’ve done some stats. Undergraduate students might be forgiven for thinking that we’re all completely mad, because no-one takes the time to answer one very simple question:

*Why do you do statistics? Why don’t scientists just use **common sense**?*

It’s a naive question in some ways, but most good questions are. There’s a lot of good answers to it,² but for my money, the best answer is a really simple one: we don’t trust ourselves enough. We worry that we’re human, and susceptible to all of the biases, temptations and frailties that humans suffer from. Much of statistics is basically a safeguard. Using “common sense” to evaluate evidence means trusting gut instincts, relying on verbal arguments and on using the raw power of human reason to come up with the right answer. Most scientists don’t think this approach is likely to work.

¹The quote comes from Auden’s 1946 poem *Under Which Lyre: A Reactionary Tract for the Times*, delivered as part of a commencement address at Harvard University. The history of the poem is kind of interesting: <http://harvardmagazine.com/2007/11/a-poets-warning.html>

²Including the suggestion that common sense is in short supply among scientists.

In fact, come to think of it, this sounds a lot like a psychological question to me, and since I do work in a psychology department, it seems like a good idea to dig a little deeper here. Is it really plausible to think that this “common sense” approach is very trustworthy? Verbal arguments have to be constructed in language, and all languages have biases – some things are harder to say than others, and not necessarily because they’re false (e.g., quantum electrodynamics is a good theory, but hard to explain in words). The instincts of our “gut” aren’t designed to solve scientific problems, they’re designed to handle day to day inferences – and given that biological evolution is slower than cultural change, we should say that they’re designed to solve the day to day problems for a *different world* than the one we live in. Most fundamentally, reasoning sensibly requires people to engage in “induction”, making wise guesses and going beyond the immediate evidence of the senses to make generalisations about the world. If you think that you can do that without being influenced by various distractors, well, I have a bridge in Brooklyn I’d like to sell you. Heck, as the next section shows, we can’t even solve “deductive” problems (ones where no guessing is required) without being influenced by our pre-existing biases.

1.1.1 The curse of belief bias

People are mostly pretty smart. We’re certainly smarter than the other species that we share the planet with (though many people might disagree). Our minds are quite amazing things, and we seem to be capable of the most incredible feats of thought and reason. That doesn’t make us perfect though. And among the many things that psychologists have shown over the years is that we really do find it hard to be neutral, to evaluate evidence impartially and without being swayed by pre-existing biases. A good example of this is the ***belief bias effect*** in logical reasoning: if you ask people to decide whether a particular argument is logically valid (i.e., conclusion would be true if the premises were true), we tend to be influenced by the believability of the conclusion, even when we shouldn’t. For instance, here’s a valid argument where the conclusion is believable:

No cigarettes are inexpensive (Premise 1)
 Some addictive things are inexpensive (Premise 2)
 Therefore, some addictive things are not cigarettes (Conclusion)

And here’s a valid argument where the conclusion is not believable:

No addictive things are inexpensive (Premise 1)
 Some cigarettes are inexpensive (Premise 2)
 Therefore, some cigarettes are not addictive (Conclusion)

The logical *structure* of argument #2 is identical to the structure of argument #1, and they’re both valid. However, in the second argument, there are good reasons to think that premise 1 is incorrect, and as a result it’s probably the case that the conclusion is also incorrect. But that’s entirely irrelevant to the topic at hand: an argument is deductively valid if the conclusion is a logical consequence of the premises. That is, a valid argument doesn’t have to involve true statements.

On the other hand, here’s an invalid argument that has a believable conclusion:

No addictive things are inexpensive (Premise 1)
 Some cigarettes are inexpensive (Premise 2)
 Therefore, some addictive things are not cigarettes (Conclusion)

And finally, an invalid argument with an unbelievable conclusion:

No cigarettes are inexpensive (Premise 1)
 Some addictive things are inexpensive (Premise 2)
 Therefore, some cigarettes are not addictive (Conclusion)

Now, suppose that people really are perfectly able to set aside their pre-existing biases about what is true and what isn't, and purely evaluate an argument on its logical merits. We'd expect 100% of people to say that the valid arguments are valid, and 0% of people to say that the invalid arguments are valid. So if you ran an experiment looking at this, you'd expect to see data like this:

	conclusion feels true	conclusion feels false
argument is valid	100% say “valid”	100% say “valid”
argument is invalid	0% say “valid”	0% say “valid”

If the psychological data looked like this (or even a good approximation to this), we might feel safe in just trusting our gut instincts. That is, it'd be perfectly okay just to let scientists evaluate data based on their common sense, and not bother with all this murky statistics stuff. However, you guys have taken psych classes, and by now you probably know where this is going...

In a classic study, Evans et al. (1983) ran an experiment looking at exactly this. What they found is that when pre-existing biases (i.e., beliefs) were in agreement with the structure of the data, everything went the way you'd hope:

	conclusion feels true	conclusion feels false
argument is valid	92% say “valid”	
argument is invalid		8% say “valid”

Not perfect, but that's pretty good. But look what happens when our intuitive feelings about the truth of the conclusion run against the logical structure of the argument:

	conclusion feels true	conclusion feels false
argument is valid	92% say “valid”	46% say “valid”
argument is invalid	92% say “valid”	8% say “valid”

Oh dear, that's not as good. Apparently, when people are presented with a strong argument that contradicts our pre-existing beliefs, we find it pretty hard to even perceive it to be a strong argument (people only did so 46% of the time). Even worse, when people are presented with a weak argument that agrees with our pre-existing biases, almost no-one can see that the argument is weak (people got that one wrong 92% of the time!)³

If you think about it, it's not as if these data are horribly damning. Overall, people did do better than chance at compensating for their prior biases, since about 60% of people's judgements were correct (you'd expect 50% by chance). Even so, if you were a professional “evaluator of evidence”, and someone came along and offered you a magic tool that improves your chances of making the right decision from 60% to (say) 95%, you'd probably jump at it, right? Of course you would. Thankfully, we actually do have a tool that can do this. But it's not magic, it's statistics. So that's reason #1 why scientists love statistics. It's just *too easy* for us to “believe what we want to believe”; so if we want to “believe in the data” instead, we're going to need a bit of help to keep our personal biases under control. That's what statistics does: it helps keep us honest.

³In my more cynical moments I feel like this fact alone explains 95% of what I read on the internet.

Table 1.5: Admission figures for the six largest departments by gender

Department	Male Applicants	Male Percent Admitted	Female Applicants	Female Percent admitted
A	825	62%	108	82%
B	560	63%	25	68%
C	325	37%	593	34%
D	417	33%	375	35%
E	191	28%	393	24%
F	272	6%	341	7%

1.2 The cautionary tale of Simpson's paradox

The following is a true story (I think...). In 1973, the University of California, Berkeley had some worries about the admissions of students into their postgraduate courses. Specifically, the thing that caused the problem was that the gender breakdown of their admissions, which looked like this...

	Number of applicants	Percent admitted
Males	8442	46%
Females	4321	35%

...and they were worried about being sued.⁴ Given that there were nearly 13,000 applicants, a difference of 9% in admission rates between males and females is just way too big to be a coincidence. Pretty compelling data, right? And if I were to say to you that these data *actually* reflect a weak bias in favour of women (sort of!), you'd probably think that I was either crazy or sexist.

Oddly, it's actually sort of true ...when people started looking more carefully at the admissions data (Bickel et al., 1975) they told a rather different story. Specifically, when they looked at it on a department by department basis, it turned out that most of the departments actually had a slightly *higher* success rate for female applicants than for male applicants. Table 1.5 shows the admission figures for the six largest departments (with the names of the departments removed for privacy reasons):

Remarkably, most departments had a *higher* rate of admissions for females than for males! Yet the overall rate of admission across the university for females was *lower* than for males. How can this be? How can both of these statements be true at the same time?

Here's what's going on. Firstly, notice that the departments are *not* equal to one another in terms of their admission percentages: some departments (e.g., engineering, chemistry) tended to admit a high percentage of the qualified applicants, whereas others (e.g., English) tended to reject most of the candidates, even if they were high quality. So, among the six departments shown above, notice that department A is the most generous, followed by B, C, D, E and F in that order. Next, notice that males and females tended to apply to different departments. If we rank the departments in terms of the total number of male applicants, we get **A>B>D>C>F>E** (the “easy” departments are in bold). On the whole, males tended to apply to the departments that had high admission rates. Now compare this to how the female applicants distributed themselves. Ranking the departments in terms of the total number of female applicants produces a quite different ordering **C>E>D>F>**A>B****. In other words, what these data seem to be suggesting is that the female applicants tended to apply to “harder” departments. And in fact, if we look at all Figure 1.1 we see that this trend is systematic, and quite striking. This effect is known as Simpson's paradox. It's not common, but it does happen in real life, and most people are very surprised by it when they first encounter it, and many people refuse to even believe that it's real. It is very real. And while there are lots of very

⁴Earlier versions of these notes incorrectly suggested that they actually were sued – apparently that's not true. There's a nice commentary on this here: <https://www.refsmmat.com/posts/2016-05-08-simpsons-paradox-berkeley.html>. A big thank you to Wilfried Van Hirtum for pointing this out to me!

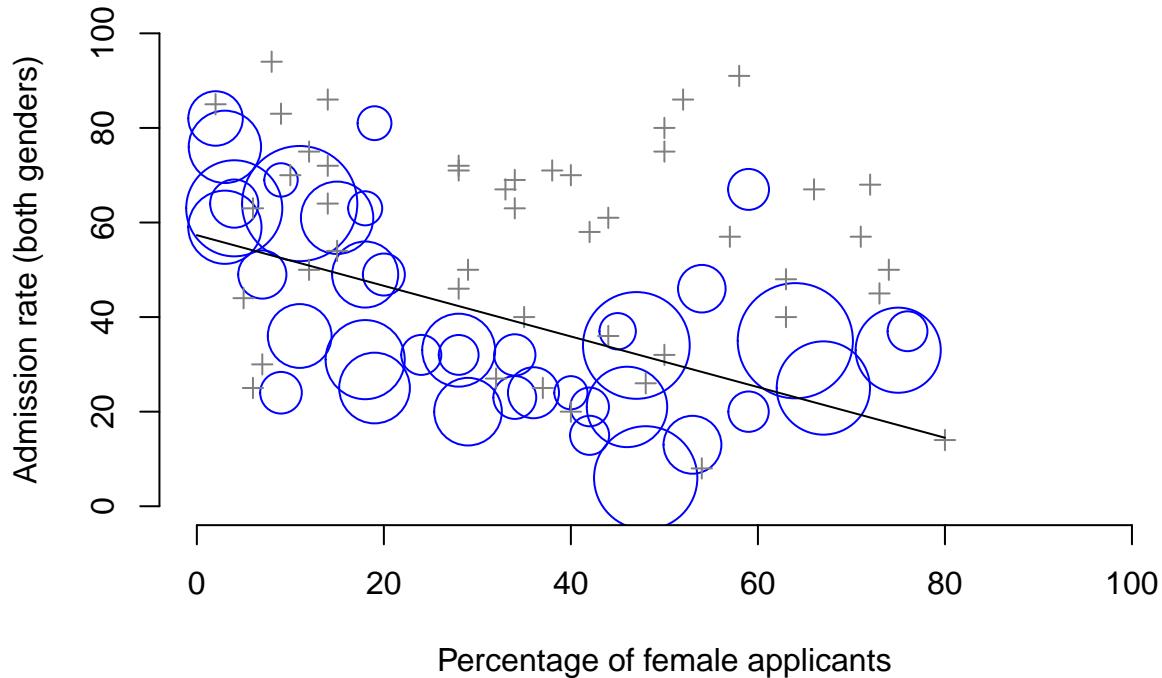


Figure 1.1: The Berkeley 1973 college admissions data. This figure plots the admission rate for the 85 departments that had at least one female applicant, as a function of the percentage of applicants that were female. The plot is a redrawing of Figure 1 from Bickel et al. (1975). Circles plot departments with more than 40 applicants; the area of the circle is proportional to the total number of applicants. The crosses plot department with fewer than 40 applicants.

subtle statistical lessons buried in there, I want to use it to make a much more important point ...doing research is hard, and there are *lots* of subtle, counterintuitive traps lying in wait for the unwary. That's reason #2 why scientists love statistics, and why we teach research methods. Because science is hard, and the truth is sometimes cunningly hidden in the nooks and crannies of complicated data.

Before leaving this topic entirely, I want to point out something else really critical that is often overlooked in a research methods class. Statistics only solves *part* of the problem. Remember that we started all this with the concern that Berkeley's admissions processes might be unfairly biased against female applicants. When we looked at the "aggregated" data, it did seem like the university was discriminating against women, but when we "disaggregate" and looked at the individual behaviour of all the departments, it turned out that the actual departments were, if anything, slightly biased in favour of women. The gender bias in total admissions was caused by the fact that women tended to self-select for harder departments. From a legal perspective, that would probably put the university in the clear. Postgraduate admissions are determined at the level of the individual department (and there are good reasons to do that), and at the level of individual departments, the decisions are more or less unbiased (the weak bias in favour of females at that level is small, and not consistent across departments). Since the university can't dictate which departments people choose to apply to, and the decision making takes place at the level of the department it can hardly be held accountable for any biases that those choices produce.

That was the basis for my somewhat glib remarks earlier, but that's not exactly the whole story, is it? After

all, if we're interested in this from a more sociological and psychological perspective, we might want to ask *why* there are such strong gender differences in applications. Why do males tend to apply to engineering more often than females, and why is this reversed for the English department? And why is it the case that the departments that tend to have a female-application bias tend to have lower overall admission rates than those departments that have a male-application bias? Might this not still reflect a gender bias, even though every single department is itself unbiased? It might. Suppose, hypothetically, that males preferred to apply to "hard sciences" and females prefer "humanities". And suppose further that the reason for why the humanities departments have low admission rates is because the government doesn't want to fund the humanities (Ph.D. places, for instance, are often tied to government funded research projects). Does that constitute a gender bias? Or just an unenlightened view of the value of the humanities? What if someone at a high level in the government cut the humanities funds because they felt that the humanities are "useless chick stuff". That seems pretty *blatantly* gender biased. None of this falls within the purview of statistics, but it matters to the research project. If you're interested in the overall structural effects of subtle gender biases, then you probably want to look at *both* the aggregated and disaggregated data. If you're interested in the decision making process at Berkeley itself then you're probably only interested in the disaggregated data.

In short there are a lot of critical questions that you can't answer with statistics, but the answers to those questions will have a huge impact on how you analyse and interpret data. And this is the reason why you should always think of statistics as a *tool* to help you learn about your data, no more and no less. It's a powerful tool to that end, but there's no substitute for careful thought.

1.3 Statistics in psychology

I hope that the discussion above helped explain why science in general is so focused on statistics. But I'm guessing that you have a lot more questions about what role statistics plays in psychology, and specifically why psychology classes always devote so many lectures to stats. So here's my attempt to answer a few of them...

Why does psychology have so much statistics?

To be perfectly honest, there's a few different reasons, some of which are better than others. The most important reason is that psychology is a statistical science. What I mean by that is that the "things" that we study are *people*. Real, complicated, gloriously messy, infuriatingly perverse people. The "things" of physics include objects like electrons, and while there are all sorts of complexities that arise in physics, electrons don't have minds of their own. They don't have opinions, they don't differ from each other in weird and arbitrary ways, they don't get bored in the middle of an experiment, and they don't get angry at the experimenter and then deliberately try to sabotage the data set (not that I've ever done that...). At a fundamental level psychology is harder than physics.⁵

Basically, we teach statistics to you as psychologists because you need to be better at stats than physicists. There's actually a saying used sometimes in physics, to the effect that "if your experiment needs statistics, you should have done a better experiment". They have the luxury of being able to say that because their objects of study are pathetically simple in comparison to the vast mess that confronts social scientists. It's not just psychology, really: most social sciences are desperately reliant on statistics. Not because we're bad experimenters, but because we've picked a harder problem to solve. We teach you stats because you really, really need it.

Can't someone else do the statistics?

To some extent, but not completely. It's true that you don't need to become a fully trained statistician just to do psychology, but you do need to reach a certain level of statistical competence. In my view, there's three reasons that every psychological researcher ought to be able to do basic statistics:

⁵Which might explain why physics is just a teensy bit further advanced as a science than we are.

- Firstly, there's the fundamental reason: statistics is deeply intertwined with research design. If you want to be good at designing psychological studies, you need to at least understand the basics of stats.
- Secondly, if you want to be good at the psychological side of the research, then you need to be able to understand the psychological literature, right? But almost every paper in the psychological literature reports the results of statistical analyses. So if you really want to understand the psychology, you need to be able to understand what other people did with their data. And that means understanding a certain amount of statistics.
- Thirdly, there's a big practical problem with being dependent on other people to do all your statistics: statistical analysis is *expensive*. If you ever get bored and want to look up how much the Australian government charges for university fees, you'll notice something interesting: statistics is designated as a "national priority" category, and so the fees are much, much lower than for any other area of study. This is because there's a massive shortage of statisticians out there. So, from your perspective as a psychological researcher, the laws of supply and demand aren't exactly on your side here! As a result, in almost any real life situation where you want to do psychological research, the cruel facts will be that you don't have enough money to afford a statistician. So the economics of the situation mean that you have to be pretty self-sufficient.

Note that a lot of these reasons generalise beyond researchers. If you want to be a practicing psychologist and stay on top of the field, it helps to be able to read the scientific literature, which relies pretty heavily on statistics.

I don't care about jobs, research, or clinical work. Do I need statistics?

Okay, now you're just messing with me. Still, I think it should matter to you too. Statistics should matter to you in the same way that statistics should matter to *everyone*: we live in the 21st century, and data are *everywhere*. Frankly, given the world in which we live these days, a basic knowledge of statistics is pretty damn close to a survival tool! Which is the topic of the next section...

1.4 Statistics in everyday life

"We are drowning in information, but we are starved for knowledge"

-Various authors, original probably John Naisbitt

When I started writing up my lecture notes I took the 20 most recent news articles posted to the ABC news website. Of those 20 articles, it turned out that 8 of them involved a discussion of something that I would call a statistical topic; 6 of those made a mistake. The most common error, if you're curious, was failing to report baseline data (e.g., the article mentions that 5% of people in situation X have some characteristic Y, but doesn't say how common the characteristic is for everyone else!) The point I'm trying to make here isn't that journalists are bad at statistics (though they almost always are), it's that a basic knowledge of statistics is very helpful for trying to figure out when someone else is either making a mistake or even lying to you. In fact, one of the biggest things that a knowledge of statistics does to you is cause you to get angry at the newspaper or the internet on a far more frequent basis: you can find a good example of this in Section 5.1.5. In later versions of this book I'll try to include more anecdotes along those lines.

1.5 There's more to research methods than statistics

So far, most of what I've talked about is statistics, and so you'd be forgiven for thinking that statistics is all I care about in life. To be fair, you wouldn't be far wrong, but research methodology is a broader concept than statistics. So most research methods courses will cover a lot of topics that relate much more to the pragmatics of research design, and in particular the issues that you encounter when trying to do research with humans. However, about 99% of student *fears* relate to the statistics part of the course, so I've focused on

the stats in this discussion, and hopefully I've convinced you that statistics matters, and more importantly, that it's not to be feared. That being said, it's pretty typical for introductory research methods classes to be very stats-heavy. This is not (usually) because the lecturers are evil people. Quite the contrary, in fact. Introductory classes focus a lot on the statistics because you almost always find yourself needing statistics before you need the other research methods training. Why? Because almost all of your assignments in other classes will rely on statistical training, to a much greater extent than they rely on other methodological tools. It's not common for undergraduate assignments to require you to design your own study from the ground up (in which case you would need to know a lot about research design), but it *is* common for assignments to ask you to analyse and interpret data that were collected in a study that someone else designed (in which case you need statistics). In that sense, from the perspective of allowing you to do well in all your other classes, the statistics is more urgent.

But note that “urgent” is different from “important” – they both matter. I really do want to stress that research design is just as important as data analysis, and this book does spend a fair amount of time on it. However, while statistics has a kind of universality, and provides a set of core tools that are useful for most types of psychological research, the research methods side isn't quite so universal. There are some general principles that everyone should think about, but a lot of research design is very idiosyncratic, and is specific to the area of research that you want to engage in. To the extent that it's the details that matter, those details don't usually show up in an introductory stats and research methods class.

Chapter 2

A brief introduction to research design

To consult the statistician after an experiment is finished is often merely to ask him to conduct a post mortem examination. He can perhaps say what the experiment died of.

– Sir Ronald Fisher¹

In this chapter, we're going to start thinking about the basic ideas that go into designing a study, collecting data, checking whether your data collection works, and so on. It won't give you enough information to allow you to design studies of your own, but it will give you a lot of the basic tools that you need to assess the studies done by other people. However, since the focus of this book is much more on data analysis than on data collection, I'm only giving a very brief overview. Note that this chapter is “special” in two ways. Firstly, it's much more psychology-specific than the later chapters. Secondly, it focuses much more heavily on the scientific problem of research methodology, and much less on the statistical problem of data analysis. Nevertheless, the two problems are related to one another, so it's traditional for stats textbooks to discuss the problem in a little detail. This chapter relies heavily on Campbell and Stanley (1963) for the discussion of study design, and Stevens (1946) for the discussion of scales of measurement. Later versions will attempt to be more precise in the citations.

2.1 Introduction to psychological measurement

The first thing to understand is data collection can be thought of as a kind of **measurement**. That is, what we're trying to do here is measure something about human behaviour or the human mind. What do I mean by “measurement”?

2.1.1 Some thoughts about psychological measurement

Measurement itself is a subtle concept, but basically it comes down to finding some way of assigning numbers, or labels, or some other kind of well-defined descriptions to “stuff”. So, any of the following would count as a psychological measurement:

- My **age** is 33 years.
- I *do not like anchovies*.

¹ Presidential Address to the First Indian Statistical Congress, 1938. Source: http://en.wikiquote.org/wiki/Ronald_Fisher

- My **chromosomal gender** is *male*.
- My **self-identified gender** is *male*.²

In the short list above, the **bolded part** is “the thing to be measured”, and the *italicised part* is “the measurement itself”. In fact, we can expand on this a little bit, by thinking about the set of possible measurements that could have arisen in each case:

- My **age** (in years) could have been *0, 1, 2, 3 ...*, etc. The upper bound on what my age could possibly be is a bit fuzzy, but in practice you’d be safe in saying that the largest possible age is *150*, since no human has ever lived that long.
- When asked if I **like anchovies**, I might have said that *I do*, or *I do not*, or *I have no opinion*, or *I sometimes do*.
- My **chromosomal gender** is almost certainly going to be *male (XY)* or *female (XX)*, but there are a few other possibilities. I could also have *Klinefelter's syndrome (XXY)*, which is more similar to male than to female. And I imagine there are other possibilities too.
- My **self-identified gender** is also very likely to be *male* or *female*, but it doesn’t have to agree with my chromosomal gender. I may also choose to identify with *neither*, or to explicitly call myself *transgender*.

As you can see, for some things (like age) it seems fairly obvious what the set of possible measurements should be, whereas for other things it gets a bit tricky. But I want to point out that even in the case of someone’s age, it’s much more subtle than this. For instance, in the example above, I assumed that it was okay to measure age in years. But if you’re a developmental psychologist, that’s way too crude, and so you often measure age in *years and months* (if a child is 2 years and 11 months, this is usually written as “2;11”). If you’re interested in newborns, you might want to measure age in *days since birth*, maybe even *hours since birth*. In other words, the way in which you specify the allowable measurement values is important.

Looking at this a bit more closely, you might also realise that the concept of “age” isn’t actually all that precise. In general, when we say “age” we implicitly mean “the length of time since birth”. But that’s not always the right way to do it. Suppose you’re interested in how newborn babies control their eye movements. If you’re interested in kids that young, you might also start to worry that “birth” is not the only meaningful point in time to care about. If Baby Alice is born 3 weeks premature and Baby Bianca is born 1 week late, would it really make sense to say that they are the “same age” if we encountered them “2 hours after birth”? In one sense, yes: by social convention, we use birth as our reference point for talking about age in everyday life, since it defines the amount of time the person has been operating as an independent entity in the world, but from a scientific perspective that’s not the only thing we care about. When we think about the biology of human beings, it’s often useful to think of ourselves as organisms that have been growing and maturing since conception, and from that perspective Alice and Bianca aren’t the same age at all. So you might want to define the concept of “age” in two different ways: the length of time since conception, and the length of time since birth. When dealing with adults, it won’t make much difference, but when dealing with newborns it might.

Moving beyond these issues, there’s the question of methodology. What specific “measurement method” are you going to use to find out someone’s age? As before, there are lots of different possibilities:

²Well... now this is awkward, isn’t it? This section is one of the oldest parts of the book, and it’s outdated in a rather embarrassing way. I wrote this in 2010, at which point all of those facts *were* true. Revisiting this in 2018... well I’m not 33 any more, but that’s not surprising I suppose. I can’t imagine my chromosomes have changed, so I’m going to guess my karyotype was then and is now XY. The self-identified gender, on the other hand... ah. I suppose the fact that the title page now refers to me as Danielle rather than Daniel might possibly be a giveaway, but I don’t typically identify as “male” on a gender questionnaire these days, and I prefer “she/her” pronouns as a default (it’s a long story)! I did think a little about how I was going to handle this in the book, actually. The book has a somewhat distinct authorial voice to it, and I feel like it would be a rather different work if I went back and wrote everything as Danielle and updated all the pronouns in the work. Besides, it would be a lot of work, so I’ve left my name as “Dan” throughout the book, and in any case “Dan” is a perfectly good nickname for “Danielle”, don’t you think? In any case, it’s not a big deal. I only wanted to mention it to make life a little easier for readers who aren’t sure how to refer to me. I still don’t like anchovies though :-)

- You could just ask people “how old are you?” The method of self-report is fast, cheap and easy, but it only works with people old enough to understand the question, and some people lie about their age.
- You could ask an authority (e.g., a parent) “how old is your child?” This method is fast, and when dealing with kids it’s not all that hard since the parent is almost always around. It doesn’t work as well if you want to know “age since conception”, since a lot of parents can’t say for sure when conception took place. For that, you might need a different authority (e.g., an obstetrician).
- You could look up official records, like birth certificates. This is time consuming and annoying, but it has its uses (e.g., if the person is now dead).

2.1.2 Operationalisation: defining your measurement

All of the ideas discussed in the previous section all relate to the concept of ***operationalisation***. To be a bit more precise about the idea, operationalisation is the process by which we take a meaningful but somewhat vague concept, and turn it into a precise measurement. The process of operationalisation can involve several different things:

- Being precise about what you are trying to measure. For instance, does “age” mean “time since birth” or “time since conception” in the context of your research?
- Determining what method you will use to measure it. Will you use self-report to measure age, ask a parent, or look up an official record? If you’re using self-report, how will you phrase the question?
- Defining the set of the allowable values that the measurement can take. Note that these values don’t always have to be numerical, though they often are. When measuring age, the values are numerical, but we still need to think carefully about what numbers are allowed. Do we want age in years, years and months, days, hours? Etc. For other types of measurements (e.g., gender), the values aren’t numerical. But, just as before, we need to think about what values are allowed. If we’re asking people to self-report their gender, what options do we allow them to choose between? Is it enough to allow only “male” or “female”? Do you need an “other” option? Or should we not give people any specific options, and let them answer in their own words? And if you open up the set of possible values to include all verbal response, how will you interpret their answers?

Operationalisation is a tricky business, and there’s no “one, true way” to do it. The way in which you choose to operationalise the informal concept of “age” or “gender” into a formal measurement depends on what you need to use the measurement for. Often you’ll find that the community of scientists who work in your area have some fairly well-established ideas for how to go about it. In other words, operationalisation needs to be thought through on a case by case basis. Nevertheless, while there are a lot of issues that are specific to each individual research project, there are some aspects to it that are pretty general.

Before moving on, I want to take a moment to clear up our terminology, and in the process introduce one more term. Here are four different things that are closely related to each other:

- **A theoretical construct.** This is the thing that you’re trying to take a measurement of, like “age”, “gender” or an “opinion”. A theoretical construct can’t be directly observed, and often they’re actually a bit vague.
- **A measure.** The measure refers to the method or the tool that you use to make your observations. A question in a survey, a behavioural observation or a brain scan could all count as a measure.
- **An operationalisation.** The term “operationalisation” refers to the logical connection between the measure and the theoretical construct, or to the process by which we try to derive a measure from a theoretical construct.
- **A variable.** Finally, a new term. A variable is what we end up with when we apply our measure to something in the world. That is, variables are the actual “data” that we end up with in our data sets.

In practice, even scientists tend to blur the distinction between these things, but it’s very helpful to try to understand the differences.

2.2 Scales of measurement

As the previous section indicates, the outcome of a psychological measurement is called a variable. But not all variables are of the same qualitative type, and it's very useful to understand what types there are. A very useful concept for distinguishing between different types of variables is what's known as *scales of measurement*.

2.2.1 Nominal scale

A *nominal scale* variable (also referred to as a *categorical* variable) is one in which there is no particular relationship between the different possibilities: for these kinds of variables it doesn't make any sense to say that one of them is "bigger" or "better" than any other one, and it absolutely doesn't make any sense to average them. The classic example for this is "eye colour". Eyes can be blue, green and brown, among other possibilities, but none of them is any "better" than any other one. As a result, it would feel really weird to talk about an "average eye colour". Similarly, gender is nominal too: male isn't better or worse than female, neither does it make sense to try to talk about an "average gender". In short, nominal scale variables are those for which the only thing you can say about the different possibilities is that they are different. That's it.

Let's take a slightly closer look at this. Suppose I was doing research on how people commute to and from work. One variable I would have to measure would be what kind of transportation people use to get to work. This "transport type" variable could have quite a few possible values, including: "train", "bus", "car", "bicycle", etc. For now, let's suppose that these four are the only possibilities, and suppose that when I ask 100 people how they got to work today, and I get this:

Transportation	Number of people
(1) Train	12
(2) Bus	30
(3) Car	48
(4) Bicycle	10

So, what's the average transportation type? Obviously, the answer here is that there isn't one. It's a silly question to ask. You can say that travel by car is the most popular method, and travel by train is the least popular method, but that's about all. Similarly, notice that the order in which I list the options isn't very interesting. I could have chosen to display the data like this

Transportation	Number of people
(3) Car	48
(1) Train	12
(4) Bicycle	10
(2) Bus	30

and nothing really changes.

2.2.2 Ordinal scale

Ordinal scale variables have a bit more structure than nominal scale variables, but not by a lot. An ordinal scale variable is one in which there is a natural, meaningful way to order the different possibilities, but you can't do anything else. The usual example given of an ordinal variable is "finishing position in a race". You

can say that the person who finished first was faster than the person who finished second, but you *don't* know how much faster. As a consequence we know that 1st > 2nd, and we know that 2nd > 3rd, but the difference between 1st and 2nd might be much larger than the difference between 2nd and 3rd.

Here's an more psychologically interesting example. Suppose I'm interested in people's attitudes to climate change, and I ask them to pick one of these four statements that most closely matches their beliefs:

- (1) Temperatures are rising, because of human activity
- (2) Temperatures are rising, but we don't know why
- (3) Temperatures are rising, but not because of humans
- (4) Temperatures are not rising

Notice that these four statements actually do have a natural ordering, in terms of "the extent to which they agree with the current science". Statement 1 is a close match, statement 2 is a reasonable match, statement 3 isn't a very good match, and statement 4 is in strong opposition to the science. So, in terms of the thing I'm interested in (the extent to which people endorse the science), I can order the items as 1 > 2 > 3 > 4. Since this ordering exists, it would be very weird to list the options like this...

- (3) Temperatures are rising, but not because of humans
- (4) Temperatures are rising, because of human activity
- (5) Temperatures are not rising
- (6) Temperatures are rising, but we don't know why

... because it seems to violate the natural "structure" to the question.

So, let's suppose I asked 100 people these questions, and got the following answers:

Response	Number
(1) Temperatures are rising, because of human activity	51
(2) Temperatures are rising, but we don't know why	20
(3) Temperatures are rising, but not because of humans	10
(4) Temperatures are not rising	19

When analysing these data, it seems quite reasonable to try to group (1), (2) and (3) together, and say that 81 of 100 people were willing to *at least partially* endorse the science. And it's *also* quite reasonable to group (2), (3) and (4) together and say that 49 of 100 people registered *at least some disagreement* with the dominant scientific view. However, it would be entirely bizarre to try to group (1), (2) and (4) together and say that 90 of 100 people said... what? There's nothing sensible that allows you to group those responses together at all.

That said, notice that while we *can* use the natural ordering of these items to construct sensible groupings, what we *can't* do is average them. For instance, in my simple example here, the "average" response to the question is 1.97. If you can tell me what that means, I'd love to know. Because that sounds like gibberish to me!

2.2.3 Interval scale

In contrast to nominal and ordinal scale variables, **interval scale** and ratio scale variables are variables for which the numerical value is genuinely meaningful. In the case of interval scale variables, the *differences* between the numbers are interpretable, but the variable doesn't have a "natural" zero value. A good example of an interval scale variable is measuring temperature in degrees celsius. For instance, if it was 15° yesterday and 18° today, then the 3° difference between the two is genuinely meaningful. Moreover, that 3° difference

is *exactly the same* as the 3° difference between 7° and 10° . In short, addition and subtraction are meaningful for interval scale variables.³

However, notice that the 0° does not mean “no temperature at all”: it actually means “the temperature at which water freezes”, which is pretty arbitrary. As a consequence, it becomes pointless to try to multiply and divide temperatures. It is wrong to say that 20° is *twice as hot* as 10° , just as it is weird and meaningless to try to claim that 20° is negative two times as hot as -10° .

Again, lets look at a more psychological example. Suppose I’m interested in looking at how the attitudes of first-year university students have changed over time. Obviously, I’m going to want to record the year in which each student started. This is an interval scale variable. A student who started in 2003 did arrive 5 years before a student who started in 2008. However, it would be completely insane for me to divide 2008 by 2003 and say that the second student started “1.0024 times later” than the first one. That doesn’t make any sense at all.

2.2.4 Ratio scale

The fourth and final type of variable to consider is a **ratio scale** variable, in which zero really means zero, and it’s okay to multiply and divide. A good psychological example of a ratio scale variable is response time (RT). In a lot of tasks it’s very common to record the amount of time somebody takes to solve a problem or answer a question, because it’s an indicator of how difficult the task is. Suppose that Alan takes 2.3 seconds to respond to a question, whereas Ben takes 3.1 seconds. As with an interval scale variable, addition and subtraction are both meaningful here. Ben really did take $3.1 - 2.3 = 0.8$ seconds longer than Alan did. However, notice that multiplication and division also make sense here too: Ben took $3.1 / 2.3 = 1.35$ times as long as Alan did to answer the question. And the reason why you can do this is that, for a ratio scale variable such as RT, “zero seconds” really does mean “no time at all”.

2.2.5 Continuous versus discrete variables

There’s a second kind of distinction that you need to be aware of, regarding what types of variables you can run into. This is the distinction between continuous variables and discrete variables. The difference between these is as follows:

- A **continuous variable** is one in which, for any two values that you can think of, it’s always logically possible to have another value in between.
- A **discrete variable** is, in effect, a variable that isn’t continuous. For a discrete variable, it’s sometimes the case that there’s nothing in the middle.

These definitions probably seem a bit abstract, but they’re pretty simple once you see some examples. For instance, response time is continuous. If Alan takes 3.1 seconds and Ben takes 2.3 seconds to respond to a question, then it’s possible for Cameron’s response time to lie in between, by taking 3.0 seconds. And of course it would also be possible for David to take 3.031 seconds to respond, meaning that his RT would lie in between Cameron’s and Alan’s. And while in practice it might be impossible to measure RT that precisely, it’s certainly possible in principle. Because we can always find a new value for RT in between any two other ones, we say that RT is continuous.

Discrete variables occur when this rule is violated. For example, nominal scale variables are always discrete: there isn’t a type of transportation that falls “in between” trains and bicycles, not in the strict mathematical way that 2.3 falls in between 2 and 3. So transportation type is discrete. Similarly, ordinal scale variables are

³Actually, I’ve been informed by readers with greater physics knowledge than I that temperature isn’t strictly an interval scale, in the sense that the amount of energy required to heat something up by 3° depends on its current temperature. So in the sense that physicists care about, temperature isn’t actually interval scale. But it still makes a cute example, so I’m going to ignore this little inconvenient truth.

Table 2.4: The relationship between the scales of measurement and the discrete/continuity distinction. Cells with a tick mark correspond to things that are possible.

	continuous	discrete
nominal		\checkmark
ordinal		\checkmark
interval	\checkmark	\checkmark
ratio	\checkmark	\checkmark

always discrete: although “2nd place” does fall between “1st place” and “3rd place”, there’s nothing that can logically fall in between “1st place” and “2nd place”. Interval scale and ratio scale variables can go either way. As we saw above, response time (a ratio scale variable) is continuous. Temperature in degrees celsius (an interval scale variable) is also continuous. However, the year you went to school (an interval scale variable) is discrete. There’s no year in between 2002 and 2003. The number of questions you get right on a true-or-false test (a ratio scale variable) is also discrete: since a true-or-false question doesn’t allow you to be “partially correct”, there’s nothing in between 5/10 and 6/10. Table 2.4 summarises the relationship between the scales of measurement and the discrete/continuity distinction. Cells with a tick mark correspond to things that are possible. I’m trying to hammer this point home, because (a) some textbooks get this wrong, and (b) people very often say things like “discrete variable” when they mean “nominal scale variable”. It’s very unfortunate.

2.2.6 Some complexities

Okay, I know you’re going to be shocked to hear this, but ... the real world is much messier than this little classification scheme suggests. Very few variables in real life actually fall into these nice neat categories, so you need to be kind of careful not to treat the scales of measurement as if they were hard and fast rules. It doesn’t work like that: they’re guidelines, intended to help you think about the situations in which you should treat different variables differently. Nothing more.

So let’s take a classic example, maybe *the* classic example, of a psychological measurement tool: the **Likert scale**. The humble Likert scale is the bread and butter tool of all survey design. You yourself have filled out hundreds, maybe thousands of them, and odds are you’ve even used one yourself. Suppose we have a survey question that looks like this:

Which of the following best describes your opinion of the statement that “all pirates are freaking awesome” ...

and then the options presented to the participant are these:

- (1) Strongly disagree
- (2) Disagree
- (3) Neither agree nor disagree
- (4) Agree
- (5) Strongly agree

This set of items is an example of a 5-point Likert scale: people are asked to choose among one of several (in this case 5) clearly ordered possibilities, generally with a verbal descriptor given in each case. However, it’s not necessary that all items be explicitly described. This is a perfectly good example of a 5-point Likert scale too:

- (1) Strongly disagree
- (2)

- (3)
- (4)
- (5) Strongly agree

Likert scales are very handy, if somewhat limited, tools. The question is, what kind of variable are they? They're obviously discrete, since you can't give a response of 2.5. They're obviously not nominal scale, since the items are ordered; and they're not ratio scale either, since there's no natural zero.

But are they ordinal scale or interval scale? One argument says that we can't really prove that the difference between "strongly agree" and "agree" is of the same size as the difference between "agree" and "neither agree nor disagree". In fact, in everyday life it's pretty obvious that they're not the same at all. So this suggests that we ought to treat Likert scales as ordinal variables. On the other hand, in practice most participants do seem to take the whole "on a scale from 1 to 5" part fairly seriously, and they tend to act as if the differences between the five response options were fairly similar to one another. As a consequence, a lot of researchers treat Likert scale data as if it were interval scale. It's not interval scale, but in practice it's close enough that we usually think of it as being *quasi-interval scale*.

2.3 Assessing the reliability of a measurement

At this point we've thought a little bit about how to operationalise a theoretical construct and thereby create a psychological measure; and we've seen that by applying psychological measures we end up with variables, which can come in many different types. At this point, we should start discussing the obvious question: is the measurement any good? We'll do this in terms of two related ideas: *reliability* and *validity*. Put simply, the ***reliability*** of a measure tells you how *precisely* you are measuring something, whereas the validity of a measure tells you how *accurate* the measure is. In this section I'll talk about reliability; we'll talk about validity in the next chapter.

Reliability is actually a very simple concept: it refers to the repeatability or consistency of your measurement. The measurement of my weight by means of a "bathroom scale" is very reliable: if I step on and off the scales over and over again, it'll keep giving me the same answer. Measuring my intelligence by means of "asking my mum" is very unreliable: some days she tells me I'm a bit thick, and other days she tells me I'm a complete moron. Notice that this concept of reliability is different to the question of whether the measurements are correct (the correctness of a measurement relates to its validity). If I'm holding a sack of potatos when I step on and off of the bathroom scales, the measurement will still be reliable: it will always give me the same answer. However, this highly reliable answer doesn't match up to my true weight at all, therefore it's wrong. In technical terms, this is a *reliable but invalid* measurement. Similarly, while my mum's estimate of my intelligence is a bit unreliable, she might be right. Maybe I'm just not too bright, and so while her estimate of my intelligence fluctuates pretty wildly from day to day, it's basically right. So that would be an *unreliable but valid* measure. Of course, to some extent, notice that if my mum's estimates are too unreliable, it's going to be very hard to figure out which one of her many claims about my intelligence is actually the right one. To some extent, then, a very unreliable measure tends to end up being invalid for practical purposes; so much so that many people would say that reliability is necessary (but not sufficient) to ensure validity.

Okay, now that we're clear on the distinction between reliability and validity, let's have a think about the different ways in which we might measure reliability:

- ***Test-retest reliability***. This relates to consistency over time: if we repeat the measurement at a later date, do we get the same answer?
- ***Inter-rater reliability***. This relates to consistency across people: if someone else repeats the measurement (e.g., someone else rates my intelligence) will they produce the same answer?
- ***Parallel forms reliability***. This relates to consistency across theoretically-equivalent measurements: if I use a different set of bathroom scales to measure my weight, does it give the same answer?

Table 2.5: The terminology used to distinguish between different roles that a variable can play when analysing a data set. Note that this book will tend to avoid the classical terminology in favour of the newer names.

role of the variable	classical name	modern name
to be explained	dependent variable (DV)	outcome
to do the explaining	independent variable (IV)	predictor

- **Internal consistency reliability.** If a measurement is constructed from lots of different parts that perform similar functions (e.g., a personality questionnaire result is added up across several questions) do the individual parts tend to give similar answers.

Not all measurements need to possess all forms of reliability. For instance, educational assessment can be thought of as a form of measurement. One of the subjects that I teach, *Computational Cognitive Science*, has an assessment structure that has a research component and an exam component (plus other things). The exam component is *intended* to measure something different from the research component, so the assessment as a whole has low internal consistency. However, within the exam there are several questions that are intended to (approximately) measure the same things, and those tend to produce similar outcomes; so the exam on its own has a fairly high internal consistency. Which is as it should be. You should only demand reliability in those situations where you want to be measure the same thing!

2.4 The “role” of variables: predictors and outcomes

Okay, I've got one last piece of terminology that I need to explain to you before moving away from variables. Normally, when we do some research we end up with lots of different variables. Then, when we analyse our data we usually try to explain some of the variables in terms of some of the other variables. It's important to keep the two roles “thing doing the explaining” and “thing being explained” distinct. So let's be clear about this now. Firstly, we might as well get used to the idea of using mathematical symbols to describe variables, since it's going to happen over and over again. Let's denote the “to be explained” variable Y , and denote the variables “doing the explaining” as X_1, X_2 , etc.

Now, when we doing an analysis, we have different names for X and Y , since they play different roles in the analysis. The classical names for these roles are *independent variable* (IV) and *dependent variable* (DV). The IV is the variable that you use to do the explaining (i.e., X) and the DV is the variable being explained (i.e., Y). The logic behind these names goes like this: if there really is a relationship between X and Y then we can say that Y depends on X , and if we have designed our study “properly” then X isn't dependent on anything else. However, I personally find those names horrible: they're hard to remember and they're highly misleading, because (a) the IV is never actually “independent of everything else” and (b) if there's no relationship, then the DV doesn't actually depend on the IV. And in fact, because I'm not the only person who thinks that IV and DV are just awful names, there are a number of alternatives that I find more appealing. The terms that I'll use in these notes are *predictors* and *outcomes*. The idea here is that what you're trying to do is use X (the predictors) to make guesses about Y (the outcomes).⁴ This is summarised in Table 2.5.

2.5 Experimental and non-experimental research

One of the big distinctions that you should be aware of is the distinction between “experimental research” and “non-experimental research”. When we make this distinction, what we're really talking about is the

⁴Annoyingly, though, there's a lot of different names used out there. I won't list all of them – there would be no point in doing that – other than to note that R often uses “response variable” where I've used “outcome”, and a traditionalist would use “dependent variable”. Sigh. This sort of terminological confusion is very common, I'm afraid.

degree of control that the researcher exercises over the people and events in the study.

2.5.1 Experimental research

The key features of ***experimental research*** is that the researcher controls all aspects of the study, especially what participants experience during the study. In particular, the researcher manipulates or varies the predictor variables (IVs), and then allows the outcome variable (DV) to vary naturally. The idea here is to deliberately vary the predictors (IVs) to see if they have any causal effects on the outcomes. Moreover, in order to ensure that there's no chance that something other than the predictor variables is causing the outcomes, everything else is kept constant or is in some other way "balanced" to ensure that they have no effect on the results. In practice, it's almost impossible to *think* of everything else that might have an influence on the outcome of an experiment, much less keep it constant. The standard solution to this is ***randomisation***: that is, we randomly assign people to different groups, and then give each group a different treatment (i.e., assign them different values of the predictor variables). We'll talk more about randomisation later in this course, but for now, it's enough to say that what randomisation does is minimise (but not eliminate) the chances that there are any systematic difference between groups.

Let's consider a very simple, completely unrealistic and grossly unethical example. Suppose you wanted to find out if smoking causes lung cancer. One way to do this would be to find people who smoke and people who don't smoke, and look to see if smokers have a higher rate of lung cancer. This is *not* a proper experiment, since the researcher doesn't have a lot of control over who is and isn't a smoker. And this really matters: for instance, it might be that people who choose to smoke cigarettes also tend to have poor diets, or maybe they tend to work in asbestos mines, or whatever. The point here is that the groups (smokers and non-smokers) actually differ on lots of things, not *just* smoking. So it might be that the higher incidence of lung cancer among smokers is caused by something else, not by smoking per se. In technical terms, these other things (e.g. diet) are called "confounds", and we'll talk about those in just a moment.

In the meantime, let's now consider what a proper experiment might look like. Recall that our concern was that smokers and non-smokers might differ in lots of ways. The solution, as long as you have no ethics, is to *control* who smokes and who doesn't. Specifically, if we randomly divide participants into two groups, and force half of them to become smokers, then it's very unlikely that the groups will differ in any respect other than the fact that half of them smoke. That way, if our smoking group gets cancer at a higher rate than the non-smoking group, then we can feel pretty confident that (a) smoking does cause cancer and (b) we're murderers.

2.5.2 Non-experimental research

Non-experimental research is a broad term that covers "any study in which the researcher doesn't have quite as much control as they do in an experiment". Obviously, control is something that scientists like to have, but as the previous example illustrates, there are lots of situations in which you can't or shouldn't try to obtain that control. Since it's grossly unethical (and almost certainly criminal) to force people to smoke in order to find out if they get cancer, this is a good example of a situation in which you really shouldn't try to obtain experimental control. But there are other reasons too. Even leaving aside the ethical issues, our "smoking experiment" does have a few other issues. For instance, when I suggested that we "force" half of the people to become smokers, I must have been talking about *starting* with a sample of non-smokers, and then forcing them to become smokers. While this sounds like the kind of solid, evil experimental design that a mad scientist would love, it might not be a very sound way of investigating the effect in the real world. For instance, suppose that smoking only causes lung cancer when people have poor diets, and suppose also that people who normally smoke do tend to have poor diets. However, since the "smokers" in our experiment aren't "natural" smokers (i.e., we forced non-smokers to become smokers; they didn't take on all of the other normal, real life characteristics that smokers might tend to possess) they probably have better diets. As such, in this silly example they wouldn't get lung cancer, and our experiment will fail, because it violates the structure of the "natural" world (the technical name for this is an "artifactual" result; see later).

One distinction worth making between two types of non-experimental research is the difference between ***quasi-experimental research*** and ***case studies***. The example I discussed earlier – in which we wanted to examine incidence of lung cancer among smokers and non-smokers, without trying to control who smokes and who doesn't – is a quasi-experimental design. That is, it's the same as an experiment, but we don't control the predictors (IVs). We can still use statistics to analyse the results, it's just that we have to be a lot more careful.

The alternative approach, case studies, aims to provide a very detailed description of one or a few instances. In general, you can't use statistics to analyse the results of case studies, and it's usually very hard to draw any general conclusions about “people in general” from a few isolated examples. However, case studies are very useful in some situations. Firstly, there are situations where you don't have any alternative: neuropsychology has this issue a lot. Sometimes, you just can't find a lot of people with brain damage in a specific area, so the only thing you can do is describe those cases that you do have in as much detail and with as much care as you can. However, there's also some genuine advantages to case studies: because you don't have as many people to study, you have the ability to invest lots of time and effort trying to understand the specific factors at play in each case. This is a very valuable thing to do. As a consequence, case studies can complement the more statistically-oriented approaches that you see in experimental and quasi-experimental designs. We won't talk much about case studies in these lectures, but they are nevertheless very valuable tools!

2.6 Assessing the validity of a study

More than any other thing, a scientist wants their research to be “valid”. The conceptual idea behind ***validity*** is very simple: can you trust the results of your study? If not, the study is invalid. However, while it's easy to state, in practice it's much harder to check validity than it is to check reliability. And in all honesty, there's no precise, clearly agreed upon notion of what validity actually is. In fact, there's lots of different kinds of validity, each of which raises its own issues, and not all forms of validity are relevant to all studies. I'm going to talk about five different types:

- Internal validity
- External validity
- Construct validity
- Face validity
- Ecological validity

To give you a quick guide as to what matters here... (1) Internal and external validity are the most important, since they tie directly to the fundamental question of whether your study really works. (2) Construct validity asks whether you're measuring what you think you are. (3) Face validity isn't terribly important except insofar as you care about “appearances”. (4) Ecological validity is a special case of face validity that corresponds to a kind of appearance that you might care about a lot.

2.6.1 Internal validity

Internal validity refers to the extent to which you are able draw the correct conclusions about the causal relationships between variables. It's called “internal” because it refers to the relationships between things “inside” the study. Let's illustrate the concept with a simple example. Suppose you're interested in finding out whether a university education makes you write better. To do so, you get a group of first year students, ask them to write a 1000 word essay, and count the number of spelling and grammatical errors they make. Then you find some third-year students, who obviously have had more of a university education than the first-years, and repeat the exercise. And let's suppose it turns out that the third-year students produce fewer errors. And so you conclude that a university education improves writing skills. Right? Except... the big problem that you have with this experiment is that the third-year students are older, and they've had more experience with writing things. So it's hard to know for sure what the causal relationship is: Do older people

write better? Or people who have had more writing experience? Or people who have had more education? Which of the above is the true *cause* of the superior performance of the third-years? Age? Experience? Education? You can't tell. This is an example of a failure of internal validity, because your study doesn't properly tease apart the *causal* relationships between the different variables.

2.6.2 External validity

External validity relates to the **generalisability** of your findings. That is, to what extent do you expect to see the same pattern of results in "real life" as you saw in your study. To put it a bit more precisely, any study that you do in psychology will involve a fairly specific set of questions or tasks, will occur in a specific environment, and will involve participants that are drawn from a particular subgroup. So, if it turns out that the results don't actually generalise to people and situations beyond the ones that you studied, then what you've got is a lack of external validity.

The classic example of this issue is the fact that a very large proportion of studies in psychology will use undergraduate psychology students as the participants. Obviously, however, the researchers don't care *only* about psychology students; they care about people in general. Given that, a study that uses only psych students as participants always carries a risk of lacking external validity. That is, if there's something "special" about psychology students that makes them different to the general populace in some *relevant* respect, then we may start worrying about a lack of external validity.

That said, it is absolutely critical to realise that a study that uses only psychology students does not necessarily have a problem with external validity. I'll talk about this again later, but it's such a common mistake that I'm going to mention it here. The external validity is threatened by the choice of population if (a) the population from which you sample your participants is very narrow (e.g., psych students), and (b) the narrow population that you sampled from is systematically different from the general population, *in some respect that is relevant to the psychological phenomenon that you intend to study*. The italicised part is the bit that lots of people forget: it is true that psychology undergraduates differ from the general population in lots of ways, and so a study that uses only psych students *may* have problems with external validity. However, if those differences aren't very relevant to the phenomenon that you're studying, then there's nothing to worry about. To make this a bit more concrete, here's two extreme examples:

- You want to measure "attitudes of the general public towards psychotherapy", but all of your participants are psychology students. This study would almost certainly have a problem with external validity.
- You want to measure the effectiveness of a visual illusion, and your participants are all psychology students. This study is very unlikely to have a problem with external validity

Having just spent the last couple of paragraphs focusing on the choice of participants (since that's the big issue that everyone tends to worry most about), it's worth remembering that external validity is a broader concept. The following are also examples of things that might pose a threat to external validity, depending on what kind of study you're doing:

- People might answer a "psychology questionnaire" in a manner that doesn't reflect what they would do in real life.
- Your lab experiment on (say) "human learning" has a different structure to the learning problems people face in real life.

2.6.3 Construct validity

Construct validity is basically a question of whether you're measuring what you want to be measuring. A measurement has good construct validity if it is actually measuring the correct theoretical construct, and bad construct validity if it doesn't. To give very simple (if ridiculous) example, suppose I'm trying to investigate

the rates with which university students cheat on their exams. And the way I attempt to measure it is by asking the cheating students to stand up in the lecture theatre so that I can count them. When I do this with a class of 300 students, 0 people claim to be cheaters. So I therefore conclude that the proportion of cheaters in my class is 0%. Clearly this is a bit ridiculous. But the point here is not that this is a very deep methodological example, but rather to explain what construct validity is. The problem with my measure is that while I'm *trying* to measure "the proportion of people who cheat" what I'm actually measuring is "the proportion of people stupid enough to own up to cheating, or bloody minded enough to pretend that they do". Obviously, these aren't the same thing! So my study has gone wrong, because my measurement has very poor construct validity.

2.6.4 Face validity

Face validity simply refers to whether or not a measure "looks like" it's doing what it's supposed to, nothing more. If I design a test of intelligence, and people look at it and they say "no, that test doesn't measure intelligence", then the measure lacks face validity. It's as simple as that. Obviously, face validity isn't very important from a pure scientific perspective. After all, what we care about is whether or not the measure *actually* does what it's supposed to do, not whether it *looks like* it does what it's supposed to do. As a consequence, we generally don't care very much about face validity. That said, the concept of face validity serves three useful pragmatic purposes:

- Sometimes, an experienced scientist will have a "hunch" that a particular measure won't work. While these sorts of hunches have no strict evidentiary value, it's often worth paying attention to them. Because often times people have knowledge that they can't quite verbalise, so there might be something to worry about even if you can't quite say why. In other words, when someone you trust criticises the face validity of your study, it's worth taking the time to think more carefully about your design to see if you can think of reasons why it might go awry. Mind you, if you don't find any reason for concern, then you should probably not worry: after all, face validity really doesn't matter much.
- Often (very often), completely uninformed people will also have a "hunch" that your research is crap. And they'll criticise it on the internet or something. On close inspection, you'll often notice that these criticisms are actually focused entirely on how the study "looks", but not on anything deeper. The concept of face validity is useful for gently explaining to people that they need to substantiate their arguments further.
- Expanding on the last point, if the beliefs of untrained people are critical (e.g., this is often the case for applied research where you actually want to convince policy makers of something or other) then you *have* to care about face validity. Simply because – whether you like it or not – a lot of people will use face validity as a proxy for real validity. If you want the government to change a law on scientific, psychological grounds, then it won't matter how good your studies "really" are. If they lack face validity, you'll find that politicians ignore you. Of course, it's somewhat unfair that policy often depends more on appearance than fact, but that's how things go.

2.6.5 Ecological validity

Ecological validity is a different notion of validity, which is similar to external validity, but less important. The idea is that, in order to be ecologically valid, the entire set up of the study should closely approximate the real world scenario that is being investigated. In a sense, ecological validity is a kind of face validity – it relates mostly to whether the study "looks" right, but with a bit more rigour to it. To be ecologically valid, the study has to look right in a fairly specific way. The idea behind it is the intuition that a study that is ecologically valid is more likely to be externally valid. It's no guarantee, of course. But the nice thing about ecological validity is that it's much easier to check whether a study is ecologically valid than it is to check whether a study is externally valid. An simple example would be eyewitness identification studies. Most of these studies tend to be done in a university setting, often with fairly simple array of faces to look at rather than a line up. The length of time between seeing the "criminal" and being asked to identify the suspect in

the “line up” is usually shorter. The “crime” isn’t real, so there’s no chance that the witness being scared, and there’s no police officers present, so there’s not as much chance of feeling pressured. These things all mean that the study *definitely* lacks ecological validity. They might (but might not) mean that it also lacks external validity.

2.7 Confounds, artifacts and other threats to validity

If we look at the issue of validity in the most general fashion, the two biggest worries that we have are *confounds* and *artifact*. These two terms are defined in the following way:

- **Confound:** A confound is an additional, often unmeasured variable⁵ that turns out to be related to both the predictors and the outcomes. The existence of confounds threatens the internal validity of the study because you can’t tell whether the predictor causes the outcome, or if the confounding variable causes it, etc.
- **Artifact:** A result is said to be “artifactual” if it only holds in the special situation that you happened to test in your study. The possibility that your result is an artifact describes a threat to your external validity, because it raises the possibility that you can’t generalise your results to the actual population that you care about.

As a general rule confounds are a bigger concern for non-experimental studies, precisely because they’re not proper experiments: by definition, you’re leaving lots of things uncontrolled, so there’s a lot of scope for confounds working their way into your study. Experimental research tends to be much less vulnerable to confounds: the more control you have over what happens during the study, the more you can prevent confounds from appearing.

However, there’s always swings and roundabouts, and when we start thinking about artifacts rather than confounds, the shoe is very firmly on the other foot. For the most part, artifactual results tend to be a concern for experimental studies than for non-experimental studies. To see this, it helps to realise that the reason that a lot of studies are non-experimental is precisely because what the researcher is trying to do is examine human behaviour in a more naturalistic context. By working in a more real-world context, you lose experimental control (making yourself vulnerable to confounds) but because you tend to be studying human psychology “in the wild” you reduce the chances of getting an artifactual result. Or, to put it another way, when you take psychology out of the wild and bring it into the lab (which we usually have to do to gain our experimental control), you always run the risk of accidentally studying something different than you wanted to study: which is more or less the definition of an artifact.

Be warned though: the above is a rough guide only. It’s absolutely possible to have confounds in an experiment, and to get artifactual results with non-experimental studies. This can happen for all sorts of reasons, not least of which is researcher error. In practice, it’s really hard to think everything through ahead of time, and even very good researchers make mistakes. But other times it’s unavoidable, simply because the researcher has ethics (e.g., see “differential attrition”).

Okay. There’s a sense in which almost any threat to validity can be characterised as a confound or an artifact: they’re pretty vague concepts. So let’s have a look at some of the most common examples...

2.7.1 History effects

History effects refer to the possibility that specific events may occur during the study itself that might influence the outcomes. For instance, something might happen in between a pre-test and a post-test. Or,

⁵The reason why I say that it’s unmeasured is that if you *have* measured it, then you can use some fancy statistical tricks to deal with the confound. Because of the existence of these statistical solutions to the problem of confounds, we often refer to a confound that we have measured and dealt with as a *covariate*. Dealing with covariates is a topic for a more advanced course, but I thought I’d mention it in passing, since it’s kind of comforting to at least know that this stuff exists.

in between testing participant 23 and participant 24. Alternatively, it might be that you're looking at an older study, which was perfectly valid for its time, but the world has changed enough since then that the conclusions are no longer trustworthy. Examples of things that would count as history effects:

- You're interested in how people think about risk and uncertainty. You started your data collection in December 2010. But finding participants and collecting data takes time, so you're still finding new people in February 2011. Unfortunately for you (and even more unfortunately for others), the Queensland floods occurred in January 2011, causing billions of dollars of damage and killing many people. Not surprisingly, the people tested in February 2011 express quite different beliefs about handling risk than the people tested in December 2010. Which (if any) of these reflects the “true” beliefs of participants? I think the answer is probably both: the Queensland floods genuinely changed the beliefs of the Australian public, though possibly only temporarily. The key thing here is that the “history” of the people tested in February is quite different to people tested in December.
- You're testing the psychological effects of a new anti-anxiety drug. So what you do is measure anxiety before administering the drug (e.g., by self-report, and taking physiological measures, let's say), then you administer the drug, and then you take the same measures afterwards. In the middle, however, because your labs are in Los Angeles, there's an earthquake, which increases the anxiety of the participants.

2.7.2 Maturation effects

As with history effects, ***maturational effects*** are fundamentally about change over time. However, maturation effects aren't in response to specific events. Rather, they relate to how people change on their own over time: we get older, we get tired, we get bored, etc. Some examples of maturation effects:

- When doing developmental psychology research, you need to be aware that children grow up quite rapidly. So, suppose that you want to find out whether some educational trick helps with vocabulary size among 3 year olds. One thing that you need to be aware of is that the vocabulary size of children that age is growing at an incredible rate (multiple words per day), all on its own. If you design your study without taking this maturational effect into account, then you won't be able to tell if your educational trick works.
- When running a very long experiment in the lab (say, something that goes for 3 hours), it's very likely that people will begin to get bored and tired, and that this maturational effect will cause performance to decline, regardless of anything else going on in the experiment

2.7.3 Repeated testing effects

An important type of history effect is the effect of ***repeated testing***. Suppose I want to take two measurements of some psychological construct (e.g., anxiety). One thing I might be worried about is if the first measurement has an effect on the second measurement. In other words, this is a history effect in which the “event” that influences the second measurement is the first measurement itself! This is not at all uncommon. Examples of this include:

- ***Learning and practice***: e.g., “intelligence” at time 2 might appear to go up relative to time 1 because participants learned the general rules of how to solve “intelligence-test-style” questions during the first testing session.
- ***Familiarity with the testing situation***: e.g., if people are nervous at time 1, this might make performance go down; after sitting through the first testing situation, they might calm down a lot precisely because they've seen what the testing looks like.
- ***Auxiliary changes caused by testing***: e.g., if a questionnaire assessing mood is boring, then mood at measurement at time 2 is more likely to become “bored”, precisely because of the boring measurement made at time 1.

2.7.4 Selection bias

Selection bias is a pretty broad term. Suppose that you're running an experiment with two groups of participants, where each group gets a different "treatment", and you want to see if the different treatments lead to different outcomes. However, suppose that, despite your best efforts, you've ended up with a gender imbalance across groups (say, group A has 80% females and group B has 50% females). It might sound like this could never happen, but trust me, it can. This is an example of a selection bias, in which the people "selected into" the two groups have different characteristics. If any of those characteristics turns out to be relevant (say, your treatment works better on females than males) then you're in a lot of trouble.

2.7.5 Differential attrition

One quite subtle danger to be aware of is called **differential attrition**, which is a kind of selection bias that is caused by the study itself. Suppose that, for the first time ever in the history of psychology, I manage to find the perfectly balanced and representative sample of people. I start running "Dan's incredibly long and tedious experiment" on my perfect sample, but then, because my study is incredibly long and tedious, lots of people start dropping out. I can't stop this: as we'll discuss later in the chapter on research ethics, participants absolutely have the right to stop doing any experiment, any time, for whatever reason they feel like, and as researchers we are morally (and professionally) obliged to remind people that they do have this right. So, suppose that "Dan's incredibly long and tedious experiment" has a very high drop out rate. What do you suppose the odds are that this drop out is random? Answer: zero. Almost certainly, the people who remain are more conscientious, more tolerant of boredom etc than those that leave. To the extent that (say) conscientiousness is relevant to the psychological phenomenon that I care about, this attrition can decrease the validity of my results.

When thinking about the effects of differential attrition, it is sometimes helpful to distinguish between two different types. The first is **homogeneous attrition**, in which the attrition effect is the same for all groups, treatments or conditions. In the example I gave above, the differential attrition would be homogeneous if (and only if) the easily bored participants are dropping out of all of the conditions in my experiment at about the same rate. In general, the main effect of homogeneous attrition is likely to be that it makes your sample unrepresentative. As such, the biggest worry that you'll have is that the generalisability of the results decreases: in other words, you lose external validity.

The second type of differential attrition is **heterogeneous attrition**, in which the attrition effect is different for different groups. This is a much bigger problem: not only do you have to worry about your external validity, you also have to worry about your internal validity too. To see why this is the case, let's consider a very dumb study in which I want to see if insulting people makes them act in a more obedient way. Why anyone would actually want to study that I don't know, but let's suppose I really, deeply cared about this. So, I design my experiment with two conditions. In the "treatment" condition, the experimenter insults the participant and then gives them a questionnaire designed to measure obedience. In the "control" condition, the experimenter engages in a bit of pointless chitchat and then gives them the questionnaire. Leaving aside the questionable scientific merits and dubious ethics of such a study, let's have a think about what might go wrong here. As a general rule, when someone insults me to my face, I tend to get much less co-operative. So, there's a pretty good chance that a lot more people are going to drop out of the treatment condition than the control condition. And this drop out isn't going to be random. The people most likely to drop out would probably be the people who don't care all that much about the importance of obediently sitting through the experiment. Since the most bloody minded and disobedient people all left the treatment group but not the control group, we've introduced a confound: the people who actually took the questionnaire in the treatment group were *already* more likely to be dutiful and obedient than the people in the control group. In short, in this study insulting people doesn't make them more obedient: it makes the more disobedient people leave the experiment! The internal validity of this experiment is completely shot.

2.7.6 Non-response bias

Non-response bias is closely related to selection bias, and to differential attrition. The simplest version of the problem goes like this. You mail out a survey to 1000 people, and only 300 of them reply. The 300 people who replied are almost certainly not a random subsample. People who respond to surveys are systematically different to people who don't. This introduces a problem when trying to generalise from those 300 people who replied, to the population at large; since you now have a very non-random sample. The issue of non-response bias is more general than this, though. Among the (say) 300 people that did respond to the survey, you might find that not everyone answers every question. If (say) 80 people chose not to answer one of your questions, does this introduce problems? As always, the answer is maybe. If the question that wasn't answered was on the last page of the questionnaire, and those 80 surveys were returned with the last page missing, there's a good chance that the missing data isn't a big deal: probably the pages just fell off. However, if the question that 80 people didn't answer was the most confrontational or invasive personal question in the questionnaire, then almost certainly you've got a problem. In essence, what you're dealing with here is what's called the problem of **missing data**. If the data that is missing was "lost" randomly, then it's not a big problem. If it's missing systematically, then it can be a big problem.

2.7.7 Regression to the mean

Regression to the mean is a curious variation on selection bias. It refers to any situation where you select data based on an extreme value on some measure. Because the measure has natural variation, it almost certainly means that when you take a subsequent measurement, that later measurement will be less extreme than the first one, purely by chance.

Here's an example. Suppose I'm interested in whether a psychology education has an adverse effect on very smart kids. To do this, I find the 20 psych I students with the best high school grades and look at how well they're doing at university. It turns out that they're doing a lot better than average, but they're not topping the class at university, even though they did top their classes at high school. What's going on? The natural first thought is that this must mean that the psychology classes must be having an adverse effect on those students. However, while that might very well be the explanation, it's more likely that what you're seeing is an example of "regression to the mean". To see how it works, let's take a moment to think about what is required to get the best mark in a class, regardless of whether that class be at high school or at university. When you've got a big class, there are going to be *lots* of very smart people enrolled. To get the best mark you have to be very smart, work very hard, and be a bit lucky. The exam has to ask just the right questions for your idiosyncratic skills, and you have to not make any dumb mistakes (we all do that sometimes) when answering them. And that's the thing: intelligence and hard work are transferrable from one class to the next. Luck isn't. The people who got lucky in high school won't be the same as the people who get lucky at university. That's the very definition of "luck". The consequence of this is that, when you select people at the very extreme values of one measurement (the top 20 students), you're selecting for hard work, skill and luck. But because the luck doesn't transfer to the second measurement (only the skill and work), these people will all be expected to drop a little bit when you measure them a second time (at university). So their scores fall back a little bit, back towards everyone else. This is regression to the mean.

Regression to the mean is surprisingly common. For instance, if two very tall people have kids, their children will tend to be taller than average, but not as tall as the parents. The reverse happens with very short parents: two very short parents will tend to have short children, but nevertheless those kids will tend to be taller than the parents. It can also be extremely subtle. For instance, there have been studies done that suggested that people learn better from negative feedback than from positive feedback. However, the way that people tried to show this was to give people positive reinforcement whenever they did good, and negative reinforcement when they did bad. And what you see is that after the positive reinforcement, people tended to do worse; but after the negative reinforcement they tended to do better. But! Notice that there's a selection bias here: when people do very well, you're selecting for "high" values, and so you should *expect* (because of regression to the mean) that performance on the next trial should be worse, regardless of whether reinforcement is given. Similarly, after a bad trial, people will tend to improve all on their own. The apparent

superiority of negative feedback is an artifact caused by regression to the mean (see Kahneman and Tversky, 1973, for discussion).

2.7.8 Experimenter bias

Experimenter bias can come in multiple forms. The basic idea is that the experimenter, despite the best of intentions, can accidentally end up influencing the results of the experiment by subtly communicating the “right answer” or the “desired behaviour” to the participants. Typically, this occurs because the experimenter has special knowledge that the participant does not – either the right answer to the questions being asked, or knowledge of the expected pattern of performance for the condition that the participant is in, and so on. The classic example of this happening is the case study of “Clever Hans”, which dates back to 1907 (Pfungst, 1911; Hothersall, 2004). Clever Hans was a horse that apparently was able to read and count, and perform other human like feats of intelligence. After Clever Hans became famous, psychologists started examining his behaviour more closely. It turned out that – not surprisingly – Hans didn’t know how to do maths. Rather, Hans was responding to the human observers around him. Because they did know how to count, and the horse had learned to change its behaviour when people changed theirs.

The general solution to the problem of experimenter bias is to engage in double blind studies, where neither the experimenter nor the participant knows which condition the participant is in, or knows what the desired behaviour is. This provides a very good solution to the problem, but it’s important to recognise that it’s not quite ideal, and hard to pull off perfectly. For instance, the obvious way that I could try to construct a double blind study is to have one of my Ph.D. students (one who doesn’t know anything about the experiment) run the study. That feels like it should be enough. The only person (me) who knows all the details (e.g., correct answers to the questions, assignments of participants to conditions) has no interaction with the participants, and the person who does all the talking to people (the Ph.D. student) doesn’t know anything. Except, that last part is very unlikely to be true. In order for the Ph.D. student to run the study effectively, they need to have been briefed by me, the researcher. And, as it happens, the Ph.D. student also knows me, and knows a bit about my general beliefs about people and psychology (e.g., I tend to think humans are much smarter than psychologists give them credit for). As a result of all this, it’s almost impossible for the experimenter to avoid knowing a little bit about what expectations I have. And even a little bit of knowledge can have an effect: suppose the experimenter accidentally conveys the fact that the participants are expected to do well in this task. Well, there’s a thing called the “Pygmalion effect”: if you expect great things of people, they’ll rise to the occasion; but if you expect them to fail, they’ll do that too. In other words, the expectations become a self-fulfilling prophecy.

2.7.9 Demand effects and reactivity

When talking about experimenter bias, the worry is that the experimenter’s knowledge or desires for the experiment are communicated to the participants, and that these effect people’s behaviour (Rosenthal, 1966). However, even if you manage to stop this from happening, it’s almost impossible to stop people from knowing that they’re part of a psychological study. And the mere fact of knowing that someone is watching/studying you can have a pretty big effect on behaviour. This is generally referred to as **reactivity** or **demand effects**. The basic idea is captured by the Hawthorne effect: people alter their performance because of the attention that the study focuses on them. The effect takes its name from the “Hawthorne Works” factory outside of Chicago (see Adair, 1984). A study done in the 1920s looking at the effects of lighting on worker productivity at the factory turned out to be an effect of the fact that the workers knew they were being studied, rather than the lighting.

To get a bit more specific about some of the ways in which the mere fact of being in a study can change how people behave, it helps to think like a social psychologist and look at some of the *roles* that people might adopt during an experiment, but might not adopt if the corresponding events were occurring in the real world:

- The *good participant* tries to be too helpful to the researcher: he or she seeks to figure out the experimenter's hypotheses and confirm them.
- The *negative participant* does the exact opposite of the good participant: he or she seeks to break or destroy the study or the hypothesis in some way.
- The *faithful participant* is unnaturally obedient: he or she seeks to follow instructions perfectly, regardless of what might have happened in a more realistic setting.
- The *apprehensive participant* gets nervous about being tested or studied, so much so that his or her behaviour becomes highly unnatural, or overly socially desirable.

2.7.10 Placebo effects

The **placebo effect** is a specific type of demand effect that we worry a lot about. It refers to the situation where the mere fact of being treated causes an improvement in outcomes. The classic example comes from clinical trials: if you give people a completely chemically inert drug and tell them that it's a cure for a disease, they will tend to get better faster than people who aren't treated at all. In other words, it is people's belief that they are being treated that causes the improved outcomes, not the drug.

2.7.11 Situation, measurement and subpopulation effects

In some respects, these terms are a catch-all term for “all other threats to external validity”. They refer to the fact that the choice of subpopulation from which you draw your participants, the location, timing and manner in which you run your study (including who collects the data) and the tools that you use to make your measurements might all be influencing the results. Specifically, the worry is that these things might be influencing the results in such a way that the results won't generalise to a wider array of people, places and measures.

2.7.12 Fraud, deception and self-deception

It is difficult to get a man to understand something, when his salary depends on his not understanding it.

– Upton Sinclair

One final thing that I feel like I should mention. While reading what the textbooks often have to say about assessing the validity of the study, I couldn't help but notice that they seem to make the assumption that the researcher is honest. I find this hilarious. While the vast majority of scientists are honest, in my experience at least, some are not.⁶ Not only that, as I mentioned earlier, scientists are not immune to belief bias – it's easy for a researcher to end up deceiving themselves into believing the wrong thing, and this can lead them to conduct subtly flawed research, and then hide those flaws when they write it up. So you need to consider not only the (probably unlikely) possibility of outright fraud, but also the (probably quite common) possibility that the research is unintentionally “slanted”. I opened a few standard textbooks and didn't find much of a discussion of this problem, so here's my own attempt to list a few ways in which these issues can arise are:

- **Data fabrication.** Sometimes, people just make up the data. This is occasionally done with “good” intentions. For instance, the researcher believes that the fabricated data do reflect the truth, and may actually reflect “slightly cleaned up” versions of actual data. On other occasions, the fraud is deliberate and malicious. Some high-profile examples where data fabrication has been alleged or

⁶Some people might argue that if you're not honest then you're not a real scientist. Which does have some truth to it I guess, but that's disingenuous (google the “No true Scotsman” fallacy). The fact is that there are lots of people who are employed ostensibly as scientists, and whose work has all of the trappings of science, but who are outright fraudulent. Pretending that they don't exist by saying that they're not scientists is just childish.

shown include Cyril Burt (a psychologist who is thought to have fabricated some of his data), Andrew Wakefield (who has been accused of fabricating his data connecting the MMR vaccine to autism) and Hwang Woo-suk (who falsified a lot of his data on stem cell research).

- **Hoaxes.** Hoaxes share a lot of similarities with data fabrication, but they differ in the intended purpose. A hoax is often a joke, and many of them are intended to be (eventually) discovered. Often, the point of a hoax is to discredit someone or some field. There's quite a few well known scientific hoaxes that have occurred over the years (e.g., Piltdown man) some of were deliberate attempts to discredit particular fields of research (e.g., the Sokal affair).
- **Data misrepresentation.** While fraud gets most of the headlines, it's much more common in my experience to see data being misrepresented. When I say this, I'm not referring to newspapers getting it wrong (which they do, almost always). I'm referring to the fact that often, the data don't actually say what the researchers think they say. My guess is that, almost always, this isn't the result of deliberate dishonesty, it's due to a lack of sophistication in the data analyses. For instance, think back to the example of Simpson's paradox that I discussed in the beginning of these notes. It's very common to see people present "aggregated" data of some kind; and sometimes, when you dig deeper and find the raw data yourself, you find that the aggregated data tell a different story to the disaggregated data. Alternatively, you might find that some aspect of the data is being hidden, because it tells an inconvenient story (e.g., the researcher might choose not to refer to a particular variable). There's a lot of variants on this; many of which are very hard to detect.
- **Study “misdesign”.** Okay, this one is subtle. Basically, the issue here is that a researcher designs a study that has built-in flaws, and those flaws are never reported in the paper. The data that are reported are completely real, and are correctly analysed, but they are produced by a study that is actually quite wrongly put together. The researcher really wants to find a particular effect, and so the study is set up in such a way as to make it “easy” to (artifactually) observe that effect. One sneaky way to do this – in case you're feeling like dabbling in a bit of fraud yourself – is to design an experiment in which it's obvious to the participants what they're “supposed” to be doing, and then let reactivity work its magic for you. If you want, you can add all the trappings of double blind experimentation etc. It won't make a difference, since the study materials themselves are subtly telling people what you want them to do. When you write up the results, the fraud won't be obvious to the reader: what's obvious to the participant when they're in the experimental context isn't always obvious to the person reading the paper. Of course, the way I've described this makes it sound like it's always fraud: probably there are cases where this is done deliberately, but in my experience the bigger concern has been with unintentional misdesign. The researcher *believes* ... and so the study just happens to end up with a built in flaw, and that flaw then magically erases itself when the study is written up for publication.
- **Data mining & post hoc hypothesising.** Another way in which the authors of a study can more or less lie about what they found is by engaging in what's referred to as “data mining”. As we'll discuss later in the class, if you keep trying to analyse your data in lots of different ways, you'll eventually find something that “looks” like a real effect but isn't. This is referred to as “data mining”. It used to be quite rare because data analysis used to take weeks, but now that everyone has very powerful statistical software on their computers, it's becoming very common. Data mining per se isn't “wrong”, but the more that you do it, the bigger the risk you're taking. The thing that is wrong, and I suspect is very common, is *unacknowledged* data mining. That is, the researcher run every possible analysis known to humanity, finds the one that works, and then pretends that this was the only analysis that they ever conducted. Worse yet, they often “invent” a hypothesis after looking at the data, to cover up the data mining. To be clear: it's not wrong to change your beliefs after looking at the data, and to reanalyse your data using your new “post hoc” hypotheses. What is wrong (and, I suspect, common) is failing to acknowledge that you've done so. If you acknowledge that you did it, then other researchers are able to take your behaviour into account. If you don't, then they can't. And that makes your behaviour deceptive. Bad!
- **Publication bias & self-censoring.** Finally, a pervasive bias is “non-reporting” of negative results. This is almost impossible to prevent. Journals don't publish every article that is submitted to them: they prefer to publish articles that find “something”. So, if 20 people run an experiment looking at whether reading *Finnegans Wake* causes insanity in humans, and 19 of them find that it doesn't, which

one do you think is going to get published? Obviously, it's the one study that did find that *Finnegans Wake* causes insanity⁷. This is an example of a *publication bias*: since no-one ever published the 19 studies that didn't find an effect, a naive reader would never know that they existed. Worse yet, most researchers "internalise" this bias, and end up *self-censoring* their research. Knowing that negative results aren't going to be accepted for publication, they never even try to report them. As a friend of mine says "for every experiment that you get published, you also have 10 failures". And she's right. The catch is, while some (maybe most) of those studies are failures for boring reasons (e.g. you stuffed something up) others might be genuine "null" results that you ought to acknowledge when you write up the "good" experiment. And telling which is which is often hard to do. A good place to start is a paper by Ioannidis (2005) with the depressing title "Why most published research findings are false". I'd also suggest taking a look at work by Kühberger et al. (2014) presenting statistical evidence that this actually happens in psychology.

There's probably a lot more issues like this to think about, but that'll do to start with. What I really want to point out is the blindingly obvious truth that real world science is conducted by actual humans, and only the most gullible of people automatically assumes that everyone else is honest and impartial. Actual scientists aren't usually *that* naive, but for some reason the world likes to pretend that we are, and the textbooks we usually write seem to reinforce that stereotype.

2.8 Summary

This chapter isn't really meant to provide a comprehensive discussion of psychological research methods: it would require another volume just as long as this one to do justice to the topic. However, in real life statistics and study design are tightly intertwined, so it's very handy to discuss some of the key topics. In this chapter, I've briefly discussed the following topics:

- Introduction to psychological measurement. What does it mean to operationalise a theoretical construct? What does it mean to have variables and take measurements?
- Scales of measurement and types of variables. Remember that there are *two* different distinctions here: there's the difference between discrete and continuous data, and there's the difference between the four different scale types (nominal, ordinal, interval and ratio).
- Reliability of a measurement. If I measure the "same" thing twice, should I expect to see the same result? Only if my measure is reliable. But what does it mean to talk about doing the "same" thing? Well, that's why we have different types of reliability. Make sure you remember what they are.
- Terminology: predictors and outcomes. What roles do variables play in an analysis? Can you remember the difference between predictors and outcomes? Dependent and independent variables? Etc.
- Experimental and non-experimental research designs. What makes an experiment an experiment? Is it a nice white lab coat, or does it have something to do with researcher control over variables?
- Validity and its threats. Does your study measure what you want it to? How might things go wrong? And is it my imagination, or was that a very long list of possible ways in which things can go wrong?

All this should make clear to you that study design is a critical part of research methodology. I built this chapter from the classic little book by Campbell and Stanley (1963), but there are of course a large number of textbooks out there on research design. Spend a few minutes with your favourite search engine and you'll find dozens.

⁷Clearly, the real effect is that only insane people would even try to read *Finnegans Wake*.

Part II. An introduction to R

Chapter 3

Getting started with R

Robots are nice to work with.

—Roger Zelazny¹

In this chapter I'll discuss how to get started in R. I'll briefly talk about how to download and install R, but most of the chapter will be focused on getting you started typing R commands. Our goal in this chapter is not to learn any statistical concepts: we're just trying to learn the basics of how R works and get comfortable interacting with the system. To do this, we'll spend a bit of time using R as a simple calculator, since that's the easiest thing to do with R. In doing so, you'll get a bit of a feel for what it's like to work in R. From there I'll introduce some very basic programming ideas: in particular, I'll talk about the idea of defining *variables* to store information, and a few things that you can do with these variables.

However, before going into any of the specifics, it's worth talking a little about why you might want to use R at all. Given that you're reading this, you've probably got your own reasons. However, if those reasons are “because that's what my stats class uses”, it might be worth explaining a little why your lecturer has chosen to use R for the class. Of course, I don't really know why *other* people choose R, so I'm really talking about why I use it.

- It's sort of obvious, but worth saying anyway: doing your statistics on a computer is faster, easier and more powerful than doing statistics by hand. Computers excel at mindless repetitive tasks, and a lot of statistical calculations are both mindless and repetitive. For most people, the only reason to ever do statistical calculations with pencil and paper is for learning purposes. In my class I do occasionally suggest doing some calculations that way, but the only real value to it is pedagogical. It does help you to get a “feel” for statistics to do some calculations yourself, so it's worth doing it once. But only once!
- Doing statistics in a spreadsheet (e.g., Microsoft Excel) is generally a bad idea in the long run. Although many people are likely feel more familiar with them, spreadsheets are very limited in terms of what analyses they allow you do. If you get into the habit of trying to do your real life data analysis using spreadsheets, then you've dug yourself into a very deep hole.
- Avoiding proprietary software is a very good idea. There are a lot of commercial packages out there that you can buy, some of which I like and some of which I don't. They're usually very glossy in their appearance, and generally very powerful (much more powerful than spreadsheets). However, they're also very expensive: usually, the company sells “student versions” (crippled versions of the real thing) very cheaply; they sell full powered “educational versions” at a price that makes me wince; and they sell commercial licences with a staggeringly high price tag. The business model here is to suck you in during your student days, and then leave you dependent on their tools when you go out into the real world. It's hard to blame them for trying, but personally I'm not in favour of shelling out thousands of dollars if I can avoid it. And you can avoid it: if you make use of packages like R that are open source and free, you never get trapped having to pay exorbitant licensing fees.

¹Source: *Dismal Light* (1968).

- Something that you might not appreciate now, but will love later on if you do anything involving data analysis, is the fact that R is highly extensible. When you download and install R, you get all the basic “packages”, and those are very powerful on their own. However, because R is so open and so widely used, it’s become something of a standard tool in statistics, and so lots of people write their own packages that extend the system. And these are freely available too. One of the consequences of this, I’ve noticed, is that if you open up an advanced textbook (a recent one, that is) rather than introductory textbooks, is that a *lot* of them use R. In other words, if you learn how to do your basic statistics in R, then you’re a lot closer to being able to use the state of the art methods than you would be if you’d started out with a “simpler” system: so if you want to become a genuine expert in psychological data analysis, learning R is a very good use of your time.
- Related to the previous point: R is a real programming language. As you get better at using R for data analysis, you’re also learning to program. To some people this might seem like a bad thing, but in truth, programming is a core research skill across a lot of the social and behavioural sciences. Think about how many surveys and experiments are done online, or presented on computers. Think about all those online social environments which you might be interested in studying; and maybe collecting data from in an automated fashion. Think about artificial intelligence systems, computer vision and speech recognition. If any of these are things that you think you might want to be involved in – as someone “doing research in psychology”, that is – you’ll need to know a bit of programming. And if you don’t already know how to program, then learning how to do statistics using R is a nice way to start.

Those are the main reasons I use R. It’s not without its flaws: it’s not easy to learn, and it has a few very annoying quirks to it that we’re all pretty much stuck with, but on the whole I think the strengths outweigh the weakness; more so than any other option I’ve encountered so far.

3.1 Installing R

Okay, enough with the sales pitch. Let’s get started. Just as with any piece of software, R needs to be installed on a “computer”, which is a magical box that does cool things and delivers free ponies. Or something along those lines: I may be confusing computers with the iPad marketing campaigns. Anyway, R is freely distributed online, and you can download it from the R homepage, which is:

<http://cran.r-project.org/>

At the top of the page – under the heading “Download and Install R” – you’ll see separate links for Windows users, Mac users, and Linux users. If you follow the relevant link, you’ll see that the online instructions are pretty self-explanatory, but I’ll walk you through the installation anyway. As of this writing, the current version of R is 3.0.2 (“Frisbee Sailing”), but they usually issue updates every six months, so you’ll probably have a newer version.²

3.1.1 Installing R on a Windows computer

The CRAN homepage changes from time to time, and it’s not particularly pretty, or all that well-designed quite frankly. But it’s not difficult to find what you’re after. In general you’ll find a link at the top of the page with the text “Download R for Windows”. If you click on that, it will take you to a page that offers you a few options. Again, at the very top of the page you’ll be told to click on a link that says to click here if you’re installing R for the first time. That’s probably what you want. This will take you to a page that has a prominent link at the top called “Download R 3.0.2 for Windows”. That’s the one you want. Click on

²Although R is updated frequently, it doesn’t usually make much of a difference for the sort of work we’ll do in this book. In fact, during the writing of the book I upgraded several times, and didn’t have to change much except these sections describing the downloading.

that and your browser should start downloading a file called `R-3.0.2-win.exe`, or whatever the equivalent version number is by the time you read this. The file for version 3.0.2 is about 54MB in size, so it may take some time depending on how fast your internet connection is. Once you've downloaded the file, double click to install it. As with any software you download online, Windows will ask you some questions about whether you trust the file and so on. After you click through those, it'll ask you where you want to install it, and what components you want to install. The default values should be fine for most people, so again, just click through. Once all that is done, you should have R installed on your system. You can access it from the Start menu, or from the desktop if you asked it to add a shortcut there. You can now open up R in the usual way if you want to, but what I'm going to suggest is that instead of doing that you should now install RStudio.

3.1.2 Installing R on a Mac

When you click on the Mac OS X link, you should find yourself on a page with the title "R for Mac OS X". The vast majority of Mac users will have a fairly recent version of the operating system: as long as you're running Mac OS X 10.6 (Snow Leopard) or higher, then you'll be fine.³ There's a fairly prominent link on the page called "`R-3.0.2.pkg`", which is the one you want. Click on that link and you'll start downloading the installer file, which is (not surprisingly) called `R-3.0.2.pkg`. It's about 61MB in size, so the download can take a while on slower internet connections.

Once you've downloaded `R-3.0.2.pkg`, all you need to do is open it by double clicking on the package file. The installation should go smoothly from there: just follow all the instructions just like you usually do when you install something. Once it's finished, you'll find a file called `R.app` in the Applications folder. You can now open up R in the usual way⁴ if you want to, but what I'm going to suggest is that instead of doing that you should now install RStudio.

3.1.3 Installing R on a Linux computer

If you're successfully managing to run a Linux box, regardless of what distribution, then you should find the instructions on the website easy enough. You can compile R from source yourself if you want, or install it through your package management system, which will probably have R in it. Alternatively, the CRAN site has precompiled binaries for Debian, Red Hat, Suse and Ubuntu and has separate instructions for each. Once you've got R installed, you can run it from the command line just by typing `R`. However, if you're feeling envious of Windows and Mac users for their fancy GUIs, you can download RStudio too.

3.1.4 Downloading and installing RStudio

Okay, so regardless of what operating system you're using, the last thing that I told you to do is to download RStudio. To understand why I've suggested this, you need to understand a little bit more about R itself. The term R doesn't really refer to a specific application on your computer. Rather, it refers to the underlying statistical language. You can use this language through lots of different applications. When you install R initially, it comes with one application that lets you do this: it's the `R.exe` application on a Windows machine, and the `R.app` application on a Mac. But that's not the only way to do it. There are lots of different applications that you can use that will let you interact with R. One of those is called RStudio, and it's the one I'm going to suggest that you use. RStudio provides a clean, professional interface to R that I find much nicer to work with than either the Windows or Mac defaults. Like R itself, RStudio is free software: you can find all the details on their webpage. In the meantime, you can download it here:

³If you're running an older version of the Mac OS, then you need to follow the link to the "old" page (<http://cran.r-project.org/bin/macosx.old/>). You should be able to find the installer file that you need at the bottom of the page.

⁴Tip for advanced Mac users. You can run R from the terminal if you want to. The command is just "R". It behaves like the normal desktop version, except that help documentation behaves like a "man" page instead of opening in a new window.

<http://www.RStudio.org/>

When you visit the RStudio website, you'll probably be struck by how much cleaner and simpler it is than the CRAN website,⁵ and how obvious it is what you need to do: click the big green button that says "Download".

When you click on the download button on the homepage it will ask you to choose whether you want the desktop version or the server version. You want the desktop version. After choosing the desktop version it will take you to a page <http://www.RStudio.org/download/desktop>) that shows several possible downloads: there's a different one for each operating system. However, the nice people at RStudio have designed the webpage so that it automatically recommends the download that is most appropriate for your computer. Click on the appropriate link, and the RStudio installer file will start downloading.

Once it's finished downloading, open the installer file in the usual way to install RStudio. After it's finished installing, you can start R by opening RStudio. You don't need to open R.app or R.exe in order to access R. RStudio will take care of that for you. To illustrate what RStudio looks like, Figure 3.1 shows a screenshot of an R session in progress. In this screenshot, you can see that it's running on a Mac, but it looks almost identical no matter what operating system you have. The Windows version looks more like a Windows application (e.g., the menus are attached to the application window and the colour scheme is slightly different), but it's more or less identical. There are a few minor differences in where things are located in the menus (I'll point them out as we go along) and in the shortcut keys, because RStudio is trying to "feel" like a proper Mac application or a proper Windows application, and this means that it has to change its behaviour a little bit depending on what computer it's running on. Even so, these differences are very small: I started out using the Mac version of RStudio and then started using the Windows version as well in order to write these notes.

The only "shortcoming" I've found with RStudio is that – as of this writing – it's still a work in progress. The "problem" is that they keep improving it. New features keep turning up the more recent releases, so there's a good chance that by the time you read this book there will be a version out that has some really neat things that weren't in the version that I'm using now.

3.1.5 Starting up R

One way or another, regardless of what operating system you're using and regardless of whether you're using RStudio, or the default GUI, or even the command line, it's time to open R and get started. When you do that, the first thing you'll see (assuming that you're looking at the **R console**, that is) is a whole lot of text that doesn't make much sense. It should look something like this:

```
R version 3.0.2 (2013-09-25) -- "Frisbee Sailing"
Copyright (C) 2013 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin10.8.0 (64-bit)
```

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

```
Natural language support but running in an English locale
```

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

⁵This is probably no coincidence: the people who design and distribute the core R language itself are focused on technical stuff. And sometimes they almost seem to forget that there's an actual human user at the end. The people who design and distribute RStudio are focused on user interface. They want to make R as usable as possible. The two websites reflect that difference.

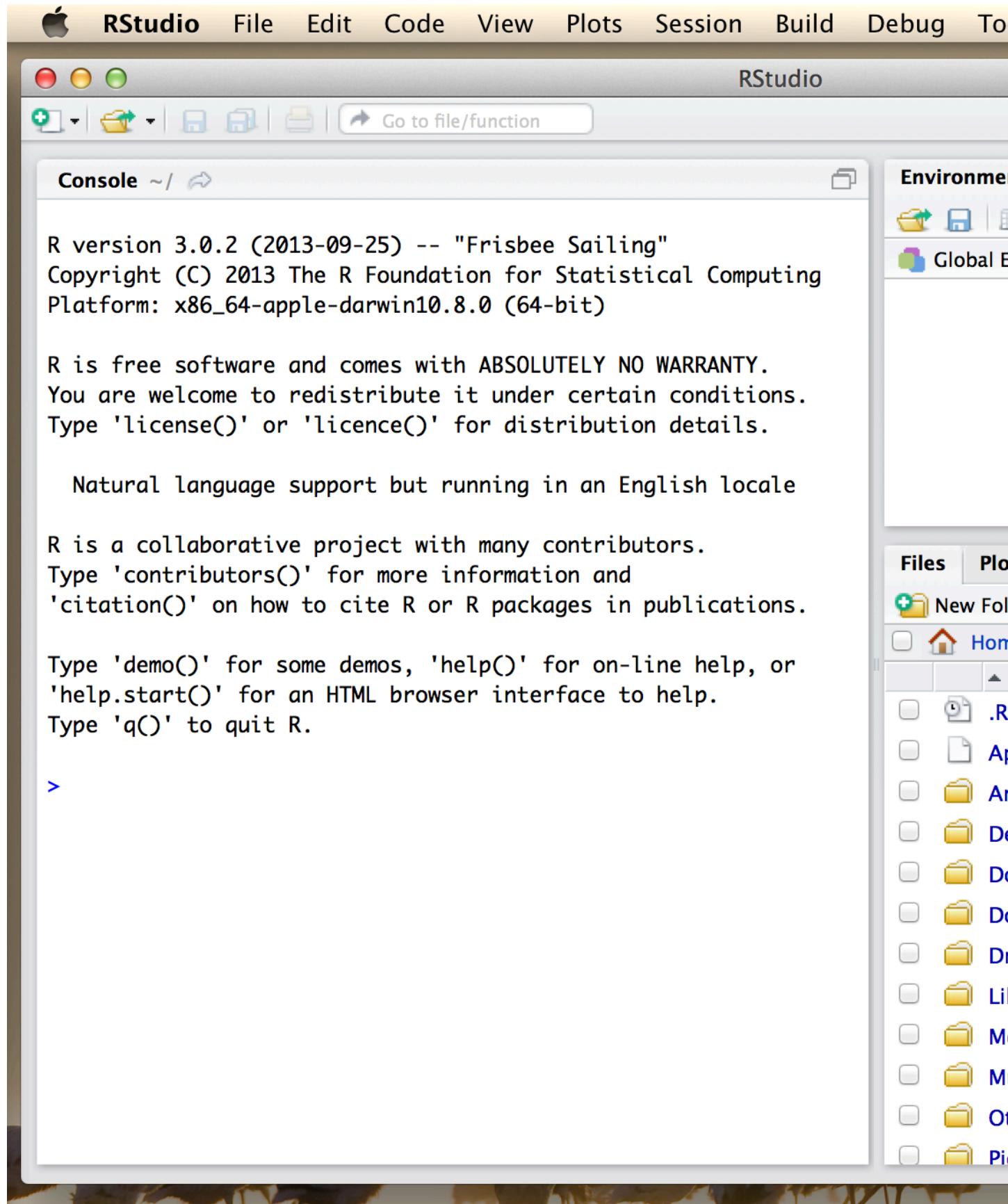


Figure 3.1: An R session in progress running through RStudio. The picture shows RStudio running on a Mac, but the Windows interface is almost identical.

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

>

Most of this text is pretty uninteresting, and when doing real data analysis you'll never really pay much attention to it. The important part of it is this...

>

... which has a flashing cursor next to it. That's the ***command prompt***. When you see this, it means that R is waiting patiently for you to do something!

3.2 Typing commands at the R console

One of the easiest things you can do with R is use it as a simple calculator, so it's a good place to start. For instance, try typing `10 + 20`, and hitting enter.⁶ When you do this, you've entered a ***command***, and R will "execute" that command. What you see on screen now will be this:

```
> 10 + 20
[1] 30
```

Not a lot of surprises in this extract. But there's a few things worth talking about, even with such a simple example. Firstly, it's important that you understand how to read the extract. In this example, what *I* typed was the `10 + 20` part. I didn't type the `>` symbol: that's just the R command prompt and isn't part of the actual command. And neither did I type the `[1] 30` part. That's what R printed out in response to my command.

Secondly, it's important to understand how the output is formatted. Obviously, the correct answer to the sum `10 + 20` is 30, and not surprisingly R has printed that out as part of its response. But it's also printed out this `[1]` part, which probably doesn't make a lot of sense to you right now. You're going to see that a lot. I'll talk about what this means in a bit more detail later on, but for now you can think of `[1] 30` as if R were saying "the answer to the 1st question you asked is 30". That's not quite the truth, but it's close enough for now. And in any case it's not really very interesting at the moment: we only asked R to calculate one thing, so obviously there's only one answer printed on the screen. Later on this will change, and the `[1]` part will start to make a bit more sense. For now, I just don't want you to get confused or concerned by it.

3.2.1 An important digression about formatting

Now that I've taught you these rules I'm going to change them pretty much immediately. That is because I want you to be able to copy code from the book directly into R if you want to test things or conduct your own analyses. However, if you copy this kind of code (that shows the command prompt and the results) directly into R you will get an error

⁶Seriously. If you're in a position to do so, open up R and start typing. The simple act of typing it rather than "just reading" makes a big difference. It makes the concepts more concrete, and it ties the abstract ideas (programming and statistics) to the actual context in which you need to use them. Statistics is something you *do*, not just something you read about in a textbook.

```
> 10 + 20
[1] 30
```

```
## Error: <text>:1:1: unexpected '>
## 1: >
##      ^
```

So instead, I'm going to provide code in a slightly different format so that it looks like this...

```
10 + 20
```

```
## [1] 30
```

There are two main differences.

- In your console, you type after the `>`, but from now on I won't show the command prompt in the book.
- In the book, output is commented out with `##`, in your console it appears directly after your code.

These two differences mean that if you're working with an electronic version of the book, you can easily copy code out of the book and into the console.

So for example if you copied the two lines of code from the book you'd get this

```
10 + 20
```

```
## [1] 30
```

```
## [1] 30
```

3.2.2 Be very careful to avoid typos

Before we go on to talk about other types of calculations that we can do with R, there's a few other things I want to point out. The first thing is that, while R is good software, it's still software. It's pretty stupid, and because it's stupid it can't handle typos. It takes it on faith that you meant to type *exactly* what you did type. For example, suppose that you forgot to hit the shift key when trying to type `+`, and as a result your command ended up being `10 = 20` rather than `10 + 20`. Here's what happens:

```
10 = 20
```

```
## Error in 10 = 20: invalid (do_set) left-hand side to assignment
```

What's happened here is that R has attempted to interpret `10 = 20` as a command, and spits out an error message because the command doesn't make any sense to it. When a *human* looks at this, and then looks down at his or her keyboard and sees that `+` and `=` are on the same key, it's pretty obvious that the command was a typo. But R doesn't know this, so it gets upset. And, if you look at it from its perspective, this makes sense. All that R "knows" is that `10` is a legitimate number, `20` is a legitimate number, and `=` is a legitimate part of the language too. In other words, from its perspective this really does look like the user meant to type `10 = 20`, since all the individual parts of that statement are legitimate and it's too stupid to realise that this is probably a typo. Therefore, R takes it on faith that this is exactly what you meant... it only

“discovers” that the command is nonsense when it tries to follow your instructions, typo and all. And then it whinges, and spits out an error.

Even more subtle is the fact that some typos won’t produce errors at all, because they happen to correspond to “well-formed” R commands. For instance, suppose that not only did I forget to hit the shift key when trying to type `10 + 20`, I also managed to press the key next to one I meant do. The resulting typo would produce the command `10 - 20`. Clearly, R has no way of knowing that you meant to *add* 20 to 10, not *subtract* 20 from 10, so what happens this time is this:

```
10 - 20
```

```
## [1] -10
```

In this case, R produces the right answer, but to the the wrong question.

To some extent, I’m stating the obvious here, but it’s important. The people who wrote R are smart. You, the user, are smart. But R itself is dumb. And because it’s dumb, it has to be mindlessly obedient. It does *exactly* what you ask it to do. There is no equivalent to “autocorrect” in R, and for good reason. When doing advanced stuff – and even the simplest of statistics is pretty advanced in a lot of ways – it’s dangerous to let a mindless automaton like R try to overrule the human user. But because of this, it’s your responsibility to be careful. Always make sure you type *exactly what you mean*. When dealing with computers, it’s not enough to type “approximately” the right thing. In general, you absolutely *must* be precise in what you say to R ... like all machines it is too stupid to be anything other than absurdly literal in its interpretation.

3.2.3 R is (a bit) flexible with spacing

Of course, now that I’ve been so uptight about the importance of always being precise, I should point out that there are some exceptions. Or, more accurately, there are some situations in which R does show a bit more flexibility than my previous description suggests. The first thing R is smart enough to do is ignore redundant spacing. What I mean by this is that, when I typed `10 + 20` before, I could equally have done this

```
10 + 20
```

```
## [1] 30
```

or this

```
10+20
```

```
## [1] 30
```

and I would get exactly the same answer. However, that doesn’t mean that you can insert spaces in any old place. When we looked at the startup documentation in Section 3.1.5 it suggested that you could type `citation()` to get some information about how to cite R. If I do so...

```
citation()
```

```
##
## To cite R in publications use:
##
##   R Core Team (2018). R: A language and environment for
```

```

##   statistical computing. R Foundation for Statistical Computing,
##   Vienna, Austria. URL https://www.R-project.org/.
##
## A BibTeX entry for LaTeX users is
##
## @Manual{,
##   title = {R: A Language and Environment for Statistical Computing},
##   author = {{R Core Team}},
##   organization = {R Foundation for Statistical Computing},
##   address = {Vienna, Austria},
##   year = {2018},
##   url = {https://www.R-project.org/},
## }
##
## We have invested a lot of time and effort in creating R, please
## cite it when using it for data analysis. See also
## 'citation("pkgname")' for citing R packages.

```

... it tells me to cite the R manual (R Core Team, 2013). Let's see what happens when I try changing the spacing. If I insert spaces in between the word and the parentheses, or inside the parentheses themselves, then all is well. That is, either of these two commands

```
citation ()
```

```
citation( )
```

will produce exactly the same response. However, what I can't do is insert spaces in the middle of the word. If I try to do this, R gets upset:

```
citat ion()
```

```

## Error: <text>:1:7: unexpected symbol
## 1: citat ion
##           ^

```

Throughout this book I'll vary the way I use spacing a little bit, just to give you a feel for the different ways in which spacing can be used. I'll try not to do it too much though, since it's generally considered to be good practice to be consistent in how you format your commands.

3.2.4 R can sometimes tell that you're not finished yet (but not often)

One more thing I should point out. If you hit enter in a situation where it's "obvious" to R that you haven't actually finished typing the command, R is just smart enough to keep waiting. For example, if you type 10 + and then press enter, even R is smart enough to realise that you probably wanted to type in another number. So here's what happens (for illustrative purposes I'm breaking my own code formatting rules in this section):

```
> 10+
+
```

and there's a blinking cursor next to the plus sign. What this means is that R is still waiting for you to finish. It “thinks” you’re still typing your command, so it hasn’t tried to execute it yet. In other words, this plus sign is actually another command prompt. It’s different from the usual one (i.e., the > symbol) to remind you that R is going to “add” whatever you type now to what you typed last time. For example, if I then go on to type 3 and hit enter, what I get is this:

```
> 10 +
+ 20
[1] 30
```

And as far as R is concerned, this is *exactly* the same as if you had typed 10 + 20. Similarly, consider the `citation()` command that we talked about in the previous section. Suppose you hit enter after typing `citation()`. Once again, R is smart enough to realise that there must be more coming – since you need to add the) character – so it waits. I can even hit enter several times and it will keep waiting:

```
> citation(
+
+
+ )
```

I’ll make use of this a lot in this book. A lot of the commands that we’ll have to type are pretty long, and they’re visually a bit easier to read if I break it up over several lines. If you start doing this yourself, you’ll eventually get yourself in trouble (it happens to us all). Maybe you start typing a command, and then you realise you’ve screwed up. For example,

```
> citblation(
+
+
```

You’d probably prefer R not to try running this command, right? If you want to get out of this situation, just hit the ‘escape’ key.⁷ R will return you to the normal command prompt (i.e. >) *without* attempting to execute the botched command.

That being said, it’s not often the case that R is smart enough to tell that there’s more coming. For instance, in the same way that I can’t add a space in the middle of a word, I can’t hit enter in the middle of a word either. If I hit enter after typing `citat` I get an error, because R thinks I’m interested in an “object” called `citat` and can’t find it:

```
> citat
Error: object 'citat' not found
```

What about if I typed `citation` and hit enter? In this case we get something very odd, something that we definitely *don’t* want, at least at this stage. Here’s what happens:

```
> [citation]
function (package = "base", lib.loc = NULL, auto = NULL)
{
  dir <- system.file(package = package, lib.loc = lib.loc)
  if (dir == "") 
    stop(gettextf("package '%s' not found", package), domain = NA)
```

BLAH BLAH BLAH

⁷If you’re running R from the terminal rather than from RStudio, escape doesn’t work: use CTRL-C instead.

Table 3.1: Basic arithmetic operations in R. These five operators are used very frequently throughout the text, so it's important to be familiar with them at the outset.

operation	operator	example input	example output
addition	'+'	10 + 2	12
subtraction	'-'	9 - 3	6
multiplication	'*'	5 * 5	25
division	'/'	10 / 3	3
power	'^'	5 ^ 2	25

where the BLAH BLAH BLAH goes on for rather a long time, and you don't know enough R yet to understand what all this gibberish actually means (of course, it doesn't actually say BLAH BLAH BLAH - it says some other things we don't understand or need to know that I've edited for length) This incomprehensible output can be quite intimidating to novice users, and unfortunately it's very easy to forget to type the parentheses; so almost certainly you'll do this by accident. Do not panic when this happens. Simply ignore the gibberish. As you become more experienced this gibberish will start to make sense, and you'll find it quite handy to print this stuff out.⁸ But for now just try to remember to add the parentheses when typing your commands.

3.3 Doing simple calculations with R

Okay, now that we've discussed some of the tedious details associated with typing R commands, let's get back to learning how to use the most powerful piece of statistical software in the world as a \$2 calculator. So far, all we know how to do is addition. Clearly, a calculator that only did addition would be a bit stupid, so I should tell you about how to perform other simple calculations using R. But first, some more terminology. Addition is an example of an "operation" that you can perform (specifically, an arithmetic operation), and the **operator** that performs it is `+`. To people with a programming or mathematics background, this terminology probably feels pretty natural, but to other people it might feel like I'm trying to make something very simple (addition) sound more complicated than it is (by calling it an arithmetic operation). To some extent, that's true: if addition was the only operation that we were interested in, it'd be a bit silly to introduce all this extra terminology. However, as we go along, we'll start using more and more different kinds of operations, so it's probably a good idea to get the language straight now, while we're still talking about very familiar concepts like addition!

3.3.1 Adding, subtracting, multiplying and dividing

So, now that we have the terminology, let's learn how to perform some arithmetic operations in R. To that end, Table 3.1 lists the operators that correspond to the basic arithmetic we learned in primary school: addition, subtraction, multiplication and division.

As you can see, R uses fairly standard symbols to denote each of the different operations you might want to perform: addition is done using the `+` operator, subtraction is performed by the `-` operator, and so on. So if I wanted to find out what 57 times 61 is (and who wouldn't?), I can use R instead of a calculator, like so:

57 * 61

```
## [1] 3477
```

So that's handy.

⁸For advanced users: yes, as you've probably guessed, R is printing out the source code for the function.

3.3.2 Taking powers

The first four operations listed in Table 3.1 are things we all learned in primary school, but they aren't the only arithmetic operations built into R. There are three other arithmetic operations that I should probably mention: taking powers, doing integer division, and calculating a modulus. Of the three, the only one that is of any real importance for the purposes of this book is taking powers, so I'll discuss that one here: the other two are discussed in Chapter 7.

For those of you who can still remember your high school maths, this should be familiar. But for some people high school maths was a long time ago, and others of us didn't listen very hard in high school. It's not complicated. As I'm sure everyone will probably remember the moment they read this, the act of multiplying a number x by itself n times is called "raising x to the n -th power". Mathematically, this is written as x^n . Some values of n have special names: in particular x^2 is called x -squared, and x^3 is called x -cubed. So, the 4th power of 5 is calculated like this:

$$5^4 = 5 \times 5 \times 5 \times 5$$

One way that we could calculate 5^4 in R would be to type in the complete multiplication as it is shown in the equation above. That is, we could do this

```
5 * 5 * 5 * 5
```

```
## [1] 625
```

but it does seem a bit tedious. It would be very annoying indeed if you wanted to calculate 5^{15} , since the command would end up being quite long. Therefore, to make our lives easier, we use the power operator instead. When we do that, our command to calculate 5^4 goes like this:

```
5 ^ 4
```

```
## [1] 625
```

Much easier.

3.3.3 Doing calculations in the right order

Okay. At this point, you know how to take one of the most powerful pieces of statistical software in the world, and use it as a \$2 calculator. And as a bonus, you've learned a few very basic programming concepts. That's not nothing (you could argue that you've just saved yourself \$2) but on the other hand, it's not very much either. In order to use R more effectively, we need to introduce more programming concepts.

In most situations where you would want to use a calculator, you might want to do multiple calculations. R lets you do this, just by typing in longer commands.⁹ In fact, we've already seen an example of this earlier, when I typed in $5 * 5 * 5 * 5$. However, let's try a slightly different example:

```
1 + 2 * 4
```

```
## [1] 9
```

⁹If you're reading this with R open, a good learning trick is to try typing in a few different variations on what I've done here. If you experiment with your commands, you'll quickly learn what works and what doesn't

Clearly, this isn't a problem for R either. However, it's worth stopping for a second, and thinking about what R just did. Clearly, since it gave us an answer of 9 it must have multiplied $2 * 4$ (to get an interim answer of 8) and then added 1 to that. But, suppose it had decided to just go from left to right: if R had decided instead to add $1+2$ (to get an interim answer of 3) and then multiplied by 4, it would have come up with an answer of 12.

To answer this, you need to know the *order of operations* that R uses. If you remember back to your high school maths classes, it's actually the same order that you got taught when you were at school: the “**BEDMAS**” order.¹⁰ That is, first calculate things inside **B**rackets (), then calculate **E**xponents ^, then **D**ivision / and **M**ultiplication *, then **A**ddition + and **S**ubtraction -. So, to continue the example above, if we want to force R to calculate the $1+2$ part before the multiplication, all we would have to do is enclose it in brackets:

```
(1 + 2) * 4
```

```
## [1] 12
```

This is a fairly useful thing to be able to do. The only other thing I should point out about order of operations is what to expect when you have two operations that have the same priority: that is, how does R resolve ties? For instance, multiplication and division are actually the same priority, but what should we expect when we give R a problem like $4 / 2 * 3$ to solve? If it evaluates the multiplication first and then the division, it would calculate a value of two-thirds. But if it evaluates the division first it calculates a value of 6. The answer, in this case, is that R goes from *left to right*, so in this case the division step would come first:

```
4 / 2 * 3
```

```
## [1] 6
```

All of the above being said, it's helpful to remember that *brackets always come first*. So, if you're ever unsure about what order R will do things in, an easy solution is to enclose the thing *you* want it to do first in brackets. There's nothing stopping you from typing $(4 / 2) * 3$. By enclosing the division in brackets we make it clear which thing is supposed to happen first. In this instance you wouldn't have needed to, since R would have done the division first anyway, but when you're first starting out it's better to make sure R does what you want!

3.4 Storing a number as a variable

One of the most important things to be able to do in R (or any programming language, for that matter) is to store information in *variables*. Variables in R aren't exactly the same thing as the variables we talked about in the last chapter on research methods, but they are similar. At a conceptual level you can think of a variable as *label* for a certain piece of information, or even several different pieces of information. When doing statistical analysis in R all of your data (the variables you measured in your study) will be stored as variables in R, but as well see later in the book you'll find that you end up creating variables for other things too. However, before we delve into all the messy details of data sets and statistical analysis, let's look at the very basics for how we create variables and work with them.

¹⁰For advanced users: if you want a table showing the complete order of operator precedence in R, type `?Syntax`. I haven't included it in this book since there are quite a few different operators, and we don't need that much detail. Besides, in practice most people seem to figure it out from seeing examples: until writing this book I never looked at the formal statement of operator precedence for any language I ever coded in, and never ran into any difficulties.

3.4.1 Variable assignment using `<-` and `->`

Since we've been working with numbers so far, let's start by creating variables to store our numbers. And since most people like concrete examples, let's invent one. Suppose I'm trying to calculate how much money I'm going to make from this book. There's several different numbers I might want to store. Firstly, I need to figure out how many copies I'll sell. This isn't exactly *Harry Potter*, so let's assume I'm only going to sell one copy per student in my class. That's 350 sales, so let's create a variable called `sales`. What I want to do is assign a *value* to my variable `sales`, and that value should be 350. We do this by using the *assignment operator*, which is `<-`. Here's how we do it:

```
sales <- 350
```

When you hit enter, R doesn't print out any output.¹¹ It just gives you another command prompt. However, behind the scenes R has created a variable called `sales` and given it a value of 350. You can check that this has happened by asking R to print the variable on screen. And the simplest way to do *that* is to type the name of the variable and hit enter¹².

```
sales
```

```
## [1] 350
```

So that's nice to know. Anytime you can't remember what R has got stored in a particular variable, you can just type the name of the variable and hit enter.

Okay, so now we know how to assign variables. Actually, there's a bit more you should know. Firstly, one of the curious features of R is that there are several different ways of making assignments. In addition to the `<-` operator, we can also use `->` and `=`, and it's pretty important to understand the differences between them.¹³ Let's start by considering `->`, since that's the easy one (we'll discuss the use of `=` in Section 3.5.1). As you might expect from just looking at the symbol, it's almost identical to `<-`. It's just that the arrow (i.e., the assignment) goes from left to right. So if I wanted to define my `sales` variable using `->`, I would write it like this:

```
350 -> sales
```

This has the same effect: and it *still* means that I'm only going to sell 350 copies. Sigh. Apart from this superficial difference, `<-` and `->` are identical. In fact, as far as R is concerned, they're actually the same operator, just in a "left form" and a "right form".¹⁴

3.4.2 Doing calculations using variables

Okay, let's get back to my original story. In my quest to become rich, I've written this textbook. To figure out how good a strategy is, I've started creating some variables in R. In addition to defining a `sales` variable that counts the number of copies I'm going to sell, I can also create a variable called `royalty`, indicating how much money I get per copy. Let's say that my royalties are about \$7 per book:

¹¹If you are using RStudio, and the "environment" panel (formerly known as the "workspace" panel) is visible when you typed the command, then you probably saw something happening there. That's to be expected, and is quite helpful. However, there's two things to note here (1) I haven't yet explained what that panel does, so for now just ignore it, and (2) this is one of the helpful things RStudio does, not a part of R itself.

¹²As we'll discuss later, by doing this we are implicitly using the `print()` function

¹³Actually, in keeping with the R tradition of providing you with a billion different screwdrivers (even when you're actually looking for a hammer) these aren't the only options. There's also `theassign()` function, and the `<<-` and `->>` operators. However, we won't be using these at all in this book.

¹⁴A quick reminder: when using operators like `<-` and `->` that span multiple characters, you can't insert spaces in the middle. That is, if you type `- >` or `< -`, R will interpret your command the wrong way. And I will cry.

```
sales <- 350
royalty <- 7
```

The nice thing about variables (in fact, the whole point of having variables) is that we can do anything with a variable that we ought to be able to do with the information that it stores. That is, since R allows me to multiply 350 by 7

```
350 * 7
```

```
## [1] 2450
```

it also allows me to multiply `sales` by `royalty`

```
sales * royalty
```

```
## [1] 2450
```

As far as R is concerned, the `sales * royalty` command is the same as the `350 * 7` command. Not surprisingly, I can assign the output of this calculation to a new variable, which I'll call `revenue`. And when we do this, the new variable `revenue` gets the value 2450. So let's do that, and then get R to print out the value of `revenue` so that we can verify that it's done what we asked:

```
revenue <- sales * royalty
revenue
```

```
## [1] 2450
```

That's fairly straightforward. A slightly more subtle thing we can do is reassign the value of my variable, based on its current value. For instance, suppose that one of my students (no doubt under the influence of psychotropic drugs) loves the book so much that he or she donates me an extra \$550. The simplest way to capture this is by a command like this:

```
revenue <- revenue + 550
revenue
```

```
## [1] 3000
```

In this calculation, R has taken the old value of `revenue` (i.e., 2450) and added 550 to that value, producing a value of 3000. This new value is assigned to the `revenue` variable, overwriting its previous value. In any case, we now know that I'm expecting to make \$3000 off this. Pretty sweet, I thinks to myself. Or at least, that's what I thinks until I do a few more calculation and work out what the implied hourly wage I'm making off this looks like.

3.4.3 Rules and conventions for naming variables

In the examples that we've seen so far, my variable names (`sales` and `revenue`) have just been English-language words written using lowercase letters. However, R allows a lot more flexibility when it comes to naming your variables, as the following list of rules¹⁵ illustrates:

¹⁵Actually, you can override any of these rules if you want to, and quite easily. All you have to do is add quote marks or backticks around your non-standard variable name. For instance ``my sales` <- 350` would work just fine, but it's almost never a good idea to do this.

- Variable names can only use the upper case alphabetic characters A-Z as well as the lower case characters a-z. You can also include numeric characters 0-9 in the variable name, as well as the period . or underscore _ character. In other words, you can use `SaL.e_s` as a variable name (though I can't think why you would want to), but you can't use `Sales?`.
- Variable names cannot include spaces: therefore `my sales` is not a valid name, but `my.sales` is.
- Variable names are case sensitive: that is, `Sales` and `sales` are *different* variable names.
- Variable names must start with a letter or a period. You can't use something like `_sales` or `1sales` as a variable name. You can use `.sales` as a variable name if you want, but it's not usually a good idea. By convention, variables starting with a . are used for special purposes, so you should avoid doing so.
- Variable names cannot be one of the reserved keywords. These are special names that R needs to keep "safe" from us mere users, so you can't use them as the names of variables. The keywords are: `if`, `else`, `repeat`, `while`, `function`, `for`, `in`, `next`, `break`, `TRUE`, `FALSE`, `NULL`, `Inf`, `NaN`, `NA`, `Rtextverb#NA_integer_#`, `Rtextverb#NA_real_#`, `NA_complex_`, and finally, `NA_character_`. Don't feel especially obliged to memorise these: if you make a mistake and try to use one of the keywords as a variable name, R will complain about it like the whiny little automaton it is.

In addition to those rules that R enforces, there are some informal conventions that people tend to follow when naming variables. One of them you've already seen: i.e., don't use variables that start with a period. But there are several others. You aren't obliged to follow these conventions, and there are many situations in which it's advisable to ignore them, but it's generally a good idea to follow them when you can:

- Use informative variable names. As a general rule, using meaningful names like `sales` and `revenue` is preferred over arbitrary ones like `variable1` and `variable2`. Otherwise it's very hard to remember what the contents of different variables are, and it becomes hard to understand what your commands actually do.
- Use short variable names. Typing is a pain and no-one likes doing it. So we much prefer to use a name like `sales` over a name like `sales.for.this.book.that.you.are.reading`. Obviously there's a bit of a tension between using informative names (which tend to be long) and using short names (which tend to be meaningless), so use a bit of common sense when trading off these two conventions.
- Use one of the conventional naming styles for multi-word variable names. Suppose I want to name a variable that stores "my new salary". Obviously I can't include spaces in the variable name, so how should I do this? There are three different conventions that you sometimes see R users employing. Firstly, you can separate the words using periods, which would give you `my.new.salary` as the variable name. Alternatively, you could separate words using underscores, as in `my_new_salary`. Finally, you could use capital letters at the beginning of each word (except the first one), which gives you `myNewSalary` as the variable name. I don't think there's any strong reason to prefer one over the other,¹⁶ but it's important to be consistent.

3.5 Using functions to do calculations

The symbols `+`, `-`, `*` and so on are examples of operators. As we've seen, you can do quite a lot of calculations just by using these operators. However, in order to do more advanced calculations (and later on, to do actual statistics), you're going to need to start using *functions*.¹⁷ Thus `10+20` is equivalent to the function call `+(20, 30)`. Not surprisingly, no-one ever uses this version. Because that would be stupid.] I'll talk in more detail about functions and how they work in Section 8.4, but for now let's just dive in and use a few. To get started, suppose I wanted to take the square root of 225. The square root, in case your high school maths is a bit rusty, is just the opposite of squaring a number. So, for instance, since "5 squared is 25" I can say that "5 is the square root of 25". The usual notation for this is

¹⁶For very advanced users: there is one exception to this. If you're naming a function, don't use . in the name unless you are intending to make use of the S3 object oriented programming system in R. If you don't know what S3 is, then you definitely don't want to be using it! For function naming, there's been a trend among R users to prefer `myFunctionName`.

¹⁷A side note for students with a programming background. Technically speaking, operators *are* functions in R: the addition operator + is actually a convenient way of calling the addition function `+()`

$$\sqrt{25} = 5$$

though sometimes you'll also see it written like this $25^{0.5} = 5$. This second way of writing it is kind of useful to "remind" you of the mathematical fact that "square root of x " is actually the same as "raising x to the power of 0.5". Personally, I've never found this to be terribly meaningful psychologically, though I have to admit it's quite convenient mathematically. Anyway, it's not important. What is important is that you remember what a square root is, since we're going to need it later on.

To calculate the square root of 25, I can do it in my head pretty easily, since I memorised my multiplication tables when I was a kid. It gets harder when the numbers get bigger, and pretty much impossible if they're not whole numbers. This is where something like R comes in very handy. Let's say I wanted to calculate $\sqrt{225}$, the square root of 225. There's two ways I could do this using R. Firstly, since the square root of 225 is the same thing as raising 225 to the power of 0.5, I could use the power operator $^$, just like we did earlier:

```
225 ^ 0.5
```

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

However, there's a second way that we can do this, since R also provides a *square root function*, `sqrt()`. To calculate the square root of 225 using this function, what I do is insert the number 225 in the parentheses. That is, the command I type is this:

```
sqrt( 225 )
```

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

and as you might expect from our previous discussion, the spaces in between the parentheses are purely cosmetic. I could have typed `sqrt(225)` or `sqrt(225)` and gotten the same result. When we use a function to do something, we generally refer to this as *calling* the function, and the values that we type into the function (there can be more than one) are referred to as the *arguments* of that function.

Obviously, the `sqrt()` function doesn't really give us any new functionality, since we already knew how to do square root calculations by using the power operator $^$, though I do think it looks nicer when we use `sqrt()`. However, there are lots of other functions in R: in fact, almost everything of interest that I'll talk about in this book is an R function of some kind. For example, one function that we will need to use in this book is the *absolute value function*. Compared to the square root function, it's extremely simple: it just converts negative numbers to positive numbers, and leaves positive numbers alone. Mathematically, the absolute value of x is written $|x|$ or sometimes `abs(x)`. Calculating absolute values in R is pretty easy, since R provides the `abs()` function that you can use for this purpose. When you feed it a positive number...

```
abs( 21 )
```

```
## [1] 21
```

the absolute value function does nothing to it at all. But when you feed it a negative number, it spits out the positive version of the same number, like this:

```
abs( -13 )
```

```
## [1] 13
```

In all honesty, there's nothing that the absolute value function does that you couldn't do just by looking at the number and erasing the minus sign if there is one. However, there's a few places later in the book where we have to use absolute values, so I thought it might be a good idea to explain the meaning of the term early on.

Before moving on, it's worth noting that – in the same way that R allows us to put multiple operations together into a longer command, like `1 + 2*4` for instance – it also lets us put functions together and even combine functions with operators if we so desire. For example, the following is a perfectly legitimate command:

```
sqrt( 1 + abs(-8) )
```

```
## [1] 3
```

When R executes this command, starts out by calculating the value of `abs(-8)`, which produces an intermediate value of 8. Having done so, the command simplifies to `sqrt(1 + 8)`. To solve the square root¹⁸ it first needs to add `1 + 8` to get 9, at which point it evaluates `sqrt(9)`, and so it finally outputs a value of 3.

3.5.1 Function arguments, their names and their defaults

There's two more fairly important things that you need to understand about how functions work in R, and that's the use of "named" arguments, and default values" for arguments. Not surprisingly, that's not to say that this is the last we'll hear about how functions work, but they are the last things we desperately need to discuss in order to get you started. To understand what these two concepts are all about, I'll introduce another function. The `round()` function can be used to round some value to the nearest whole number. For example, I could type this:

```
round( 3.1415 )
```

```
## [1] 3
```

Pretty straightforward, really. However, suppose I only wanted to round it to two decimal places: that is, I want to get 3.14 as the output. The `round()` function supports this, by allowing you to input a second argument to the function that specifies the number of decimal places that you want to round the number to. In other words, I could do this:

```
round( 3.14165, 2 )
```

```
## [1] 3.14
```

What's happening here is that I've specified *two* arguments: the first argument is the number that needs to be rounded (i.e., `3.1415`), the second argument is the number of decimal places that it should be rounded to (i.e., `2`), and the two arguments are separated by a comma. In this simple example, it's quite easy to remember which one argument comes first and which one comes second, but for more complicated functions this is not easy. Fortunately, most R functions make use of **argument names**. For the `round()` function, for example the number that needs to be rounded is specified using the `x` argument, and the number of decimal points that you want it rounded to is specified using the `digits` argument. Because we have these names available to us, we can specify the arguments to the function by name. We do so like this:

¹⁸A note for the mathematically inclined: R does support complex numbers, but unless you explicitly specify that you want them it assumes all calculations must be real valued. By default, the square root of a negative number is treated as undefined: `sqrt(-9)` will produce `NaN` (not a number) as its output. To get complex numbers, you would type `sqrt(-9+0i)` and R would now return `0+3i`. However, since we won't have any need for complex numbers in this book, I won't refer to them again.

```
round( x = 3.1415, digits = 2 )
## [1] 3.14
```

Notice that this is kind of similar in spirit to variable assignment (Section ??(assign), except that I used `=` here, rather than `<-`. In both cases we're specifying specific values to be associated with a label. However, there are some differences between what I was doing earlier on when creating variables, and what I'm doing here when specifying arguments, and so as a consequence it's important that you use `=` in this context.

As you can see, specifying the arguments by name involves a lot more typing, but it's also a lot easier to read. Because of this, the commands in this book will usually specify arguments by name,¹⁹ since that makes it clearer to you what I'm doing. However, one important thing to note is that when specifying the arguments using their names, it doesn't matter what order you type them in. But if you don't use the argument names, then you have to input the arguments in the correct order. In other words, these three commands all produce the same output...

```
round( 3.14165, 2 )
## [1] 3.14
round( x = 3.1415, digits = 2 )
## [1] 3.14
round( digits = 2, x = 3.1415 )
```

but this one does not...

```
round( 2, 3.14165 )
## [1] 2
```

How do you find out what the correct order is? There's a few different ways, but the easiest one is to look at the help documentation for the function (see Section 4.12). However, if you're ever unsure, it's probably best to actually type in the argument name.

Okay, so that's the first thing I said you'd need to know: argument names. The second thing you need to know about is default values. Notice that the first time I called the `round()` function I didn't actually specify the `digits` argument at all, and yet R somehow knew that this meant it should round to the nearest whole number. How did that happen? The answer is that the `digits` argument has a *default value* of 0, meaning that if you decide not to specify a value for `digits` then R will act as if you had typed `digits = 0`. This is quite handy: the vast majority of the time when you want to round a number you want to round it to the nearest whole number, and it would be pretty annoying to have to specify the `digits` argument every single time. On the other hand, sometimes you actually do want to round to something other than the nearest whole number, and it would be even more annoying if R didn't allow this! Thus, by having `digits = 0` as the default value, we get the best of both worlds.

¹⁹The two functions discussed previously, `sqrt()` and `abs()`, both only have a single argument, `x`. So I could have typed something like `sqrt(x = 225)` or `abs(x = -13)` earlier. The fact that all these functions use `x` as the name of the argument that corresponds the “main” variable that you’re working with is no coincidence. That’s a fairly widely used convention. Quite often, the writers of R functions will try to use conventional names like this to make your life easier. Or at least that’s the theory. In practice it doesn’t always work as well as you’d hope.

3.6 Letting RStudio help you with your commands

Time for a bit of a digression. At this stage you know how to type in basic commands, including how to use R functions. And it's probably beginning to dawn on you that there are a *lot* of R functions, all of which have their own arguments. You're probably also worried that you're going to have to remember all of them! Thankfully, it's not that bad. In fact, very few data analysts bother to try to remember all the commands. What they really do is use tricks to make their lives easier. The first (and arguably most important one) is to use the internet. If you don't know how a particular R function works, Google it. Second, you can look up the R help documentation. I'll talk more about these two tricks in Section 4.12. But right now I want to call your attention to a couple of simple tricks that RStudio makes available to you.

3.6.1 Autocomplete using “tab”

The first thing I want to call your attention to is the *autocomplete* ability in RStudio.²⁰

Let's stick to our example above and assume that what you want to do is to round a number. This time around, start typing the name of the function that you want, and then hit the “tab” key. RStudio will then display a little window like the one shown in Figure 3.2. In this figure, I've typed the letters `ro` at the command line, and then hit tab. The window has two panels. On the left, there's a list of variables and functions that start with the letters that I've typed shown in black text, and some grey text that tells you where that variable/function is stored. Ignore the grey text for now: it won't make much sense to you until we've talked about packages in Section 4.2. In Figure 3.2 you can see that there's quite a few things that start with the letters `ro`: there's something called `rock`, something called `round`, something called `round.Date` and so on. The one we want is `round`, but if you're typing this yourself you'll notice that when you hit the tab key the window pops up with the top entry (i.e., `rock`) highlighted. You can use the up and down arrow keys to select the one that you want. Or, if none of the options look right to you, you can hit the escape key (“esc”) or the left arrow key to make the window go away.

In our case, the thing we want is the `round` option, so we'll select that. When you do this, you'll see that the panel on the right changes. Previously, it had been telling us something about the `rock` data set (i.e., “Measurements on 48 rock samples...”) that is distributed as part of R. But when we select `round`, it displays information about the `round()` function, exactly as it is shown in Figure 3.2. This display is really handy. The very first thing it says is `round(x, digits = 0)`: what this is telling you is that the `round()` function has two arguments. The first argument is called `x`, and it doesn't have a default value. The second argument is `digits`, and it has a default value of 0. In a lot of situations, that's all the information you need. But RStudio goes a bit further, and provides some additional information about the function underneath. Sometimes that additional information is very helpful, sometimes it's not: RStudio pulls that text from the R help documentation, and my experience is that the helpfulness of that documentation varies wildly. Anyway, if you've decided that `round()` is the function that you want to use, you can hit the right arrow or the enter key, and RStudio will finish typing the rest of the function name for you.

Start typing the name of a function or a variable, and hit the “tab” key. RStudio brings up a little dialog box like this one that lets you select the one you want, and even prints out a little information about it.

The RStudio autocomplete tool works slightly differently if you've already got the name of the function typed and you're now trying to type the arguments. For instance, suppose I've typed `round(` into the console, and *then* I hit tab. RStudio is smart enough to recognise that I already know the name of the function that I want, because I've already typed it! Instead, it figures that what I'm interested in is the *arguments* to that function. So that's what pops up in the little window. You can see this in Figure ???. Again, the window has two panels, and you can interact with this window in exactly the same way that you did with the window shown in 3.2. On the left hand panel, you can see a list of the argument names. On the right hand side, it displays some information about what the selected argument does.

²⁰For advanced users: obviously, this isn't just an RStudio thing. If you're running R in a terminal window, tab autocomplete still works, and does so in exactly the way you'd expect. It's not as visually pretty as the RStudio version, of course, and lacks some of the cooler features that RStudio provides. I don't bother to document that here: my assumption is that if you are

The screenshot shows an RStudio session with the following code:

```
>  
>  
>  
> rock {datasets}  
> round {base} <-- Selected  
> round.Date {base}  
> round.POSIXt {base}  
> row {base}  
> row.names {base}  
> row.names data frame {base}  
> ro
```

A tooltip on the right side provides information about the selected function:

round(x, digits)
ceiling takes a single numeric vector containing corresponding elements
floor takes a single vector containing ..
Press F1 for additional help

Figure 3.2: Start typing the name of a function or a variable, and hit the "tab" key. RStudio brings up a little dialog box like this one that lets you select the one you want, and even prints out a little information about it.

```
>
>
>          x=
> digits=      a numeric vector. Or, for round and
> ...=
>
>
>
>
>
>
> round( |
```

The screenshot shows an RStudio session with the following code:

```
>
>
>          x=
> digits=      a numeric vector. Or, for round and
> ...=
>
>
>
>
>
>
> round( |
```

A tooltip window is open over the word "digits". The tooltip has a blue header bar with the text "x" and a white body containing the argument information: "digits=" followed by the description "a numeric vector. Or, for round and". At the bottom of the tooltip is a grey bar with the text "Press F1 for additional help".

Figure 3.3: If you've typed the name of a function already along with the left parenthesis and then hit the "tab" key, RStudio brings up a different window to the one shown above. This one lists all the arguments to the function on the left, and information about each argument on the right.

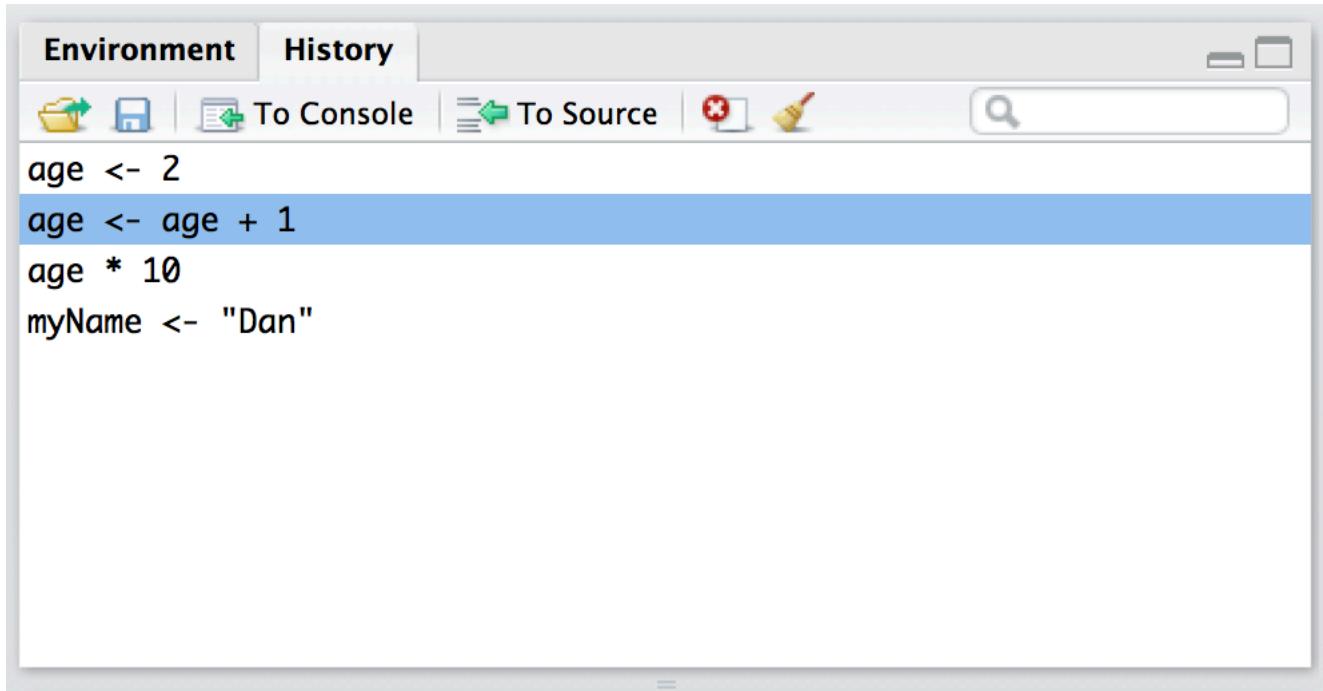


Figure 3.4: The history panel is located in the top right hand side of the RStudio window. Click on the word "History" and it displays this panel.

If you've typed the name of a function already along with the left parenthesis and then hit the "tab" key, RStudio brings up a different window to the one shown in Figure 3.2. This one lists all the arguments to the function on the left, and information about each argument on the right.

3.6.2 Browsing your command history

One thing that R does automatically is keep track of your “command history”. That is, it remembers all the commands that you’ve previously typed. You can access this history in a few different ways. The simplest way is to use the up and down arrow keys. If you hit the up key, the R console will show you the most recent command that you’ve typed. Hit it again, and it will show you the command before that. If you want the text on the screen to go away, hit escape²¹ Using the up and down keys can be really handy if you’ve typed a long command that had one typo in it. Rather than having to type it all again from scratch, you can use the up key to bring up the command and fix it.

The second way to get access to your command history is to look at the history panel in RStudio. On the upper right hand side of the RStudio window you’ll see a tab labelled “History”. Click on that, and you’ll see a list of all your recent commands displayed in that panel: it should look something like Figure ???. If you double click on one of the commands, it will be copied to the R console. (You can achieve the same result by selecting the command you want with the mouse and then clicking the “To Console” button).²²

²¹running R in the terminal then you’re already familiar with using tab autocomplete.

²²Incidentally, that always works: if you’ve started typing a command and you want to clear it and start again, hit escape.

²²Another method is to start typing some text and then hit the Control key and the up arrow together (on Windows or Linux) or the Command key and the up arrow together (on a Mac). This will bring up a window showing all your recent commands that started with the same text as what you’ve currently typed. That can come in quite handy sometimes.

3.7 Storing many numbers as a vector

At this point we've covered functions in enough detail to get us safely through the next couple of chapters (with one small exception: see Section 4.11, so let's return to our discussion of variables. When I introduced variables in Section 3.4 I showed you how we can use variables to store a single number. In this section, we'll extend this idea and look at how to store multiple numbers within the one variable. In R, the name for a variable that can store multiple values is a *vector*. So let's create one.

3.7.1 Creating a vector

Let's stick to my silly "get rich quick by textbook writing" example. Suppose the textbook company (if I actually had one, that is) sends me sales data on a monthly basis. Since my class start in late February, we might expect most of the sales to occur towards the start of the year. Let's suppose that I have 100 sales in February, 200 sales in March and 50 sales in April, and no other sales for the rest of the year. What I would like to do is have a variable – let's call it `sales.by.month` – that stores all this sales data. The first number stored should be 0 since I had no sales in January, the second should be 100, and so on. The simplest way to do this in R is to use the `combine` function, `c()`. To do so, all we have to do is type all the numbers you want to store in a comma separated list, like this:²³

```
sales.by.month <- c(0, 100, 200, 50, 0, 0, 0, 0, 0, 0, 0)
sales.by.month

## [1] 0 100 200 50 0 0 0 0 0 0 0
```

To use the correct terminology here, we have a single variable here called `sales.by.month`: this variable is a vector that consists of 12 *elements*.

3.7.2 A handy digression

Now that we've learned how to put information into a vector, the next thing to understand is how to pull that information back out again. However, before I do so it's worth taking a slight detour. If you've been following along, typing all the commands into R yourself, it's possible that the output that you saw when we printed out the `sales.by.month` vector was slightly different to what I showed above. This would have happened if the window (or the RStudio panel) that contains the R console is really, really narrow. If that were the case, you might have seen output that looks something like this:

```
sales.by.month

## [1] 0 100 200 50 0 0 0 0 0 0
```

Because there wasn't much room on the screen, R has printed out the results over two lines. But that's not the important thing to notice. The important point is that the first line has a `[1]` in front of it, whereas the second line starts with `[9]`. It's pretty clear what's happening here. For the first row, R has printed out the 1st element through to the 8th element, so it starts that row with a `[1]`. For the second row, R has printed out the 9th element of the vector through to the 12th one, and so it begins that row with a `[9]` so that you can tell where it's up to at a glance. It might seem a bit odd to you that R does this, but in some ways it's a kindness, especially when dealing with larger data sets!

²³Notice that I didn't specify any argument names here. The `c()` function is one of those cases where we don't use names. We just type all the numbers, and R just dumps them all in a single variable.

3.7.3 Getting information out of vectors

To get back to the main story, let's consider the problem of how to get information out of a vector. At this point, you might have a sneaking suspicion that the answer has something to do with the [1] and [9] things that R has been printing out. And of course you are correct. Suppose I want to pull out the February sales data only. February is the second month of the year, so let's try this:

```
sales.by.month[2]
```

```
## [1] 100
```

Yep, that's the February sales all right. But there's a subtle detail to be aware of here: notice that R outputs [1] 100, *not* [2] 100. This is because R is being extremely literal. When we typed in `sales.by.month[2]`, we asked R to find exactly *one* thing, and that one thing happens to be the second element of our `sales.by.month` vector. So, when it outputs [1] 100 what R is saying is that the first number *that we just asked for* is 100. This behaviour makes more sense when you realise that we can use this trick to create new variables. For example, I could create a `february.sales` variable like this:

```
february.sales <- sales.by.month[2]
february.sales
```

```
## [1] 100
```

Obviously, the new variable `february.sales` should only have one element and so when I print it out this new variable, the R output begins with a [1] because 100 is the value of the first (and only) element of `february.sales`. The fact that this also happens to be the value of the second element of `sales.by.month` is irrelevant. We'll pick this topic up again shortly (Section 3.10).

3.7.4 Altering the elements of a vector

Sometimes you'll want to change the values stored in a vector. Imagine my surprise when the publisher rings me up to tell me that the sales data for May are wrong. There were actually an additional 25 books sold in May, but there was an error or something so they hadn't told me about it. How can I fix my `sales.by.month` variable? One possibility would be to assign the whole vector again from the beginning, using `c()`. But that's a lot of typing. Also, it's a little wasteful: why should R have to redefine the sales figures for all 12 months, when only the 5th one is wrong? Fortunately, we can tell R to change only the 5th element, using this trick:

```
sales.by.month[5] <- 25
sales.by.month
```

```
## [1] 0 100 200 50 25 0 0 0 0 0 0
```

Another way to edit variables is to use the `edit()` or `fix()` functions. I won't discuss them in detail right now, but you can check them out on your own.

3.7.5 Useful things to know about vectors

Before moving on, I want to mention a couple of other things about vectors. Firstly, you often find yourself wanting to know how many elements there are in a vector (usually because you've forgotten). You can use the `length()` function to do this. It's quite straightforward:

```
length( x = sales.by.month )
```

```
## [1] 12
```

Secondly, you often want to alter all of the elements of a vector at once. For instance, suppose I wanted to figure out how much money I made in each month. Since I'm earning an exciting \$7 per book (no seriously, that's actually pretty close to what authors get on the very expensive textbooks that you're expected to purchase), what I want to do is multiply each element in the `sales.by.month` vector by 7. R makes this pretty easy, as the following example shows:

```
sales.by.month * 7
```

```
## [1] 0 700 1400 350 175 0 0 0 0 0 0 0
```

In other words, when you multiply a vector by a single number, all elements in the vector get multiplied. The same is true for addition, subtraction, division and taking powers. So that's neat. On the other hand, suppose I wanted to know how much money I was making per day, rather than per month. Since not every month has the same number of days, I need to do something slightly different. Firstly, I'll create two new vectors:

```
days.per.month <- c(31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)
profit <- sales.by.month * 7
```

Obviously, the `profit` variable is the same one we created earlier, and the `days.per.month` variable is pretty straightforward. What I want to do is divide every element of `profit` by the *corresponding* element of `days.per.month`. Again, R makes this pretty easy:

```
profit / days.per.month
```

```
## [1] 0.000000 25.000000 45.161290 11.666667 5.645161 0.000000 0.000000
## [8] 0.000000 0.000000 0.000000 0.000000 0.000000
```

I still don't like all those zeros, but that's not what matters here. Notice that the second element of the output is 25, because R has divided the second element of `profit` (i.e. 700) by the second element of `days.per.month` (i.e. 28). Similarly, the third element of the output is equal to 1400 divided by 31, and so on. We'll talk more about calculations involving vectors later on (and in particular a thing called the "recycling rule"; Section 7.12.2, but that's enough detail for now).

3.8 Storing text data

A lot of the time your data will be numeric in nature, but not always. Sometimes your data really needs to be described using text, not using numbers. To address this, we need to consider the situation where our variables store text. To create a variable that stores the word "hello", we can type this:

```
greeting <- "hello"
greeting
```

```
## [1] "hello"
```

When interpreting this, it's important to recognise that the quote marks here *aren't* part of the string itself. They're just something that we use to make sure that R knows to treat the characters that they enclose as a piece of text data, known as a ***character string***. In other words, R treats "hello" as a string containing the word "hello"; but if I had typed `he11o` instead, R would go looking for a variable by that name! You can also use '`hello`' to specify a character string.

Okay, so that's how we store the text. Next, it's important to recognise that when we do this, R stores the entire word "hello" as a *single* element: our `greeting` variable is *not* a vector of five different letters. Rather, it has only the one element, and that element corresponds to the entire character string "hello". To illustrate this, if I actually ask R to find the first element of `greeting`, it prints the whole string:

```
greeting[1]
```

```
## [1] "hello"
```

Of course, there's no reason why I can't create a vector of character strings. For instance, if we were to continue with the example of my attempts to look at the monthly sales data for my book, one variable I might want would include the names of all 12 months.⁷[Though actually there's no real need to do this, since R has an inbuilt variable called `month.name`] that you can use for this purpose. To do so, I could type in a command like this

```
months <- c("January", "February", "March", "April", "May", "June",
          "July", "August", "September", "October", "November",
          "December")
```

This is a ***character vector*** containing 12 elements, each of which is the name of a month. So if I wanted R to tell me the name of the fourth month, all I would do is this:

```
months[4]
```

```
## [1] "April"
```

3.8.1 Working with text

Working with text data is somewhat more complicated than working with numeric data, and I discuss some of the basic ideas in Section 7.8, but for purposes of the current chapter we only need this bare bones sketch. The only other thing I want to do before moving on is show you an example of a function that can be applied to text data. So far, most of the functions that we have seen (i.e., `sqrt()`, `abs()` and `round()`) only make sense when applied to numeric data (e.g., you can't calculate the square root of "hello"), and we've seen one function that can be applied to pretty much any variable or vector (i.e., `length()`). So it might be nice to see an example of a function that can be applied to text.

The function I'm going to introduce you to is called `nchar()`, and what it does is count the number of individual characters that make up a string. Recall earlier that when we tried to calculate the `length()` of our `greeting` variable it returned a value of 1: the `greeting` variable contains only the one string, which happens to be "hello". But what if I want to know how many letters there are in the word? Sure, I could *count* them, but that's boring, and more to the point it's a terrible strategy if what I wanted to know was the number of letters in *War and Peace*. That's where the `nchar()` function is helpful:

```
nchar( x = greeting )
```

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

That makes sense, since there are in fact 5 letters in the string "hello". Better yet, you can apply `nchar()` to whole vectors. So, for instance, if I want R to tell me how many letters there are in the names of each of the 12 months, I can do this:

```
nchar( x = months )
```

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

So that's nice to know. The `nchar()` function can do a bit more than this, and there's a lot of other functions that you can do to extract more information from text or do all sorts of fancy things. However, the goal here is not to teach any of that! The goal right now is just to see an example of a function that actually does work when applied to text.

3.9 Storing “true or false” data

Time to move onto a third kind of data. A key concept in that a lot of R relies on is the idea of a *logical value*. A logical value is an assertion about whether something is true or false. This is implemented in R in a pretty straightforward way. There are two logical values, namely `TRUE` and `FALSE`. Despite the simplicity, a logical values are very useful things. Let's see how they work.

3.9.1 Assessing mathematical truths

In George Orwell's classic book *1984*, one of the slogans used by the totalitarian Party was “two plus two equals five”, the idea being that the political domination of human freedom becomes complete when it is possible to subvert even the most basic of truths. It's a terrifying thought, especially when the protagonist Winston Smith finally breaks down under torture and agrees to the proposition. “Man is infinitely malleable”, the book says. I'm pretty sure that this isn't true of humans²⁴ but it's definitely not true of R. R is not infinitely malleable. It has rather firm opinions on the topic of what is and isn't true, at least as regards basic mathematics. If I ask it to calculate $2 + 2$, it always gives the same answer, and it's not bloody 5:

```
2 + 2
```

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

Of course, so far R is just doing the calculations. I haven't asked it to explicitly assert that $2 + 2 = 4$ is a true statement. If I want R to make an explicit judgement, I can use a command like this:

```
2 + 2 == 4
```

```
## [1] TRUE
```

²⁴I offer up my teenage attempts to be “cool” as evidence that some things just can't be done.

What I've done here is use the *equality operator*, `==`, to force R to make a “true or false” judgement.²⁵ Okay, let's see what R thinks of the Party slogan:

```
2+2 == 5
```

```
## [1] FALSE
```

Booyah! Freedom and ponies for all! Or something like that. Anyway, it's worth having a look at what happens if I try to *force* R to believe that two plus two is five by making an assignment statement like `2 + 2 = 5` or `2 + 2 <- 5`. When I do this, here's what happens:

```
2 + 2 = 5
```

```
## Error in 2 + 2 = 5: target of assignment expands to non-language object
```

R doesn't like this very much. It recognises that `2 + 2` is *not* a variable (that's what the “non-language object” part is saying), and it won't let you try to “reassign” it. While R is pretty flexible, and actually does let you do some quite remarkable things to redefine parts of R itself, there are just some basic, primitive truths that it refuses to give up. It won't change the laws of addition, and it won't change the definition of the number 2.

That's probably for the best.

3.9.2 Logical operations

So now we've seen logical operations at work, but so far we've only seen the simplest possible example. You probably won't be surprised to discover that we can combine logical operations with other operations and functions in a more complicated way, like this:

```
3*3 + 4*4 == 5*5
```

```
## [1] TRUE
```

or this

```
sqrt( 25 ) == 5
```

```
## [1] TRUE
```

Not only that, but as Table 3.2 illustrates, there are several other logical operators that you can use, corresponding to some basic mathematical concepts.

Hopefully these are all pretty self-explanatory: for example, the *less than* operator `<` checks to see if the number on the left is less than the number on the right. If it's less, then R returns an answer of `TRUE`:

²⁵Note that this is a very different operator to the assignment operator `=` that I talked about in Section 3.4. A common typo that people make when trying to write logical commands in R (or other languages, since the “`=` versus `==`” distinction is important in most programming languages) is to accidentally type `=` when you really mean `==`. Be especially cautious with this – I've been programming in various languages since I was a teenager, and I *still* screw this up a lot. Hm. I think I see why I wasn't cool as a teenager. And why I'm still not cool.

Table 3.2: Some logical operators. Technically I should be calling these "binary relational operators", but quite frankly I don't want to. It's my book so no-one can make me.

operation	operator	example input	answer
less than	<	2 < 3	'TRUE'
less than or equal to	<=	2 <= 2	'TRUE'
greater than	>	2 > 3	'FALSE'
greater than or equal to	>=	2 >= 2	'TRUE'
equal to	==	2 == 3	'FALSE'
not equal to	!=	2 != 3	'TRUE'

```
99 < 100
```

```
## [1] TRUE
```

but if the two numbers are equal, or if the one on the right is larger, then R returns an answer of FALSE, as the following two examples illustrate:

```
100 < 100
```

```
## [1] FALSE
```

```
100 < 99
```

```
## [1] FALSE
```

In contrast, the *less than or equal to* operator `<=` will do exactly what it says. It returns a value of TRUE if the number of the left hand side is less than or equal to the number on the right hand side. So if we repeat the previous two examples using `<=`, here's what we get:

```
100 <= 100
```

```
## [1] TRUE
```

```
100 <= 99
```

```
## [1] FALSE
```

And at this point I hope it's pretty obvious what the *greater than* operator `>` and the *greater than or equal to* operator `>=` do! Next on the list of logical operators is the *not equal to* operator `!=` which – as with all the others – does what it says it does. It returns a value of TRUE when things on either side are not identical to each other. Therefore, since $2 + 2$ isn't equal to 5, we get:

```
2 + 2 != 5
```

```
## [1] TRUE
```

Table 3.3: Some more logical operators.

operation	operator	example input	answer
not	!	<code>!(1==1)</code>	‘FALSE’
or		<code>(1==1) (2==3)</code>	‘TRUE’
and	&	<code>(1==1) & (2==3)</code>	‘FALSE’

We’re not quite done yet. There are three more logical operations that are worth knowing about, listed in Table 3.3.

These are the **not** operator `!`, the **and** operator `&`, and the **or** operator `|`. Like the other logical operators, their behaviour is more or less exactly what you’d expect given their names. For instance, if I ask you to assess the claim that “either $2 + 2 = 4$ or $2 + 2 = 5$ ” you’d say that it’s true. Since it’s an “either-or” statement, all we need is for one of the two parts to be true. That’s what the `|` operator does:

```
(2+2 == 4) | (2+2 == 5)
```

```
## [1] TRUE
```

On the other hand, if I ask you to assess the claim that “both $2 + 2 = 4$ and $2 + 2 = 5$ ” you’d say that it’s false. Since this is an **and** statement we need both parts to be true. And that’s what the `&` operator does:

```
(2+2 == 4) & (2+2 == 5)
```

```
## [1] FALSE
```

Finally, there’s the **not** operator, which is simple but annoying to describe in English. If I ask you to assess my claim that “it is not true that $2 + 2 = 5$ ” then you would say that my claim is true; because my claim is that “ $2 + 2 = 5$ is false”. And I’m right. If we write this as an R command we get this:

```
! (2+2 == 5)
```

```
## [1] TRUE
```

In other words, since $2+2 == 5$ is a **FALSE** statement, it must be the case that `!(2+2 == 5)` is a **TRUE** one. Essentially, what we’ve really done is claim that “not false” is the same thing as “true”. Obviously, this isn’t really quite right in real life. But R lives in a much more black or white world: for R everything is either true or false. No shades of gray are allowed. We can actually see this much more explicitly, like this:

```
! FALSE
```

```
## [1] TRUE
```

Of course, in our $2 + 2 = 5$ example, we didn’t really need to use “not” `!` and “equals to” `==` as two separate operators. We could have just used the “not equals to” operator `!=` like this:

```
2+2 != 5
```

```
## [1] TRUE
```

But there are many situations where you really do need to use the `!` operator. We'll see some later on.²⁶

3.9.3 Storing and using logical data

Up to this point, I've introduced *numeric data* (in Sections 3.4 and `@ref(#vectors)`) and *character data* (in Section 3.8). So you might not be surprised to discover that these `TRUE` and `FALSE` values that R has been producing are actually a third kind of data, called *logical data*. That is, when I asked R if `2 + 2 == 5` and it said `[1] FALSE` in reply, it was actually producing information that we can store in variables. For instance, I could create a variable called `is.the.Party.correct`, which would store R's opinion:

```
is.the.Party.correct <- 2 + 2 == 5
is.the.Party.correct
```

```
## [1] FALSE
```

Alternatively, you can assign the value directly, by typing `TRUE` or `FALSE` in your command. Like this:

```
is.the.Party.correct <- FALSE
is.the.Party.correct
```

```
## [1] FALSE
```

Better yet, because it's kind of tedious to type `TRUE` or `FALSE` over and over again, R provides you with a shortcut: you can use `T` and `F` instead (but it's case sensitive: `t` and `f` won't work).²⁷ So this works:

```
is.the.Party.correct <- F
is.the.Party.correct
```

```
## [1] FALSE
```

but this doesn't:

```
is.the.Party.correct <- f
## Error in eval(expr, envir, enclos): object 'f' not found
```

3.9.4 Vectors of logicals

The next thing to mention is that you can store vectors of logical values in exactly the same way that you can store vectors of numbers (Section 3.7) and vectors of text data (Section 3.8). Again, we can define them directly via the `c()` function, like this:

²⁶A note for those of you who have taken a computer science class: yes, R does have a function for exclusive-or, namely `xor()`. Also worth noting is the fact that R makes the distinction between element-wise operators `&` and `|` and operators that look only at the first element of the vector, namely `&&` and `||`. To see the distinction, compare the behaviour of a command like `c(FALSE, TRUE) & c(TRUE, TRUE)` to the behaviour of something like `c(FALSE, TRUE) && c(TRUE, TRUE)`. If this doesn't mean anything to you, ignore this footnote entirely. It's not important for the content of this book.

²⁷Warning! `TRUE` and `FALSE` are reserved keywords in R, so you can trust that they always mean what they say they do. Unfortunately, the shortcut versions `T` and `F` do not have this property. It's even possible to create variables that set up the reverse meanings, by typing commands like `T <- FALSE` and `F <- TRUE`. This is kind of insane, and something that is generally thought to be a design flaw in R. Anyway, the long and short of it is that it's safer to use `TRUE` and `FALSE`.

```
x <- c(TRUE, TRUE, FALSE)
x
```

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

or you can produce a vector of logicals by applying a logical operator to a vector. This might not make a lot of sense to you, so let’s unpack it slowly. First, let’s suppose we have a vector of numbers (i.e., a “non-logical vector”). For instance, we could use the `sales.by.month` vector that we were using in Section@ref(#vectors). Suppose I wanted R to tell me, for each month of the year, whether I actually sold a book in that month. I can do that by typing this:

```
sales.by.month > 0
```

```
## [1] FALSE TRUE TRUE TRUE TRUE FALSE FALSE FALSE FALSE FALSE
## [12] FALSE
```

and again, I can store this in a vector if I want, as the example below illustrates:

```
any.sales.this.month <- sales.by.month > 0
any.sales.this.month
```

```
## [1] FALSE TRUE TRUE TRUE TRUE FALSE FALSE FALSE FALSE FALSE
## [12] FALSE
```

In other words, `any.sales.this.month` is a logical vector whose elements are `TRUE` only if the corresponding element of `sales.by.month` is greater than zero. For instance, since I sold zero books in January, the first element is `FALSE`.

3.9.5 Applying logical operation to text

In a moment (Section 3.10) I’ll show you why these logical operations and logical vectors are so handy, but before I do so I want to very briefly point out that you can apply them to text as well as to logical data. It’s just that we need to be a bit more careful in understanding how R interprets the different operations. In this section I’ll talk about how the equal to operator `==` applies to text, since this is the most important one. Obviously, the not equal to operator `!=` gives the exact opposite answers to `==` so I’m implicitly talking about that one too, but I won’t give specific commands showing the use of `!=`. As for the other operators, I’ll defer a more detailed discussion of this topic to Section 7.8.5.

Okay, let’s see how it works. In one sense, it’s very simple. For instance, I can ask R if the word “cat” is the same as the word “dog”, like this:

```
"cat" == "dog"
```

```
## [1] FALSE
```

That’s pretty obvious, and it’s good to know that even R can figure that out. Similarly, R does recognise that a “cat” is a “cat”:

```
"cat" == "cat"
## [1] TRUE
```

Again, that's exactly what we'd expect. However, what you need to keep in mind is that R is not at all tolerant when it comes to grammar and spacing. If two strings differ in any way whatsoever, R will say that they're not equal to each other, as the following examples indicate:

```
"cat" == "cat"
## [1] TRUE

"cat" == "CAT"
## [1] FALSE

"cat" == "c a t"
## [1] FALSE
```

3.10 Indexing vectors

One last thing to add before finishing up this chapter. So far, whenever I've had to get information out of a vector, all I've done is typed something like `months[4]`; and when I do this R prints out the fourth element of the `months` vector. In this section, I'll show you two additional tricks for getting information out of the vector.

3.10.1 Extracting multiple elements

One very useful thing we can do is pull out more than one element at a time. In the previous example, we only used a single number (i.e., 2) to indicate which element we wanted. Alternatively, we can use a vector. So, suppose I wanted the data for February, March and April. What I could do is use the vector `c(2,3,4)` to indicate which elements I want R to pull out. That is, I'd type this:

```
sales.by.month[ c(2,3,4) ]
```

```
## [1] 100 200 50
```

Notice that the order matters here. If I asked for the data in the reverse order (i.e., April first, then March, then February) by using the vector `c(4,3,2)`, then R outputs the data in the reverse order:

```
sales.by.month[ c(4,3,2) ]
```

```
## [1] 50 200 100
```

A second thing to be aware of is that R provides you with handy shortcuts for very common situations. For instance, suppose that I wanted to extract everything from the 2nd month through to the 8th month. One way to do this is to do the same thing I did above, and use the vector `c(2,3,4,5,6,7,8)` to indicate the elements that I want. That works just fine

```
sales.by.month[ c(2,3,4,5,6,7,8) ]  
  
## [1] 100 200 50 25 0 0 0
```

but it's kind of a lot of typing. To help make this easier, R lets you use `2:8` as shorthand for `c(2,3,4,5,6,7,8)`, which makes things a lot simpler. First, let's just check that this is true:

```
2:8
```

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

Next, let's check that we can use the `2:8` shorthand as a way to pull out the 2nd through 8th elements of `sales.by.months`:

```
sales.by.month[2:8]
```

```
## [1] 100 200 50 25 0 0 0
```

So that's kind of neat.

3.10.2 Logical indexing

At this point, I can introduce an extremely useful tool called *logical indexing*. In the last section, I created a logical vector `any.sales.this.month`, whose elements are TRUE for any month in which I sold at least one book, and FALSE for all the others. However, that big long list of TRUEs and FALSEs is a little bit hard to read, so what I'd like to do is to have R select the names of the months for which I sold any books. Earlier on, I created a vector `months` that contains the names of each of the months. This is where logical indexing is handy. What I need to do is this:

```
months[ sales.by.month > 0 ]  
  
## [1] "February" "March"    "April"     "May"
```

To understand what's happening here, it's helpful to notice that `sales.by.month > 0` is the same logical expression that we used to create the `any.sales.this.month` vector in the last section. In fact, I could have just done this:

```
months[ any.sales.this.month ]  
  
## [1] "February" "March"    "April"     "May"
```

and gotten exactly the same result. In order to figure out which elements of `months` to include in the output, what R does is look to see if the corresponding element in `any.sales.this.month` is TRUE. Thus, since element 1 of `any.sales.this.month` is FALSE, R does not include "January" as part of the output; but since element 2 of `any.sales.this.month` is TRUE, R does include "February" in the output. Note that there's no reason why I can't use the same trick to find the actual sales numbers for those months. The command to do that would just be this:

```
sales.by.month [ sales.by.month > 0 ]
```

```
## [1] 100 200 50 25
```

In fact, we can do the same thing with text. Here's an example. Suppose that – to continue the saga of the textbook sales – I later find out that the bookshop only had sufficient stocks for a few months of the year. They tell me that early in the year they had "high" stocks, which then dropped to "low" levels, and in fact for one month they were "out" of copies of the book for a while before they were able to replenish them. Thus I might have a variable called `stock.levels` which looks like this:

```
stock.levels<-c("high", "high", "low", "out", "out", "high",
                 "high", "high", "high", "high", "high")
```

```
stock.levels
```

```
## [1] "high" "high" "low"   "out"   "out"   "high" "high" "high" "high"
## [11] "high" "high"
```

Thus, if I want to know the months for which the bookshop was out of my book, I could apply the logical indexing trick, but with the character vector `stock.levels`, like this:

```
months[stock.levels == "out"]
```

```
## [1] "April" "May"
```

Alternatively, if I want to know when the bookshop was either low on copies or out of copies, I could do this:

```
months[stock.levels == "out" | stock.levels == "low"]
```

```
## [1] "March" "April" "May"
```

or this

```
months[stock.levels != "high" ]
```

```
## [1] "March" "April" "May"
```

Either way, I get the answer I want.

At this point, I hope you can see why logical indexing is such a useful thing. It's a very basic, yet very powerful way to manipulate data. We'll talk a lot more about how to manipulate data in Chapter 7, since it's a critical skill for real world research that is often overlooked in introductory research methods classes (or at least, that's been my experience). It does take a bit of practice to become completely comfortable using logical indexing, so it's a good idea to play around with these sorts of commands. Try creating a few different variables of your own, and then ask yourself questions like "how do I get R to spit out all the elements that are [blah]". Practice makes perfect, and it's only by practicing logical indexing that you'll perfect the art of yelling frustrated insults at your computer.²⁸

²⁸Well, I say that... but in my personal experience it wasn't until I started learning "regular expressions" that my loathing of computers reached its peak.

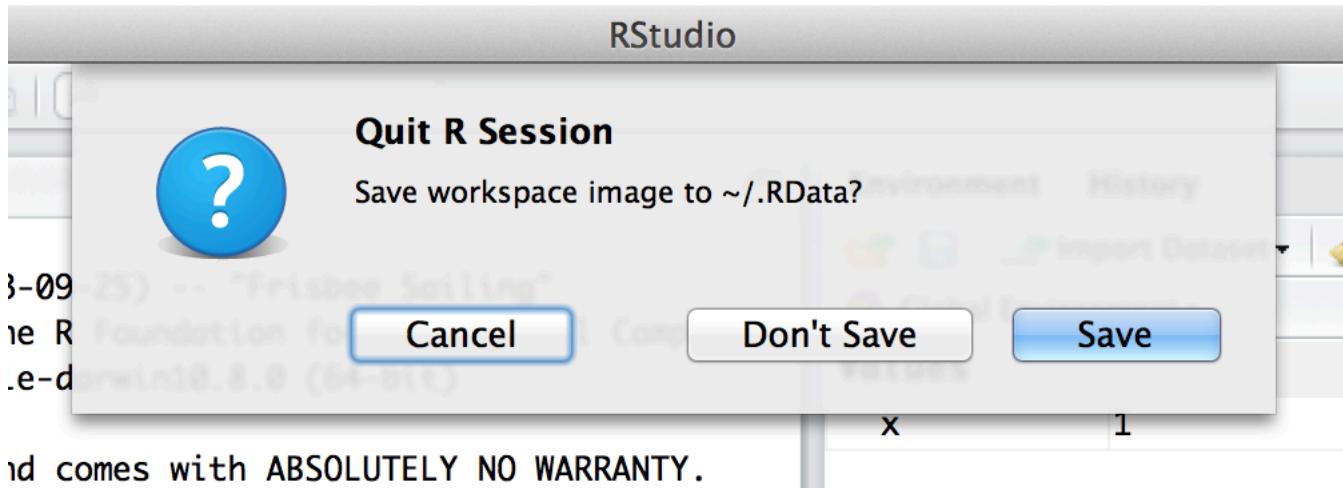


Figure 3.5: The dialog box that shows up when you try to close RStudio.

3.11 Quitting R

```
knitr::include_graphics("./rbook-master/img/introR/Rstudio_quit.png")
```

There's one last thing I should cover in this chapter: how to quit R. When I say this, I'm not trying to imply that R is some kind of pathological addiction and that you need to call the R QuitLine or wear patches to control the cravings (although you certainly might argue that there's something seriously pathological about being addicted to R). I just mean how to exit the program. Assuming you're running R in the usual way (i.e., through RStudio or the default GUI on a Windows or Mac computer), then you can just shut down the application in the normal way. However, R also has a function, called `q()` that you can use to quit, which is pretty handy if you're running R in a terminal window.

Regardless of what method you use to quit R, when you do so for the first time R will probably ask you if you want to save the "workspace image". We'll talk a lot more about loading and saving data in Section 4.5, but I figured we'd better quickly cover this now otherwise you're going to get annoyed when you close R at the end of the chapter. If you're using RStudio, you'll see a dialog box that looks like the one shown in Figure 3.5. If you're using a text based interface you'll see this:

```
q()
## Save workspace image? [y/n/c]:
```

The `y/n/c` part here is short for "yes / no / cancel". Type `y` if you want to save, `n` if you don't, and `c` if you've changed your mind and you don't want to quit after all.

What does this actually *mean*? What's going on is that R wants to know if you want to save all those variables that you've been creating, so that you can use them later. This sounds like a great idea, so it's really tempting to type `y` or click the "Save" button. To be honest though, I very rarely do this, and it kind of annoys me a little bit... what R is *really* asking is if you want it to store these variables in a "default" data file, which it will automatically reload for you next time you open R. And quite frankly, if I'd wanted to save the variables, then I'd have already saved them before trying to quit. Not only that, I'd have saved them to a location of *my* choice, so that I can find it again later. So I personally never bother with this.

In fact, every time I install R on a new machine one of the first things I do is change the settings so that it never asks me again. You can do this in RStudio really easily: use the menu system to find the RStudio

option; the dialog box that comes up will give you an option to tell R never to whine about this again (see Figure 3.6. On a Mac, you can open this window by going to the “RStudio” menu and selecting “Preferences”. On a Windows machine you go to the “Tools” menu and select “Global Options”. Under the “General” tab you’ll see an option that reads “Save workspace to .Rdata on exit”. By default this is set to “ask”. If you want R to stop asking, change it to “never”.

```
knitr::include_graphics("./rbook-master/img/introR/Rstudio_options.png")
```

3.12 Summary

Every book that tries to introduce basic programming ideas to novices has to cover roughly the same topics, and in roughly the same order. Mine is no exception, and so in the grand tradition of doing it just the same way everyone else did it, this chapter covered the following topics:

- Getting started. We downloaded and installed R and RStudio
- Basic commands. We talked a bit about the logic of how R works and in particular how to type commands into the R console (Section@ref(#firstcommand), and in doing so learned how to perform basic calculations using the arithmetic operators +, -, *, / and ^.
- Introduction to functions. We saw several different functions, three that are used to perform numeric calculations (`sqrt()`, `abs()`, `round()`, one that applies to text (`nchar()`; Section@ref(#simpletext)), and one that works on any variable (`length()`; Section@ref(#veclength)). In doing so, we talked a bit about how argument names work, and learned about default values for arguments. (Section@ref(#functionarguments))
- Introduction to variables. We learned the basic idea behind variables, and how to assign values to variables using the assignment operator `<-` (Section@ref(#assign)). We also learned how to create vectors using the combine function `c()`. (Section@ref(#vectors))
- Data types. Learned the distinction between numeric, character and logical data; including the basics of how to enter and use each of them. (Sections@ref(#assign) to Sections3.9)
- Logical operations. (#logicals) Learned how to use the logical operators ==, !=, <, >, <=, =>, !, & and |. And learned how to use logical indexing. (Section 3.10)

We still haven’t arrived at anything that resembles a “data set”, of course. Maybe the next Chapter will get us a bit closer...

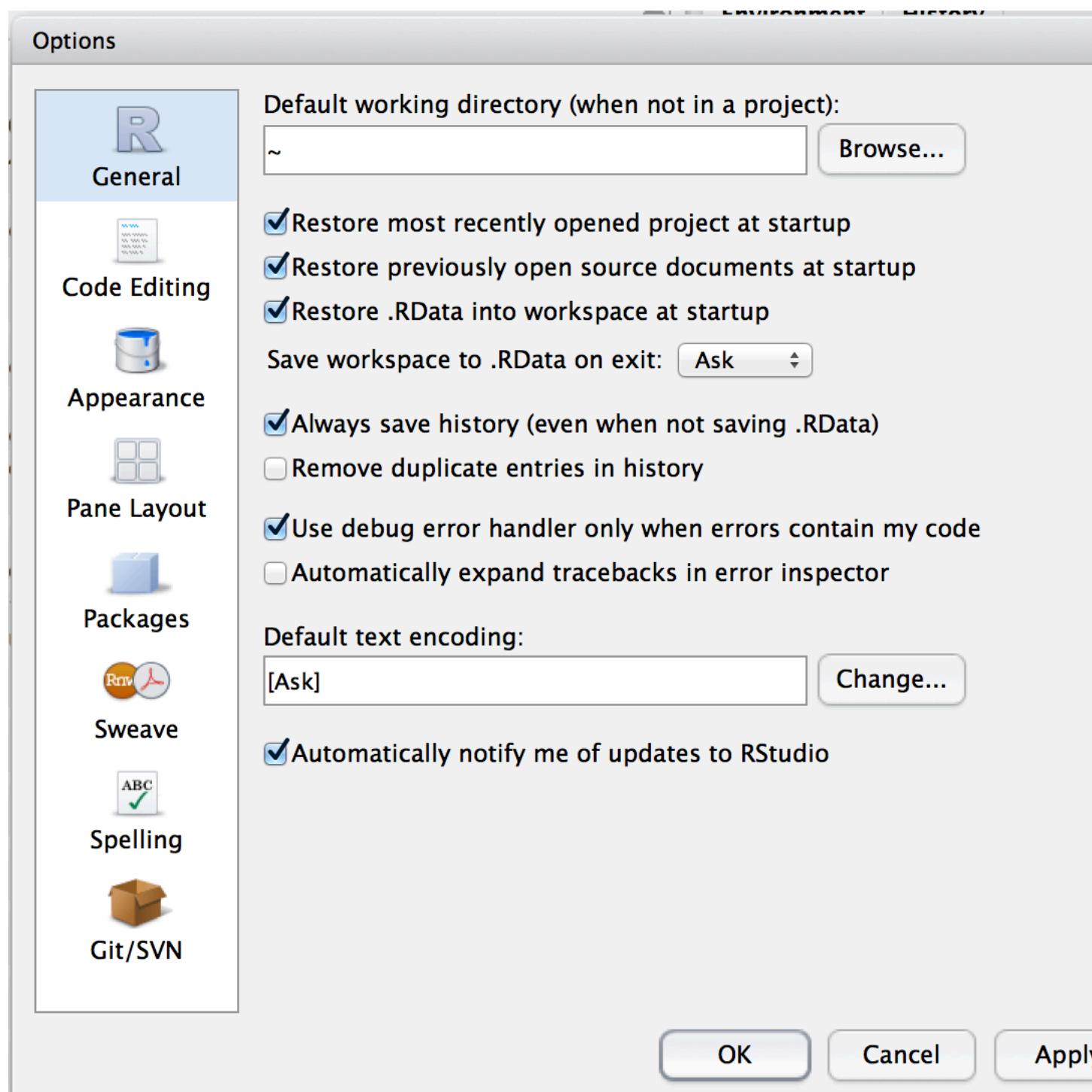


Figure 3.6: The options window in RStudio. On a Mac, you can open this window by going to the "RStudio" menu and selecting "Preferences". On a Windows machine you go to the "Tools" menu and select "Global Options"

Chapter 4

Additional R concepts

Form follows function

– Louis Sullivan

In Chapter 3 our main goal was to get started in R. As we go through the book we'll run into a lot of new R concepts, which I'll explain alongside the relevant data analysis concepts. However, there's still quite a few things that I need to talk about now, otherwise we'll run into problems when we start trying to work with data and do statistics. So that's the goal in this chapter: to build on the introductory content from the last chapter, to get you to the point that we can start using R for statistics. Broadly speaking, the chapter comes in two parts. The first half of the chapter is devoted to the “mechanics” of R: installing and loading packages, managing the workspace, navigating the file system, and loading and saving data. In the second half, I'll talk more about what kinds of variables exist in R, and introduce three new kinds of variables: factors, data frames and formulas. I'll finish up by talking a little bit about the help documentation in R as well as some other avenues for finding assistance. In general, I'm not trying to be comprehensive in this chapter, I'm trying to make sure that you've got the basic foundations needed to tackle the content that comes later in the book. However, a lot of the topics are revisited in more detail later, especially in Chapters 7 and 8.

4.1 Using comments

Before discussing any of the more complicated stuff, I want to introduce the **comment** character, `#`. It has a simple meaning: it tells R to ignore everything else you've written on this line. You won't have much need of the `#` character immediately, but it's very useful later on when writing scripts (see Chapter 8). However, while you don't need to use it, I want to be able to include comments in my R extracts. For instance, if you read this:¹

```
seeker <- 3.1415      # create the first variable
lover <- 2.7183        # create the second variable
keeper <- seeker * lover # now multiply them to create a third one
print( keeper )         # print out the value of 'keeper'

## [1] 8.539539
```

it's a lot easier to understand what I'm doing than if I just write this:

¹Notice that I used `print(keeper)` rather than just typing `keeper`. Later on in the text I'll sometimes use the `print()` function to display things because I think it helps make clear what I'm doing, but in practice people rarely do this.

```
seeker <- 3.1415
lover <- 2.7183
keeper <- seeker * lover
print( keeper )
```

```
## [1] 8.539539
```

You might have already noticed that the code extracts in Chapter 3 included the `#` character, but from now on, you'll start seeing `#` characters appearing in the extracts, with some human-readable explanatory remarks next to them. These are still perfectly legitimate commands, since R knows that it should ignore the `#` character and everything after it. But hopefully they'll help make things a little easier to understand.

4.2 Installing and loading packages

In this section I discuss R *packages*, since almost all of the functions you might want to use in R come in packages. A package is basically just a big collection of functions, data sets and other R objects that are all grouped together under a common name. Some packages are already installed when you put R on your computer, but the vast majority of them of R packages are out there on the internet, waiting for you to download, install and use them.

When I first started writing this book, Rstudio didn't really exist as a viable option for using R, and as a consequence I wrote a very lengthy section that explained how to do package management using raw R commands. It's not actually terribly hard to work with packages that way, but it's clunky and unpleasant. Fortunately, we don't have to do things that way anymore. In this section, I'll describe how to work with packages using the Rstudio tools, because they're so much simpler. Along the way, you'll see that whenever you get Rstudio to do something (e.g., install a package), you'll actually see the R commands that get created. I'll explain them as we go, because I think that helps you understand what's going on.

However, before we get started, there's a critical distinction that you need to understand, which is the difference between having a package *installed* on your computer, and having a package *loaded* in R. As of this writing, there are just over 5000 R packages freely available "out there" on the internet.² When you install R on your computer, you don't get all of them: only about 30 or so come bundled with the basic R installation. So right now there are about 30 packages "installed" on your computer, and another 5000 or so that are not installed. So that's what installed means: it means "it's on your computer somewhere". The critical thing to remember is that just because something is on your computer doesn't mean R can use it. In order for R to be able to *use* one of your 30 or so installed packages, that package must also be "loaded". Generally, when you open up R, only a few of these packages (about 7 or 8) are actually loaded. Basically what it boils down to is this:

A package must be installed before it can be loaded.

A package must be loaded before it can be used.

This two step process might seem a little odd at first, but the designers of R had very good reasons to do it this way,³ and you get the hang of it pretty quickly.

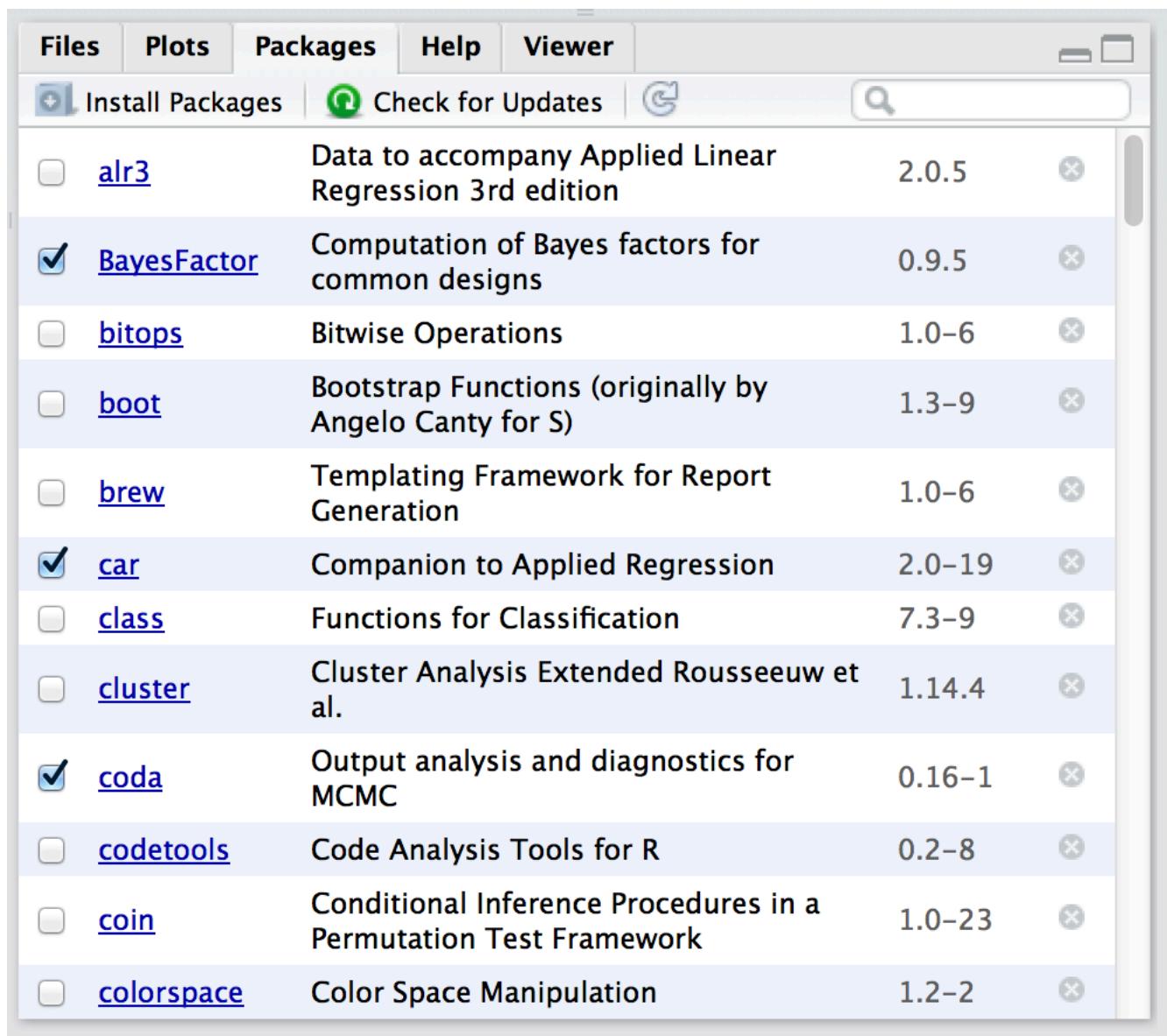


Figure 4.1: The packages panel.

4.2.1 The package panel in Rstudio

Right, lets get started. The first thing you need to do is look in the lower right hand panel in Rstudio. You'll see a tab labelled "Packages". Click on the tab, and you'll see a list of packages that looks something like Figure 4.1. Every row in the panel corresponds to a different package, and every column is a useful piece of information about that package.⁴ Going from left to right, here's what each column is telling you:

- The check box on the far left column indicates whether or not the package is loaded.
- The one word of text immediately to the right of the check box is the name of the package.
- The short passage of text next to the name is a brief description of the package.
- The number next to the description tells you what version of the package you have installed.
- The little x-mark next to the version number is a button that you can push to uninstall the package from your computer (you almost never need this).

4.2.2 Loading a package

That seems straightforward enough, so let's try loading and unloading packades. For this example, I'll use the `foreign` package. The `foreign` package is a collection of tools that are very handy when R needs to interact with files that are produced by other software packages (e.g., SPSS). It comes bundled with R, so it's one of the ones that you have installed already, but it won't be one of the ones loaded. Inside the `foreign` package is a function called `read.spss()`. It's a handy little function that you can use to import an SPSS data file into R, so let's pretend we want to use it. Currently, the `foreign` package isn't loaded, so if I ask R to tell me if it knows about a function called `read.spss()` it tells me that there's no such thing...

```
exists( "read.spss" )
```

```
## [1] FALSE
```

Now let's load the package. In Rstudio, the process is dead simple: go to the package tab, find the entry for the `foreign` package, and check the box on the left hand side. The moment that you do this, you'll see a command like this appear in the R console:

```
library("foreign", lib.loc="/Library/Frameworks/R.framework/Versions/3.0/Resources/library")
```

The `lib.loc` bit will look slightly different on Macs versus on Windows, because that part of the command is just Rstudio telling R where to look to find the installed packages. What I've shown you above is the Mac version. On a Windows machine, you'll probably see something that looks like this:

```
library("foreign", lib.loc="C:/Program Files/R/R-3.0.2/library")
```

But actually it doesn't matter much. The `lib.loc` bit is almost always unnecessary. Unless you've taken to installing packages in idiosyncratic places (which is something that you can do if you really want) R already knows where to look. So in the vast majority of cases, the command to load the `foreign` package is just this:

²More precisely, there are 5000 or so packages on CRAN, the Comprehensive R Archive Network.

³Basically, the reason is that there are 5000 packages, and probably about 4000 authors of packages, and no-one really knows what all of them do. Keeping the installation separate from the loading minimizes the chances that two packages will interact with each other in a nasty way.

⁴If you're using the command line, you can get the same information by typing `library()` at the command line.

```
library("foreign")
```

Throughout this book, you'll often see me typing in `library()` commands. You don't actually have to type them in yourself: you can use the Rstudio package panel to do all your package loading for you. The only reason I include the `library()` commands sometimes is as a reminder to you to make sure that you have the relevant package loaded. Oh, and I suppose we should check to see if our attempt to load the package actually worked. Let's see if R now knows about the existence of the `read.spss()` function...

```
exists( "read.spss" )
```

```
## [1] TRUE
```

Yep. All good.

4.2.3 Unloading a package

Sometimes, especially after a long session of working with R, you find yourself wanting to get rid of some of those packages that you've loaded. The Rstudio package panel makes this exactly as easy as loading the package in the first place. Find the entry corresponding to the package you want to unload, and uncheck the box. When you do that for the `foreign` package, you'll see this command appear on screen:

```
detach("package:foreign", unload=TRUE)
```

```
## Warning: 'foreign' namespace cannot be unloaded:
##   namespace 'foreign' is imported by 'rio', 'psych' so cannot be unloaded
```

And the package is unloaded. We can verify this by seeing if the `read.spss()` function still `exists()`:

```
exists( "read.spss" )
```

```
## [1] FALSE
```

Nope. Definitely gone.

4.2.4 A few extra comments

Sections 4.2.2 and 4.2.3 cover the main things you need to know about loading and unloading packages. However, there's a couple of other details that I want to draw your attention to. A concrete example is the best way to illustrate. One of the other packages that you already have installed on your computer is the `Matrix` package, so let's load that one and see what happens:

```
library( Matrix )
```

```
## Loading required package: lattice
```

This is slightly more complex than the output that we got last time, but it's not too complicated. The `Matrix` package makes use of some of the tools in the `lattice` package, and R has kept track of this dependency. So when you try to load the `Matrix` package, R recognises that you're also going to need to have the `lattice`

package loaded too. As a consequence, *both* packages get loaded, and R prints out a helpful little note on screen to tell you that it's done so.

R is pretty aggressive about enforcing these dependencies. Suppose, for example, I try to unload the `lattice` package while the `Matrix` package is still loaded. This is easy enough to try: all I have to do is uncheck the box next to “`lattice`” in the packages panel. But if I try this, here's what happens:

```
detach("package:lattice", unload=TRUE)
## Error: package `lattice' is required by `Matrix' so will not be detached
```

R refuses to do it. This can be quite useful, since it stops you from accidentally removing something that you still need. So, if I want to remove both `Matrix` and `lattice`, I need to do it in the correct order

Something else you should be aware of. Sometimes you'll attempt to load a package, and R will print out a message on screen telling you that something or other has been “masked”. This will be confusing to you if I don't explain it now, and it actually ties very closely to the whole reason why R forces you to load packages separately from installing them. Here's an example. Two of the package that I'll refer to a lot in this book are called `car` and `psych`. The `car` package is short for “Companion to Applied Regression” (which is a really great book, I'll add), and it has a lot of tools that I'm quite fond of. The `car` package was written by a guy called John Fox, who has written a lot of great statistical tools for social science applications. The `psych` package was written by William Revelle, and it has a lot of functions that are very useful for psychologists in particular, especially in regards to psychometric techniques. For the most part, `car` and `psych` are quite unrelated to each other. They do different things, so not surprisingly almost all of the function names are different. But... there's one exception to that. The `car` package and the `psych` package *both* contain a function called `logit()`.⁵ This creates a naming conflict. If I load both packages into R, an ambiguity is created. If the user types in `logit(100)`, should R use the `logit()` function in the `car` package, or the one in the `psych` package? The answer is: R uses whichever package you loaded most recently, and it tells you this very explicitly. Here's what happens when I load the `car` package, and then afterwards load the `psych` package:

```
library(car)
library(psych)
```

The output here is telling you that the `logit` object (i.e., function) in the `car` package is no longer accessible to you. It's been hidden (or “masked”) from you by the one in the `psych` package.⁶

4.2.5 Downloading new packages

One of the main selling points for R is that there are thousands of packages that have been written for it, and these are all available online. So whereabouts online are these packages to be found, and how do we download and install them? There is a big repository of packages called the “Comprehensive R Archive Network” (CRAN), and the easiest way of getting and installing a new package is from one of the many CRAN mirror sites. Conveniently for us, R provides a function called `install.packages()` that you can use to do this. Even *more* conveniently, the Rstudio team runs its own CRAN mirror and Rstudio has a clean interface that lets you install packages without having to learn how to use the `install.packages()` command⁷

Using the Rstudio tools is, again, dead simple. In the top left hand corner of the packages panel (Figure 4.1) you'll see a button called “Install Packages”. If you click on that, it will bring up a window like the one shown in Figure 4.2.

⁵The `logit` function a simple mathematical function that happens not to have been included in the basic R distribution.

⁶Tip for advanced users. You can get R to use the one from the `car` package by using `car::logit()` as your command rather than `logit()`, since the `car::` part tells R explicitly which package to use. See also `:::` if you're especially keen to force R to use functions it otherwise wouldn't, but take care, since `:::` can be dangerous.

⁷It is not very difficult.

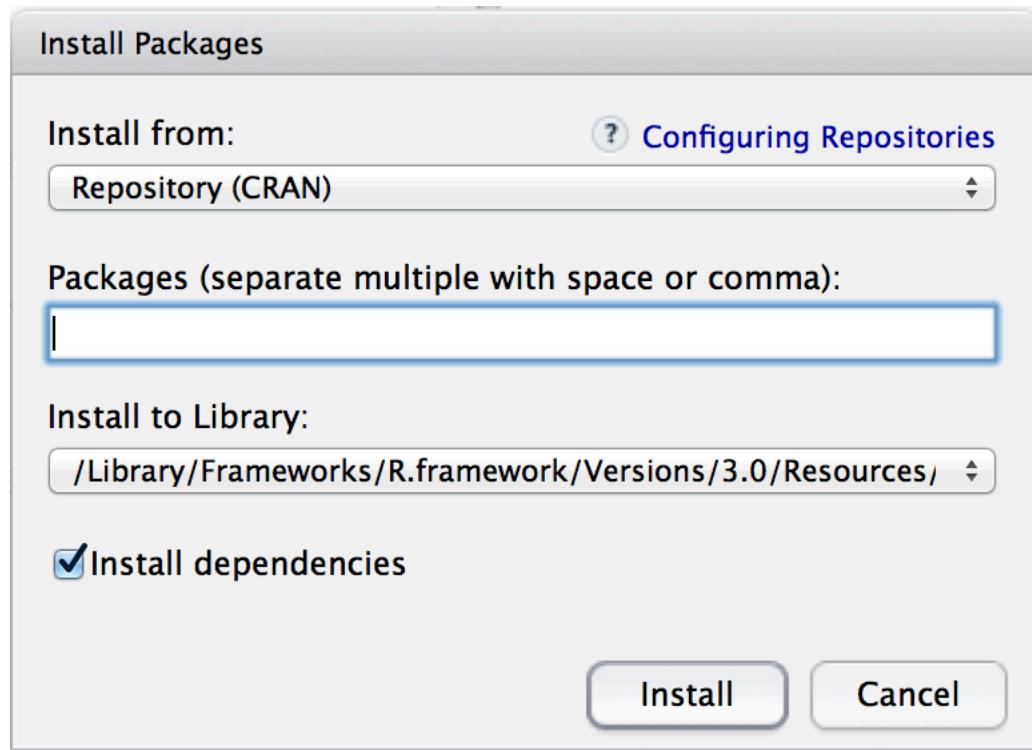


Figure 4.2: The package installation dialog box in Rstudio

There are a few different buttons and boxes you can play with. Ignore most of them. Just go to the line that says “Packages” and start typing the name of the package that you want. As you type, you’ll see a dropdown menu appear (Figure 4.3), listing names of packages that start with the letters that you’ve typed so far.

You can select from this list, or just keep typing. Either way, once you’ve got the package name that you want, click on the install button at the bottom of the window. When you do, you’ll see the following command appear in the R console:

```
install.packages("psych")
```

This is the R command that does all the work. R then goes off to the internet, has a conversation with CRAN, downloads some stuff, and installs it on your computer. You probably don’t care about all the details of R’s little adventure on the web, but the `install.packages()` function is rather chatty, so it reports a bunch of gibberish that you really aren’t all that interested in:

```
trying URL 'http://cran.rstudio.com/bin/macosx/contrib/3.0/psych_1.4.1.tgz'
Content type 'application/x-gzip' length 2737873 bytes (2.6 Mb)
opened URL
=====
downloaded 2.6 Mb
```

```
The downloaded binary packages are in
/var/folders/c1/thhsyrz53g73q0w1kb5z3l_80000gn/T//RtmpmQ9VT3 downloaded_packages
```

Despite the long and tedious response, all that really means is “I’ve installed the `psych` package”. I find it

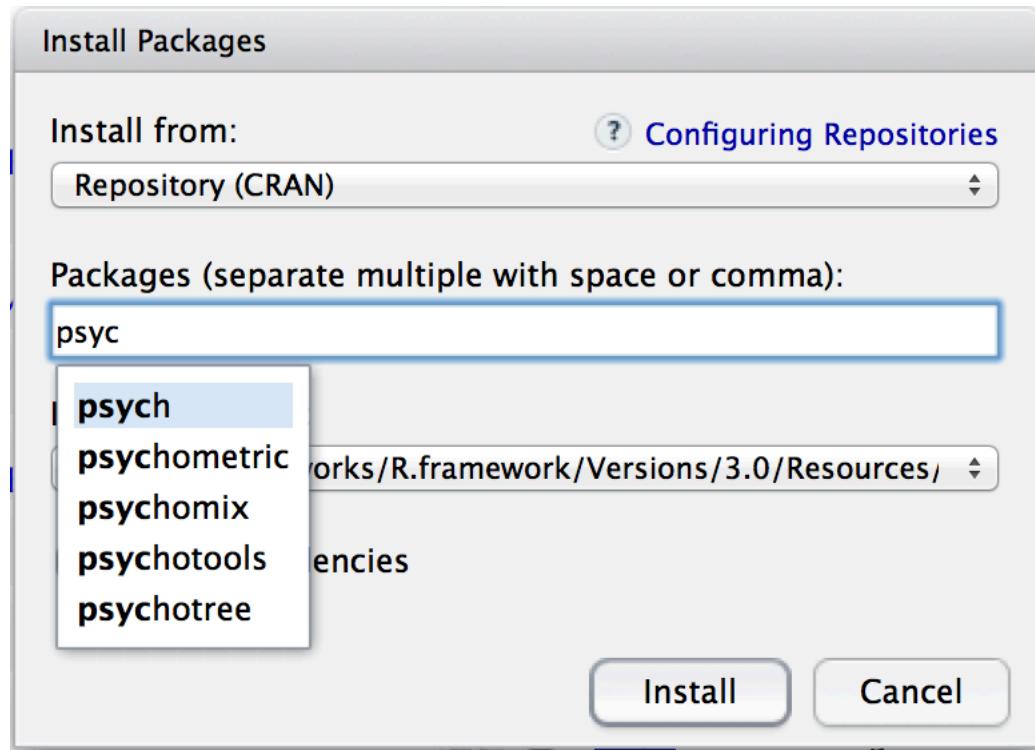


Figure 4.3: When you start typing, you'll see a dropdown menu suggest a list of possible packages that you might want to install

best to humour the talkative little automaton. I don't actually read any of this garbage, I just politely say "thanks" and go back to whatever I was doing.

4.2.6 Updating R and R packages

Every now and then the authors of packages release updated versions. The updated versions often add new functionality, fix bugs, and so on. It's generally a good idea to update your packages periodically. There's an `update.packages()` function that you can use to do this, but it's probably easier to stick with the Rstudio tool. In the packages panel, click on the "Update Packages" button. This will bring up a window that looks like the one shown in Figure 4.4. In this window, each row refers to a package that needs to be updated. You can tell R which updates you want to install by checking the boxes on the left. If you're feeling lazy and just want to update everything, click the "Select All" button, and then click the "Install Updates" button. R then prints out a *lot* of garbage on the screen, individually downloading and installing all the new packages. This might take a while to complete depending on how good your internet connection is. Go make a cup of coffee. Come back, and all will be well.

About every six months or so, a new version of R is released. You can't update R from within Rstudio (not to my knowledge, at least): to get the new version you can go to the CRAN website and download the most recent version of R, and install it in the same way you did when you originally installed R on your computer. This used to be a slightly frustrating event, because whenever you downloaded the new version of R, you would lose all the packages that you'd downloaded and installed, and would have to repeat the process of re-installing them. This was pretty annoying, and there were some neat tricks you could use to get around this. However, newer versions of R don't have this problem so I no longer bother explaining the workarounds for that issue.

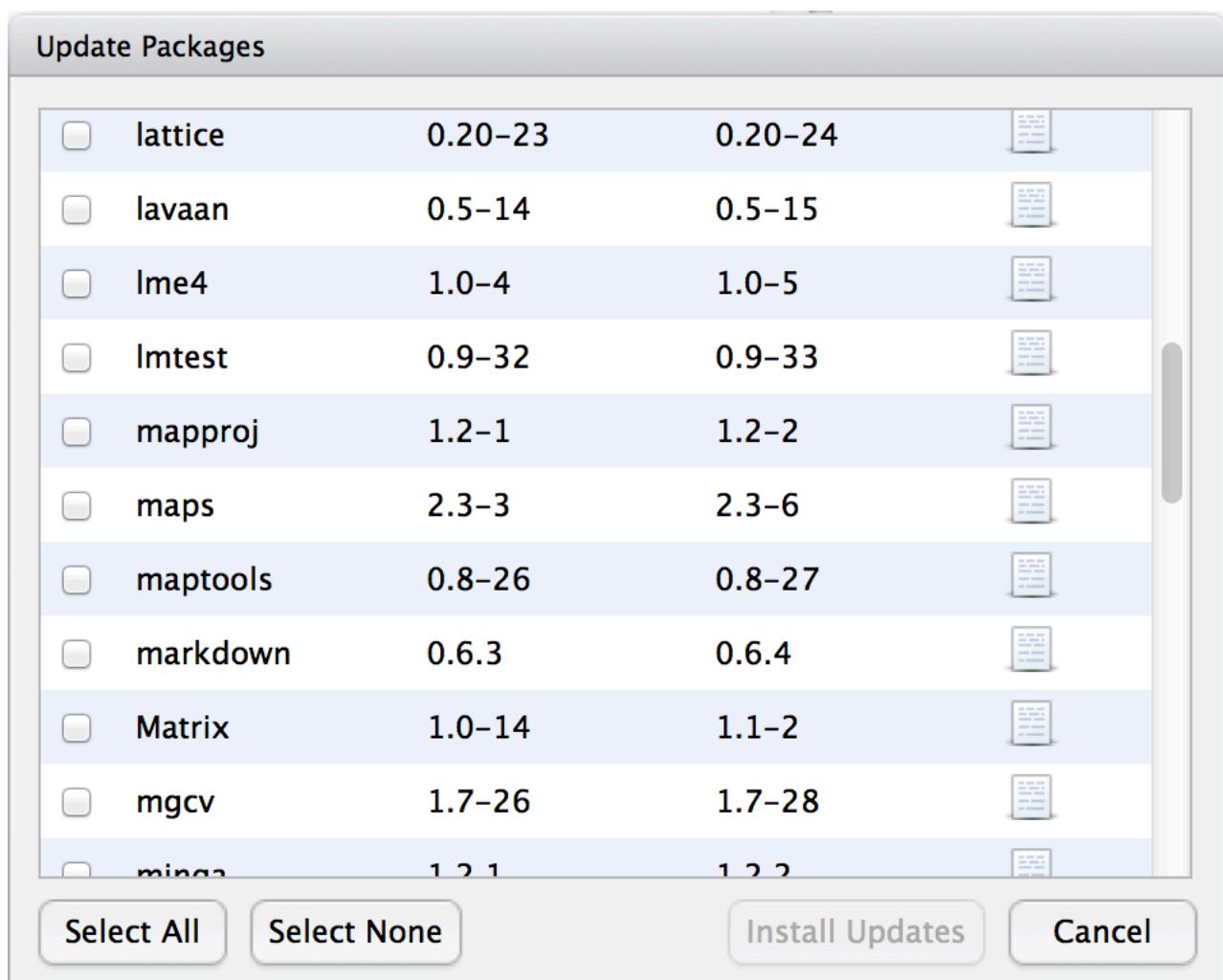


Figure 4.4: The Rstudio dialog box for updating packages

4.2.7 What packages does this book use?

There are several packages that I make use of in this book. The most prominent ones are:

- **lsr**. This is the *Learning Statistics with R* package that accompanies this book. It doesn't have a lot of interesting high-powered tools: it's just a small collection of handy little things that I think can be useful to novice users. As you get more comfortable with R this package should start to feel pretty useless to you.
- **psych**. This package, written by William Revelle, includes a lot of tools that are of particular use to psychologists. In particular, there's several functions that are particularly convenient for producing analyses or summaries that are very common in psych, but less common in other disciplines.
- **car**. This is the *Companion to Applied Regression* package, which accompanies the excellent book of the same name by (Fox and Weisberg, 2011). It provides a lot of very powerful tools, only some of which we'll touch in this book.

Besides these three, there are a number of packages that I use in a more limited fashion: **gplots**, **sciplot**, **foreign**, **effects**, **R.matlab**, **gdata**, **lmtest**, and probably one or two others that I've missed. There are also a number of packages that I refer to but don't actually use in this book, such as **reshape**, **compute.es**, **HistData** and **multcomp** among others. Finally, there are a number of packages that provide more advanced tools that I hope to talk about in future versions of the book, such as **sem**, **ez**, **nlme** and **lme4**. In any case, whenever I'm using a function that isn't in the core packages, I'll make sure to note this in the text.

4.3 Managing the workspace

Let's suppose that you're reading through this book, and what you're doing is sitting down with it once a week and working through a whole chapter in each sitting. Not only that, you've been following my advice and typing in all these commands into R. So far during this chapter, you'd have typed quite a few commands, although the only ones that actually involved creating variables were the ones you typed during Section 4.1. As a result, you currently have three variables; **seeker**, **lover**, and **keeper**. These three variables are the contents of your **workspace**, also referred to as the **global environment**. The workspace is a key concept in R, so in this section we'll talk a lot about what it is and how to manage its contents.

4.3.1 Listing the contents of the workspace

The first thing that you need to know how to do is examine the contents of the workspace. If you're using Rstudio, you will probably find that the easiest way to do this is to use the "Environment" panel in the top right hand corner. Click on that, and you'll see a list that looks very much like the one shown in Figures 4.5 and 4.6. If you're using the command line, then the **objects()** function may come in handy:

```
objects()
```

```
## [1] "any.sales.this.month"   "berkeley"           "berkeley.small"
## [4] "coef"                  "days.per.month"      "february.sales"
## [7] "greeting"              "is.the.Party.correct" "keeper"
## [10] "lover"                 "months"              "profit"
## [13] "revenue"                "royalty"             "sales"
## [16] "sales.by.month"        "seeker"              "simpson"
## [19] "stock.levels"          "x"                   "xlu"
```

Values				
keeper	8.53953945			
lover	2.7183			
seeker	3.1415			

Figure 4.5: The Rstudio Environment panel shows you the contents of the workspace. The view shown above is the list view. To switch to the grid view, click on the menu item on the top right that currently reads list. Select grid from the dropdown menu, and then it will switch to a view like the one shown in the other workspace figure.

<input type="checkbox"/>	Name	Type	Length	Size	Value
<input type="checkbox"/>	keeper	numeric	1	48 B	8.53953945
<input type="checkbox"/>	lover	numeric	1	48 B	2.7183
<input type="checkbox"/>	seeker	numeric	1	48 B	3.1415

Figure 4.6: The Rstudio "Environment" panel shows you the contents of the workspace. Compare this "grid" view to the "list" earlier

Of course, in the true R tradition, the `objects()` function has a lot of fancy capabilities that I'm glossing over in this example. Moreover there are also several other functions that you can use, including `ls()` which is pretty much identical to `objects()`, and `ls.str()` which you can use to get a fairly detailed description of all the variables in the workspace. In fact, the `lsr` package actually includes its own function that you can use for this purpose, called `who()`. The reason for using the `who()` function is pretty straightforward: in my everyday work I find that the output produced by the `objects()` command isn't *quite* informative enough, because the only thing it prints out is the name of each variable; but the `ls.str()` function is *too* informative, because it prints out a lot of additional information that I really don't like to look at. The `who()` function is a compromise between the two. First, now that we've got the `lsr` package installed, we need to load it:

```
library(lsr)
```

and now we can use the `who()` function:

```
who()
```

	-- Name --	-- Class --	-- Size --
##	any.sales.this.month	logical	12
##	berkeley	data.frame	39 x 3
##	berkeley.small	data.frame	46 x 2
##	coef	numeric	2
##	days.per.month	numeric	12
##	february.sales	numeric	1
##	greeting	character	1
##	is.the.Party.correct	logical	1
##	keeper	numeric	1
##	lover	numeric	1
##	months	character	12
##	profit	numeric	12
##	revenue	numeric	1
##	royalty	numeric	1
##	sales	numeric	1
##	sales.by.month	numeric	12
##	seeker	numeric	1
##	simpson	matrix	6 x 5
##	stock.levels	character	12
##	x	logical	3
##	xlu	numeric	1

As you can see, the `who()` function lists all the variables and provides some basic information about what kind of variable each one is and how many elements it contains. Personally, I find this output much easier more useful than the very compact output of the `objects()` function, but less overwhelming than the extremely verbose `ls.str()` function. Throughout this book you'll see me using the `who()` function a lot. You don't have to use it yourself: in fact, I suspect you'll find it easier to look at the Rstudio environment panel. But for the purposes of writing a textbook I found it handy to have a nice text based description: otherwise there would be about another 100 or so screenshots added to the book.⁸

4.3.2 Removing variables from the workspace

Looking over that list of variables, it occurs to me that I really don't need them any more. I created them originally just to make a point, but they don't serve any useful purpose anymore, and now I want to get rid

⁸This would be especially annoying if you're reading an electronic copy of the book because the text displayed by the `who()` function is searchable, whereas text shown in a screen shot isn't!

of them. I'll show you how to do this, but first I want to warn you – there's no “undo” option for variable removal. Once a variable is removed, it's gone forever unless you save it to disk. I'll show you how to do *that* in Section 4.5, but quite clearly we have no need for these variables at all, so we can safely get rid of them.

In Rstudio, the easiest way to remove variables is to use the environment panel. Assuming that you're in grid view (i.e., Figure 4.6), check the boxes next to the variables that you want to delete, then click on the “Clear” button at the top of the panel. When you do this, Rstudio will show a dialog box asking you to confirm that you really do want to delete the variables. It's always worth checking that you really do, because as Rstudio is at pains to point out, you can't undo this. Once a variable is deleted, it's gone.⁹ In any case, if you click “yes”, that variable will disappear from the workspace: it will no longer appear in the environment panel, and it won't show up when you use the `who()` command.

Suppose you don't access to Rstudio, and you still want to remove variables. This is where the **`remove`** function `rm()` comes in handy. The simplest way to use `rm()` is just to type in a (comma separated) list of all the variables you want to remove. Let's say I want to get rid of `seeker` and `lover`, but I would like to keep `keeper`. To do this, all I have to do is type:

```
rm( seeker, lover )
```

There's no visible output, but if I now inspect the workspace

```
who()
```

```
##   -- Name --      -- Class --  -- Size --
## any.sales.this.month logical     12
## berkeley             data.frame 39 x 3
## berkeley.small       data.frame 46 x 2
## coef                numeric    2
## days.per.month      numeric    12
## february.sales      numeric    1
## greeting            character  1
## is.the.Party.correct logical    1
## keeper               numeric    1
## months              character 12
## profit              numeric    12
## revenue             numeric    1
## royalty             numeric    1
## sales               numeric    1
## sales.by.month      numeric    12
## simpson             matrix     6 x 5
## stock.levels        character 12
## x                   logical    3
## xlu                 numeric    1
```

I see that there's only the `keeper` variable left. As you can see, `rm()` can be very handy for keeping the workspace tidy.

4.4 Navigating the file system

In this section I talk a little about how R interacts with the file system on your computer. It's not a terribly interesting topic, but it's useful. As background to this discussion, I'll talk a bit about how file system

⁹Mind you, all that means is that it's been removed from the workspace. If you've got the data saved to file somewhere, then that *file* is perfectly safe.

locations work in Section 4.4.1. Once upon a time *everyone* who used computers could safely be assumed to understand how the file system worked, because it was impossible to successfully use a computer if you didn't! However, modern operating systems are much more "user friendly", and as a consequence of this they go to great lengths to hide the file system from users. So these days it's not at all uncommon for people to have used computers most of their life and not be familiar with the way that computers organise files. If you already know this stuff, skip straight to Section 4.4.2. Otherwise, read on. I'll try to give a brief introduction that will be useful for those of you who have never been forced to learn how to navigate around a computer using a DOS or UNIX shell.

4.4.1 The file system itself

In this section I describe the basic idea behind file locations and file paths. Regardless of whether you're using Window, Mac OS or Linux, every file on the computer is assigned a (fairly) human readable address, and every address has the same basic structure: it describes a *path* that starts from a *root* location , through a series of *folders* (or if you're an old-school computer user, *directories*), and finally ends up at the file.

On a Windows computer the root is the physical drive¹⁰ on which the file is stored, and for most home computers the name of the hard drive that stores all your files is C: and therefore most file names on Windows begin with C:. After that comes the folders, and on Windows the folder names are separated by a \ symbol. So, the complete path to this book on my Windows computer might be something like this:

C:\Users\danRbook\LSR.pdf

and what that *means* is that the book is called LSR.pdf, and it's in a folder called book which itself is in a folder called dan which itself is ... well, you get the idea. On Linux, Unix and Mac OS systems, the addresses look a little different, but they're more or less identical in spirit. Instead of using the backslash, folders are separated using a forward slash, and unlike Windows, they don't treat the physical drive as being the root of the file system. So, the path to this book on my Mac might be something like this:

/Users/dan/Rbook/LSR.pdf

So that's what we mean by the "path" to a file. The next concept to grasp is the idea of a ***working directory*** and how to change it. For those of you who have used command line interfaces previously, this should be obvious already. But if not, here's what I mean. The working directory is just "whatever folder I'm currently looking at". Suppose that I'm currently looking for files in Explorer (if you're using Windows) or using Finder (on a Mac). The folder I currently have open is my user directory (i.e., C:\Users\dan or /Users/dan). That's my current working directory.

The fact that we can imagine that the program is "in" a particular directory means that we can talk about moving *from* our current location *to* a new one. What that means is that we might want to specify a new location in relation to our current location. To do so, we need to introduce two new conventions. Regardless of what operating system you're using, we use . to refer to the current working directory, and .. to refer to the directory above it. This allows us to specify a path to a new location in relation to our current location, as the following examples illustrate. Let's assume that I'm using my Windows computer, and my working directory is C:\Users\danRbook). The table below shows several addresses in relation to my current one:

There's one last thing I want to call attention to: the ~ directory. I normally wouldn't bother, but R makes reference to this concept sometimes. It's quite common on computers that have multiple users to define ~ to be the user's home directory. On my Mac, for instance, the home directory ~ for the "dan" user is \Users\dan\. And so, not surprisingly, it is possible to define other directories in terms of their relationship to the home directory. For example, an alternative way to describe the location of the LSR.pdf file on my Mac would be

¹⁰Well, the partition, technically.

Table 4.1: Basic arithmetic operations in R. These five operators are used very frequently throughout the text, so it's important to be familiar with them at the outset.

absolute path (i.e., from root)	relative path (i.e. from C:\Users\ danRbook)
C:\\\\Users\\\\dan	..
C:\\\\Users	..\\\\.. \\\\
C:\\\\Users\\\\danRbook\\\\source	.\\\\source
C:\\\\Users\\\\dan\\\\nerdstuff	..\\\\nerdstuff

~Rbook\LSR.pdf

That's about all you really need to know about file paths. And since this section already feels too long, it's time to look at how to navigate the file system in R.

4.4.2 Navigating the file system using the R console

In this section I'll talk about how to navigate this file system from within R itself. It's not particularly user friendly, and so you'll probably be happy to know that Rstudio provides you with an easier method, and I will describe it in Section 4.4.4. So in practice, you won't *really* need to use the commands that I babble on about in this section, but I do think it helps to see them in operation at least once before forgetting about them forever.

Okay, let's get started. When you want to load or save a file in R it's important to know what the working directory is. You can find out by using the `getwd()` command. For the moment, let's assume that I'm using Mac OS or Linux, since there's some subtleties to Windows. Here's what happens:

```
getwd()
## [1] "/Users/dan"
```

We can change the working directory quite easily using `setwd()`. The `setwd()` function has only the one argument, `dir`, is a character string specifying a path to a directory, or a path relative to the working directory. Since I'm currently located at `/Users/dan`, the following two are equivalent:

```
setwd("/Users/dan/Rbook/data")
setwd("./Rbook/data")
```

Now that we're here, we can type `list.files()` command to get a listing of all the files in that directory. Since this is the directory in which I store all of the data files that we'll use in this book, here's what we get as the result:

```
list.files()
## [1] "afl124.Rdata"           "aflsmall.Rdata"          "aflsmall2.Rdata"
## [4] "agpp.Rdata"             "all.zip"                 "annoying.Rdata"
## [7] "anscombesquartet.Rdata" "awesome.Rdata"            "awesome2.Rdata"
## [10] "booksales.csv"          "booksales.Rdata"          "booksales2.csv"
## [13] "cakes.Rdata"            "cards.Rdata"              "chapek9.Rdata"
## [16] "chico.Rdata"            "clinicaltrial_old.Rdata" "clinicaltrial.Rdata"
## [19] "coffee.Rdata"            "drugs.wmc.rt.Rdata"      "dwr_all.Rdata"
## [22] "effort.Rdata"           "happy.Rdata"              "harpo.Rdata"
## [25] "harpo2.Rdata"           "likert.Rdata"             "nightgarden.Rdata"
```

```
## [28] "nightgarden2.Rdata"      "parenthood.Rdata"       "parenthood2.Rdata"
## [31] "randomness.Rdata"        "repeated.Rdata"         "rtfm.Rdata"
## [34] "salem.Rdata"             "zeppo.Rdata"
```

Not terribly exciting, I'll admit, but it's useful to know about. In any case, there's only one more thing I want to make a note of, which is that R also makes use of the home directory. You can find out what it is by using the `path.expand()` function, like this:

```
path.expand("~")
## [1] "/Users/dan"
```

You can change the user directory if you want, but we're not going to make use of it very much so there's no reason to. The only reason I'm even bothering to mention it at all is that when you use Rstudio to open a file, you'll see output on screen that defines the path to the file relative to the `#~#` directory. I'd prefer you not to be confused when you see it.¹¹

4.4.3 Why do the Windows paths use the wrong slash?

Let's suppose I'm on Windows. As before, I can find out what my current working directory is like this:

```
getwd()
## [1] "C:/Users/dan/"
```

This seems about right, but you might be wondering why R is displaying a Windows path using the wrong type of slash. The answer is slightly complicated, and has to do with the fact that R treats the \ character as “special” (see Section 7.8.7). If you're deeply wedded to the idea of specifying a path using the Windows style slashes, then what you need to do is to type / whenever you mean \. In other words, if you want to specify the working directory on a Windows computer, you need to use one of the following commands:

```
setwd( "C:/Users/dan" )
setwd( "C:\\\\Users\\\\dan" )
```

It's kind of annoying to have to do it this way, but as you'll see later on in Section 7.8.7 it's a necessary evil. Fortunately, as we'll see in the next section, Rstudio provides a much simpler way of changing directories...

4.4.4 Navigating the file system using the Rstudio file panel

Although I think it's important to understand how all this command line stuff works, in many (maybe even most) situations there's an easier way. For our purposes, the easiest way to navigate the file system is to make use of Rstudio's built in tools. The “file” panel – the lower right hand area in Figure 4.7 – is actually a pretty decent file browser. Not only can you just point and click on the names to move around the file system, you can also use it to set the working directory, and even load files.

Here's what you need to do to change the working directory using the file panel. Let's say I'm looking at the actual screen shown in Figure 4.7. At the top of the file panel you see some text that says “Home > Rbook > data”. What that means is that it's *displaying* the files that are stored in the

¹¹One additional thing worth calling your attention to is the `file.choose()` function. Suppose you want to load a file and you don't quite remember where it is, but would like to browse for it. Typing `file.choose()` at the command line will open a window in which you can browse to find the file; when you click on the file you want, R will print out the full path to that file. This is kind of handy.

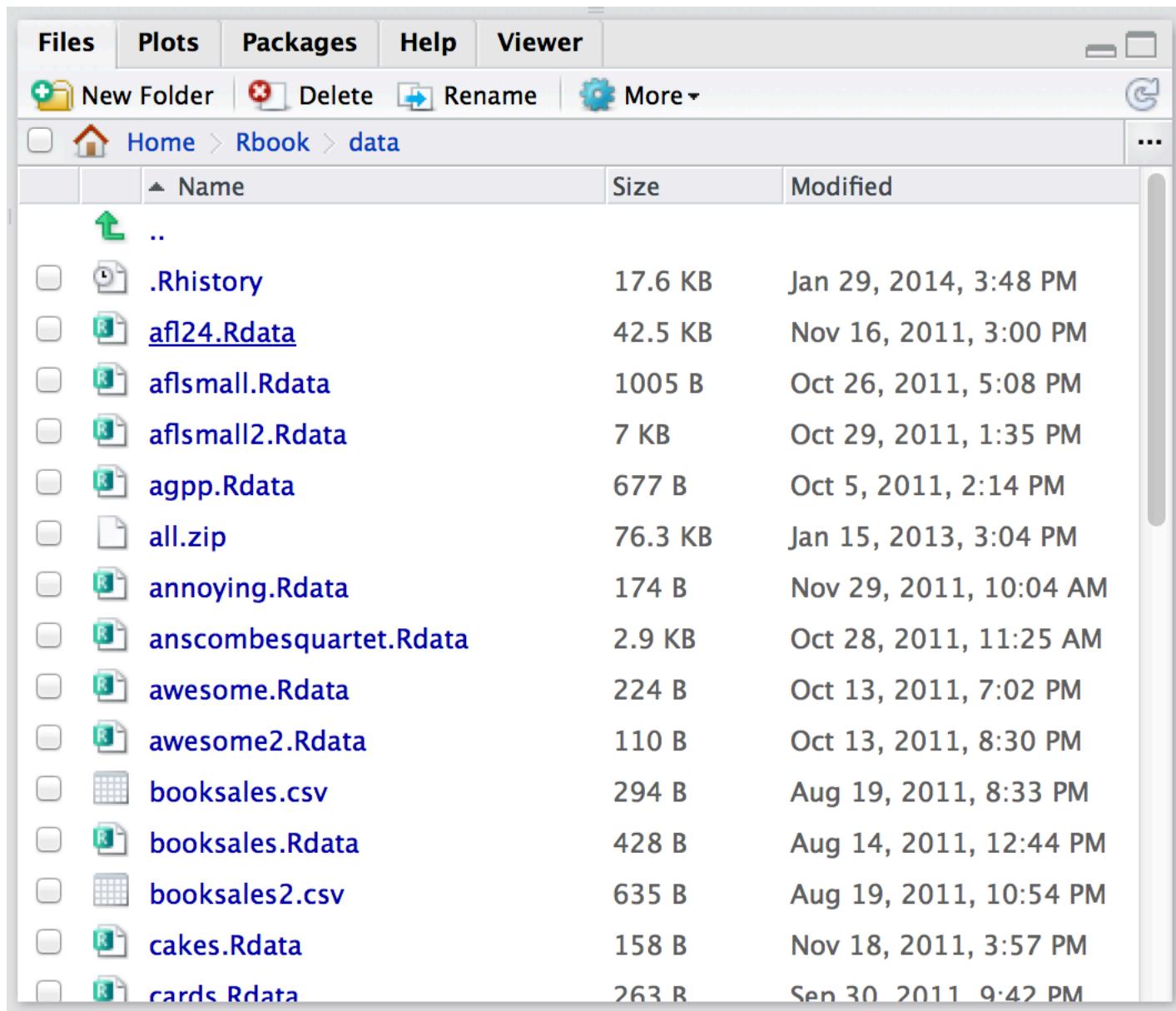


Figure 4.7: The "file panel" is the area shown in the lower right hand corner. It provides a very easy way to browse and navigate your computer using R. See main text for details.

```
/Users/dan/Rbook/data
```

directory on my computer. It does *not* mean that this is the R working directory. If you want to change the R working directory, using the file panel, you need to click on the button that reads “More”. This will bring up a little menu, and one of the options will be “Set as Working Directory”. If you select that option, then R really will change the working directory. You can tell that it has done so because this command appears in the console:

```
setwd("~/Rbook/data")
```

In other words, Rstudio sends a command to the R console, exactly as if you’d typed it yourself. The file panel can be used to do other things too. If you want to move “up” to the parent folder (e.g., from /Users/dan/Rbook/data to /Users/dan/Rbook click on the “..” link in the file panel. To move to a subfolder, click on the name of the folder that you want to open. You can open some types of file by clicking on them. You can delete files from your computer using the “delete” button, rename them with the “rename” button, and so on.

As you can tell, the file panel is a very handy little tool for navigating the file system. But it can do more than just navigate. As we’ll see later, it can be used to open files. And if you look at the buttons and menu options that it presents, you can even use it to rename, delete, copy or move files, and create new folders. However, since most of that functionality isn’t critical to the basic goals of this book, I’ll let you discover those on your own.

4.5 Loading and saving data

There are several different types of files that are likely to be relevant to us when doing data analysis. There are three in particular that are especially important from the perspective of this book:

- *Workspace files* are those with a .Rdata file extension. This is the standard kind of file that R uses to store data and variables. They’re called “workspace files” because you can use them to save your whole workspace.
- *Comma separated value (CSV) files* are those with a .csv file extension. These are just regular old text files, and they can be opened with almost any software. It’s quite typical for people to store data in CSV files, precisely because they’re so simple.
- *Script files* are those with a .R file extension. These aren’t data files at all; rather, they’re used to save a collection of commands that you want R to execute later. They’re just text files, but we won’t make use of them until Chapter 8.

There are also several other types of file that R makes use of,¹² but they’re not really all that central to our interests. There are also several other kinds of data file that you might want to import into R. For instance, you might want to open Microsoft Excel spreadsheets (.xlsx files), or data files that have been saved in the native file formats for other statistics software, such as SPSS, SAS, Minitab, Stata or Systat. Finally, you might have to handle databases. R tries hard to play nicely with other software, so it has tools that let you open and work with any of these and many others. I’ll discuss some of these other possibilities elsewhere in this book (Section 7.9), but for now I want to focus primarily on the two kinds of data file that you’re most likely to need: .Rdata files and .csv files. In this section I’ll talk about how to load a workspace file, how to import data from a CSV file, and how to save your workspace to a workspace file. Throughout this section I’ll first describe the (sometimes awkward) R commands that do all the work, and then I’ll show you the (much easier) way to do it using Rstudio.

¹²Notably those with .rda, .Rd, .Rhistry, .rdb and .rdx extensions

4.5.1 Loading workspace files using R

When I used the `list.files()` command to list the contents of the `/Users/dan/Rbook/data` directory (in Section 4.4.2), the output referred to a file called `booksales.Rdata`. Let's say I want to load the data from this file into my workspace. The way I do this is with the `load()` function. There are two arguments to this function, but the only one we're interested in is

- `file`. This should be a character string that specifies a path to the file that needs to be loaded. You can use an absolute path or a relative path to do so.

Using the absolute file path, the command would look like this:

```
load( file = "/Users/dan/Rbook/data/booksales.Rdata" )
```

but this is pretty lengthy. Given that the working directory (remember, we changed the directory at the end of Section 4.4.4) is `/Users/dan/Rbook/data`, I could use a relative file path, like so:

```
load( file = "../data/booksales.Rdata" )
```

However, my preference is usually to change the working directory first, and *then* load the file. What that would look like is this:

```
setwd( "../data" )           # move to the data directory
load( "booksales.Rdata" )    # load the data
```

If I were then to type `who()` I'd see that there are several new variables in my workspace now. Throughout this book, whenever you see me loading a file, I will assume that the file is actually stored in the working directory, or that you've changed the working directory so that R is pointing at the directory that contains the file. Obviously, *you* don't need type that command yourself: you can use the Rstudio file panel to do the work.

4.5.2 Loading workspace files using Rstudio

Okay, so how do we open an `.Rdata` file using the Rstudio file panel? It's terribly simple. First, use the file panel to find the folder that contains the file you want to load. If you look at Figure 4.7, you can see that there are several `.Rdata` files listed. Let's say I want to load the `booksales.Rdata` file. All I have to do is click on the file name. Rstudio brings up a little dialog box asking me to confirm that I do want to load this file. I click yes. The following command then turns up in the console,

```
load("~/Rbook/data/booksales.Rdata")
```

and the new variables will appear in the workspace (you'll see them in the Environment panel in Rstudio, or if you type `who()`). So easy it barely warrants having its own section.

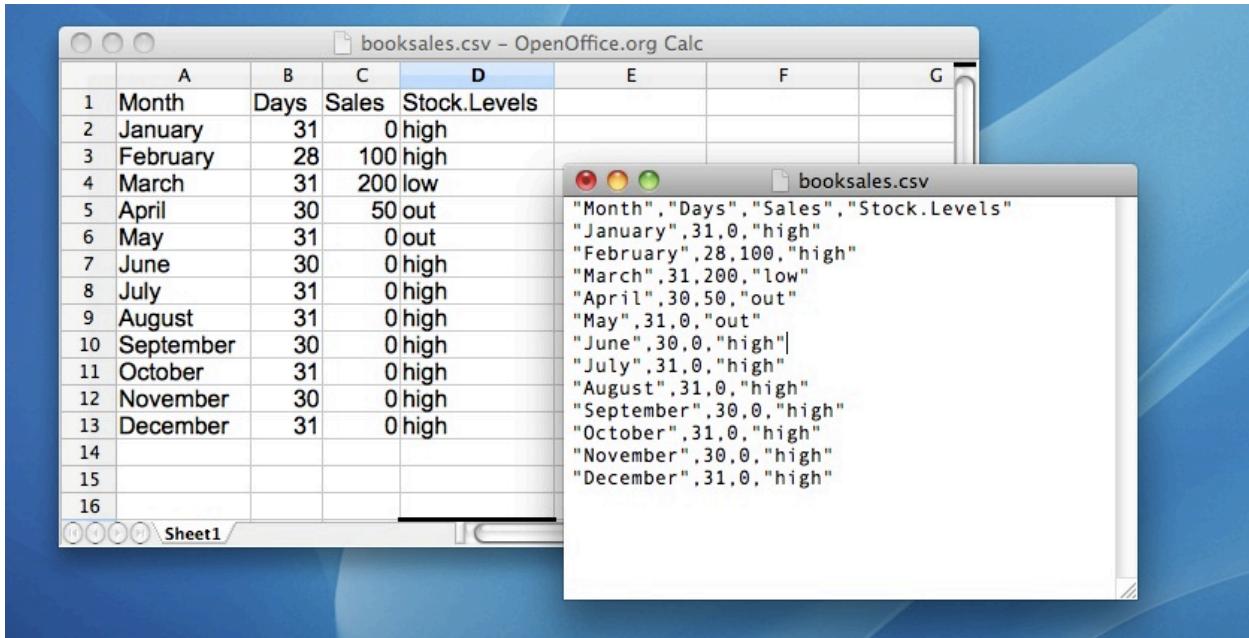


Figure 4.8: The booksales.csv data file. On the left, I've opened the file in using a spreadsheet program (OpenOffice), which shows that the file is basically a table. On the right, the same file is open in a standard text editor (theTextEdit program on a Mac), which shows how the file is formatted. The entries in the table are wrapped in quote marks and separated by commas.

4.5.3 Importing data from CSV files using `loadingcsv`

One quite commonly used data format is the humble “comma separated value” file, also called a CSV file, and usually bearing the file extension .csv. CSV files are just plain old-fashioned text files, and what they store is basically just a table of data. This is illustrated in Figure 4.8, which shows a file called booksales.csv that I’ve created. As you can see, each row corresponds to a variable, and each row represents the book sales data for one month. The first row doesn’t contain actual data though: it has the names of the variables.

If Rstudio were not available to you, the easiest way to open this file would be to use the `read.csv()` function.¹³ This function is pretty flexible, and I’ll talk a lot more about its capabilities in Section 7.9 for more details, but for now there’s only two arguments to the function that I’ll mention:

- `file`. This should be a character string that specifies a path to the file that needs to be loaded. You can use an absolute path or a relative path to do so.
- `header`. This is a logical value indicating whether or not the first row of the file contains variable names. The default value is `TRUE`.

Therefore, to import the CSV file, the command I need is:

```
books <- read.csv( file = "booksales.csv" )
```

There are two very important points to notice here. Firstly, notice that I *didn’t* try to use the `load()` function, because that function is only meant to be used for .Rdata files. If you try to use `load()` on other types of data, you get an error. Secondly, notice that when I imported the CSV file I assigned the result to

¹³In a lot of books you’ll see the `read.table()` function used for this purpose instead of `read.csv()`. They’re more or less identical functions, with the same arguments and everything. They differ only in the default values.

a variable, which I imaginatively called `books`.¹⁴ file. There's a reason for this. The idea behind an `.Rdata` file is that it stores a whole workspace. So, if you had the ability to look inside the file yourself you'd see that the data file keeps track of all the variables and their names. So when you `load()` the file, R restores all those original names. CSV files are treated differently: as far as R is concerned, the CSV only stores *one* variable, but that variable is big table. So when you import that table into the workspace, R expects *you* to give it a name.] Let's have a look at what we've got:

```
print( books )
```

```
##      Month Days Sales Stock.Levels
## 1   January    31     0      high
## 2 February    28   100      high
## 3   March     31   200       low
## 4   April     30     50      out
## 5    May      31     0      out
## 6   June      30     0      high
## 7   July      31     0      high
## 8  August     31     0      high
## 9 September    30     0      high
## 10 October    31     0      high
## 11 November   30     0      high
## 12 December   31     0      high
```

Clearly, it's worked, but the format of this output is a bit unfamiliar. We haven't seen anything like this before. What you're looking at is a *data frame*, which is a very important kind of variable in R, and one I'll discuss in Section 4.8. For now, let's just be happy that we imported the data and that it looks about right.

4.5.4 Importing data from CSV files using Rstudio

Yet again, it's easier in Rstudio. In the environment panel in Rstudio you should see a button called "Import Dataset". Click on that, and it will give you a couple of options: select the "From Text File..." option, and it will open up a very familiar dialog box asking you to select a file: if you're on a Mac, it'll look like the usual Finder window that you use to choose a file; on Windows it looks like an Explorer window. An example of what it looks like on a Mac is shown in Figure 4.9. I'm assuming that you're familiar with your own computer, so you should have no problem finding the CSV file that you want to import! Find the one you want, then click on the "Open" button. When you do this, you'll see a window that looks like the one in Figure 4.10.

The import data set window is relatively straightforward to understand.

In the top left corner, you need to type the name of the variable you R to create. By default, that will be the same as the file name: our file is called `booksales.csv`, so Rstudio suggests the name `booksales`. If you're happy with that, leave it alone. If not, type something else. Immediately below this are a few things that you can tweak to make sure that the data gets imported correctly:

- Heading. Does the first row of the file contain raw data, or does it contain headings for each variable? The `booksales.csv` file has a header at the top, so I selected "yes".
- Separator. What character is used to separate different entries? In most CSV files this will be a comma (it is "comma separated" after all). But you can change this if your file is different.
- Decimal. What character is used to specify the decimal point? In English speaking countries, this is almost always a period (i.e., `.`). That's not universally true: many European countries use a comma. So you can change that if you need to.

¹⁴Note that I didn't do this in my earlier example when loading the `.Rdata`

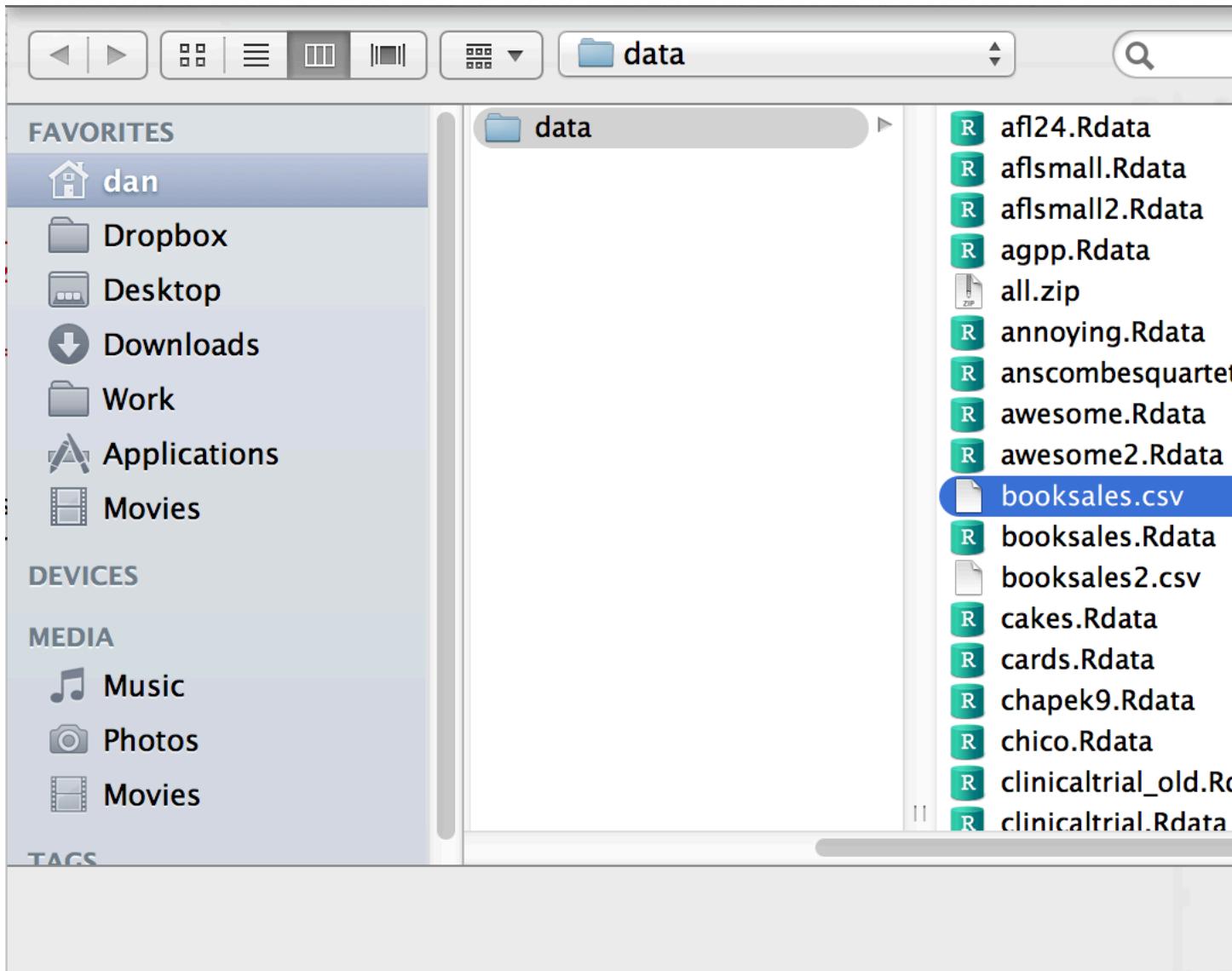


Figure 4.9: A dialog box on a Mac asking you to select the CSV file R should try to import. Mac users will recognise this immediately: it's the usual way in which a Mac asks you to find a file. Windows users won't see this: they'll see the usual explorer window that Windows always gives you when it wants you to select a file.

Import Dataset

Name	booksales
Heading	<input checked="" type="radio"/> Yes <input type="radio"/> No
Separator	Comma
Decimal	Period
Quote	Double quote ("")

Input File

```
"Month", "Days", "Sales", "Stock.Levels"  
"January", 31, 0, "high"  
"February", 28, 100, "high"  
"March", 31, 200, "low"  
"April", 30, 50, "out"  
"May", 31, 0, "out"  
"June", 30, 0, "high"  
"July", 31, 0, "high"  
"August", 31, 0, "high"  
"September", 30, 0, "high"  
"October", 31, 0, "high"  
"November", 30, 0, "high"  
"December", 31, 0, "high"
```

Data Frame

Month	Days	Sales	Stock.Levels
January	31	0	high
February	28	100	high
March	31	200	low
April	30	50	out
May	31	0	out
June	30	0	high
July	31	0	high
August	31	0	high
September	30	0	high
October	31	0	high
November	30	0	high
December	31	0	high

Import

Figure 4.10: The Rstudio window for importing a CSV file into R

- Quote. What character is used to denote a block of text? That's usually going to be a double quote mark. It is for the `booksales.csv` file, so that's what I selected.

The nice thing about the Rstudio window is that it shows you the raw data file at the top of the window, and it shows you a preview of the data at the bottom. If the data at the bottom doesn't look right, try changing some of the settings on the left hand side. Once you're happy, click "Import". When you do, two commands appear in the R console:

```
booksales <- read.csv("~/Rbook/data/booksales.csv")
View(booksales)
```

The first of these commands is the one that loads the data. The second one will display a pretty table showing the data in Rstudio.

4.5.5 Saving a workspace file using `save`

Not surprisingly, saving data is very similar to loading data. Although Rstudio provides a simple way to save files (see below), it's worth understanding the actual commands involved. There are two commands you can use to do this, `save()` and `save.image()`. If you're happy to save *all* of the variables in your workspace into the data file, then you should use `save.image()`. And if you're happy for R to save the file into the current working directory, all you have to do is this:

```
save.image( file = "myfile.Rdata" )
```

Since `file` is the first argument, you can shorten this to `save.image("myfile.Rdata")`; and if you want to save to a different directory, then (as always) you need to be more explicit about specifying the path to the file, just as we discussed in Section 4.4. Suppose, however, I have several variables in my workspace, and I only want to save some of them. For instance, I might have this as my workspace:

```
who()
##   -- Name --   -- Class --   -- Size --
##   data         data.frame     3 x 2
##   handy        character      1
##   junk         numeric       1
```

I want to save `data` and `handy`, but not `junk`. But I don't want to delete `junk` right now, because I want to use it for something else later on. This is where the `save()` function is useful, since it lets me indicate exactly which variables I want to save. Here is one way I can use the `save` function to solve my problem:

```
save(data, handy, file = "myfile.Rdata")
```

Importantly, you *must* specify the name of the `file` argument. The reason is that if you don't do so, R will think that `"myfile.Rdata"` is actually a *variable* that you want to save, and you'll get an error message. Finally, I should mention a second way to specify which variables the `save()` function should save, which is to use the `list` argument. You do so like this:

```
save.me <- c("data", "handy")    # the variables to be saved
save( file = "booksales2.Rdata", list = save.me )    # the command to save them
```

4.5.6 Saving a workspace file using Rstudio

Rstudio allows you to save the workspace pretty easily. In the environment panel (Figures 4.5 and 4.6) you can see the “save” button. There’s no text, but it’s the same icon that gets used on every computer everywhere: it’s the one that looks like a floppy disk. You know, those things that haven’t been used in about 20 years. Alternatively, go to the “Session” menu and click on the “Save Workspace As...” option.¹⁵ This will bring up the standard “save” dialog box for your operating system (e.g., on a Mac it’ll look a little bit like the loading dialog box in Figure 4.9). Type in the name of the file that you want to save it to, and all the variables in your workspace will be saved to disk. You’ll see an R command like this one

```
save.image("~/Desktop/Untitled.RData")
```

Pretty straightforward, really.

4.5.7 Other things you might want to save

Until now, we’ve talked mostly about loading and saving *data*. Other things you might want to save include:

- *The output.* Sometimes you might also want to keep a copy of all your interactions with R, including everything that you typed in and everything that R did in response. There are some functions that you can use to get R to write its output to a file rather than to print onscreen (e.g., `sink()`), but to be honest, if you do want to save the R output, the easiest thing to do is to use the mouse to select the relevant text in the R console, go to the “Edit” menu in Rstudio and select “Copy”. The output has now been copied to the clipboard. Now open up your favourite text editor or word processing software, and paste it. And you’re done. However, this will only save the contents of the console, not the plots you’ve drawn (assuming you’ve drawn some). We’ll talk about saving images later on.
- *A script.* While it is possible – and sometimes handy – to save the R output as a method for keeping a copy of your statistical analyses, another option that people use a lot (especially when you move beyond simple “toy” analyses) is to write *scripts*. A script is a text file in which you write out all the commands that you want R to run. You can write your script using whatever software you like. In real world data analysis writing scripts is a key skill – and as you become familiar with R you’ll probably find that most of what you do involves scripting rather than typing commands at the R prompt. However, you won’t need to do much scripting initially, so we’ll leave that until Chapter 8.

4.6 Useful things to know about variables

In Chapter 3 I talked a lot about variables, how they’re assigned and some of the things you can do with them, but there’s a lot of additional complexities. That’s not a surprise of course. However, some of those issues are worth drawing your attention to now. So that’s the goal of this section; to cover a few extra topics. As a consequence, this section is basically a bunch of things that I want to briefly mention, but don’t really fit in anywhere else. In short, I’ll talk about several different issues in this section, which are only loosely connected to one another.

4.6.1 Special values

The first thing I want to mention are some of the “special” values that you might see R produce. Most likely you’ll see them in situations where you were expecting a number, but there are quite a few other ways you

¹⁵A word of warning: what you *don’t* want to do is use the “File” menu. If you look in the “File” menu you will see “Save” and “Save As...” options, but they don’t save the workspace. Those options are used for dealing with *scripts*, and so they’ll produce .R files. We won’t get to those until Chapter 8.

can encounter them. These values are `Inf`, `NaN`, `NA` and `NULL`. These values can crop up in various different places, and so it's important to understand what they mean.

- *Infinity (Inf)*. The easiest of the special values to explain is `Inf`, since it corresponds to a value that is infinitely large. You can also have `-Inf`. The easiest way to get `Inf` is to divide a positive number by 0:

```
1 / 0
```

```
## [1] Inf
```

In most real world data analysis situations, if you're ending up with infinite numbers in your data, then something has gone awry. Hopefully you'll never have to see them.

- *Not a Number (NaN)*. The special value of `NaN` is short for “not a number”, and it's basically a reserved keyword that means “there isn't a mathematically defined number for this”. If you can remember your high school maths, remember that it is conventional to say that $0/0$ doesn't have a proper answer: mathematicians would say that $0/0$ is *undefined*. R says that it's not a number:

```
0 / 0
```

```
## [1] NaN
```

Nevertheless, it's still treated as a “numeric” value. To oversimplify, `NaN` corresponds to cases where you asked a proper numerical question that genuinely has *no meaningful answer*.

- *Not available (NA)*. `NA` indicates that the value that is “supposed” to be stored here is missing. To understand what this means, it helps to recognise that the `NA` value is something that you're most likely to see when analysing data from real world experiments. Sometimes you get equipment failures, or you lose some of the data, or whatever. The point is that some of the information that you were “expecting” to get from your study is just plain missing. Note the difference between `NA` and `NaN`. For `NaN`, we really do know what's supposed to be stored; it's just that it happens to correspond to something like $0/0$ that doesn't make any sense at all. In contrast, `NA` indicates that we actually don't know what was supposed to be there. The information is *missing*.
- *No value (NULL)*. The `NULL` value takes this “absence” concept even further. It basically asserts that the variable genuinely has no value whatsoever. This is quite different to both `NaN` and `NA`. For `NaN` we actually know what the value is, because it's something insane like $0/0$. For `NA`, we believe that there is supposed to be a value “out there”, but a dog ate our homework and so we don't quite know what it is. But for `NULL` we strongly believe that there is *no value at all*.

4.6.2 Assigning names to vector elements

One thing that is sometimes a little unsatisfying about the way that R prints out a vector is that the elements come out unlabelled. Here's what I mean. Suppose I've got data reporting the quarterly profits for some company. If I just create a no-frills vector, I have to rely on memory to know which element corresponds to which event. That is:

```
profit <- c( 3.1, 0.1, -1.4, 1.1 )
profit
```

```
## [1] 3.1 0.1 -1.4 1.1
```

You can probably guess that the first element corresponds to the first quarter, the second element to the second quarter, and so on, but that's only because I've told you the back story and because this happens to be a very simple example. In general, it can be quite difficult. This is where it can be helpful to assign `names` to each of the elements. Here's how you do it:

```
names(profit) <- c("Q1", "Q2", "Q3", "Q4")
profit
```

```
##   Q1   Q2   Q3   Q4
## 3.1 0.1 -1.4 1.1
```

This is a slightly odd looking command, admittedly, but it's not too difficult to follow. All we're doing is assigning a vector of labels (character strings) to `names(profit)`. You can always delete the names again by using the command `names(profit) <- NULL`. It's also worth noting that you don't have to do this as a two stage process. You can get the same result with this command:

```
profit <- c( "Q1" = 3.1, "Q2" = 0.1, "Q3" = -1.4, "Q4" = 1.1 )
profit
```

```
##   Q1   Q2   Q3   Q4
## 3.1 0.1 -1.4 1.1
```

The important things to notice are that (a) this does make things much easier to read, but (b) the names at the top aren't the “real” data. The *value* of `profit[1]` is still 3.1; all I've done is added a *name* to `profit[1]` as well. Nevertheless, names aren't purely cosmetic, since R allows you to pull out particular elements of the vector by referring to their names:

```
profit["Q1"]
```

```
##   Q1
## 3.1
```

And if I ever need to pull out the names themselves, then I just type `names(profit)`.

4.6.3 Variable classes

As we've seen, R allows you to store different kinds of data. In particular, the variables we've defined so far have either been character data (text), numeric data, or logical data.¹⁶ It's important that we remember what kind of information each variable stores (and even more important that R remembers) since different kinds of variables allow you to do different things to them. For instance, if your variables have numerical information in them, then it's okay to multiply them together:

```
x <- 5    # x is numeric
y <- 4    # y is numeric
x * y
```

```
## [1] 20
```

¹⁶Or functions. But let's ignore functions for the moment.

But if they contain character data, multiplication makes no sense whatsoever, and R will complain if you try to do it:

```
x <- "apples"  # x is character
y <- "oranges" # y is character
x * y

## Error in x * y: non-numeric argument to binary operator
```

Even R is smart enough to know you can't multiply "apples" by "oranges". It knows this because the quote marks are indicators that the variable is supposed to be treated as text, not as a number.

This is quite useful, but notice that it means that R makes a big distinction between 5 and "5". Without quote marks, R treats 5 as the number five, and will allow you to do calculations with it. With the quote marks, R treats "5" as the textual character five, and doesn't recognise it as a number any more than it recognises "p" or "five" as numbers. As a consequence, there's a big difference between typing `x <- 5` and typing `x <- "5"`. In the former, we're storing the number 5; in the latter, we're storing the character "5". Thus, if we try to do multiplication with the character versions, R gets stroppy:

```
x <- "5"  # x is character
y <- "4"  # y is character
x * y

## Error in x * y: non-numeric argument to binary operator
```

Okay, let's suppose that I've forgotten what kind of data I stored in the variable `x` (which happens depressingly often). R provides a function that will let us find out. Or, more precisely, it provides *three* functions: `class()`, `mode()` and `typeof()`. Why the heck does it provide three functions, you might be wondering? Basically, because R actually keeps track of three different kinds of information about a variable:

1. The **`class`** of a variable is a “high level” classification, and it captures psychologically (or statistically) meaningful distinctions. For instance "2011-09-12" and "my birthday" are both text strings, but there's an important difference between the two: one of them is a date. So it would be nice if we could get R to recognise that "2011-09-12" is a date, and allow us to do things like add or subtract from it. The class of a variable is what R uses to keep track of things like that. Because the class of a variable is critical for determining what R can or can't do with it, the `class()` function is very handy.
2. The **`mode`** of a variable refers to the format of the information that the variable stores. It tells you whether R has stored text data or numeric data, for instance, which is kind of useful, but it only makes these “simple” distinctions. It can be useful to know about, but it's not the main thing we care about. So I'm not going to use the `mode()` function very much.¹⁷
3. The **`type`** of a variable is a very low level classification. We won't use it in this book, but (for those of you that care about these details) this is where you can see the distinction between integer data, double precision numeric, etc. Almost none of you actually will care about this, so I'm not even going to bother demonstrating the `typeof()` function.

For purposes, it's the `class()` of the variable that we care most about. Later on, I'll talk a bit about how you can convince R to “coerce” a variable to change from one class to another (Section 7.10). That's a useful skill for real world data analysis, but it's not something that we need right now. In the meantime, the following examples illustrate the use of the `class()` function:

¹⁷Actually, I don't think I ever use this in practice. I don't know why I bother to talk about it in the book anymore.

```
x <- "hello world"      # x is text
class(x)

## [1] "character"

x <- TRUE      # x is logical
class(x)

## [1] "logical"

x <- 100      # x is a number
class(x)

## [1] "numeric"
```

Exciting, no?

4.7 Factors

Okay, it's time to start introducing some of the data types that are somewhat more specific to statistics. If you remember back to Chapter 2, when we assign numbers to possible outcomes, these numbers can mean quite different things depending on what kind of variable we are attempting to measure. In particular, we commonly make the distinction between *nominal*, *ordinal*, *interval* and *ratio* scale data. How do we capture this distinction in R? Currently, we only seem to have a single numeric data type. That's probably not going to be enough, is it?

A little thought suggests that the numeric variable class in R is perfectly suited for capturing ratio scale data. For instance, if I were to measure response time (RT) for five different events, I could store the data in R like this:

```
RT <- c(342, 401, 590, 391, 554)
```

where the data here are measured in milliseconds, as is conventional in the psychological literature. It's perfectly sensible to talk about "twice the response time", $2 \times RT$, or the "response time plus 1 second", $RT + 1000$, and so both of the following are perfectly reasonable things for R to do:

```
2 * RT
```

```
## [1] 684 802 1180 782 1108
```

```
RT + 1000
```

```
## [1] 1342 1401 1590 1391 1554
```

And to a lesser extent, the "numeric" class is okay for interval scale data, as long as we remember that multiplication and division aren't terribly interesting for these sorts of variables. That is, if my IQ score is 110 and yours is 120, it's perfectly okay to say that you're 10 IQ points smarter than me¹⁸, but it's not okay

¹⁸Taking all the usual caveats that attach to IQ measurement as a given, of course.

to say that I'm only 92% as smart as you are, because intelligence doesn't have a natural zero.¹⁹ We might even be willing to tolerate the use of numeric variables to represent ordinal scale variables, such as those that you typically get when you ask people to rank order items (e.g., like we do in Australian elections), though as we will see R actually has a built in tool for representing ordinal data (see Section 7.11.2) However, when it comes to nominal scale data, it becomes completely unacceptable, because almost all of the "usual" rules for what you're allowed to do with numbers don't apply to nominal scale data. It is for this reason that R has **factors**.

4.7.1 Introducing factors

Suppose, I was doing a study in which people could belong to one of three different treatment conditions. Each group of people were asked to complete the same task, but each group received different instructions. Not surprisingly, I might want to have a variable that keeps track of what group people were in. So I could type in something like this

```
group <- c(1,1,1,2,2,2,3,3,3)
```

so that `group[i]` contains the group membership of the `i`-th person in my study. Clearly, this is numeric data, but equally obviously this is a nominal scale variable. There's no sense in which "group 1" plus "group 2" equals "group 3", but nevertheless if I try to do that, R won't stop me because it doesn't know any better:

```
group + 2
```

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

Apparently R seems to think that it's allowed to invent "group 4" and "group 5", even though they didn't actually exist. Unfortunately, R is too stupid to know any better: it thinks that 3 is an ordinary number in this context, so it sees no problem in calculating `3 + 2`. But since *we're* not that stupid, we'd like to stop R from doing this. We can do so by instructing R to treat `group` as a factor. This is easy to do using the `as.factor()` function.²⁰

```
group <- as.factor(group)
group
```

```
## [1] 1 1 1 2 2 2 3 3 3
## Levels: 1 2 3
```

It looks more or less the same as before (though it's not immediately obvious what all that `Levels` rubbish is about), but if we ask R to tell us what the class of the `group` variable is now, it's clear that it has done what we asked:

```
class(group)
```

```
## [1] "factor"
```

Neat. Better yet, now that I've converted `group` to a factor, look what happens when I try to add 2 to it:

¹⁹Or, more precisely, we don't know how to measure it. Arguably, a rock has zero intelligence. But it doesn't make sense to say that the IQ of a rock is 0 in the same way that we can say that the average human has an IQ of 100. And without knowing what the IQ value is that corresponds to a literal absence of any capacity to think, reason or learn, then we really can't multiply or divide IQ scores and expect a meaningful answer.

²⁰Once again, this is an example of *coercing* a variable from one class to another. I'll talk about coercion in more detail in Section 7.10.

```
group + 2

## Warning in Ops.factor(group, 2): '+' not meaningful for factors

## [1] NA NA NA NA NA NA NA NA NA
```

This time even R is smart enough to know that I'm being an idiot, so it tells me off and then produces a vector of missing values. (i.e., `NA`: see Section 4.6.1).

4.7.2 Labelling the factor levels

I have a confession to make. My memory is not infinite in capacity; and it seems to be getting worse as I get older. So it kind of annoys me when I get data sets where there's a nominal scale variable called `gender`, with two levels corresponding to males and females. But when I go to print out the variable I get something like this:

```
gender

## [1] 1 1 1 1 1 2 2 2 2
## Levels: 1 2
```

Okaaaay. That's not helpful at all, and it makes me very sad. Which number corresponds to the males and which one corresponds to the females? Wouldn't it be nice if R could actually keep track of this? It's way too hard to remember which number corresponds to which gender. And besides, the problem that this causes is much more serious than a single sad nerd... because R has no way of knowing that the 1s in the `group` variable are a very different kind of thing to the 1s in the `gender` variable. So if I try to ask which elements of the `group` variable are equal to the corresponding elements in `gender`, R thinks this is totally kosher, and gives me this:

```
group == gender

## Error in Ops.factor(group, gender): level sets of factors are different
```

Well, that's ... especially stupid.²¹ The problem here is that R is very literal minded. Even though you've declared both `group` and `gender` to be factors, it still assumes that a 1 is a 1 no matter which variable it appears in.

To fix both of these problems (my memory problem, and R's infuriating literal interpretations), what we need to do is assign meaningful labels to the different *levels* of each factor. We can do that like this:

```
levels(group) <- c("group 1", "group 2", "group 3")
print(group)
```

```
## [1] group 1 group 1 group 1 group 2 group 2 group 2 group 3 group 3 group 3
## Levels: group 1 group 2 group 3
```

²¹Some users might wonder why R even allows the `==` operator for factors. The reason is that sometimes you really do have different factors that have the same levels. For instance, if I was analysing data associated with football games, I might have a factor called `home.team`, and another factor called `winning.team`. In that situation I really should be able to ask if `home.team == winning.team`.

```
levels(gender) <- c("male", "female")
print(gender)

## [1] male   male   male   male   male   female female female female
## Levels: male female
```

That's much easier on the eye, and better yet, R is smart enough to know that "female" is not equal to "group 2", so now when I try to ask which group memberships are "equal to" the gender of the corresponding person,

```
group == gender
```

```
## Error in Ops.factor(group, gender): level sets of factors are different
```

R correctly tells me that I'm an idiot.

4.7.3 Moving on...

Factors are very useful things, and we'll use them a lot in this book: they're *the* main way to represent a nominal scale variable. And there are lots of nominal scale variables out there. I'll talk more about factors in 7.11.2, but for now you know enough to be able to get started.

4.8 Data frames

It's now time to go back and deal with the somewhat confusing thing that happened in Section ?? when we tried to open up a CSV file. Apparently we succeeded in loading the data, but it came to us in a very odd looking format. At the time, I told you that this was a *data frame*. Now I'd better explain what that means.

4.8.1 Introducing data frames

In order to understand why R has created this funny thing called a data frame, it helps to try to see what problem it solves. So let's go back to the little scenario that I used when introducing factors in Section 4.7. In that section I recorded the `group` and `gender` for all 9 participants in my study. Let's also suppose I recorded their ages and their `score` on "Dan's Terribly Exciting Psychological Test":

```
age <- c(17, 19, 21, 37, 18, 19, 47, 18, 19)
score <- c(12, 10, 11, 15, 16, 14, 25, 21, 29)
```

Assuming no other variables are in the workspace, if I type `who()` I get this:

```
who()
```

##	-- Name --	-- Class --	-- Size --
##	age	numeric	9
##	any.sales.this.month	logical	12
##	berkeley	data.frame	39 x 3
##	berkeley.small	data.frame	46 x 2

```

##   coef          numeric    2
##   days.per.month numeric    12
##   february.sales numeric    1
##   gender         factor     9
##   greeting      character   1
##   group          factor     9
##   is.the.Party.correct logical   1
##   months         character  12
##   revenue        numeric    1
##   royalty        numeric    1
##   sales          numeric    1
##   sales.by.month numeric    12
##   score          numeric    9
##   simpson        matrix     6 x 5
##   stock.levels   character  12
##   xlu            numeric    1

```

So there are four variables in the workspace, `age`, `gender`, `group` and `score`. And it just so happens that all four of them are the same size (i.e., they're all vectors with 9 elements). Aaaand it just so happens that `age[1]` corresponds to the age of the first person, and `gender[1]` is the gender of that very same person, etc. In other words, you and I both know that all four of these variables correspond to the *same* data set, and all four of them are organised in exactly the same way.

However, R *doesn't* know this! As far as it's concerned, there's no reason why the `age` variable has to be the same length as the `gender` variable; and there's no particular reason to think that `age[1]` has any special relationship to `gender[1]` any more than it has a special relationship to `gender[4]`. In other words, when we store everything in separate variables like this, R doesn't know anything about the relationships between things. It doesn't even really know that these variables actually refer to a proper data set. The data frame fixes this: if we store our variables inside a data frame, we're telling R to treat these variables as a single, fairly coherent data set.

To see how they do this, let's create one. So how do we create a data frame? One way we've already seen: if we import our data from a CSV file, R will store it as a data frame. A second way is to create it directly from some existing variables using the `data.frame()` function. All you have to do is type a list of variables that you want to include in the data frame. The output of a `data.frame()` command is, well, a data frame. So, if I want to store all four variables from my experiment in a data frame called `expt` I can do so like this:

```
expt <- data.frame ( age, gender, group, score )
expt
```

```

##   age gender  group score
## 1  17 male   group 1    12
## 2  19 male   group 1    10
## 3  21 male   group 1    11
## 4  37 male   group 2    15
## 5  18 male   group 2    16
## 6  19 female group 2    14
## 7  47 female group 3    25
## 8  18 female group 3    21
## 9  19 female group 3    29

```

Note that `expt` is a completely self-contained variable. Once you've created it, it no longer depends on the original variables from which it was constructed. That is, if we make changes to the original `age` variable, it will *not* lead to any changes to the age data stored in `expt`.

4.8.2 Pulling out the contents of the data frame using \$

At this point, our workspace contains only the one variable, a data frame called `expt`. But as we can see when we told R to print the variable out, this data frame contains 4 variables, each of which has 9 observations. So how do we get this information out again? After all, there's no point in storing information if you don't use it, and there's no way to use information if you can't access it. So let's talk a bit about how to pull information out of a data frame.

The first thing we might want to do is pull out one of our stored variables, let's say `score`. One thing you might try to do is ignore the fact that `score` is locked up inside the `expt` data frame. For instance, you might try to print it out like this:

```
score
## Error in eval(expr, envir, enclos): object 'score' not found
```

This doesn't work, because R doesn't go "peeking" inside the data frame unless you explicitly tell it to do so. There's actually a very good reason for this, which I'll explain in a moment, but for now let's just assume R knows what it's doing. How do we tell R to look inside the data frame? As is always the case with R there are several ways. The simplest way is to use the `$` operator to extract the variable you're interested in, like this:

```
expt$score
## [1] 12 10 11 15 16 14 25 21 29
```

4.8.3 Getting information about a data frame

One problem that sometimes comes up in practice is that you forget what you called all your variables. Normally you might try to type `objects()` or `who()`, but neither of those commands will tell you what the names are for those variables inside a data frame! One way is to ask R to tell you what the *names* of all the variables stored in the data frame are, which you can do using the `names()` function:

```
names(expt)
## [1] "age"      "gender"    "group"     "score"
```

An alternative method is to use the `who()` function, as long as you tell it to look at the variables inside data frames. If you set `expand = TRUE` then it will not only list the variables in the workspace, but it will "expand" any data frames that you've got in the workspace, so that you can see what they look like. That is:

```
who(expand = TRUE)
##   -- Name --          -- Class --   -- Size --
##   any.sales.this.month logical      12
##   berkeley              data.frame  39 x 3
##   $women.apply          numeric     39
##   $total.admit          numeric     39
##   $number.apply          numeric     39
##   berkeley.small        data.frame  46 x 2
```

```

##   $women.apply      numeric    46
##   $total.admit     numeric    46
##   coef             numeric     2
##   days.per.month   numeric    12
##   expt            data.frame  9 x 4
##   $age              numeric     9
##   $gender           factor      9
##   $group            factor      9
##   $score            numeric     9
##   february.sales   numeric     1
##   greeting          character    1
##   is.the.Party.correct logical    1
##   months            character   12
##   revenue           numeric     1
##   royalty           numeric     1
##   sales              numeric     1
##   sales.by.month   numeric    12
##   simpson           matrix     6 x 5
##   stock.levels     character   12
##   xlu               numeric     1

```

or, since `expand` is the first argument in the `who()` function you can just type `who(TRUE)`. I'll do that a lot in this book.

4.8.4 Looking for more on data frames?

There's a lot more that can be said about data frames: they're fairly complicated beasts, and the longer you use R the more important it is to make sure you really understand them. We'll talk a lot more about them in Chapter 7.

4.9 Lists

The next kind of data I want to mention are *lists*. Lists are an extremely fundamental data structure in R, and as you start making the transition from a novice to a savvy R user you will use lists all the time. I don't use lists very often in this book – not directly – but most of the advanced data structures in R are built from lists (e.g., data frames are actually a specific type of list). Because lists are so important to how R stores things, it's useful to have a basic understanding of them. Okay, so what is a list, exactly? Like data frames, lists are just “collections of variables.” However, unlike data frames – which are basically supposed to look like a nice “rectangular” table of data – there are no constraints on what kinds of variables we include, and no requirement that the variables have any particular relationship to one another. In order to understand what this actually *means*, the best thing to do is create a list, which we can do using the `list()` function. If I type this as my command:

```

Dan <- list( age = 34,
            nerd = TRUE,
            parents = c("Joe", "Liz")
)

```

R creates a new list variable called `Dan`, which is a bundle of three different variables: `age`, `nerd` and `parents`. Notice, that the `parents` variable is longer than the others. This is perfectly acceptable for a list, but it wouldn't be for a data frame. If we now print out the variable, you can see the way that R stores the list:

```
print( Dan )
```

```
## $age
## [1] 34
##
## $nerd
## [1] TRUE
##
## $parents
## [1] "Joe" "Liz"
```

As you might have guessed from those `$` symbols everywhere, the variables are stored in exactly the same way that they are for a data frame (again, this is not surprising: data frames *are* a type of list). So you will (I hope) be entirely unsurprised and probably quite bored when I tell you that you can extract the variables from the list using the `$` operator, like so:

```
Dan$nerd
```

```
## [1] TRUE
```

If you need to add new entries to the list, the easiest way to do so is to again use `$`, as the following example illustrates. If I type a command like this

```
Dan$children <- "Alex"
```

then R creates a new entry to the end of the list called `children`, and assigns it a value of `"Alex"`. If I were now to `print()` this list out, you'd see a new entry at the bottom of the printout. Finally, it's actually possible for lists to contain other lists, so it's quite possible that I would end up using a command like `Dan$children$age` to find out how old my son is. Or I could try to remember it myself I suppose.

4.10 Formulas

The last kind of variable that I want to introduce before finally being able to start talking about statistics is the *formula*. Formulas were originally introduced into R as a convenient way to specify a particular type of statistical model (see Chapter??) but they're such handy things that they've spread. Formulas are now used in a lot of different contexts, so it makes sense to introduce them early.

Stated simply, a formula object is a variable, but it's a special type of variable that specifies a relationship between other variables. A formula is specified using the “tilde operator” `#~#`. A very simple example of a formula is shown below:²²

```
formula1 <- out ~ pred
formula1
```

```
## out ~ pred
```

²²Note that, when I write out the formula, R doesn't check to see if the `out` and `pred` variables actually exist: it's only later on when you try to use the formula for something that this happens.

The *precise* meaning of this formula depends on exactly what you want to do with it, but in broad terms it means “the `out` (outcome) variable, analysed in terms of the `pred` (predictor) variable”. That said, although the simplest and most common form of a formula uses the “one variable on the left, one variable on the right” format, there are others. For instance, the following examples are all reasonably common

```
formula2 <- out ~ pred1 + pred2    # more than one variable on the right
formula3 <- out ~ pred1 * pred2    # different relationship between predictors
formula4 <- ~ var1 + var2          # a 'one-sided' formula
```

and there are many more variants besides. Formulas are pretty flexible things, and so different functions will make use of different formats, depending on what the function is intended to do.

4.11 Generic functions

There’s one really important thing that I omitted when I discussed functions earlier on in Section 3.5, and that’s the concept of a *generic function*. The two most notable examples that you’ll see in the next few chapters are `summary()` and `plot()`, although you’ve already seen an example of one working behind the scenes, and that’s the `print()` function. The thing that makes generics different from the other functions is that their behaviour changes, often quite dramatically, depending on the `class()` of the input you give it. The easiest way to explain the concept is with an example. With that in mind, let’s take a closer look at what the `print()` function actually does. I’ll do this by creating a formula, and printing it out in a few different ways. First, let’s stick with what we know:

```
my.formula <- blah ~ blah.blah      # create a variable of class "formula"
print( my.formula )                  # print it out using the generic print() function
```

```
## blah ~ blah.blah
```

So far, there’s nothing very surprising here. But there’s actually a lot going on behind the scenes here. When I type `print(my.formula)`, what actually happens is the `print()` function checks the class of the `my.formula` variable. When the function discovers that the variable it’s been given is a formula, it goes looking for a function called `print.formula()`, and then delegates the whole business of printing out the variable to the `print.formula()` function.²³ For what it’s worth, the name for a “dedicated” function like `print.formula()` that exists only to be a special case of a generic function like `print()` is a *method*, and the name for the process in which the generic function passes off all the hard work onto a method is called *method dispatch*. You won’t need to understand the details at all for this book, but you do need to know the gist of it; if only because a lot of the functions we’ll use are actually generics. Anyway, to help expose a little more of the workings to you, let’s bypass the `print()` function entirely and call the formula method directly:

```
print.formula( my.formula )          # print it out using the print.formula() method
## Appears to be deprecated
```

There’s no difference in the output at all. But this shouldn’t surprise you because it was actually the `print.formula()` method that was doing all the hard work in the first place. The `print()` function itself

²³For readers with a programming background: what I’m describing is the very basics of how S3 methods work. However, you should be aware that R has two entirely distinct systems for doing object oriented programming, known as S3 and S4. Of the two, S3 is simpler and more informal, whereas S4 supports all the stuff that you might expect of a fully object oriented language. Most of the generics we’ll run into in this book use the S3 system, which is convenient for me because I’m still trying to figure out S4.

is a lazy bastard that doesn't do anything other than select which of the methods is going to do the actual printing.

Okay, fair enough, but you might be wondering what would have happened if `print.formula()` didn't exist? That is, what happens if there isn't a specific method defined for the class of variable that you're using? In that case, the generic function passes off the hard work to a "default" method, whose name in this case would be `print.default()`. Let's see what happens if we bypass the `print()` formula, and try to print out `my.formula` using the `print.default()` function:

```
print.default( my.formula )      # print it out using the print.default() method
```

```
## blah ~ blah.blah
## attr(,"class")
## [1] "formula"
## attr(,".Environment")
## <environment: R_GlobalEnv>
```

Hm. You can kind of see that it is trying to print out the same formula, but there's a bunch of ugly low-level details that have also turned up on screen. This is because the `print.default()` method doesn't know anything about formulas, and doesn't know that it's supposed to be hiding the obnoxious internal gibberish that R produces sometimes.

At this stage, this is about as much as we need to know about generic functions and their methods. In fact, you can get through the entire book without learning any more about them than this, so it's probably a good idea to end this discussion here.

4.12 Getting help

The very last topic I want to mention in this chapter is where to go to find help. Obviously, I've tried to make this book as helpful as possible, but it's not even close to being a comprehensive guide, and there's thousands of things it doesn't cover. So where should you go for help?

4.12.1 Other resources

- The Rseek website (www.rseek.org). One thing that I really find annoying about the R help documentation is that it's hard to search properly. When coupled with the fact that the documentation is dense and highly technical, it's often a better idea to search or ask online for answers to your questions. With that in mind, the Rseek website is great: it's an R specific search engine. I find it really useful, and it's almost always my first port of call when I'm looking around.
- The R-help mailing list (see <http://www.r-project.org/mail.html> for details). This is the official R help mailing list. It can be very helpful, but it's *very* important that you do your homework before posting a question. The list gets a lot of traffic. While the people on the list try as hard as they can to answer questions, they do so for free, and you *really* don't want to know how much money they could charge on an hourly rate if they wanted to apply market rates. In short, they are doing you a favour, so be polite. Don't waste their time asking questions that can be easily answered by a quick search on Rseek (it's rude), make sure your question is clear, and all of the relevant information is included. In short, read the posting guidelines carefully (<http://www.r-project.org/posting-guide.html>), and make use of the `help.request()` function that R provides to check that you're actually doing what you're expected.

4.13 Summary

This chapter continued where Chapter 3 left off. The focus was still primarily on introducing basic R concepts, but this time at least you can see how those concepts are related to data analysis:

- *Installing, loading and updating packages.* Knowing how to extend the functionality of R by installing and using packages is critical to becoming an effective R user (Section 4.2)
- *Getting around.* Section 4.3 talked about how to manage your workspace and how to keep it tidy. Similarly, Section 4.4 talked about how to get R to interact with the rest of the file system.
- *Loading and saving data.* Finally, we encountered actual data files. Loading and saving data is obviously a crucial skill, one we discussed in Section 4.5.
- *Useful things to know about variables.* In particular, we talked about special values, element names and classes (Section 4.6).
- *More complex types of variables.* R has a number of important variable types that will be useful when analysing real data. I talked about factors in Section 4.7, data frames in Section 4.8, lists in Section 4.9 and formulas in Section 4.10.
- *Generic functions.* How is it that some function seem to be able to do lots of different things? Section 4.11 tells you how.
- *Getting help.* Assuming that you're not looking for counselling, Section 4.12 covers several possibilities. If you are looking for counselling, well, this book really can't help you there. Sorry.

Taken together, Chapters 3 and 4 provide enough of a background that you can finally get started doing some statistics! Yes, there's a lot more R concepts that you ought to know (and we'll talk about some of them in Chapters 7 and 8), but I think that we've talked quite enough about programming for the moment. It's time to see how your experience with programming can be used to do some data analysis...

Part III. Working with data

Chapter 5

Descriptive statistics

Any time that you get a new data set to look at, one of the first tasks that you have to do is find ways of summarising the data in a compact, easily understood fashion. This is what *descriptive statistics* (as opposed to inferential statistics) is all about. In fact, to many people the term “statistics” is synonymous with descriptive statistics. It is this topic that we’ll consider in this chapter, but before going into any details, let’s take a moment to get a sense of why we need descriptive statistics. To do this, let’s load the `aflsmall.Rdata` file, and use the `who()` function in the `lsr` package to see what variables are stored in the file:

```
load( "./data/aflsmall.Rdata" )
library(lsr)
who()

##   -- Name --           -- Class --   -- Size --
##   afl.finalists        factor      400
##   afl.margins          numeric     176
##   any.sales.this.month logical     12
##   berkeley              data.frame 39 x 3
##   berkeley.small        data.frame 46 x 2
##   coef                  numeric     2
##   Dan                   list       4
##   days.per.month        numeric     12
##   expt                 data.frame 9 x 4
##   february.sales        numeric     1
##   formula1              formula    
##   formula2              formula    
##   formula3              formula    
##   formula4              formula    
##   greeting              character   1
##   is.the.Party.correct logical    1
##   months                character   12
##   my.formula            formula    
##   revenue               numeric     1
##   royalty               numeric     1
##   sales                 numeric     1
##   sales.by.month        numeric     12
##   simpson               matrix      6 x 5
##   stock.levels          character   12
##   xlu                   numeric     1
```

There are two variables here, `afl.finalists` and `afl.margins`. We'll focus a bit on these two variables in this chapter, so I'd better tell you what they are. Unlike most of data sets in this book, these are actually real data, relating to the Australian Football League (AFL)¹ The `afl.margins` variable contains the winning margin (number of points) for all 176 home and away games played during the 2010 season. The `afl.finalists` variable contains the names of all 400 teams that played in all 200 finals matches played during the period 1987 to 2010. Let's have a look at the `afl.margins` variable:

```
print(afl.margins)
```

```
## [1] 56 31 56 8 32 14 36 56 19 1 3 104 43 44 72 9 28
## [18] 25 27 55 20 16 16 7 23 40 48 64 22 55 95 15 49 52
## [35] 50 10 65 12 39 36 3 26 23 20 43 108 53 38 4 8 3
## [52] 13 66 67 50 61 36 38 29 9 81 3 26 12 36 37 70 1
## [69] 35 12 50 35 9 54 47 8 47 2 29 61 38 41 23 24 1
## [86] 9 11 10 29 47 71 38 49 65 18 0 16 9 19 36 60 24
## [103] 25 44 55 3 57 83 84 35 4 35 26 22 2 14 19 30 19
## [120] 68 11 75 48 32 36 39 50 11 0 63 82 26 3 82 73 19
## [137] 33 48 8 10 53 20 71 75 76 54 44 5 22 94 29 8 98
## [154] 9 89 1 101 7 21 52 42 21 116 3 44 29 27 16 6 44
## [171] 3 28 38 29 10 10
```

This output doesn't make it easy to get a sense of what the data are actually saying. Just "looking at the data" isn't a terribly effective way of understanding data. In order to get some idea about what's going on, we need to calculate some descriptive statistics (this chapter) and draw some nice pictures (Chapter 6). Since the descriptive statistics are the easier of the two topics, I'll start with those, but nevertheless I'll show you a histogram of the `afl.margins` data, since it should help you get a sense of what the data we're trying to describe actually look like. But for what it's worth, this histogram – which is shown in Figure 5.1 – was generated using the `hist()` function. We'll talk a lot more about how to draw histograms in Section 6.3. For now, it's enough to look at the histogram and note that it provides a fairly interpretable representation of the `afl.margins` data.

5.1 Measures of central tendency

Drawing pictures of the data, as I did in Figure 5.1 is an excellent way to convey the "gist" of what the data is trying to tell you, it's often extremely useful to try to condense the data into a few simple "summary" statistics. In most situations, the first thing that you'll want to calculate is a measure of *central tendency*. That is, you'd like to know something about the "average" or "middle" of your data lies. The two most commonly used measures are the mean, median and mode; occasionally people will also report a trimmed mean. I'll explain each of these in turn, and then discuss when each of them is useful.

5.1.1 The mean

The *mean* of a set of observations is just a normal, old-fashioned average: add all of the values up, and then divide by the total number of values. The first five AFL margins were 56, 31, 56, 8 and 32, so the mean of these observations is just:

$$\frac{56 + 31 + 56 + 8 + 32}{5} = \frac{183}{5} = 36.60$$

Of course, this definition of the mean isn't news to anyone: averages (i.e., means) are used so often in everyday life that this is pretty familiar stuff. However, since the concept of a mean is something that

¹Note for non-Australians: the AFL is an Australian rules football competition. You don't need to know anything about Australian rules in order to follow this section.

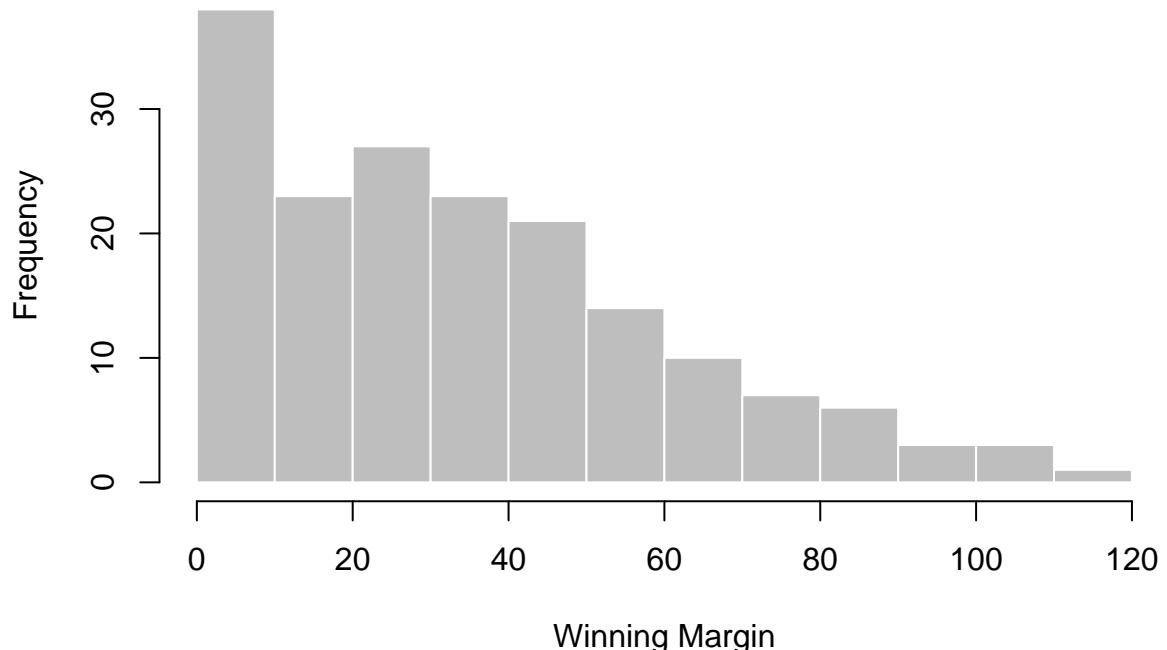


Figure 5.1: A histogram of the AFL 2010 winning margin data (the `afl.margins` variable). As you might expect, the larger the margin the less frequently you tend to see it.

everyone already understands, I'll use this as an excuse to start introducing some of the mathematical notation that statisticians use to describe this calculation, and talk about how the calculations would be done in R.

The first piece of notation to introduce is N , which we'll use to refer to the number of observations that we're averaging (in this case $N = 5$). Next, we need to attach a label to the observations themselves. It's traditional to use X for this, and to use subscripts to indicate which observation we're actually talking about. That is, we'll use X_1 to refer to the first observation, X_2 to refer to the second observation, and so on, all the way up to X_N for the last one. Or, to say the same thing in a slightly more abstract way, we use X_i to refer to the i -th observation. Just to make sure we're clear on the notation, the following table lists the 5 observations in the `afl.margins` variable, along with the mathematical symbol used to refer to it, and the actual value that the observation corresponds to:

the observation	its symbol	the observed value
winning margin, game 1	$\$X_1\$$	56 points
winning margin, game 2	$\$X_2\$$	31 points
winning margin, game 3	$\$X_3\$$	56 points
winning margin, game 4	$\$X_4\$$	8 points
winning margin, game 5	$\$X_5\$$	32 points

Okay, now let's try to write a formula for the mean. By tradition, we use \bar{X} as the notation for the mean. So the calculation for the mean could be expressed using the following formula:

$$\bar{X} = \frac{X_1 + X_2 + \dots + X_{N-1} + X_N}{N}$$

This formula is entirely correct, but it's terribly long, so we make use of the **summation symbol** \sum to shorten it.² If I want to add up the first five observations, I could write out the sum the long way, $X_1 + X_2 + X_3 + X_4 + X_5$ or I could use the summation symbol to shorten it to this:

$$\sum_{i=1}^5 X_i$$

Taken literally, this could be read as "the sum, taken over all i values from 1 to 5, of the value X_i ". But basically, what it means is "add up the first five observations". In any case, we can use this notation to write out the formula for the mean, which looks like this:

$$\bar{X} = \frac{1}{N} \sum_{i=1}^N X_i$$

In all honesty, I can't imagine that all this mathematical notation helps clarify the concept of the mean at all. In fact, it's really just a fancy way of writing out the same thing I said in words: add all the values up, and then divide by the total number of items. However, that's not really the reason I went into all that detail. My goal was to try to make sure that everyone reading this book is clear on the notation that we'll be using throughout the book: \bar{X} for the mean, \sum for the idea of summation, X_i for the i th observation, and N for the total number of observations. We're going to be re-using these symbols a fair bit, so it's important that you understand them well enough to be able to "read" the equations, and to be able to see that it's just saying "add up lots of things and then divide by another thing".

²The choice to use Σ to denote summation isn't arbitrary: it's the Greek upper case letter sigma, which is the analogue of the letter S in that alphabet. Similarly, there's an equivalent symbol used to denote the multiplication of lots of numbers: because multiplications are also called "products", we use the Π symbol for this; the Greek upper case pi, which is the analogue of the letter P.

5.1.2 Calculating the mean in R

Okay that's the maths, how do we get the magic computing box to do the work for us? If you really wanted to, you could do this calculation directly in R. For the first 5 AFL scores, do this just by typing it in as if R were a calculator...

```
(56 + 31 + 56 + 8 + 32) / 5
```

```
## [1] 36.6
```

... in which case R outputs the answer 36.6, just as if it were a calculator. However, that's not the only way to do the calculations, and when the number of observations starts to become large, it's easily the most tedious. Besides, in almost every real world scenario, you've already got the actual numbers stored in a variable of some kind, just like we have with the `afl.margins` variable. Under those circumstances, what you want is a function that will just add up all the values stored in a numeric vector. That's what the `sum()` function does. If we want to add up all 176 winning margins in the data set, we can do so using the following command:³

```
sum( afl.margins )
```

```
## [1] 6213
```

If we only want the sum of the first five observations, then we can use square brackets to pull out only the first five elements of the vector. So the command would now be:

```
sum( afl.margins[1:5] )
```

```
## [1] 183
```

To calculate the mean, we now tell R to divide the output of this summation by five, so the command that we need to type now becomes the following:

```
sum( afl.margins[1:5] ) / 5
```

```
## [1] 36.6
```

Although it's pretty easy to calculate the mean using the `sum()` function, we can do it in an even easier way, since R also provides us with the `mean()` function. To calculate the mean for all 176 games, we would use the following command:

```
mean( x = afl.margins )
```

```
## [1] 35.30114
```

However, since `x` is the first argument to the function, I could have omitted the argument name. In any case, just to show you that there's nothing funny going on, here's what we would do to calculate the mean for the first five observations:

³Note that, just as we saw with the combine function `c()` and the remove function `rm()`, the `sum()` function has unnamed arguments. I'll talk about unnamed arguments later in Section 8.4.1, but for now let's just ignore this detail.

```
mean( afl.margins[1:5] )
```

```
## [1] 36.6
```

As you can see, this gives exactly the same answers as the previous calculations.

5.1.3 The median

The second measure of central tendency that people use a lot is the *median*, and it's even easier to describe than the mean. The median of a set of observations is just the middle value. As before let's imagine we were interested only in the first 5 AFL winning margins: 56, 31, 56, 8 and 32. To figure out the median, we sort these numbers into ascending order:

```
8, 31, 32, 56, 56
```

From inspection, it's obvious that the median value of these 5 observations is 32, since that's the middle one in the sorted list (I've put it in bold to make it even more obvious). Easy stuff. But what should we do if we were interested in the first 6 games rather than the first 5? Since the sixth game in the season had a winning margin of 14 points, our sorted list is now

```
8, 14, 31, 32, 56, 56
```

and there are *two* middle numbers, 31 and 32. The median is defined as the average of those two numbers, which is of course 31.5. As before, it's very tedious to do this by hand when you've got lots of numbers. To illustrate this, here's what happens when you use R to sort all 176 winning margins. First, I'll use the `sort()` function (discussed in Chapter 7) to display the winning margins in increasing numerical order:

```
sort( x = afl.margins )
```

```
## [1]  0   0   1   1   1   1   2   2   3   3   3   3   3   3   3   3   3   4
## [18]  4   5   6   7   7   8   8   8   8   9   9   9   9   9   9   9   10
## [35] 10  10  10  10  11  11  11  12  12  12  13  14  14  15  16  16  16
## [52] 16  18  19  19  19  19  19  20  20  20  21  21  22  22  22  23  23
## [69] 23  24  24  25  25  26  26  26  27  27  28  28  28  29  29  29  29
## [86] 29  29  30  31  32  32  33  35  35  35  35  36  36  36  36  36  36
## [103] 37  38  38  38  38  38  39  39  39  40  41  42  43  43  44  44  44
## [120] 44  47  47  47  48  48  48  49  49  49  50  50  50  50  52  52  53
## [137] 54  54  55  55  55  56  56  56  57  60  61  61  63  64  65  65  66
## [154] 67  68  70  71  71  72  73  75  75  76  81  82  82  83  84  89  94
## [171] 95  98  101 104 108 116
```

The middle values are 30 and 31, so the median winning margin for 2010 was 30.5 points. In real life, of course, no-one actually calculates the median by sorting the data and then looking for the middle value. In real life, we use the `median` command:

```
median( x = afl.margins )
```

```
## [1] 30.5
```

which outputs the median value of 30.5.

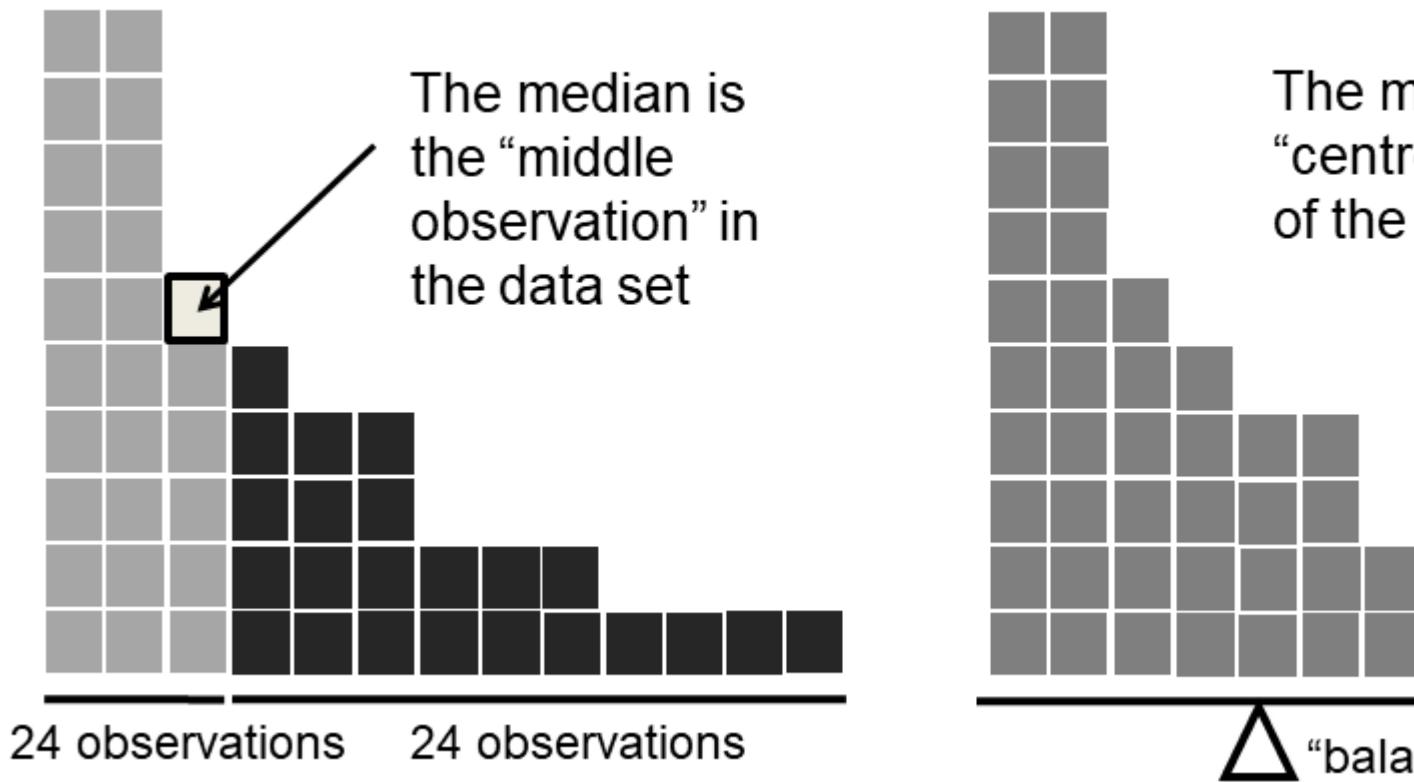


Figure 5.2: An illustration of the difference between how the mean and the median should be interpreted. The mean is basically the “centre of gravity” of the data set: if you imagine that the histogram of the data is a solid object, then the point on which you could balance it (as if on a see-saw) is the mean. In contrast, the median is the middle observation. Half of the observations are smaller, and half of the observations are larger.

5.1.4 Mean or median? What's the difference?

Knowing how to calculate means and medians is only a part of the story. You also need to understand what each one is saying about the data, and what that implies for when you should use each one. This is illustrated in Figure 5.2 the mean is kind of like the “centre of gravity” of the data set, whereas the median is the “middle value” in the data. What this implies, as far as which one you should use, depends a little on what type of data you’ve got and what you’re trying to achieve. As a rough guide:

- If your data are nominal scale, you probably shouldn’t be using either the mean or the median. Both the mean and the median rely on the idea that the numbers assigned to values are meaningful. If the numbering scheme is arbitrary, then it’s probably best to use the mode (Section 5.1.7) instead.
- If your data are ordinal scale, you’re more likely to want to use the median than the mean. The median only makes use of the order information in your data (i.e., which numbers are bigger), but doesn’t depend on the precise numbers involved. That’s exactly the situation that applies when your data are ordinal scale. The mean, on the other hand, makes use of the precise numeric values assigned to the observations, so it’s not really appropriate for ordinal data.
- For interval and ratio scale data, either one is generally acceptable. Which one you pick depends a bit on what you’re trying to achieve. The mean has the advantage that it uses all the information in the data (which is useful when you don’t have a lot of data), but it’s very sensitive to extreme values, as we’ll see in Section 5.1.6.

Let’s expand on that last part a little. One consequence is that there’s systematic differences between the mean and the median when the histogram is asymmetric (skewed; see Section ??). This is illustrated in Figure 5.2 notice that the median (right hand side) is located closer to the “body” of the histogram, whereas the mean (left hand side) gets dragged towards the “tail” (where the extreme values are). To give a concrete example, suppose Bob (income \$50,000), Kate (income \$60,000) and Jane (income \$65,000) are sitting at a table: the average income at the table is \$58,333 and the median income is \$60,000. Then Bill sits down with them (income \$100,000,000). The average income has now jumped to \$25,043,750 but the median rises only to \$62,500. If you’re interested in looking at the overall income at the table, the mean might be the right answer; but if you’re interested in what counts as a typical income at the table, the median would be a better choice here.

5.1.5 A real life example

To try to get a sense of why you need to pay attention to the differences between the mean and the median, let’s consider a real life example. Since I tend to mock journalists for their poor scientific and statistical knowledge, I should give credit where credit is due. This is from an excellent article on the ABC news website⁴ 24 September, 2010:

Senior Commonwealth Bank executives have travelled the world in the past couple of weeks with a presentation showing how Australian house prices, and the key price to income ratios, compare favourably with similar countries. “Housing affordability has actually been going sideways for the last five to six years,” said Craig James, the chief economist of the bank’s trading arm, CommSec.

This probably comes as a huge surprise to anyone with a mortgage, or who wants a mortgage, or pays rent, or isn’t completely oblivious to what’s been going on in the Australian housing market over the last several years. Back to the article:

CBA has waged its war against what it believes are housing doomsayers with graphs, numbers and international comparisons. In its presentation, the bank rejects arguments that Australia’s

⁴www.abc.net.au/news/stories/2010/09/24/3021480.htm

housing is relatively expensive compared to incomes. It says Australia's house price to household income ratio of 5.6 in the major cities, and 4.3 nationwide, is comparable to many other developed nations. It says San Francisco and New York have ratios of 7, Auckland's is 6.7, and Vancouver comes in at 9.3.

More excellent news! Except, the article goes on to make the observation that...

Many analysts say that has led the bank to use misleading figures and comparisons. If you go to page four of CBA's presentation and read the source information at the bottom of the graph and table, you would notice there is an additional source on the international comparison – Demographia. However, if the Commonwealth Bank had also used Demographia's analysis of Australia's house price to income ratio, it would have come up with a figure closer to 9 rather than 5.6 or 4.3

That's, um, a rather serious discrepancy. One group of people say 9, another says 4.5. Should we just split the difference, and say the truth lies somewhere in between? Absolutely not: this is a situation where there is a right answer and a wrong answer. Demographia are correct, and the Commonwealth Bank is incorrect. As the article points out

[An] obvious problem with the Commonwealth Bank's domestic price to income figures is they compare average incomes with median house prices (unlike the Demographia figures that compare median incomes to median prices). The median is the mid-point, effectively cutting out the highs and lows, and that means the average is generally higher when it comes to incomes and asset prices, because it includes the earnings of Australia's wealthiest people. To put it another way: the Commonwealth Bank's figures count Ralph Norris' multi-million dollar pay packet on the income side, but not his (no doubt) very expensive house in the property price figures, thus understating the house price to income ratio for middle-income Australians.

Couldn't have put it better myself. The way that Demographia calculated the ratio is the right thing to do. The way that the Bank did it is incorrect. As for why an extremely quantitatively sophisticated organisation such as a major bank made such an elementary mistake, well... I can't say for sure, since I have no special insight into their thinking, but the article itself does happen to mention the following facts, which may or may not be relevant:

[As] Australia's largest home lender, the Commonwealth Bank has one of the biggest vested interests in house prices rising. It effectively owns a massive swathe of Australian housing as security for its home loans as well as many small business loans.

My, my.

5.1.6 Trimmed mean

One of the fundamental rules of applied statistics is that the data are messy. Real life is never simple, and so the data sets that you obtain are never as straightforward as the statistical theory says.⁵ This can have awkward consequences. To illustrate, consider this rather strange looking data set:

$$-100, 2, 3, 4, 5, 6, 7, 8, 9, 10$$

If you were to observe this in a real life data set, you'd probably suspect that something funny was going on with the -100 value. It's probably an *outlier*, a value that doesn't really belong with the others. You

⁵Or at least, the basic statistical theory – these days there is a whole subfield of statistics called *robust statistics* that tries to grapple with the messiness of real data and develop theory that can cope with it.

might consider removing it from the data set entirely, and in this particular case I'd probably agree with that course of action. In real life, however, you don't always get such cut-and-dried examples. For instance, you might get this instead:

−15, 2, 3, 4, 5, 6, 7, 8, 9, 12

The −15 looks a bit suspicious, but not anywhere near as much as that −100 did. In this case, it's a little trickier. It *might* be a legitimate observation, it might not.

When faced with a situation where some of the most extreme-valued observations might not be quite trustworthy, the mean is not necessarily a good measure of central tendency. It is highly sensitive to one or two extreme values, and is thus not considered to be a ***robust*** measure. One remedy that we've seen is to use the median. A more general solution is to use a "trimmed mean". To calculate a trimmed mean, what you do is "discard" the most extreme examples on both ends (i.e., the largest and the smallest), and then take the mean of everything else. The goal is to preserve the best characteristics of the mean and the median: just like a median, you aren't highly influenced by extreme outliers, but like the mean, you "use" more than one of the observations. Generally, we describe a trimmed mean in terms of the percentage of observation on either side that are discarded. So, for instance, a 10% trimmed mean discards the largest 10% of the observations *and* the smallest 10% of the observations, and then takes the mean of the remaining 80% of the observations. Not surprisingly, the 0% trimmed mean is just the regular mean, and the 50% trimmed mean is the median. In that sense, trimmed means provide a whole family of central tendency measures that span the range from the mean to the median.

For our toy example above, we have 10 observations, and so a 10% trimmed mean is calculated by ignoring the largest value (i.e., 12) and the smallest value (i.e., −15) and taking the mean of the remaining values. First, let's enter the data

```
dataset <- c( -15, 2, 3, 4, 5, 6, 7, 8, 9, 12 )
```

Next, let's calculate means and medians:

```
mean( x = dataset )
```

```
## [1] 4.1
```

```
median( x = dataset )
```

```
## [1] 5.5
```

That's a fairly substantial difference, but I'm tempted to think that the mean is being influenced a bit too much by the extreme values at either end of the data set, especially the −15 one. So let's just try trimming the mean a bit. If I take a 10% trimmed mean, we'll drop the extreme values on either side, and take the mean of the rest:

```
mean( x = dataset, trim = .1 )
```

```
## [1] 5.5
```

which in this case gives exactly the same answer as the median. Note that, to get a 10% trimmed mean you write `trim = .1`, not `trim = 10`. In any case, let's finish up by calculating the 5% trimmed mean for the `afl.margins` data,

```
mean( x = afl.margins, trim = .05)

## [1] 33.75
```

5.1.7 Mode

The mode of a sample is very simple: it is the value that occurs most frequently. To illustrate the mode using the AFL data, let's examine a different aspect to the data set. Who has played in the most finals? The `afl.finalists` variable is a factor that contains the name of every team that played in any AFL final from 1987-2010, so let's have a look at it. To do this we will use the `head()` command. `head()` is useful when you're working with a `data.frame` with a lot of rows since you can use it to tell you how many rows to return. There have been a lot of finals in this period so printing `afl.finalists` using `print(afl.finalists)` will just fill us the screen. The command below tells R we just want the first 25 rows of the `data.frame`.

```
head(afl.finalists, 25)
```

```
## [1] Hawthorn    Melbourne   Carlton    Melbourne   Hawthorn
## [6] Carlton     Melbourne   Carlton    Hawthorn   Melbourne
## [11] Melbourne   Hawthorn   Melbourne   Essendon   Hawthorn
## [16] Geelong     Geelong    Hawthorn   Collingwood Melbourne
## [21] Collingwood West Coast Collingwood Essendon   Collingwood
## 17 Levels: Adelaide Brisbane Carlton Collingwood Essendon ... Western Bulldogs
```

There are actually 400 entries (aren't you glad we didn't print them all?). We *could* read through all 400, and count the number of occasions on which each team name appears in our list of finalists, thereby producing a **frequency table**. However, that would be mindless and boring: exactly the sort of task that computers are great at. So let's use the `table()` function (discussed in more detail in Section 7.1) to do this task for us:

```
table( afl.finalists )

## afl.finalists
##      Adelaide      Brisbane      Carlton      Collingwood
##             26              25              26              28
##      Essendon      Fitzroy      Fremantle      Geelong
##             32                  0                  6                  39
##      Hawthorn      Melbourne      North Melbourne      Port Adelaide
##             27              28              28              17
##      Richmond      St Kilda      Sydney      West Coast
##             6                  24                  26                  38
##      Western Bulldogs
##             24
```

Now that we have our frequency table, we can just look at it and see that, over the 24 years for which we have data, Geelong has played in more finals than any other team. Thus, the mode of the `finalists` data is "Geelong". The core packages in R don't have a function for calculating the mode⁶. However, I've included a function in the `lsr` package that does this. The function is called `mode0f()`, and here's how you use it:

⁶As we saw earlier, it *does* have a function called `mode()`, but it does something completely different.

```
modeOf( x = afl.finalists )

## [1] "Geelong"
```

There's also a function called `maxFreq()` that tells you what the modal frequency is. If we apply this function to our `finalists` data, we obtain the following:

```
maxFreq( x = afl.finalists )

## [1] 39
```

Taken together, we observe that Geelong (39 finals) played in more finals than any other team during the 1987-2010 period.

One last point to make with respect to the mode. While it's generally true that the mode is most often calculated when you have nominal scale data (because means and medians are useless for those sorts of variables), there are some situations in which you really do want to know the mode of an ordinal, interval or ratio scale variable. For instance, let's go back to thinking about our `afl.margins` variable. This variable is clearly ratio scale (if it's not clear to you, it may help to re-read Section 2.2), and so in most situations the mean or the median is the measure of central tendency that you want. But consider this scenario... a friend of yours is offering a bet. They pick a football game at random, and (without knowing who is playing) you have to guess the *exact* margin. If you guess correctly, you win \$50. If you don't, you lose \$1. There are no consolation prizes for “almost” getting the right answer. You have to guess exactly the right margin⁷ For this bet, the mean and the median are completely useless to you. It is the mode that you should bet on. So, we calculate this modal value

```
modeOf( x = afl.margins )

## [1] 3

maxFreq( x = afl.margins )

## [1] 8
```

So the 2010 data suggest you should bet on a 3 point margin, and since this was observed in 8 of the 176 games (4.5% of games) the odds are firmly in your favour.

5.2 Measures of variability

The statistics that we've discussed so far all relate to *central tendency*. That is, they all talk about which values are “in the middle” or “popular” in the data. However, central tendency is not the only type of summary statistic that we want to calculate. The second thing that we really want is a measure of the **variability** of the data. That is, how “spread out” are the data? How “far” away from the mean or median do the observed values tend to be? For now, let's assume that the data are interval or ratio scale, so we'll continue to use the `afl.margins` data. We'll use this data to discuss several different measures of spread, each with different strengths and weaknesses.

⁷This is called a “0-1 loss function”, meaning that you either win (1) or you lose (0), with no middle ground.

5.2.1 Range

The **range** of a variable is very simple: it's the biggest value minus the smallest value. For the AFL winning margins data, the maximum value is 116, and the minimum value is 0. We can calculate these values in R using the `max()` and `min()` functions:

```
max( afl.margins )
```

```
## [1] 116
```

```
min( afl.margins )
```

```
## [1] 0
```

where I've omitted the output because it's not interesting. The other possibility is to use the `range()` function; which outputs both the minimum value and the maximum value in a vector, like this:

```
range( afl.margins )
```

```
## [1] 0 116
```

Although the range is the simplest way to quantify the notion of “variability”, it’s one of the worst. Recall from our discussion of the mean that we want our summary measure to be robust. If the data set has one or two extremely bad values in it, we’d like our statistics not to be unduly influenced by these cases. If we look once again at our toy example of a data set containing very extreme outliers...

–100, 2, 3, 4, 5, 6, 7, 8, 9, 10

... it is clear that the range is not robust, since this has a range of 110, but if the outlier were removed we would have a range of only 8.

5.2.2 Interquartile range

The **interquartile range** (IQR) is like the range, but instead of calculating the difference between the biggest and smallest value, it calculates the difference between the 25th quantile and the 75th quantile. Probably you already know what a **quantile** is (they’re more commonly called percentiles), but if not: the 10th percentile of a data set is the smallest number x such that 10% of the data is less than x . In fact, we’ve already come across the idea: the median of a data set is its 50th quantile / percentile! R actually provides you with a way of calculating quantiles, using the (surprise, surprise) `quantile()` function. Let’s use it to calculate the median AFL winning margin:

```
quantile( x = afl.margins, probs = .5)
```

```
## 50%
## 30.5
```

And not surprisingly, this agrees with the answer that we saw earlier with the `median()` function. Now, we can actually input lots of quantiles at once, by specifying a vector for the `probs` argument. So let’s do that, and get the 25th and 75th percentile:

```
quantile( x = afl.margins, probs = c(.25, .75) )

##    25%    75%
## 12.75 50.50
```

And, by noting that $50.5 - 12.75 = 37.75$, we can see that the interquartile range for the 2010 AFL winning margins data is 37.75. Of course, that seems like too much work to do all that typing, so R has a built in function called `IQR()` that we can use:

```
IQR( x = afl.margins )

## [1] 37.75
```

While it's obvious how to interpret the range, it's a little less obvious how to interpret the IQR. The simplest way to think about it is like this: the interquartile range is the range spanned by the "middle half" of the data. That is, one quarter of the data falls below the 25th percentile, one quarter of the data is above the 75th percentile, leaving the "middle half" of the data lying in between the two. And the IQR is the range covered by that middle half.

5.2.3 Mean absolute deviation

The two measures we've looked at so far, the range and the interquartile range, both rely on the idea that we can measure the spread of the data by looking at the quantiles of the data. However, this isn't the only way to think about the problem. A different approach is to select a meaningful reference point (usually the mean or the median) and then report the "typical" deviations from that reference point. What do we mean by "typical" deviation? Usually, the mean or median value of these deviations! In practice, this leads to two different measures, the "mean absolute deviation (from the mean)" and the "median absolute deviation (from the median)". From what I've read, the measure based on the median seems to be used in statistics, and does seem to be the better of the two, but to be honest I don't think I've seen it used much in psychology. The measure based on the mean does occasionally show up in psychology though. In this section I'll talk about the first one, and I'll come back to talk about the second one later.

Since the previous paragraph might sound a little abstract, let's go through the ***mean absolute deviation*** from the mean a little more slowly. One useful thing about this measure is that the name actually tells you exactly how to calculate it. Let's think about our AFL winning margins data, and once again we'll start by pretending that there's only 5 games in total, with winning margins of 56, 31, 56, 8 and 32. Since our calculations rely on an examination of the deviation from some reference point (in this case the mean), the first thing we need to calculate is the mean, \bar{X} . For these five observations, our mean is $\bar{X} = 36.6$. The next step is to convert each of our observations X_i into a deviation score. We do this by calculating the difference between the observation X_i and the mean \bar{X} . That is, the deviation score is defined to be $X_i - \bar{X}$. For the first observation in our sample, this is equal to $56 - 36.6 = 19.4$. Okay, that's simple enough. The next step in the process is to convert these deviations to absolute deviations. As we discussed earlier when talking about the `abs()` function in R (Section 3.5), we do this by converting any negative values to positive ones. Mathematically, we would denote the absolute value of -3 as $|-3|$, and so we say that $|-3| = 3$. We use the absolute value function here because we don't really care whether the value is higher than the mean or lower than the mean, we're just interested in how *close* it is to the mean. To help make this process as obvious as possible, the table below shows these calculations for all five observations:

the observation	its symbol	the observed value
winning margin, game 1	$\$X_1\$$	56 points
winning margin, game 2	$\$X_2\$$	31 points
winning margin, game 3	$\$X_3\$$	56 points
winning margin, game 4	$\$X_4\$$	8 points
winning margin, game 5	$\$X_5\$$	32 points

Now that we have calculated the absolute deviation score for every observation in the data set, all that we have to do is calculate the mean of these scores. Let's do that:

$$\frac{19.4 + 5.6 + 19.4 + 28.6 + 4.6}{5} = 15.52$$

And we're done. The mean absolute deviation for these five scores is 15.52.

However, while our calculations for this little example are at an end, we do have a couple of things left to talk about. Firstly, we should really try to write down a proper mathematical formula. But in order to do this I need some mathematical notation to refer to the mean absolute deviation. Irritatingly, "mean absolute deviation" and "median absolute deviation" have the same acronym (MAD), which leads to a certain amount of ambiguity, and since R tends to use MAD to refer to the median absolute deviation, I'd better come up with something different for the mean absolute deviation. Sigh. What I'll do is use AAD instead, short for *average absolute deviation*. Now that we have some unambiguous notation, here's the formula that describes what we just calculated:

$$(X) = \frac{1}{N} \sum_{i=1}^N |X_i - \bar{X}|$$

The last thing we need to talk about is how to calculate AAD in R. One possibility would be to do everything using low level commands, laboriously following the same steps that I used when describing the calculations above. However, that's pretty tedious. You'd end up with a series of commands that might look like this:

```
X <- c(56, 31, 56, 8, 32)      # enter the data
X.bar <- mean(X)                # step 1. the mean of the data
AD <- abs(X - X.bar)            # step 2. the absolute deviations from the mean
AAD <- mean(AD)                 # step 3. the mean absolute deviations
print(AAD)                      # print the results

## [1] 15.52
```

Each of those commands is pretty simple, but there's just too many of them. And because I find that to be too much typing, the `lsr` package has a very simple function called `aad()` that does the calculations for you. If we apply the `aad()` function to our data, we get this:

```
library(lsr)
aad(X)
```

```
## [1] 15.52
```

No surprises there.

5.2.4 Variance

Although the mean absolute deviation measure has its uses, it's not the best measure of variability to use. From a purely mathematical perspective, there are some solid reasons to prefer squared deviations rather

Table 5.1: Basic arithmetic operations in R. These five operators are used very frequently throughout the text, so it's important to be familiar with them at the outset.

Notation [English]	<code>\$i\$</code> [which game]	<code>\$X_i\$</code> [value]	<code>\$X_i - \bar{X}\$</code> [deviation from mean]	<code>\$(X_i - \bar{X})^2\$</code> [squared deviation]
1		56	19.4	376.36
2		31	-5.6	31.36
3		56	19.4	376.36
4		8	-28.6	817.96
5		32	-4.6	21.16

than absolute deviations. If we do that, we obtain a measure is called the *variance*, which has a lot of really nice statistical properties that I'm going to ignore,⁸ and $\text{Var}(Y)$ respectively. Now imagine I want to define a new variable Z that is the sum of the two, $Z = X + Y$. As it turns out, the variance of Z is equal to $\text{Var}(X) + \text{Var}(Y)$. This is a *very* useful property, but it's not true of the other measures that I talk about in this section.] and one massive psychological flaw that I'm going to make a big deal out of in a moment. The variance of a data set X is sometimes written as $\text{Var}(X)$, but it's more commonly denoted s^2 (the reason for this will become clearer shortly). The formula that we use to calculate the variance of a set of observations is as follows:

$$\text{Var}(X) = \frac{1}{N} \sum_{i=1}^N (X_i - \bar{X})^2$$

$$\text{Var}(X) = \frac{\sum_{i=1}^N (X_i - \bar{X})^2}{N}$$

As you can see, it's basically the same formula that we used to calculate the mean absolute deviation, except that instead of using “absolute deviations” we use “squared deviations”. It is for this reason that the variance is sometimes referred to as the “mean square deviation”.

Now that we've got the basic idea, let's have a look at a concrete example. Once again, let's use the first five AFL games as our data. If we follow the same approach that we took last time, we end up with the following table:

That last column contains all of our squared deviations, so all we have to do is average them. If we do that by typing all the numbers into R by hand...

```
( 376.36 + 31.36 + 376.36 + 817.96 + 21.16 ) / 5
```

```
## [1] 324.64
```

... we end up with a variance of 324.64. Exciting, isn't it? For the moment, let's ignore the burning question that you're all probably thinking (i.e., what the heck does a variance of 324.64 actually mean?) and instead talk a bit more about how to do the calculations in R, because this will reveal something very weird.

As always, we want to avoid having to type in a whole lot of numbers ourselves. And as it happens, we have the vector X lying around, which we created in the previous section. With this in mind, we can calculate the variance of X by using the following command,

```
mean( (X - mean(X))^2 )
```

```
## [1] 324.64
```

⁸Well, I will very briefly mention the one that I think is coolest, for a very particular definition of “cool”, that is. Variances are *additive*. Here's what that means: suppose I have two variables X and Y , whose variances are Var

and as usual we get the same answer as the one that we got when we did everything by hand. However, I *still* think that this is too much typing. Fortunately, R has a built in function called `var()` which does calculate variances. So we could also do this...

```
var(X)
## [1] 405.8
```

and you get the same... no, wait... you get a completely *different* answer. That's just weird. Is R broken? Is this a typo? Is Dan an idiot?

As it happens, the answer is no.⁹ It's not a typo, and R is not making a mistake. To get a feel for what's happening, let's stop using the tiny data set containing only 5 data points, and switch to the full set of 176 games that we've got stored in our `afl.margins` vector. First, let's calculate the variance by using the formula that I described above:

```
mean( (afl.margins - mean(afl.margins) )^2)
## [1] 675.9718
```

Now let's use the `var()` function:

```
var( afl.margins )
## [1] 679.8345
```

Hm. These two numbers are very similar this time. That seems like too much of a coincidence to be a mistake. And of course it isn't a mistake. In fact, it's very simple to explain what R is doing here, but slightly trickier to explain *why* R is doing it. So let's start with the "what". What R is doing is evaluating a slightly different formula to the one I showed you above. Instead of averaging the squared deviations, which requires you to divide by the number of data points N , R has chosen to divide by $N - 1$. In other words, the formula that R is using is this one

$$\frac{1}{N-1} \sum_{i=1}^N (X_i - \bar{X})^2$$

It's easy enough to verify that this is what's happening, as the following command illustrates:

```
sum( (X-mean(X))^2 ) / 4
## [1] 405.8
```

This is the same answer that R gave us originally when we calculated `var(X)` originally. So that's the *what*. The real question is *why* R is dividing by $N - 1$ and not by N . After all, the variance is supposed to be the *mean* squared deviation, right? So shouldn't we be dividing by N , the actual number of observations in the sample? Well, yes, we should. However, as we'll discuss in Chapter 10, there's a subtle distinction between "describing a sample" and "making guesses about the population from which the sample came". Up to this point, it's been a distinction without a difference. Regardless of whether you're describing a sample or drawing inferences about the population, the mean is calculated exactly the same way. Not so for the variance, or the standard deviation, or for many other measures besides. What I outlined to you

⁹With the possible exception of the third question.

initially (i.e., take the actual average, and thus divide by N) assumes that you literally intend to calculate the variance of the sample. Most of the time, however, you're not terribly interested in the sample *in and of itself*. Rather, the sample exists to tell you something about the world. If so, you're actually starting to move away from calculating a “sample statistic”, and towards the idea of estimating a “population parameter”. However, I'm getting ahead of myself. For now, let's just take it on faith that R knows what it's doing, and we'll revisit the question later on when we talk about estimation in Chapter 10.

Okay, one last thing. This section so far has read a bit like a mystery novel. I've shown you how to calculate the variance, described the weird “ $N - 1$ ” thing that R does and hinted at the reason why it's there, but I haven't mentioned the single most important thing... how do you *interpret* the variance? Descriptive statistics are supposed to describe things, after all, and right now the variance is really just a gibberish number. Unfortunately, the reason why I haven't given you the human-friendly interpretation of the variance is that there really isn't one. This is the most serious problem with the variance. Although it has some elegant mathematical properties that suggest that it really is a fundamental quantity for expressing variation, it's completely useless if you want to communicate with an actual human... variances are completely uninterpretable in terms of the original variable! All the numbers have been squared, and they don't mean anything anymore. This is a huge issue. For instance, according to the table I presented earlier, the margin in game 1 was “376.36 points-squared higher than the average margin”. This is *exactly* as stupid as it sounds; and so when we calculate a variance of 324.64, we're in the same situation. I've watched a lot of footy games, and never has anyone referred to “points squared”. It's *not* a real unit of measurement, and since the variance is expressed in terms of this gibberish unit, it is totally meaningless to a human.

5.2.5 Standard deviation

Okay, suppose that you like the idea of using the variance because of those nice mathematical properties that I haven't talked about, but – since you're a human and not a robot – you'd like to have a measure that is expressed in the same units as the data itself (i.e., points, not points-squared). What should you do? The solution to the problem is obvious: take the square root of the variance, known as the **standard deviation**, also called the “root mean squared deviation”, or RMSD. This solves out problem fairly neatly: while nobody has a clue what “a variance of 324.68 points-squared” really means, it's much easier to understand “a standard deviation of 18.01 points”, since it's expressed in the original units. It is traditional to refer to the standard deviation of a sample of data as s , though “sd” and “std dev.” are also used at times. Because the standard deviation is equal to the square root of the variance, you probably won't be surprised to see that the formula is:

$$s = \sqrt{\frac{1}{N} \sum_{i=1}^N (X_i - \bar{X})^2}$$

and the R function that we use to calculate it is `sd()`. However, as you might have guessed from our discussion of the variance, what R actually calculates is slightly different to the formula given above. Just like the we saw with the variance, what R calculates is a version that divides by $N - 1$ rather than N . For reasons that will make sense when we return to this topic in Chapter@refch:estimation I'll refer to this new quantity as $\hat{\sigma}$ (read as: “sigma hat”), and the formula for this is

$$\hat{\sigma} = \sqrt{\frac{1}{N-1} \sum_{i=1}^N (X_i - \bar{X})^2}$$

With that in mind, calculating standard deviations in R is simple:

```
sd( afl.margins )
```

```
## [1] 26.07364
```

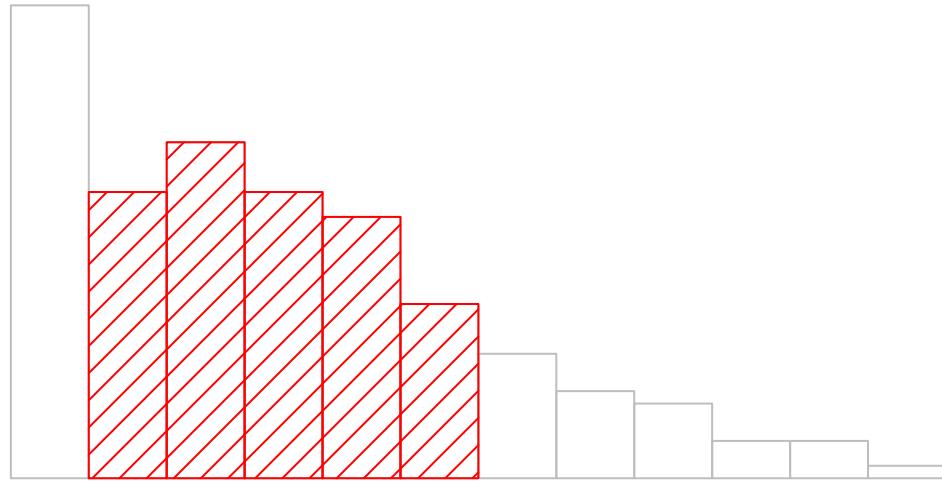


Figure 5.3: An illustration of the standard deviation, applied to the AFL winning margins data. The shaded bars in the histogram show how much of the data fall within one standard deviation of the mean. In this case, 65.3% of the data set lies within this range, which is pretty consistent with the “approximately 68% rule” discussed in the main text.

Interpreting standard deviations is slightly more complex. Because the standard deviation is derived from the variance, and the variance is a quantity that has little to no meaning that makes sense to us humans, the standard deviation doesn’t have a simple interpretation. As a consequence, most of us just rely on a simple rule of thumb: in general, you should expect 68% of the data to fall within 1 standard deviation of the mean, 95% of the data to fall within 2 standard deviation of the mean, and 99.7% of the data to fall within 3 standard deviations of the mean. This rule tends to work pretty well most of the time, but it’s not exact: it’s actually calculated based on an *assumption* that the histogram is symmetric and “bell shaped”.¹⁰ As you can tell from looking at the AFL winning margins histogram in Figure 5.1, this isn’t exactly true of our data! Even so, the rule is approximately correct. As it turns out, 65.3% of the AFL margins data fall within one standard deviation of the mean. This is shown visually in Figure 5.3.

5.2.6 Median absolute deviation

The last measure of variability that I want to talk about is the ***median absolute deviation*** (MAD). The basic idea behind MAD is very simple, and is pretty much identical to the idea behind the mean absolute deviation (Section 5.2.3). The difference is that you use the median everywhere. If we were to frame this idea as a pair of R commands, they would look like this:

¹⁰Strictly, the assumption is that the data are *normally* distributed, which is an important concept that we’ll discuss more in Chapter 9, and will turn up over and over again later in the book.

```
# mean absolute deviation from the mean:
mean( abs(afl.margins - mean(afl.margins)) )

## [1] 21.10124

# *median* absolute deviation from the *median*:
median( abs(afl.margins - median(afl.margins)) )

## [1] 19.5
```

This has a straightforward interpretation: every observation in the data set lies some distance away from the typical value (the median). So the MAD is an attempt to describe a *typical deviation from a typical value* in the data set. It wouldn't be unreasonable to interpret the MAD value of 19.5 for our AFL data by saying something like this:

The median winning margin in 2010 was 30.5, indicating that a typical game involved a winning margin of about 30 points. However, there was a fair amount of variation from game to game: the MAD value was 19.5, indicating that a typical winning margin would differ from this median value by about 19-20 points.

As you'd expect, R has a built in function for calculating MAD, and you will be shocked no doubt to hear that it's called `mad()`. However, it's a little bit more complicated than the functions that we've been using previously. If you want to use it to calculate MAD in the exact same way that I have described it above, the command that you need to use specifies two arguments: the data set itself `x`, and a `constant` that I'll explain in a moment. For our purposes, the constant is 1, so our command becomes

```
mad( x = afl.margins, constant = 1 )

## [1] 19.5
```

Apart from the weirdness of having to type that `constant = 1` part, this is pretty straightforward.

Okay, so what exactly is this `constant = 1` argument? I won't go into all the details here, but here's the gist. Although the "raw" MAD value that I've described above is completely interpretable on its own terms, that's not actually how it's used in a lot of real world contexts. Instead, what happens a lot is that the researcher *actually* wants to calculate the standard deviation. However, in the same way that the mean is very sensitive to extreme values, the standard deviation is vulnerable to the exact same issue. So, in much the same way that people sometimes use the median as a "robust" way of calculating "something that is like the mean", it's not uncommon to use MAD as a method for calculating "something that is like the standard deviation". Unfortunately, the *raw* MAD value doesn't do this. Our raw MAD value is 19.5, and our standard deviation was 26.07. However, what some clever person has shown is that, under certain assumptions¹¹, you can multiply the raw MAD value by 1.4826 and obtain a number that is directly comparable to the standard deviation. As a consequence, the default value of `constant` is 1.4826, and so when you use the `mad()` command without manually setting a value, here's what you get:

```
mad( afl.margins )

## [1] 28.9107
```

I should point out, though, that if you want to use this "corrected" MAD value as a robust version of the standard deviation, you really are relying on the assumption that the data are (or at least, are "supposed to be" in some sense) symmetric and basically shaped like a bell curve. That's really *not* true for our `afl.margins` data, so in this case I wouldn't try to use the MAD value this way.

¹¹The assumption again being that the data are normally-distributed!

5.2.7 Which measure to use?

We've discussed quite a few measures of spread (range, IQR, MAD, variance and standard deviation), and hinted at their strengths and weaknesses. Here's a quick summary:

- *Range*. Gives you the full spread of the data. It's very vulnerable to outliers, and as a consequence it isn't often used unless you have good reasons to care about the extremes in the data.
- *Interquartile range*. Tells you where the "middle half" of the data sits. It's pretty robust, and complements the median nicely. This is used a lot.
- *Mean absolute deviation*. Tells you how far "on average" the observations are from the mean. It's very interpretable, but has a few minor issues (not discussed here) that make it less attractive to statisticians than the standard deviation. Used sometimes, but not often.
- *Variance*. Tells you the average squared deviation from the mean. It's mathematically elegant, and is probably the "right" way to describe variation around the mean, but it's completely uninterpretable because it doesn't use the same units as the data. Almost never used except as a mathematical tool; but it's buried "under the hood" of a very large number of statistical tools.
- *Standard deviation*. This is the square root of the variance. It's fairly elegant mathematically, and it's expressed in the same units as the data so it can be interpreted pretty well. In situations where the mean is the measure of central tendency, this is the default. This is by far the most popular measure of variation.
- *Median absolute deviation*. The typical (i.e., median) deviation from the median value. In the raw form it's simple and interpretable; in the corrected form it's a robust way to estimate the standard deviation, for some kinds of data sets. Not used very often, but it does get reported sometimes.

In short, the IQR and the standard deviation are easily the two most common measures used to report the variability of the data; but there are situations in which the others are used. I've described all of them in this book because there's a fair chance you'll run into most of these somewhere.

5.3 Skew and kurtosis{#skew}

There are two more descriptive statistics that you will sometimes see reported in the psychological literature, known as skew and kurtosis. In practice, neither one is used anywhere near as frequently as the measures of central tendency and variability that we've been talking about. Skew is pretty important, so you do see it mentioned a fair bit; but I've actually never seen kurtosis reported in a scientific article to date.

```
## [1] -0.919309
## [1] 0.01220865
## [1] 0.9218069
```

Since it's the more interesting of the two, let's start by talking about the *skewness*. Skewness is basically a measure of asymmetry, and the easiest way to explain it is by drawing some pictures. As Figure 5.4 illustrates, if the data tend to have a lot of extreme small values (i.e., the lower tail is "longer" than the upper tail) and not so many extremely large values (left panel), then we say that the data are *negatively skewed*. On the other hand, if there are more extremely large values than extremely small ones (right panel) we say that the data are *positively skewed*. That's the qualitative idea behind skewness. The actual formula for the skewness of a data set is as follows

$$\text{skewness}(X) = \frac{1}{N\hat{\sigma}^3} \sum_{i=1}^N (X_i - \bar{X})^3$$

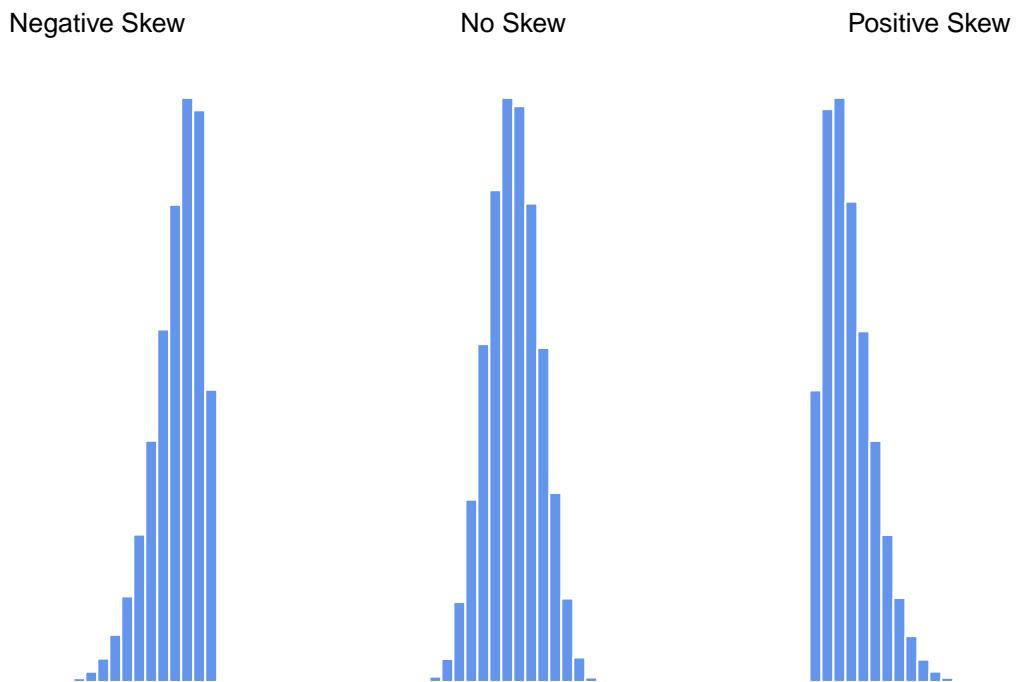


Figure 5.4: An illustration of skewness. On the left we have a negatively skewed data set (skewness = $-.93$), in the middle we have a data set with no skew (technically, skewness = $-.006$), and on the right we have a positively skewed data set (skewness = $.93$).

where N is the number of observations, \bar{X} is the sample mean, and $\hat{\sigma}$ is the standard deviation (the “divide by $N - 1$ ” version, that is). Perhaps more helpfully, it might be useful to point out that the `psych` package contains a `skew()` function that you can use to calculate skewness. So if we wanted to use this function to calculate the skewness of the `afl.margins` data, we’d first need to load the package

```
library( psych )
```

which now makes it possible to use the following command:

```
skew( x = afl.margins )
```

```
## [1] 0.7671555
```

Not surprisingly, it turns out that the AFL winning margins data is fairly skewed.

The final measure that is sometimes referred to, though very rarely in practice, is the **kurtosis** of a data set. Put simply, kurtosis is a measure of the “pointiness” of a data set, as illustrated in Figure 5.5.

```
## [1] -0.9579798
```

```
## [1] 0.01636593
```

```
## [1] 2.021532
```

By convention, we say that the “normal curve” (black lines) has zero kurtosis, so the pointiness of a data set is assessed relative to this curve. In this Figure, the data on the left are not pointy enough, so the kurtosis is negative and we call the data *platykurtic*. The data on the right are too pointy, so the kurtosis is positive and we say that the data is *leptokurtic*. But the data in the middle are just pointy enough, so we say that it is *mesokurtic* and has kurtosis zero. This is summarised in the table below:

informal term	technical name	kurtosis value
too flat	platykurtic	negative
just pointy enough	mesokurtic	zero
too pointy	leptokurtic	positive

The equation for kurtosis is pretty similar in spirit to the formulas we’ve seen already for the variance and the skewness; except that where the variance involved squared deviations and the skewness involved cubed deviations, the kurtosis involves raising the deviations to the fourth power:¹²

$$\text{kurtosis}(X) = \frac{1}{N\hat{\sigma}^4} \sum_{i=1}^N (X_i - \bar{X})^4 - 3$$

I know, it’s not terribly interesting to me either. More to the point, the `psych` package has a function called `kurtosi()` that you can use to calculate the kurtosis of your data. For instance, if we were to do this for the AFL margins,

```
kurtosi( x = afl.margins )
```

```
## [1] 0.02962633
```

we discover that the AFL winning margins data are just pointy enough.

¹²The “-3” part is something that statisticians tack on to ensure that the normal curve has kurtosis zero. It looks a bit stupid, just sticking a “-3” at the end of the formula, but there are good mathematical reasons for doing this.

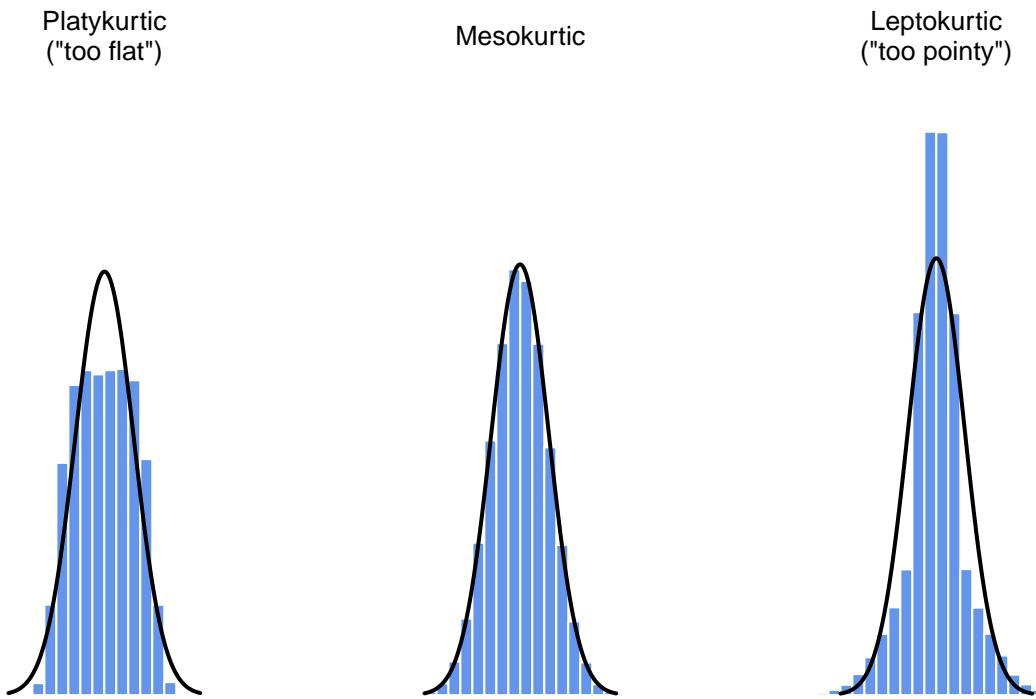


Figure 5.5: An illustration of kurtosis. On the left, we have a “platykurtic” data set ($kurtosis = -.95$), meaning that the data set is “too flat”. In the middle we have a “mesokurtic” data set ($kurtosis$ is almost exactly 0), which means that the pointiness of the data is just about right. Finally, on the right, we have a “leptokurtic” data set ($kurtosis = 2.12$) indicating that the data set is “too pointy”. Note that kurtosis is measured with respect to a normal curve (black line)

5.4 Getting an overall summary of a variable

Up to this point in the chapter I've explained several different summary statistics that are commonly used when analysing data, along with specific functions that you can use in R to calculate each one. However, it's kind of annoying to have to separately calculate means, medians, standard deviations, skewness etc. Wouldn't it be nice if R had some helpful functions that would do all these tedious calculations at once? Something like `summary()` or `describe()`, perhaps? Why yes, yes it would. So much so that both of these functions exist. The `summary()` function is in the `base` package, so it comes with every installation of R. The `describe()` function is part of the `psych` package, which we loaded earlier in the chapter.

5.4.1 “Summarising” a variable

The `summary()` function is an easy thing to use, but a tricky thing to understand in full, since it's a generic function (see Section 4.11). The basic idea behind the `summary()` function is that it prints out some useful information about whatever object (i.e., variable, as far as we're concerned) you specify as the `object` argument. As a consequence, the behaviour of the `summary()` function differs quite dramatically depending on the class of the object that you give it. Let's start by giving it a *numeric* object:

```
summary( object = afl.margins )
```

```
##      Min. 1st Qu. Median     Mean 3rd Qu.    Max.
##      0.00   12.75  30.50  35.30  50.50 116.00
```

For numeric variables, we get a whole bunch of useful descriptive statistics. It gives us the minimum and maximum values (i.e., the range), the first and third quartiles (25th and 75th percentiles; i.e., the IQR), the mean and the median. In other words, it gives us a pretty good collection of descriptive statistics related to the central tendency and the spread of the data.

Okay, what about if we feed it a logical vector instead? Let's say I want to know something about how many “blowouts” there were in the 2010 AFL season. I operationalise the concept of a blowout (see Chapter 2) as a game in which the winning margin exceeds 50 points. Let's create a logical variable `blowouts` in which the *i*-th element is `TRUE` if that game was a blowout according to my definition,

```
blowouts <- afl.margins > 50
blowouts
```

```
## [1] TRUE FALSE TRUE FALSE FALSE FALSE TRUE FALSE FALSE FALSE
## [12] TRUE FALSE FALSE TRUE FALSE FALSE FALSE FALSE TRUE FALSE FALSE
## [23] FALSE FALSE FALSE FALSE FALSE TRUE FALSE TRUE TRUE FALSE FALSE
## [34] TRUE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [45] FALSE TRUE TRUE FALSE FALSE FALSE FALSE FALSE TRUE TRUE FALSE
## [56] TRUE FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE
## [67] TRUE FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE
## [78] FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [89] FALSE FALSE TRUE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE
## [100] FALSE TRUE FALSE FALSE FALSE TRUE FALSE TRUE TRUE TRUE FALSE
## [111] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE
## [122] TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE TRUE FALSE
## [133] FALSE TRUE TRUE FALSE FALSE FALSE FALSE TRUE FALSE TRUE
## [144] TRUE TRUE TRUE FALSE FALSE FALSE TRUE FALSE FALSE TRUE FALSE
## [155] TRUE FALSE TRUE FALSE FALSE TRUE FALSE FALSE TRUE FALSE FALSE
## [166] FALSE FALSE
```

So that's what the `blowouts` variable looks like. Now let's ask R for a `summary()`

```
summary( object = blowouts )
```

```
##      Mode    FALSE     TRUE
## logical     132      44
```

In this context, the `summary()` function gives us a count of the number of `TRUE` values, the number of `FALSE` values, and the number of missing values (i.e., the `NAs`). Pretty reasonable behaviour.

Next, let's try to give it a factor. If you recall, I've defined the `afl.finalists` vector as a factor, so let's use that:

```
summary( object = afl.finalists )
```

```
##          Adelaide      Brisbane      Carlton      Collingwood
##                  26             25             26              28
##          Essendon      Fitzroy      Fremantle      Geelong
##                  32                 0               6              39
##          Hawthorn      Melbourne  North Melbourne  Port Adelaide
##                  27                 28               28              17
##          Richmond      St Kilda      Sydney      West Coast
##                  6                  24               26              38
##  Western Bulldogs
##                  24
```

For factors, we get a frequency table, just like we got when we used the `table()` function. Interestingly, however, if we convert this to a character vector using the `as.character()` function (see Section 7.10, we don't get the same results:

```
f2 <- as.character( afl.finalists )
summary( object = f2 )
```

```
##      Length     Class      Mode
##        400  character  character
```

This is one of those situations I was referring to in Section 4.7, in which it is helpful to declare your nominal scale variable as a factor rather than a character vector. Because I've defined `afl.finalists` as a factor, R *knows* that it should treat it as a nominal scale variable, and so it gives you a much more detailed (and helpful) summary than it would have if I'd left it as a character vector.

5.4.2 “Summarising” a data frame

Okay what about data frames? When you pass a data frame to the `summary()` function, it produces a slightly condensed summary of each variable inside the data frame. To give you a sense of how this can be useful, let's try this for a new data set, one that you've never seen before. The data is stored in the `clinicaltrial.Rdata` file, and we'll use it a lot in Chapter ?? (you can find a complete description of the data at the start of that chapter). Let's load it, and see what we've got:

```
load( "./data/clinicaltrial.Rdata" )
who(TRUE)

##      -- Name --      -- Class --      -- Size --
##   clin.trial    data.frame     18 x 3
##   $drug        factor         18
##   $therapy     factor         18
##   $mood.gain  numeric        18
```

There's a single data frame called `clin.trial` which contains three variables, `drug`, `therapy` and `mood.gain`. Presumably then, this data is from a clinical trial of some kind, in which people were administered different drugs; and the researchers looked to see what the drugs did to their mood. Let's see if the `summary()` function sheds a little more light on this situation:

```
summary( clin.trial )

##          drug        therapy      mood.gain
## placebo :6    no.therapy:9   Min.   :0.1000
## anxifree:6   CBT       :9   1st Qu.:0.4250
## joyzepam:6   Median    :0.8500
##                   Mean     :0.8833
##                   3rd Qu.:1.3000
##                   Max.    :1.8000
```

Evidently there were three drugs: a placebo, something called “anxitfree” and something called “joyzepam”; and there were 6 people administered each drug. There were 9 people treated using cognitive behavioural therapy (CBT) and 9 people who received no psychological treatment. And we can see from looking at the summary of the `mood.gain` variable that most people did show a mood gain (mean = .88), though without knowing what the scale is here it's hard to say much more than that. Still, that's not too bad. Overall, I feel that I learned something from that.

5.4.3 “Describing” a data frame

The `describe()` function (in the `psych` package) is a little different, and it's really only intended to be useful when your data are interval or ratio scale. Unlike the `summary()` function, it calculates the same descriptive statistics for any type of variable you give it. By default, these are:

- `var`. This is just an index: 1 for the first variable, 2 for the second variable, and so on.
- `n`. This is the sample size: more precisely, it's the number of non-missing values.
- `mean`. This is the sample mean (Section 5.1.1).
- `sd`. This is the (bias corrected) standard deviation (Section 5.2.5).
- `median`. The median (Section 5.1.3).
- `trimmed`. This is trimmed mean. By default it's the 10% trimmed mean (Section 5.1.6).
- `mad`. The median absolute deviation (Section 5.2.6).
- `min`. The minimum value.
- `max`. The maximum value.
- `range`. The range spanned by the data (Section 5.2.1).
- `skew`. The skewness (Section ??).
- `kurtosis`. The kurtosis (Section 5.3).
- `se`. The standard error of the mean (Chapter 10).

Notice that these descriptive statistics generally only make sense for data that are interval or ratio scale (usually encoded as numeric vectors). For nominal or ordinal variables (usually encoded as factors), most of these descriptive statistics are not all that useful. What the `describe()` function does is convert factors and logical variables to numeric vectors in order to do the calculations. These variables are marked with * and most of the time, the descriptive statistics for those variables won't make much sense. If you try to feed it a data frame that includes a character vector as a variable, it produces an error.

With those caveats in mind, let's use the `describe()` function to have a look at the `clin.trial` data frame. Here's what we get:

```
describe( x = clin.trial )

##           vars   n  mean    sd median trimmed  mad min max range skew
## drug*          1 18  2.00  0.84    2.00    2.00 1.48 1.0 3.0    2.0  0.00
## therapy*       2 18  1.50  0.51    1.50    1.50 0.74 1.0 2.0    1.0  0.00
## mood.gain      3 18  0.88  0.53    0.85    0.88 0.67 0.1 1.8    1.7  0.13
##             kurtosis   se
## drug*          -1.66 0.20
## therapy*        -2.11 0.12
## mood.gain      -1.44 0.13
```

As you can see, the output for the asterisked variables is pretty meaningless, and should be ignored. However, for the `mood.gain` variable, there's a lot of useful information.

5.5 Descriptive statistics separately for each group

It is very commonly the case that you find yourself needing to look at descriptive statistics, broken down by some grouping variable. This is pretty easy to do in R, and there are three functions in particular that are worth knowing about: `by()`, `describeBy()` and `aggregate()`. Let's start with the `describeBy()` function, which is part of the `psych` package. The `describeBy()` function is very similar to the `describe()` function, except that it has an additional argument called `group` which specifies a grouping variable. For instance, let's say, I want to look at the descriptive statistics for the `clin.trial` data, broken down separately by `therapy` type. The command I would use here is:

```
describeBy( x=clin.trial, group=clin.trial$therapy )

##
##  Descriptive statistics by group
##  group: no.therapy
##           vars   n  mean    sd median trimmed  mad min max range skew
## drug*          1  9  2.00  0.87    2.00    2.00 1.48 1.0 3.0    2.0  0.00    -1.81
## therapy*       2  9  1.00  0.00    1.00    1.00 0.00 1.0 1.0     0.0  NaN      NaN
## mood.gain      3  9  0.72  0.59    0.72    0.72 0.44 0.1 1.7    1.6  0.51    -1.59
##             se
## drug*          0.29
## therapy*        0.00
## mood.gain      0.20
## -----
##  group: CBT
##           vars   n  mean    sd median trimmed  mad min max range skew
## drug*          1  9  2.00  0.87    2.00    2.00 1.48 1.0 3.0    2.0  0.00
## therapy*       2  9  2.00  0.00    2.00    2.00 0.00 2.0 2.0     0.0  NaN
```

```
## mood.gain    3 9 1.04 0.45    1.1    1.04 0.44 0.3 1.8    1.5 -0.03
##                 kurtosis   se
## drug*        -1.81 0.29
## therapy*      NaN 0.00
## mood.gain    -1.12 0.15
```

As you can see, the output is essentially identical to the output that the `describe()` function produce, except that the output now gives you means, standard deviations etc separately for the CBT group and the no.therapy group. Notice that, as before, the output displays asterisks for factor variables, in order to draw your attention to the fact that the descriptive statistics that it has calculated won't be very meaningful for those variables. Nevertheless, this command has given us some really useful descriptive statistics `mood.gain` variable, broken down as a function of `therapy`.

A somewhat more general solution is offered by the `by()` function. There are three arguments that you need to specify when using this function: the `data` argument specifies the data set, the `INDICES` argument specifies the grouping variable, and the `FUN` argument specifies the name of a function that you want to apply separately to each group. To give a sense of how powerful this is, you can reproduce the `describeBy()` function by using a command like this:

```
by( data=clin.trial, INDICES=clin.trial$therapy, FUN=describe )
```

```
## clin.trial$therapy: no.therapy
##           vars n mean   sd median trimmed  mad min max range skew kurtosis
## drug*       1 9 2.00 0.87    2.0    2.00 1.48 1.0 3.0    2.0 0.00    -1.81
## therapy*    2 9 1.00 0.00    1.0    1.00 0.00 1.0 1.0    0.0  NaN     NaN
## mood.gain   3 9 0.72 0.59    0.5    0.72 0.44 0.1 1.7    1.6 0.51    -1.59
##                 se
## drug*       0.29
## therapy*    0.00
## mood.gain   0.20
##
## -----
## clin.trial$therapy: CBT
##           vars n mean   sd median trimmed  mad min max range skew
## drug*       1 9 2.00 0.87    2.0    2.00 1.48 1.0 3.0    2.0 0.00
## therapy*    2 9 2.00 0.00    2.0    2.00 0.00 2.0 2.0    0.0  NaN
## mood.gain   3 9 1.04 0.45    1.1    1.04 0.44 0.3 1.8    1.5 -0.03
##                 kurtosis   se
## drug*        -1.81 0.29
## therapy*      NaN 0.00
## mood.gain    -1.12 0.15
```

This will produce the exact same output as the command shown earlier. However, there's nothing special about the `describe()` function. You could just as easily use the `by()` function in conjunction with the `summary()` function. For example:

```
by( data=clin.trial, INDICES=clin.trial$therapy, FUN=summary )
```

```
## clin.trial$therapy: no.therapy
##          drug      therapy      mood.gain
## placebo :3  no.therapy:9  Min.  :0.1000
## anxifree:3    CBT      :0  1st Qu.:0.3000
## joyzepam:3            Median :0.5000
##                      Mean   :0.7222
```

```

##                               3rd Qu.:1.3000
##                               Max.    :1.7000
## -----
## clin.trial$therapy: CBT
##      drug      therapy mood.gain
##  placebo :3  no.therapy:0   Min.    :0.300
##  anxifree:3    CBT       :9   1st Qu.:0.800
##  joyzepam:3          Median :1.100
##                               Mean    :1.044
##                               3rd Qu.:1.300
##                               Max.    :1.800

```

Again, this output is pretty easy to interpret. It's the output of the `summary()` function, applied separately to CBT group and the no.therapy group. For the two factors (`drug` and `therapy`) it prints out a frequency table, whereas for the numeric variable (`mood.gain`) it prints out the range, interquartile range, mean and median.

What if you have multiple grouping variables? Suppose, for example, you would like to look at the average mood gain separately for all possible combinations of drug and therapy. It is actually possible to do this using the `by()` and `describeBy()` functions, but I usually find it more convenient to use the `aggregate()` function in this situation. There are again three arguments that you need to specify. The `formula` argument is used to indicate which variable you want to analyse, and which variables are used to specify the groups. For instance, if you want to look at `mood.gain` separately for each possible combination of `drug` and `therapy`, the formula you want is `mood.gain ~ drug + therapy`. The `data` argument is used to specify the data frame containing all the data, and the `FUN` argument is used to indicate what function you want to calculate for each group (e.g., the `mean`). So, to obtain group means, use this command:

```

aggregate( formula = mood.gain ~ drug + therapy, # mood.gain by drug/therapy combination
            data = clin.trial,                      # data is in the clin.trial data frame
            FUN = mean)                           # print out group means
)

```

```

##      drug      therapy mood.gain
## 1 placebo no.therapy  0.300000
## 2 anxifree no.therapy  0.400000
## 3 joyzepam no.therapy 1.466667
## 4 placebo       CBT  0.600000
## 5 anxifree       CBT  1.033333
## 6 joyzepam       CBT  1.500000

```

or, alternatively, if you want to calculate the standard deviations for each group, you would use the following command (argument names omitted this time):

```
aggregate( mood.gain ~ drug + therapy, clin.trial, sd )
```

```

##      drug      therapy mood.gain
## 1 placebo no.therapy 0.2000000
## 2 anxifree no.therapy 0.2000000
## 3 joyzepam no.therapy 0.2081666
## 4 placebo       CBT  0.3000000
## 5 anxifree       CBT  0.2081666
## 6 joyzepam       CBT  0.2645751

```

5.6 Standard scores

Suppose my friend is putting together a new questionnaire intended to measure “grumpiness”. The survey has 50 questions, which you can answer in a grumpy way or not. Across a big sample (hypothetically, let’s imagine a million people or so!) the data are fairly normally distributed, with the mean grumpiness score being 17 out of 50 questions answered in a grumpy way, and the standard deviation is 5. In contrast, when I take the questionnaire, I answer 35 out of 50 questions in a grumpy way. So, how grumpy am I? One way to think about would be to say that I have grumpiness of $35/50$, so you might say that I’m 70% grumpy. But that’s a bit weird, when you think about it. If my friend had phrased her questions a bit differently, people might have answered them in a different way, so the overall distribution of answers could easily move up or down depending on the precise way in which the questions were asked. So, I’m only 70% grumpy *with respect to this set of survey questions*. Even if it’s a very good questionnaire, this isn’t very a informative statement.

A simpler way around this is to describe my grumpiness by comparing me to other people. Shockingly, out of my friend’s sample of 1,000,000 people, only 159 people were as grumpy as me (that’s not at all unrealistic, frankly), suggesting that I’m in the top 0.016% of people for grumpiness. This makes much more sense than trying to interpret the raw data. This idea – that we should describe my grumpiness in terms of the overall distribution of the grumpiness of humans – is the qualitative idea that standardisation attempts to get at. One way to do this is to do exactly what I just did, and describe everything in terms of percentiles. However, the problem with doing this is that “it’s lonely at the top”. Suppose that my friend had only collected a sample of 1000 people (still a pretty big sample for the purposes of testing a new questionnaire, I’d like to add), and this time gotten a mean of 16 out of 50 with a standard deviation of 5, let’s say. The problem is that almost certainly, not a single person in that sample would be as grumpy as me.

However, all is not lost. A different approach is to convert my grumpiness score into a *standard score*, also referred to as a *z-score*. The standard score is defined as the number of standard deviations above the mean that my grumpiness score lies. To phrase it in “pseudo-maths” the standard score is calculated like this:

$$\text{standard score} = \frac{\text{raw score} - \text{mean}}{\text{standard deviation}}$$

In actual maths, the equation for the *z-score* is

$$z_i = \frac{X_i - \bar{X}}{\hat{\sigma}}$$

So, going back to the grumpiness data, we can now transform Dan’s raw grumpiness into a standardised grumpiness score.¹³ If the mean is 17 and the standard deviation is 5 then my standardised grumpiness score would be¹⁴

$$z = \frac{35 - 17}{5} = 3.6$$

To interpret this value, recall the rough heuristic that I provided in Section 5.2.5, in which I noted that 99.7% of values are expected to lie within 3 standard deviations of the mean. So the fact that my grumpiness corresponds to a *z* score of 3.6 indicates that I’m very grumpy indeed. Later on, in Section 9.5, I’ll introduce a function called `pnorm()` that allows us to be a bit more precise than this. Specifically, it allows us to calculate a theoretical percentile rank for my grumpiness, as follows:

¹³I haven’t discussed how to compute *z*-scores, explicitly, but you can probably guess. For a variable `X`, the simplest way is to use a command like `(X - mean(X)) / sd(X)`. There’s also a fancier function called `scale()` that you can use, but it relies on somewhat more complicated R concepts that I haven’t explained yet.

¹⁴Technically, because I’m calculating means and standard deviations from a sample of data, but want to talk about my grumpiness relative to a population, what I’m actually doing is *estimating* a *z* score. However, since we haven’t talked about estimation yet (see Chapter 10) I think it’s best to ignore this subtlety, especially as it makes very little difference to our calculations.

```
pnorm( 3.6 )
```

```
## [1] 0.9998409
```

At this stage, this command doesn't make too much sense, but don't worry too much about it. It's not important for now. But the output is fairly straightforward: it suggests that I'm grumpier than 99.98% of people. Sounds about right.

In addition to allowing you to interpret a raw score in relation to a larger population (and thereby allowing you to make sense of variables that lie on arbitrary scales), standard scores serve a second useful function. Standard scores can be compared to one another in situations where the raw scores can't. Suppose, for instance, my friend also had another questionnaire that measured extraversion using a 24 items questionnaire. The overall mean for this measure turns out to be 13 with standard deviation 4; and I scored a 2. As you can imagine, it doesn't make a lot of sense to try to compare my raw score of 2 on the extraversion questionnaire to my raw score of 35 on the grumpiness questionnaire. The raw scores for the two variables are "about" fundamentally different things, so this would be like comparing apples to oranges.

What about the standard scores? Well, this is a little different. If we calculate the standard scores, we get $z = (35 - 17)/5 = 3.6$ for grumpiness and $z = (2 - 13)/4 = -2.75$ for extraversion. These two numbers *can* be compared to each other.¹⁵ I'm much less extraverted than most people ($z = -2.75$) and much grumpier than most people ($z = 3.6$): but the extent of my unusualness is much more extreme for grumpiness (since 3.6 is a bigger number than 2.75). Because each standardised score is a statement about where an observation falls *relative to its own population*, it *is* possible to compare standardised scores across completely different variables.

5.7 Correlations

Up to this point we have focused entirely on how to construct descriptive statistics for a single variable. What we haven't done is talked about how to describe the relationships *between* variables in the data. To do that, we want to talk mostly about the ***correlation*** between variables. But first, we need some data.

5.7.1 The data

After spending so much time looking at the AFL data, I'm starting to get bored with sports. Instead, let's turn to a topic close to every parent's heart: sleep. The following data set is fictitious, but based on real events. Suppose I'm curious to find out how much my infant son's sleeping habits affect my mood. Let's say that I can rate my grumpiness very precisely, on a scale from 0 (not at all grumpy) to 100 (grumpy as a very, very grumpy old man). And, lets also assume that I've been measuring my grumpiness, my sleeping patterns and my son's sleeping patterns for quite some time now. Let's say, for 100 days. And, being a nerd, I've saved the data as a file called `parenthood.Rdata`. If we load the data...

```
load( "./data/parenthood.Rdata" )
who(TRUE)
```

```
##    -- Name --    -- Class --    -- Size --
##    parenthood    data.frame    100 x 4
##    $dan.sleep    numeric      100
##    $baby.sleep   numeric      100
```

¹⁵Though some caution is usually warranted. It's not always the case that one standard deviation on variable A corresponds to the same "kind" of thing as one standard deviation on variable B. Use common sense when trying to determine whether or not the z scores of two variables can be meaningfully compared.

```
## $dan.grump    numeric      100
## $day         integer      100
```

... we see that the file contains a single data frame called `parenthood`, which contains four variables `dan.sleep`, `baby.sleep`, `dangrump` and `day`. If we peek at the data using `head()` out the data, here's what we get:

```
head(parenthood, 10)
```

```
##   dan.sleep baby.sleep dan.grump day
## 1     7.59     10.18      56    1
## 2     7.91     11.66      60    2
## 3     5.14      7.92      82    3
## 4     7.71      9.61      55    4
## 5     6.68      9.75      67    5
## 6     5.99      5.04      72    6
## 7     8.19     10.45      53    7
## 8     7.19      8.27      60    8
## 9     7.40      6.06      60    9
## 10    6.58      7.09      71   10
```

Next, I'll calculate some basic descriptive statistics:

```
describe( parenthood )
```

```
##          vars   n  mean     sd median trimmed   mad   min   max range
## dan.sleep    1 100  6.97  1.02    7.03    7.00  1.09  4.84  9.00  4.16
## baby.sleep   2 100  8.05  2.07    7.95    8.05  2.33  3.25 12.07  8.82
## dan.grump   3 100 63.71 10.05   62.00   63.16  9.64 41.00 91.00 50.00
## day         4 100 50.50 29.01   50.50   50.50 37.06  1.00 100.00 99.00
##                  skew kurtosis    se
## dan.sleep  -0.29    -0.72 0.10
## baby.sleep -0.02    -0.69 0.21
## dan.grump  0.43    -0.16 1.00
## day        0.00    -1.24 2.90
```

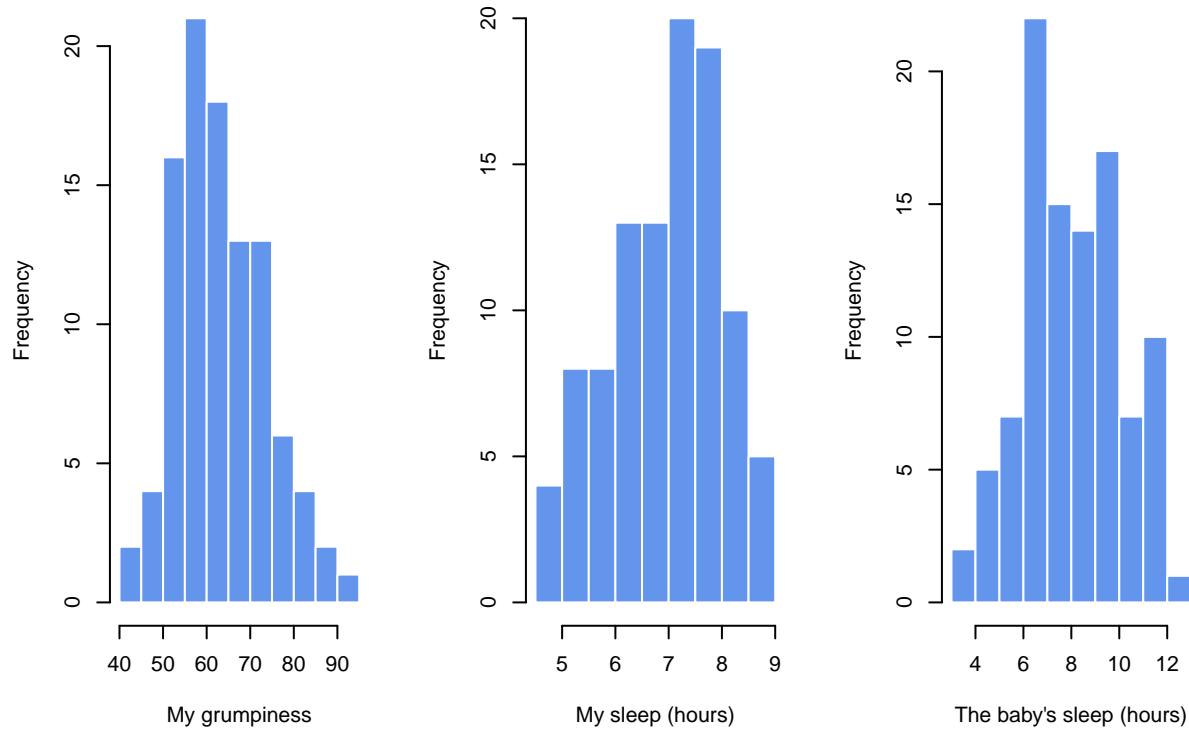
Finally, to give a graphical depiction of what each of the three interesting variables looks like, Figure 5.6 plots histograms.

One thing to note: just because R can calculate dozens of different statistics doesn't mean you should report all of them. If I were writing this up for a report, I'd probably pick out those statistics that are of most interest to me (and to my readership), and then put them into a nice, simple table like the one in Table ??.¹⁶ Notice that when I put it into a table, I gave everything "human readable" names. This is always good practice. Notice also that I'm not getting enough sleep. This isn't good practice, but other parents tell me that it's standard practice.

5.7.2 The strength and direction of a relationship

We can draw scatterplots to give us a general sense of how closely related two variables are. Ideally though, we might want to say a bit more about it than that. For instance, let's compare the relationship between

¹⁶Actually, even that table is more than I'd bother with. In practice most people pick *one* measure of central tendency, and *one* measure of variability only.

Figure 5.6: Histograms for the three interesting variables in the `parenthood` data setTable 5.2: Descriptive statistics for the `parenthood` data.

variable	min	max	mean	median	std. dev	IQR
Dan's grumpiness	41	91	63.71	62	10.05	14
Dan's hours slept	4.84	9	6.97	7.03	1.02	1.45
Dan's son's hours slept	3.25	12.07	8.05	7.95	2.07	3.21

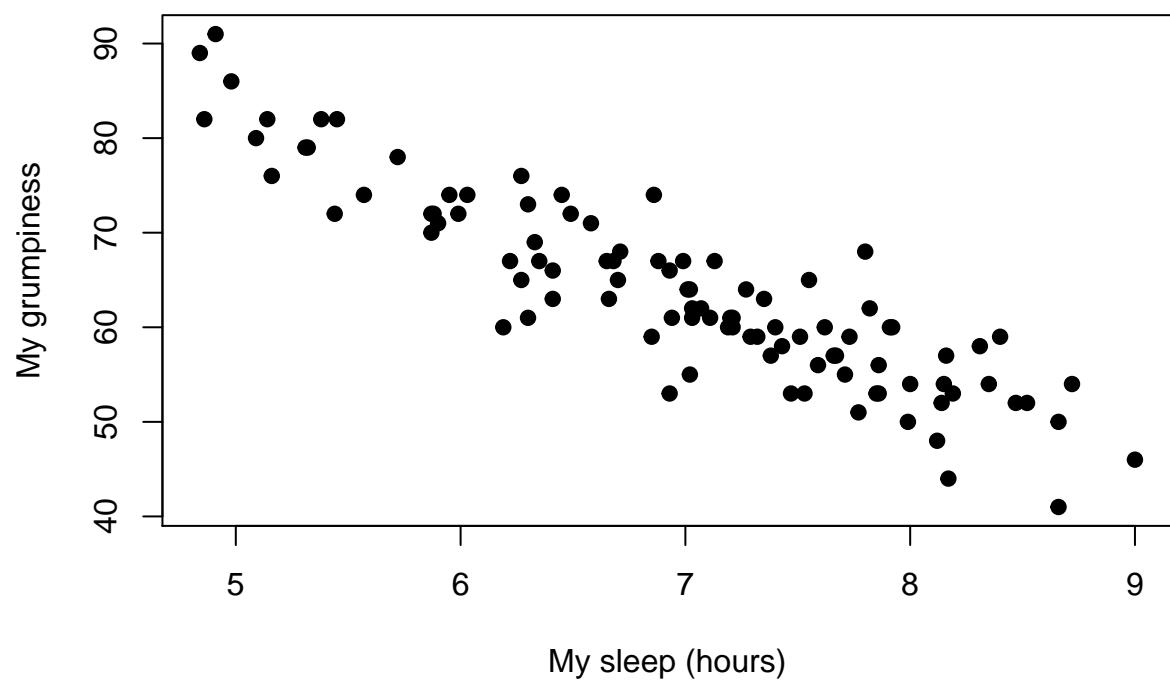


Figure 5.7: Scatterplot showing the relationship between `dan.sleep` and `dan.grump`

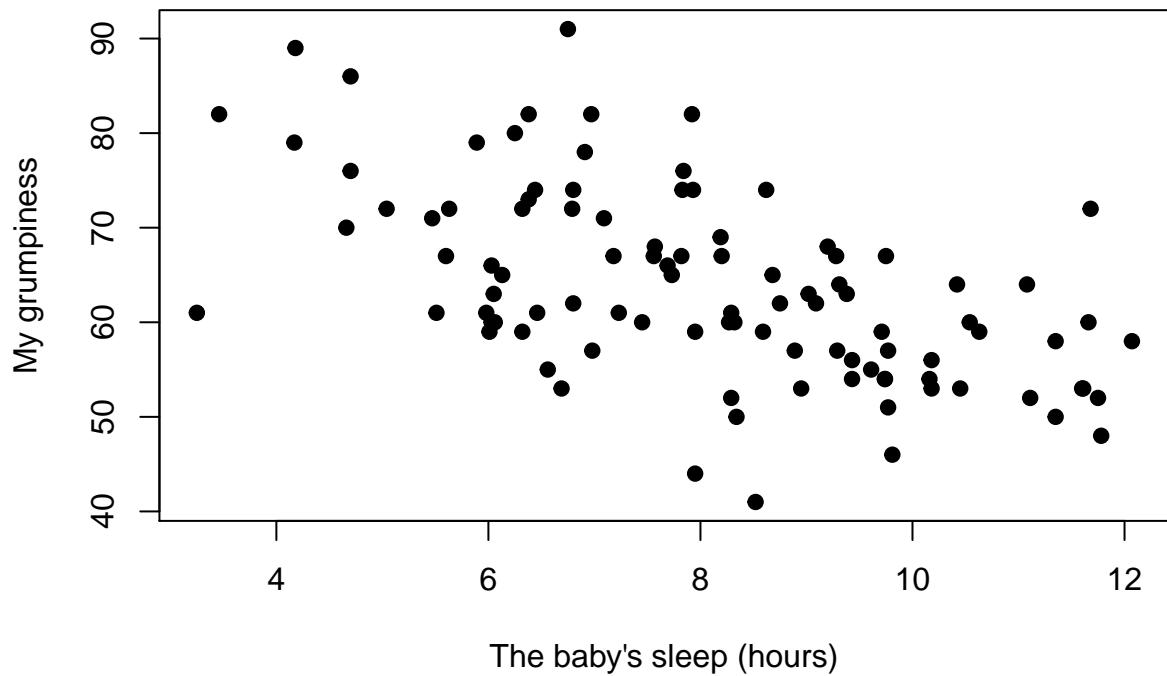


Figure 5.8: Scatterplot showing the relationship between `baby.sleep` and `dan.grump`

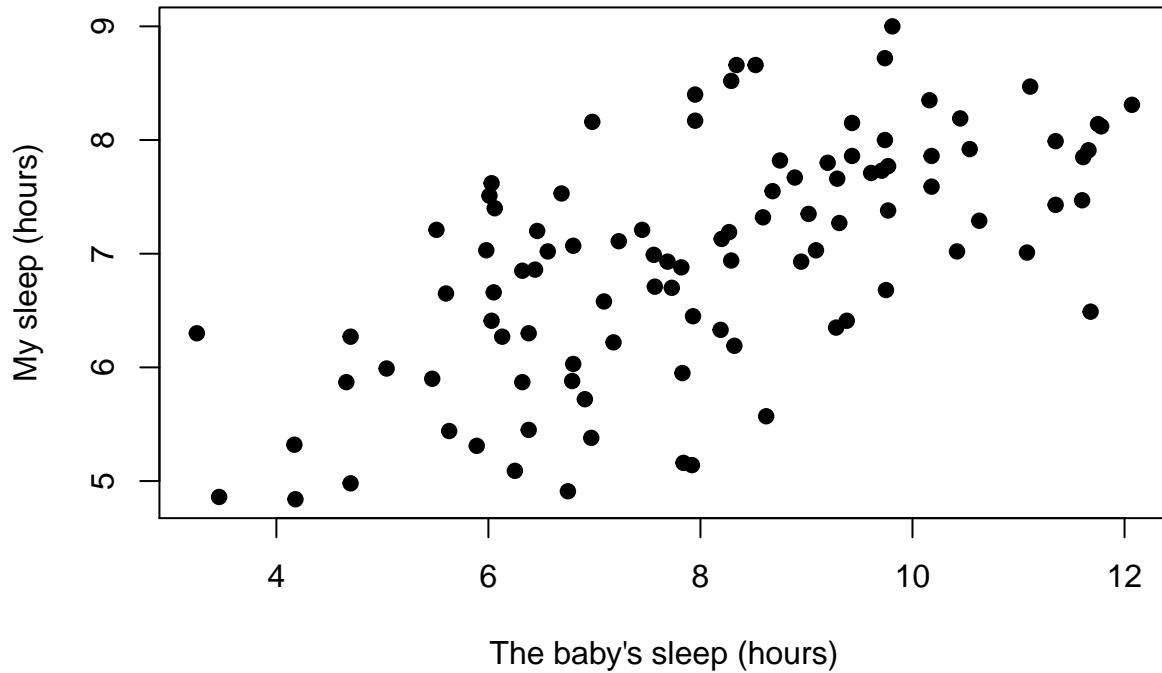


Figure 5.9: Scatterplot showing the relationship between `baby.sleep` and `dan.sleep`

`dan.sleep` and `dan.grump` (Figure 5.7 with that between `baby.sleep` and `dan.grump` (Figure 5.8. When looking at these two plots side by side, it's clear that the relationship is *qualitatively* the same in both cases: more sleep equals less grump! However, it's also pretty obvious that the relationship between `dan.sleep` and `dan.grump` is *stronger* than the relationship between `baby.sleep` and `dan.grump`. The plot on the left is “neater” than the one on the right. What it feels like is that if you want to predict what my mood is, it'd help you a little bit to know how many hours my son slept, but it'd be *more* helpful to know how many hours I slept.

In contrast, let's consider Figure 5.8 vs. Figure 5.9. If we compare the scatterplot of “`baby.sleep v dan.grump`” to the scatterplot of “`baby.sleep v dan.sleep`”, the overall strength of the relationship is the same, but the direction is different. That is, if my son sleeps more, I get *more* sleep (positive relationship), but if he sleeps more then I get *less* grumpy (negative relationship).

5.7.3 The correlation coefficient

We can make these ideas a bit more explicit by introducing the idea of a **correlation coefficient** (or, more specifically, Pearson's correlation coefficient), which is traditionally denoted by r . The correlation coefficient between two variables X and Y (sometimes denoted r_{XY}), which we'll define more precisely in the next section, is a measure that varies from -1 to 1 . When $r = -1$ it means that we have a perfect negative relationship, and when $r = 1$ it means we have a perfect positive relationship. When $r = 0$, there's no relationship at all. If you look at Figure 5.10, you can see several plots showing what different correlations look like.

The formula for the Pearson's correlation coefficient can be written in several different ways. I think the

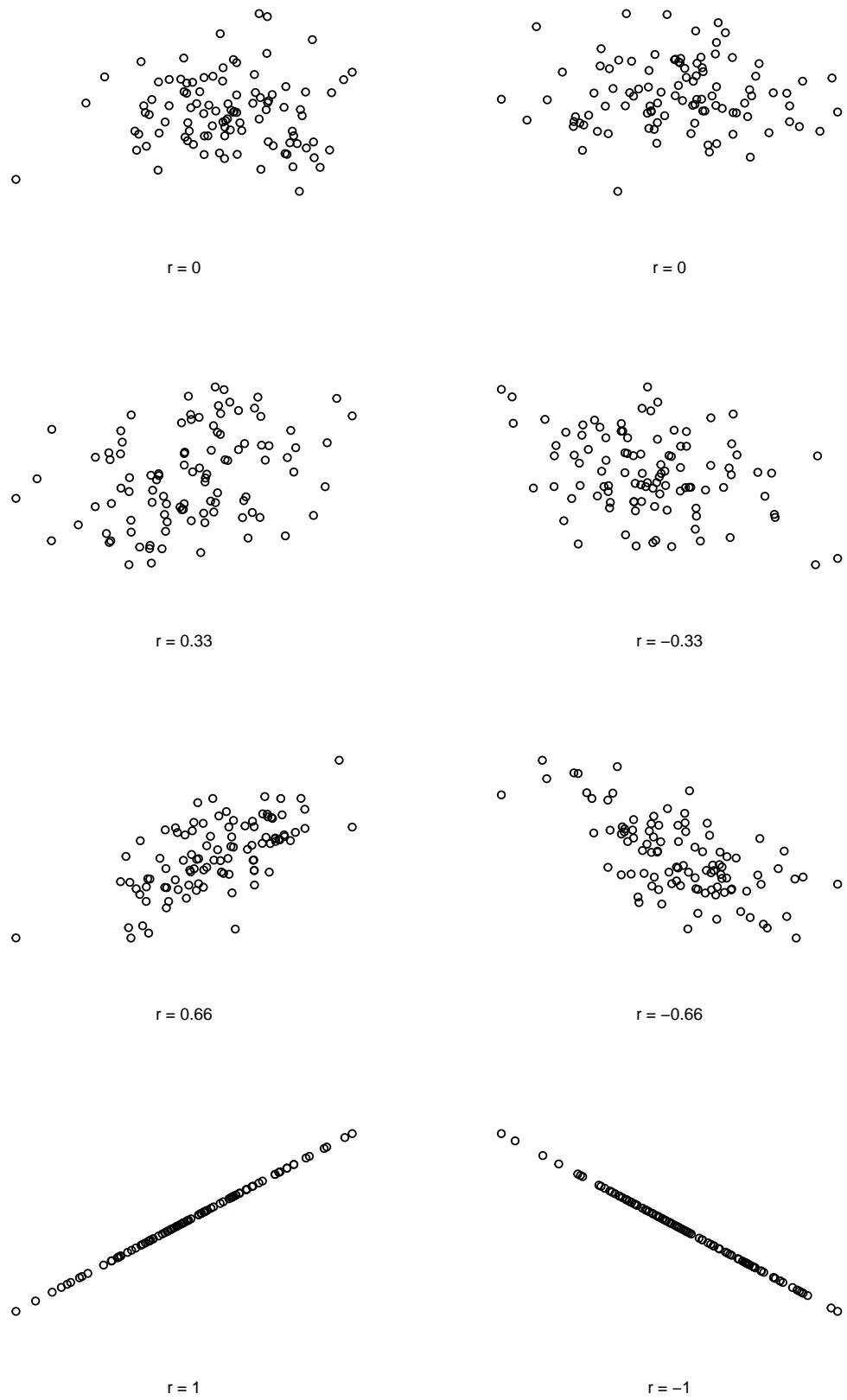


Figure 5.10: Illustration of the effect of varying the strength and direction of a correlation

simplest way to write down the formula is to break it into two steps. Firstly, let's introduce the idea of a **covariance**. The covariance between two variables X and Y is a generalisation of the notion of the variance; it's a mathematically simple way of describing the relationship between two variables that isn't terribly informative to humans:

$$\text{Cov}(X, Y) = \frac{1}{N-1} \sum_{i=1}^N (X_i - \bar{X})(Y_i - \bar{Y})$$

Because we're multiplying (i.e., taking the “product” of) a quantity that depends on X by a quantity that depends on Y and then averaging¹⁷, you can think of the formula for the covariance as an “average cross product” between X and Y . The covariance has the nice property that, if X and Y are entirely unrelated, then the covariance is exactly zero. If the relationship between them is positive (in the sense shown in Figure@reffig:corr) then the covariance is also positive; and if the relationship is negative then the covariance is also negative. In other words, the covariance captures the basic qualitative idea of correlation. Unfortunately, the raw magnitude of the covariance isn't easy to interpret: it depends on the units in which X and Y are expressed, and worse yet, the actual units that the covariance itself is expressed in are really weird. For instance, if X refers to the `dan.sleep` variable (units: hours) and Y refers to the `dan.grump` variable (units: grumps), then the units for their covariance are “hours \times grumps”. And I have no freaking idea what that would even mean.

The Pearson correlation coefficient r fixes this interpretation problem by standardising the covariance, in pretty much the exact same way that the z -score standardises a raw score: by dividing by the standard deviation. However, because we have two variables that contribute to the covariance, the standardisation only works if we divide by both standard deviations.¹⁸ In other words, the correlation between X and Y can be written as follows:

$$r_{XY} = \frac{\text{Cov}(X, Y)}{\hat{\sigma}_X \hat{\sigma}_Y}$$

By doing this standardisation, not only do we keep all of the nice properties of the covariance discussed earlier, but the actual values of r are on a meaningful scale: $r = 1$ implies a perfect positive relationship, and $r = -1$ implies a perfect negative relationship. I'll expand a little more on this point later, in Section@refsec:interpretingcorrelations. But before I do, let's look at how to calculate correlations in R.

5.7.4 Calculating correlations in R

Calculating correlations in R can be done using the `cor()` command. The simplest way to use the command is to specify two input arguments `x` and `y`, each one corresponding to one of the variables. The following extract illustrates the basic usage of the function:¹⁹

```
cor( x = parenthood$dan.sleep, y = parenthood$dan.grump )
## [1] -0.903384
```

However, the `cor()` function is a bit more powerful than this simple example suggests. For example, you can also calculate a complete “correlation matrix”, between all pairs of variables in the data frame:²⁰

¹⁷Just like we saw with the variance and the standard deviation, in practice we divide by $N - 1$ rather than N .

¹⁸This is an oversimplification, but it'll do for our purposes.

¹⁹If you are reading this after having already completed Chapter 11 you might be wondering about hypothesis tests for correlations. R has a function called `cor.test()` that runs a hypothesis test for a single correlation, and the `psych` package contains a version called `corr.test()` that can run tests for every correlation in a correlation matrix; hypothesis tests for correlations are discussed in more detail in Section ??.

²⁰An alternative usage of `cor()` is to correlate one set of variables with another subset of variables. If `X` and `Y` are both data frames with the same number of rows, then `cor(x = X, y = Y)` will produce a correlation matrix that correlates all variables in `X` with all variables in `Y`.

```
# correlate all pairs of variables in "parenthood":
cor( x = parenthood )
```

```
##          dan.sleep  baby.sleep  dan.grump      day
## dan.sleep  1.00000000  0.62794934 -0.90338404 -0.09840768
## baby.sleep  0.62794934  1.00000000 -0.56596373 -0.01043394
## dan.grump -0.90338404 -0.56596373  1.00000000  0.07647926
## day        -0.09840768 -0.01043394  0.07647926  1.00000000
```

5.7.5 Interpreting a correlation

Naturally, in real life you don't see many correlations of 1. So how should you interpret a correlation of, say $r = .4$? The honest answer is that it really depends on what you want to use the data for, and on how strong the correlations in your field tend to be. A friend of mine in engineering once argued that any correlation less than .95 is completely useless (I think he was exaggerating, even for engineering). On the other hand there are real cases – even in psychology – where you should really expect correlations that strong. For instance, one of the benchmark data sets used to test theories of how people judge similarities is so clean that any theory that can't achieve a correlation of at least .9 really isn't deemed to be successful. However, when looking for (say) elementary correlates of intelligence (e.g., inspection time, response time), if you get a correlation above .3 you're doing very very well. In short, the interpretation of a correlation depends a lot on the context. That said, the rough guide in Table ?? is pretty typical.

```
knitr::kable(
rbind(
c("-1.0 to -0.9", "Very strong", "Negative"),
c("-0.9 to -0.7", "Strong", "Negative"),
c("-0.7 to -0.4", "Moderate", "Negative"),
c("-0.4 to -0.2", "Weak", "Negative"),
c("-0.2 to 0", "Negligible", "Negative"),
c("0 to 0.2", "Negligible", "Positive"),
c("0.2 to 0.4", "Weak", "Positive"),
c("0.4 to 0.7", "Moderate", "Positive"),
c("0.7 to 0.9", "Strong", "Positive"),
c("0.9 to 1.0", "Very strong", "Positive")), col.names=c("Correlation", "Strength", "Direction"),
booktabs = TRUE)
```

Correlation	Strength	Direction
-1.0 to -0.9	Very strong	Negative
-0.9 to -0.7	Strong	Negative
-0.7 to -0.4	Moderate	Negative
-0.4 to -0.2	Weak	Negative
-0.2 to 0	Negligible	Negative
0 to 0.2	Negligible	Positive
0.2 to 0.4	Weak	Positive
0.4 to 0.7	Moderate	Positive
0.7 to 0.9	Strong	Positive
0.9 to 1.0	Very strong	Positive

However, something that can never be stressed enough is that you should *always* look at the scatterplot before attaching any interpretation to the data. A correlation might not mean what you think it means. The classic illustration of this is “Anscombe’s Quartet” ?, Anscombe1973, which is a collection of four data

sets. Each data set has two variables, an X and a Y . For all four data sets the mean value for X is 9 and the mean for Y is 7.5. The standard deviations for all X variables are almost identical, as are those for the Y variables. And in each case the correlation between X and Y is $r = 0.816$. You can verify this yourself, since the dataset comes distributed with R. The commands would be:

```
cor( anscombe$x1, anscombe$y1 )
```

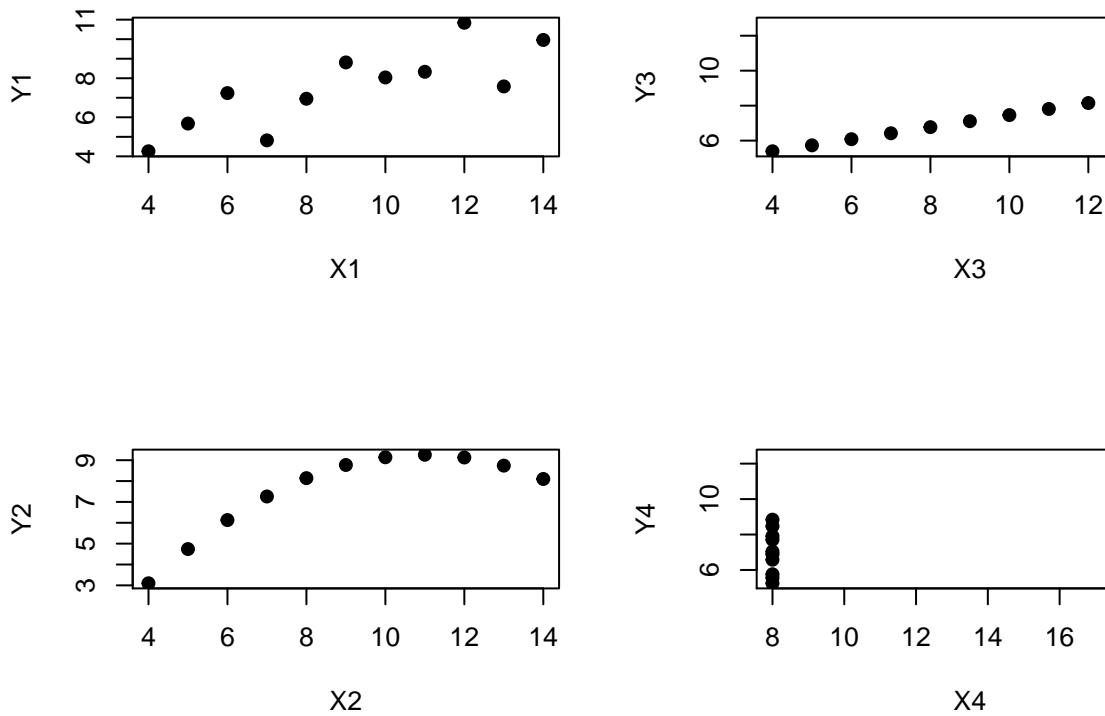
```
## [1] 0.8164205
```

```
cor( anscombe$x2, anscombe$y2 )
```

```
## [1] 0.8162365
```

and so on.

You'd think that these four data sets would look pretty similar to one another. They do not. If we draw scatterplots of X against Y for all four variables, as shown in Figure ?? we see that all four of these are *spectacularly* different to each other.



larly different to each other.

The lesson here, which so very many people seem to forget in real life is “*always graph your raw data*”. This will be the focus of Chapter 6.

5.7.6 Spearman's rank correlations

The Pearson correlation coefficient is useful for a lot of things, but it does have shortcomings. One issue in particular stands out: what it actually measures is the strength of the *linear* relationship between two

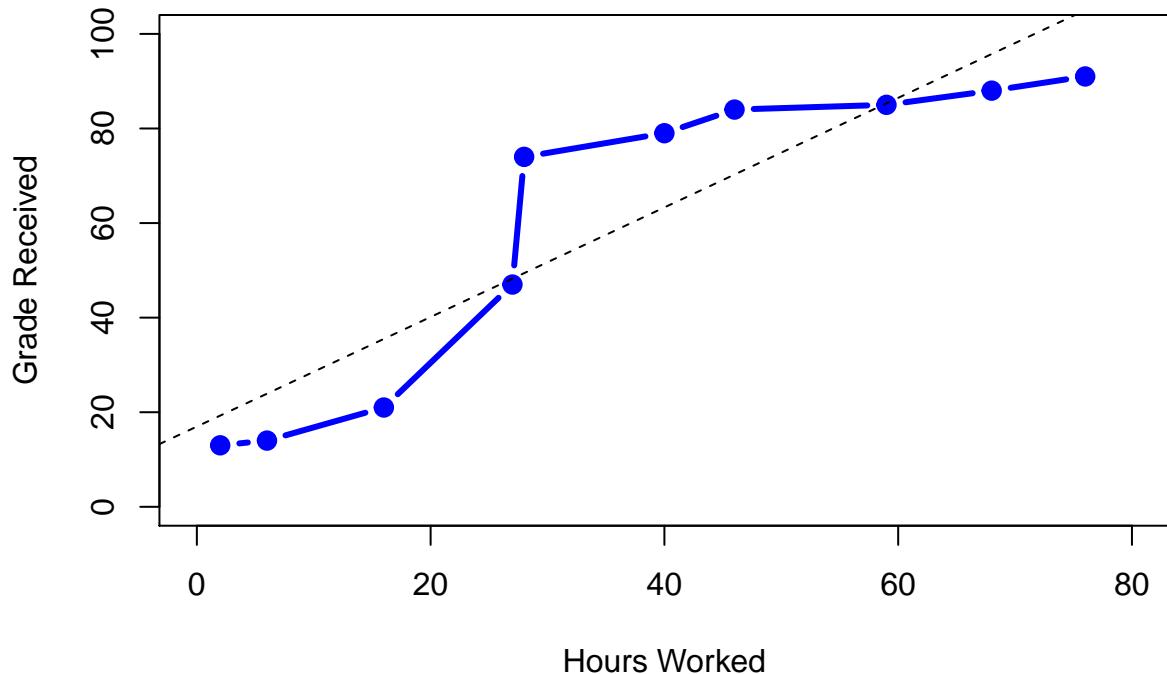


Figure 5.11: The relationship between hours worked and grade received, for a toy data set consisting of only 10 students (each circle corresponds to one student). The dashed line through the middle shows the linear relationship between the two variables. This produces a strong Pearson correlation of $r = .91$. However, the interesting thing to note here is that there's actually a perfect monotonic relationship between the two variables: in this toy example at least, increasing the hours worked always increases the grade received, as illustrated by the solid line. This is reflected in a Spearman correlation of $\rho = 1$. With such a small data set, however, it's an open question as to which version better describes the actual relationship involved.

variables. In other words, what it gives you is a measure of the extent to which the data all tend to fall on a single, perfectly straight line. Often, this is a pretty good approximation to what we mean when we say “relationship”, and so the Pearson correlation is a good thing to calculate. Sometimes, it isn’t.

One very common situation where the Pearson correlation isn’t quite the right thing to use arises when an increase in one variable X really is reflected in an increase in another variable Y , but the nature of the relationship isn’t necessarily linear. An example of this might be the relationship between effort and reward when studying for an exam. If you put in zero effort (X) into learning a subject, then you should expect a grade of 0% (Y). However, a little bit of effort will cause a *massive* improvement: just turning up to lectures means that you learn a fair bit, and if you just turn up to classes, and scribble a few things down so your grade might rise to 35%, all without a lot of effort. However, you just don’t get the same effect at the other end of the scale. As everyone knows, it takes *a lot* more effort to get a grade of 90% than it takes to get a grade of 55%. What this means is that, if I’ve got data looking at study effort and grades, there’s a pretty good chance that Pearson correlations will be misleading.

To illustrate, consider the data plotted in Figure 5.11, showing the relationship between hours worked and grade received for 10 students taking some class. The curious thing about this – highly fictitious – data set is that increasing your effort *always* increases your grade. It might be by a lot or it might be by a little, but increasing effort will never decrease your grade. The data are stored in `effort.Rdata`:

```
> load("effort.Rdata")
> who(TRUE)
-- Name --  -- Class --  -- Size --
effort      data.frame    10 x 2
$hours       numeric       10
$grade       numeric       10
```

The raw data look like this:

```
> effort
  hours grade
1     2    13
2    76    91
3    40    79
4     6    14
5    16    21
6    28    74
7    27    47
8    59    85
9    46    84
10   68    88
```

If we run a standard Pearson correlation, it shows a strong relationship between hours worked and grade received,

```
> cor(effort$hours, effort$grade)
[1] 0.909402
```

but this doesn’t actually capture the observation that increasing hours worked *always* increases the grade. There’s a sense here in which we want to be able to say that the correlation is *perfect* but for a somewhat different notion of what a “relationship” is. What we’re looking for is something that captures the fact that there is a perfect ***ordinal relationship*** here. That is, if student 1 works more hours than student 2, then we can guarantee that student 1 will get the better grade. That’s not what a correlation of $r = .91$ says at all.

How should we address this? Actually, it's really easy: if we're looking for ordinal relationships, all we have to do is treat the data as if it were ordinal scale! So, instead of measuring effort in terms of "hours worked", let's rank all 10 of our students in order of hours worked. That is, student 1 did the least work out of anyone (2 hours) so they get the lowest rank (rank = 1). Student 4 was the next laziest, putting in only 6 hours of work in over the whole semester, so they get the next lowest rank (rank = 2). Notice that I'm using "rank = 1" to mean "low rank". Sometimes in everyday language we talk about "rank = 1" to mean "top rank" rather than "bottom rank". So be careful: you can rank "from smallest value to largest value" (i.e., small equals rank 1) or you can rank "from largest value to smallest value" (i.e., large equals rank 1). In this case, I'm ranking from smallest to largest, because that's the default way that R does it. But in real life, it's really easy to forget which way you set things up, so you have to put a bit of effort into remembering!

Okay, so let's have a look at our students when we rank them from worst to best in terms of effort and reward:

	rank (hours worked)	rank (grade received)
student 1	1	1
student 2		10
student 3		6
student 4		2
student 5		3
student 6		5
student 7		4
student 8		8
student 9		7
student 10		9

Hm. These are *identical*. The student who put in the most effort got the best grade, the student with the least effort got the worst grade, etc. We can get R to construct these rankings using the `rank()` function, like this:

```
> hours.rank <- rank( effort$hours )    # rank students by hours worked
> grade.rank <- rank( effort$grade )    # rank students by grade received
```

As the table above shows, these two rankings are identical, so if we now correlate them we get a perfect relationship:

```
> cor( hours.rank, grade.rank )
[1] 1
```

What we've just re-invented is *Spearman's rank order correlation*, usually denoted ρ to distinguish it from the Pearson correlation r . We can calculate Spearman's ρ using R in two different ways. Firstly we could do it the way I just showed, using the `rank()` function to construct the rankings, and then calculate the Pearson correlation on these ranks. However, that's way too much effort to do every time. It's much easier to just specify the `method` argument of the `cor()` function.

```
> cor( effort$hours, effort$grade, method = "spearman")
[1] 1
```

The default value of the `method` argument is "pearson", which is why we didn't have to specify it earlier on when we were doing Pearson correlations.

5.7.7 The `correlate()` function

As we've seen, the `cor()` function works pretty well, and handles many of the situations that you might be interested in. One thing that many beginners find frustrating, however, is the fact that it's not built to handle non-numeric variables. From a statistical perspective, this is perfectly sensible: Pearson and Spearman correlations are only designed to work for numeric variables, so the `cor()` function spits out an error.

Here's what I mean. Suppose you were keeping track of how many `hours` you worked in any given day, and counted how many `tasks` you completed. If you were doing the tasks for money, you might also want to keep track of how much `pay` you got for each job. It would also be sensible to keep track of the `weekday` on which you actually did the work: most of us don't work as much on Saturdays or Sundays. If you did this for 7 weeks, you might end up with a data set that looks like this one:

```
> load("work.Rdata")

> who(TRUE)
-- Name -- -- Class -- -- Size --
work      data.frame   49 x 7
$hours    numeric      49
$tasks    numeric      49
$pay      numeric      49
$day     integer      49
$weekday factor      49
$week    numeric      49
$day.type factor      49

> head(work)
  hours tasks pay day weekday week day.type
1  7.2    14  41   1 Tuesday    1 weekday
2  7.4    11  39   2 Wednesday   1 weekday
3  6.6    14  13   3 Thursday   1 weekday
4  6.5    22  47   4 Friday     1 weekday
5  3.1     5   4   5 Saturday   1 weekend
6  3.0     7  12   6 Sunday     1 weekend
```

Obviously, I'd like to know something about how all these variables correlate with one another. I could correlate `hours` with `pay` quite using `cor()`, like so:

```
> cor(work$hours, work$pay)
[1] 0.7604283
```

But what if I wanted a quick and easy way to calculate all pairwise correlations between the numeric variables? I can't just input the `work` data frame, because it contains two factor variables, `weekday` and `day.type`. If I try this, I get an error:

```
> cor(work)
Error in cor(work) : 'x' must be numeric
```

In order to get the correlations that I want using the `cor()` function, is create a new data frame that doesn't contain the factor variables, and then feed that new data frame into the `cor()` function. It's not actually very hard to do that, and I'll talk about how to do it properly in Section@refsec:subsetdataframe. But it would be nice to have some function that is smart enough to just ignore the factor variables. That's where the `correlate()` function in the `lsr` package can be handy. If you feed it a data frame that contains factors, it knows to ignore them, and returns the pairwise correlations only between the numeric variables:

```
> correlate(work)

CORRELATIONS
=====
- correlation type: pearson
- correlations shown only when both variables are numeric

      hours  tasks   pay    day weekday   week day.type
hours       .  0.800  0.760 -0.049      .  0.018      .
tasks     0.800       .  0.720 -0.072      . -0.013      .
pay        0.760  0.720       .  0.137      .  0.196      .
day      -0.049 -0.072  0.137       .      .  0.990      .
weekday      .       .       .       .      .      .      .
week       0.018 -0.013  0.196  0.990      .      .      .
day.type     .       .       .       .      .      .      .
```

The output here shows a . whenever one of the variables is non-numeric. It also shows a . whenever a variable is correlated with itself (it's not a meaningful thing to do). The `correlate()` function can also do Spearman correlations, by specifying the `corr.method` to use:

```
> correlate( work, corr.method="spearman" )

CORRELATIONS
=====
- correlation type: spearman
- correlations shown only when both variables are numeric

      hours  tasks   pay    day weekday   week day.type
hours       .  0.805  0.745 -0.047      .  0.010      .
tasks     0.805       .  0.730 -0.068      . -0.008      .
pay        0.745  0.730       .  0.094      .  0.154      .
day      -0.047 -0.068  0.094       .      .  0.990      .
weekday      .       .       .       .      .      .      .
week       0.010 -0.008  0.154  0.990      .      .      .
day.type     .       .       .       .      .      .      .
```

Obviously, there's no new functionality in the `correlate()` function, and any advanced R user would be perfectly capable of using the `cor()` function to get these numbers out. But if you're not yet comfortable with extracting a subset of a data frame, the `correlate()` function is for you.

5.8 Handling missing values

There's one last topic that I want to discuss briefly in this chapter, and that's the issue of *missing data*. Real data sets very frequently turn out to have missing values: perhaps someone forgot to fill in a particular survey question, for instance. Missing data can be the source of a lot of tricky issues, most of which I'm going to gloss over. However, at a minimum, you need to understand the basics of handling missing data in R.

5.8.1 The single variable case

Let's start with the simplest case, in which you're trying to calculate descriptive statistics for a single variable which has missing data. In R, this means that there will be `NA` values in your data vector. Let's create a

variable like that:

```
> partial <- c(10, 20, NA, 30)
```

Let's assume that you want to calculate the mean of this variable. By default, R assumes that you want to calculate the mean using all four elements of this vector, which is probably the safest thing for a dumb automaton to do, but it's rarely what you actually want. Why not? Well, remember that the basic interpretation of `NA` is "I don't know what this number is". This means that $1 + NA = NA$: if I add 1 to some number that I don't know (i.e., the `NA`) then the answer is *also* a number that I don't know. As a consequence, if you don't explicitly tell R to ignore the `NA` values, and the data set does have missing values, then the output will itself be a missing value. If I try to calculate the mean of the `partial` vector, without doing anything about the missing value, here's what happens:

```
> mean( x = partial )
[1] NA
```

Technically correct, but deeply unhelpful.

To fix this, all of the descriptive statistics functions that I've discussed in this chapter (with the exception of `cor()` which is a special case I'll discuss below) have an optional argument called `na.rm`, which is shorthand for "remove NA values". By default, `na.rm = FALSE`, so R does nothing about the missing data problem. Let's try setting `na.rm = TRUE` and see what happens:

When calculating sums and means when missing data are present (i.e., when there are `NA` values) there's actually an additional argument to the function that you should be aware of. This argument is called `na.rm`, and is a logical value indicating whether R should ignore (or "remove") the missing data for the purposes of doing the calculations. By default, R assumes that you want to keep the missing values, so unless you say otherwise it will set `na.rm = FALSE`. However, R assumes that $1 + NA = NA$: if I add 1 to some number that I don't know (i.e., the `NA`) then the answer is *also* a number that I don't know. As a consequence, if you don't explicitly tell R to ignore the `NA` values, and the data set does have missing values, then the output will itself be a missing value. This is illustrated in the following extract:

```
> mean( x = partial, na.rm = TRUE )
[1] 20
```

Notice that the mean is 20 (i.e., $60 / 3$) and *not* 15. When R ignores a `NA` value, it genuinely ignores it. In effect, the calculation above is identical to what you'd get if you asked for the mean of the three-element vector `c(10, 20, 30)`.

As indicated above, this isn't unique to the `mean()` function. Pretty much all of the other functions that I've talked about in this chapter have an `na.rm` argument that indicates whether it should ignore missing values. However, its behaviour is the same for all these functions, so I won't waste everyone's time by demonstrating it separately for each one.

5.8.2 Missing values in pairwise calculations

I mentioned earlier that the `cor()` function is a special case. It doesn't have an `na.rm` argument, because the story becomes a lot more complicated when more than one variable is involved. What it does have is an argument called `use` which does roughly the same thing, but you need to think little more carefully about what you want this time. To illustrate the issues, let's open up a data set that has missing values, `parenthood2.Rdata`. This file contains the same data as the original `parenthood` data, but with some values deleted. It contains a single data frame, `parenthood2`:

```
> load( "parenthood2.Rdata" )
> print( parenthood2 )
  dan.sleep baby.sleep dan.grump day
1      7.59        NA       56   1
2      7.91     11.66       60   2
3      5.14      7.92       82   3
4      7.71      9.61       55   4
5      6.68      9.75       NA   5
6      5.99      5.04       72   6
BLAH BLAH BLAH
```

If I calculate my descriptive statistics using the `describe()` function

```
> describe( parenthood2 )
    var   n   mean     sd median trimmed   mad   min   max   BLAH
dan.sleep  1  91  6.98  1.02   7.03   7.02  1.13  4.84  9.00   BLAH
baby.sleep 2  89  8.11  2.05   8.20   8.13  2.28  3.25 12.07   BLAH
dan.grump  3  92 63.15  9.85  61.00  62.66 10.38 41.00 89.00   BLAH
day        4 100 50.50 29.01  50.50  50.50 37.06  1.00 100.00   BLAH
```

we can see from the `n` column that there are 9 missing values for `dan.sleep`, 11 missing values for `baby.sleep` and 8 missing values for `dan.grump`.²¹ Suppose what I would like is a correlation matrix. And let's also suppose that I don't bother to tell R how to handle those missing values. Here's what happens:

```
> cor( parenthood2 )
            dan.sleep baby.sleep dan.grump day
dan.sleep      1          NA        NA   NA
baby.sleep     NA         1          NA   NA
dan.grump      NA         NA         1   NA
day            NA         NA         NA   1
```

Annoying, but it kind of makes sense. If I don't *know* what some of the values of `dan.sleep` and `baby.sleep` actually are, then I can't possibly *know* what the correlation between these two variables is either, since the formula for the correlation coefficient makes use of every single observation in the data set. Once again, it makes sense: it's just not particularly *helpful*.

To make R behave more sensibly in this situation, you need to specify the `use` argument to the `cor()` function. There are several different values that you can specify for this, but the two that we care most about in practice tend to be "`complete.obs`" and "`pairwise.complete.obs`". If we specify `use = "complete.obs"`, R will completely ignore all cases (i.e., all rows in our `parenthood2` data frame) that have any missing values at all. So, for instance, if you look back at the extract earlier when I used the `head()` function, notice that observation 1 (i.e., day 1) of the `parenthood2` data set is missing the value for `baby.sleep`, but is otherwise complete? Well, if you choose `use = "complete.obs"` R will ignore that row completely: that is, even when it's trying to calculate the correlation between `dan.sleep` and `dan.grump`, observation 1 will be ignored, because the value of `baby.sleep` is missing for that observation. Here's what we get:

```
> cor(parenthood2, use = "complete.obs")
            dan.sleep baby.sleep dan.grump      day
dan.sleep  1.00000000  0.6394985 -0.89951468  0.06132891
baby.sleep  0.63949845  1.0000000 -0.58656066  0.14555814
dan.grump -0.89951468 -0.5865607  1.00000000 -0.06816586
day        0.06132891  0.1455581 -0.06816586  1.00000000
```

²¹It's worth noting that, even though we have missing data for each of these variables, the output doesn't contain any `NA` values. This is because, while `describe()` also has an `na.rm` argument, the default value for this function is `na.rm = TRUE`.

The other possibility that we care about, and the one that tends to get used more often in practice, is to set `use = "pairwise.complete.obs"`. When we do that, R only looks at the variables that it's trying to correlate when determining what to drop. So, for instance, since the only missing value for observation 1 of `parenthood2` is for `baby.sleep` R will only drop observation 1 when `baby.sleep` is one of the variables involved: and so R keeps observation 1 when trying to correlate `dan.sleep` and `dan.grump`. When we do it this way, here's what we get:

```
> cor(parenthood2, use = "pairwise.complete.obs")
      dan.sleep  baby.sleep  dan.grump       day
dan.sleep    1.00000000  0.61472303 -0.903442442 -0.076796665
baby.sleep   0.61472303  1.00000000 -0.567802669  0.058309485
dan.grump   -0.90344244 -0.56780267  1.000000000  0.005833399
day         -0.07679667  0.05830949  0.005833399  1.000000000
```

Similar, but not quite the same. It's also worth noting that the `correlate()` function (in the `lsr` package) automatically uses the “pairwise complete” method:

```
> correlate(parenthood2)

CORRELATIONS
=====
- correlation type:  pearson
- correlations shown only when both variables are numeric

      dan.sleep  baby.sleep  dan.grump       day
dan.sleep     .        0.615   -0.903 -0.077
baby.sleep   0.615     .      -0.568  0.058
dan.grump   -0.903   -0.568     .   0.006
day         -0.077    0.058   0.006     .
```

The two approaches have different strengths and weaknesses. The “pairwise complete” approach has the advantage that it keeps more observations, so you're making use of more of your data and (as we'll discuss in tedious detail in Chapter 10) it improves the reliability of your estimated correlation. On the other hand, it means that every correlation in your correlation matrix is being computed from a slightly different set of observations, which can be awkward when you want to compare the different correlations that you've got.

So which method should you use? It depends a lot on *why* you think your values are missing, and probably depends a little on how paranoid you are. For instance, if you think that the missing values were “chosen” completely randomly²² then you'll probably want to use the pairwise method. If you think that missing data are a cue to thinking that the whole observation might be rubbish (e.g., someone just selecting arbitrary responses in your questionnaire), but that there's no pattern to which observations are “rubbish” then it's probably safer to keep only those observations that are complete. If you think there's something systematic going on, in that some observations are more likely to be missing than others, then you have a much trickier problem to solve, and one that is beyond the scope of this book.

5.9 Summary

Calculating some basic descriptive statistics is one of the very first things you do when analysing real data, and descriptive statistics are much simpler to understand than inferential statistics, so like every other statistics textbook I've started with descriptives. In this chapter, we talked about the following topics:

²²The technical term here is “missing completely at random” (often written MCAR for short). Makes sense, I suppose, but it does sound ungrammatical to me.

- *Measures of central tendency.* Broadly speaking, central tendency measures tell you where the data are. There's three measures that are typically reported in the literature: the mean, median and mode. (Section 5.1)
- *Measures of variability.* In contrast, measures of variability tell you about how "spread out" the data are. The key measures are: range, standard deviation, interquartile range (Section 5.2)
- *Getting summaries of variables in R.* Since this book focuses on doing data analysis in R, we spent a bit of time talking about how descriptive statistics are computed in R. (Section ?? and 5.5)
- *Standard scores.* The z-score is a slightly unusual beast. It's not quite a descriptive statistic, and not quite an inference. We talked about it in Section 5.6. Make sure you understand that section: it'll come up again later.
- *Correlations.* Want to know how strong the relationship is between two variables? Calculate a correlation. (Section 5.7)
- *Missing data.* Dealing with missing data is one of those frustrating things that data analysts really wish they didn't have to think about. In real life it can be hard to do well. For the purpose of this book, we only touched on the basics in Section 5.8

In the next section we'll move on to a discussion of how to draw pictures! Everyone loves a pretty picture, right? But before we do, I want to end on an important point. A traditional first course in statistics spends only a small proportion of the class on descriptive statistics, maybe one or two lectures at most. The vast majority of the lecturer's time is spent on inferential statistics, because that's where all the hard stuff is. That makes sense, but it hides the practical everyday importance of choosing good descriptives. With that in mind...

5.10 Epilogue: Good descriptive statistics are descriptive!

The death of one man is a tragedy. The death of millions is a statistic.

– Josef Stalin, Potsdam 1945

950,000 – 1,200,000

– Estimate of Soviet repression deaths, 1937-1938 (Ellman, 2002)

Stalin's infamous quote about the statistical character death of millions is worth giving some thought. The clear intent of his statement is that the death of an individual touches us personally and its force cannot be denied, but that the deaths of a multitude are incomprehensible, and as a consequence mere statistics, more easily ignored. I'd argue that Stalin was half right. A statistic is an abstraction, a description of events beyond our personal experience, and so hard to visualise. Few if any of us can imagine what the deaths of millions is "really" like, but we can imagine one death, and this gives the lone death its feeling of immediate tragedy, a feeling that is missing from Ellman's cold statistical description.

Yet it is not so simple: without numbers, without counts, without a description of what happened, we have *no chance* of understanding what really happened, no opportunity even to try to summon the missing feeling. And in truth, as I write this, sitting in comfort on a Saturday morning, half a world and a whole lifetime away from the Gulags, when I put the Ellman estimate next to the Stalin quote a dull dread settles in my stomach and a chill settles over me. The Stalinist repression is something truly beyond my experience, but with a combination of statistical data and those recorded personal histories that have come down to us, it is not entirely beyond my comprehension. Because what Ellman's numbers tell us is this: over a two year period, Stalinist repression wiped out the equivalent of every man, woman and child currently alive in the city where I live. Each one of those deaths had its own story, was its own tragedy, and only some of those are known to us now. Even so, with a few carefully chosen statistics, the scale of the atrocity starts to come into focus.

Thus it is no small thing to say that the first task of the statistician and the scientist is to summarise the data, to find some collection of numbers that can convey to an audience a sense of what has happened. This

is the job of descriptive statistics, but it's not a job that can be told solely using the numbers. You are a data analyst, not a statistical software package. Part of your job is to take these *statistics* and turn them into a *description*. When you analyse data, it is not sufficient to list off a collection of numbers. Always remember that what you're really trying to do is communicate with a human audience. The numbers are important, but they need to be put together into a meaningful story that your audience can interpret. That means you need to think about framing. You need to think about context. And you need to think about the individual events that your statistics are summarising.

Chapter 6

Drawing graphs

Above all else show the data.

—Edward Tufte¹

Visualising data is one of the most important tasks facing the data analyst. It's important for two distinct but closely related reasons. Firstly, there's the matter of drawing "presentation graphics": displaying your data in a clean, visually appealing fashion makes it easier for your reader to understand what you're trying to tell them. Equally important, perhaps even more important, is the fact that drawing graphs helps *you* to understand the data. To that end, it's important to draw "exploratory graphics" that help you learn about the data as you go about analysing it. These points might seem pretty obvious, but I cannot count the number of times I've seen people forget them.

To give a sense of the importance of this chapter, I want to start with a classic illustration of just how powerful a good graph can be. To that end, Figure 6.1 shows a redrawing of one of the most famous data visualisations of all time: John Snow's 1854 map of cholera deaths. The map is elegant in its simplicity. In the background we have a street map, which helps orient the viewer. Over the top, we see a large number of small dots, each one representing the location of a cholera case. The larger symbols show the location of water pumps, labelled by name. Even the most casual inspection of the graph makes it very clear that the source of the outbreak is almost certainly the Broad Street pump. Upon viewing this graph, Dr Snow arranged to have the handle removed from the pump, ending the outbreak that had killed over 500 people. Such is the power of a good data visualisation.

The goals in this chapter are twofold: firstly, to discuss several fairly standard graphs that we use a lot when analysing and presenting data, and secondly, to show you how to create these graphs in R. The graphs themselves tend to be pretty straightforward, so in that respect this chapter is pretty simple. Where people usually struggle is learning how to produce graphs, and especially, learning how to produce good graphs.² Fortunately, learning how to draw graphs in R is reasonably simple, as long as you're not too picky about what your graph looks like. What I mean when I say this is that R has a lot of *very* good graphing functions, and most of the time you can produce a clean, high-quality graphic without having to learn very much about the low-level details of how R handles graphics. Unfortunately, on those occasions when you do want to do something non-standard, or if you need to make highly specific changes to the figure, you actually do need to learn a fair bit about the these details; and those details are both complicated and boring. With that in mind, the structure of this chapter is as follows: I'll start out by giving you a very quick overview of

¹The origin of this quote is Tufte's lovely book *The Visual Display of Quantitative Information*.

²I should add that this isn't unique to R. Like everything in R there's a pretty steep learning curve to learning how to draw graphs, and like always there's a massive payoff at the end in terms of the quality of what you can produce. But to be honest, I've seen the same problems show up regardless of what system people use. I suspect that the hardest thing to do is to force yourself to take the time to think deeply about what your graphs are doing. I say that in full knowledge that only about half of my graphs turn out as well as they ought to. Understanding what makes a good graph is easy: actually designing a good graph is *hard*.

Snow's Cholera Map of London

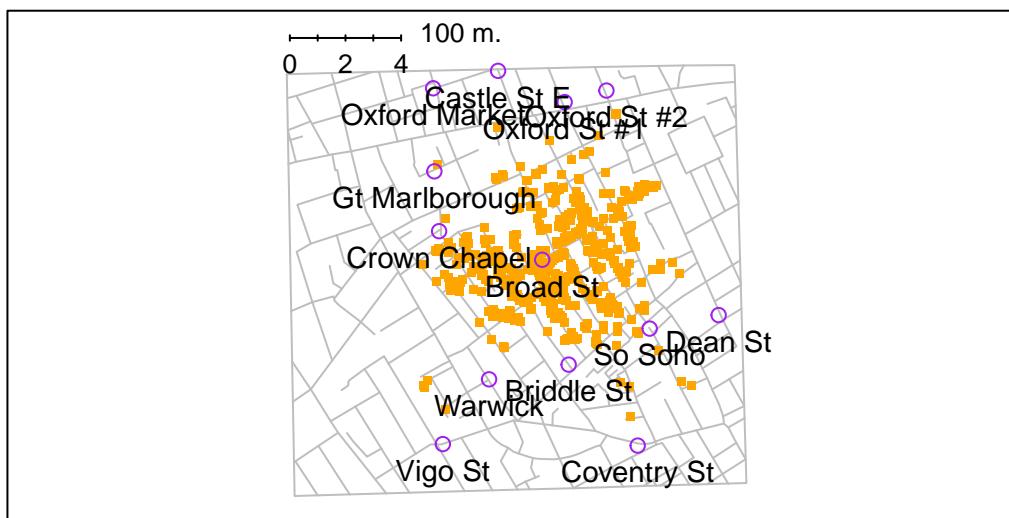


Figure 6.1: A stylised redrawing of John Snow's original cholera map. Each small dot represents the location of a cholera case, and each large circle shows the location of a well. As the plot makes clear, the cholera outbreak is centred very closely on the Broad St pump. This image uses the data from the `HistData` package @ [Friendly2011], and was drawn using minor alterations to the commands provided in the help files. Note that Snow's original hand drawn map used different symbols and labels, but you get the idea.

how graphics work in R. I'll then discuss several different kinds of graph and how to draw them, as well as showing the basics of how to customise these plots. I'll then talk in more detail about R graphics, discussing some of those complicated and boring issues. In a future version of this book, I intend to finish this chapter off by talking about what makes a good or a bad graph, but I haven't yet had the time to write that section.

6.1 An overview of R graphics

Reduced to its simplest form, you can think of an R graphic as being much like a painting. You start out with an empty canvas. Every time you use a graphics function, it paints some new things onto your canvas. Later on, you can paint more things over the top if you want; but just like painting, you can't "undo" your strokes. If you make a mistake, you have to throw away your painting and start over. Fortunately, this is way more easy to do when using R than it is when painting a picture in real life: you delete the plot and then type a new set of commands.³ This way of thinking about drawing graphs is referred to as the *painter's model*. So far, this probably doesn't sound particularly complicated, and for the vast majority of graphs you'll want to draw it's exactly as simple as it sounds. Much like painting in real life, the headaches usually start when we dig into details. To see why, I'll expand this "painting metaphor" a bit further just to show you the basics of what's going on under the hood, but before I do I want to stress that you really don't need to understand all these complexities in order to draw graphs. I'd been using R for years before I even realised that most of these issues existed! However, I don't want you to go through the same pain I went through every time I inadvertently discovered one of these things, so here's a quick overview.

Firstly, if you want to paint a picture, you need to paint it **on** something. In real life, you can paint on lots of different things. Painting onto canvas isn't the same as painting onto paper, and neither one is the same as painting on a wall. In R, the thing that you paint your graphic onto is called a **device**. For most applications that we'll look at in this book, this "device" will be a window on your computer. If you're using Windows as your operating system, then the name for this device is `windows`; on a Mac it's called `quartz` because that's the name of the software that the Mac OS uses to draw pretty pictures; and on Linux/Unix, you're probably using `X11`. On the other hand, if you're using Rstudio (regardless of which operating system you're on), there's a separate device called `RStudioGD` that forces R to paint inside the "plots" panel in Rstudio. However, from the computers perspective there's nothing terribly special about drawing pictures on screen: and so R is quite happy to paint pictures directly into a file. R can paint several different types of image files: `jpeg`, `png`, `pdf`, `postscript`, `tiff` and `bmp` files are all among the options that you have available to you. For the most part, these different devices all behave the same way, so you don't really need to know much about the differences between them when learning how to draw pictures. But, just like real life painting, sometimes the specifics do matter. Unless stated otherwise, you can assume that I'm drawing a picture on screen, using the appropriate device (i.e., `windows`, `quartz`, `X11` or `RStudioGD`). One the rare occasions where these behave differently from one another, I'll try to point it out in the text.

Secondly, when you paint a picture you need to paint it **with** something. Maybe you want to do an oil painting, but maybe you want to use watercolour. And, generally speaking, you pretty much have to pick one or the other. The analog to this in R is a "graphics system". A graphics system defines a collection of very **low-level graphics** commands about what to draw and where to draw it. Something that surprises most new R users is the discovery that R actually has *two* completely independent graphics systems, known as **traditional graphics** (in the `graphics` package) and **grid graphics** (in the `grid` package).⁴ Not surprisingly, the traditional graphics system is the older of the two: in fact, it's actually older than R since it has its origins in S, the system from which R is descended. Grid graphics are newer, and in some respects more powerful, so many of the more recent, fancier graphical tools in R make use of grid graphics. However, grid graphics are somewhat more complicated beasts, so most people start out by learning the traditional

³Or, since you can always use the up and down keys to scroll through your recent command history, you can just pull up your most recent commands and edit them to fix your mistake. It becomes even easier once you start using scripts (Section 8.1, since all you have to do is edit your script and then run it again).

⁴Of course, even that is a slightly misleading description, since some R graphics tools make use of external graphical rendering systems like OpenGL (e.g., the `rgl` package). I absolutely will not be talking about OpenGL or the like in this book, but as it happens there is one graph in this book that relies on them: Figure ??.

graphics system. Nevertheless, as long as you don't want to use any low-level commands yourself, then you don't really need to care about whether you're using traditional graphics or grid graphics. However, the moment you do want to tweak your figure by using some low-level commands you do need to care. Because these two different systems are pretty much incompatible with each other, there's a pretty big divide in R graphics universe. Unless stated otherwise, you can assume that everything I'm saying pertains to traditional graphics.

Thirdly, a painting is usually done in a particular **style**. Maybe it's a still life, maybe it's an impressionist piece, or maybe you're trying to annoy me by pretending that cubism is a legitimate artistic style. Regardless, each artistic style imposes some overarching aesthetic and perhaps even constraints on what can (or should) be painted using that style. In the same vein, R has quite a number of different packages, each of which provide a collection of **high-level graphics** commands. A single high-level command is capable of drawing an entire graph, complete with a range of customisation options. Most but not all of the high-level commands that I'll talk about in this book come from the **graphics** package itself, and so belong to the world of traditional graphics. These commands all tend to share a common visual style, although there are a few graphics that I'll use that come from other packages that differ in style somewhat. On the other side of the great divide, the grid universe relies heavily on two different packages – **lattice** and **ggplots2** – each of which provides a quite different visual style. As you've probably guessed, there's a whole separate bunch of functions that you'd need to learn if you want to use **lattice** graphics or make use of the **ggplots2**. However, for the purposes of this book I'll restrict myself to talking about the basic **graphics** tools.

At this point, I think we've covered more than enough background material. The point that I'm trying to make by providing this discussion isn't to scare you with all these horrible details, but rather to try to convey to you the fact that R doesn't really provide a single coherent graphics system. Instead, R itself provides a platform, and different people have built different graphical tools using that platform. As a consequence of this fact, there's two different universes of graphics, and a great multitude of packages that live in them. At this stage you don't need to understand these complexities, but it's useful to know that they're there. But for now, I think we can be happy with a simpler view of things: we'll draw pictures on screen using the traditional graphics system, and as much as possible we'll stick to high level commands only.

So let's start painting.

6.2 An introduction to plotting

Before I discuss any specialised graphics, let's start by drawing a few very simple graphs just to get a feel for what it's like to draw pictures using R. To that end, let's create a small vector **Fibonacci** that contains a few numbers we'd like R to draw for us. Then, we'll ask R to **plot()** those numbers:

```
> Fibonacci <- c( 1,1,2,3,5,8,13 )
> plot( Fibonacci )
```

The result is Figure 6.2.

As you can see, what R has done is plot the *values* stored in the **Fibonacci** variable on the vertical axis (y-axis) and the corresponding *index* on the horizontal axis (x-axis). In other words, since the 4th element of the vector has a value of 3, we get a dot plotted at the location (4,3). That's pretty straightforward, and the image in Figure 6.2 is probably pretty close to what you would have had in mind when I suggested that we plot the **Fibonacci** data. However, there's quite a lot of customisation options available to you, so we should probably spend a bit of time looking at some of those options. So, be warned: this ends up being a fairly long section, because there's so many possibilities open to you. Don't let it overwhelm you though... while all of the options discussed here are handy to know about, you can get by just fine only knowing a few of them. The only reason I've included all this stuff right at the beginning is that it ends up making the rest of the chapter a lot more readable!

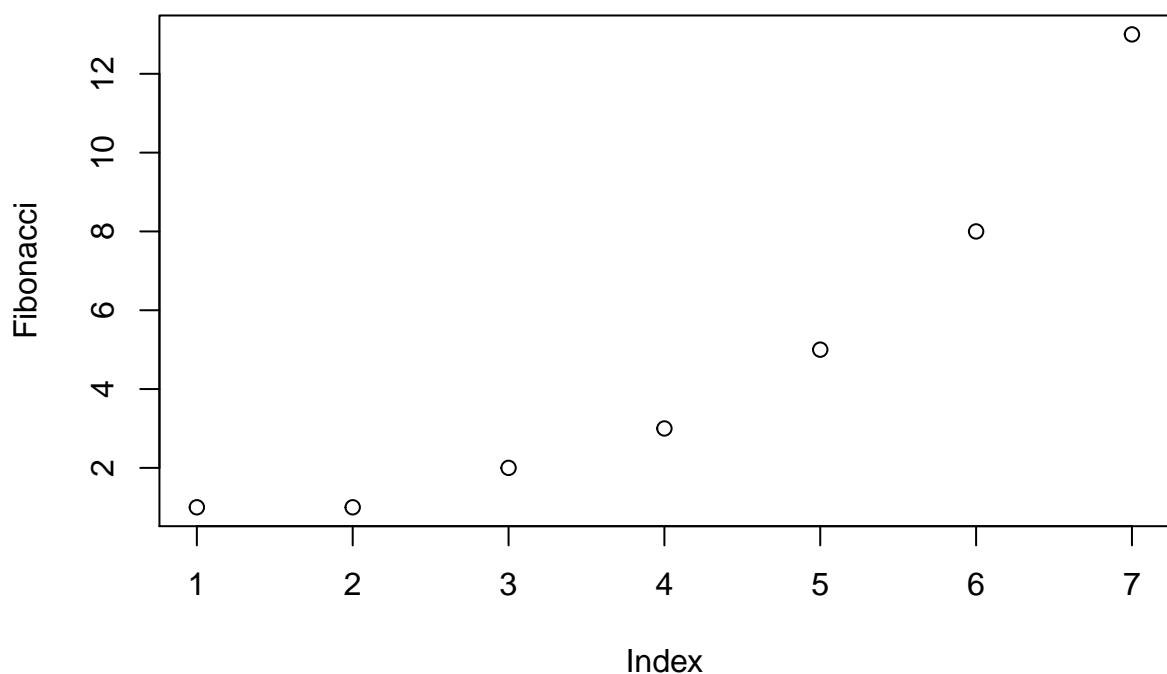


Figure 6.2: Our first plot

6.2.1 A tedious digression

Before we go into any discussion of customising plots, we need a little more background. The important thing to note when using the `plot()` function, is that it's another example of a *generic* function (Section 4.11, much like `print()` and `summary()`, and so its behaviour changes depending on what kind of input you give it. However, the `plot()` function is somewhat fancier than the other two, and its behaviour depends on *two* arguments, `x` (the first input, which is required) and `y` (which is optional). This makes it (a) extremely powerful once you get the hang of it, and (b) hilariously unpredictable, when you're not sure what you're doing. As much as possible, I'll try to make clear what type of inputs produce what kinds of outputs. For now, however, it's enough to note that I'm only doing very basic plotting, and as a consequence all of the work is being done by the `plot.default()` function.

What kinds of customisations might we be interested in? If you look at the help documentation for the default plotting method (i.e., type `?plot.default` or `help("plot.default")`) you'll see a very long list of arguments that you can specify to customise your plot. I'll talk about several of them in a moment, but first I want to point out something that might seem quite wacky. When you look at all the different options that the help file talks about, you'll notice that *some* of the options that it refers to are “proper” arguments to the `plot.default()` function, but it also goes on to mention a bunch of things that *look* like they're supposed to be arguments, but they're not listed in the “Usage” section of the file, and the documentation calls them **graphical parameters** instead. Even so, it's usually possible to treat them as if they were arguments of the plotting function. Very odd. In order to stop my readers trying to find a brick and look up my home address, I'd better explain what's going on; or at least give the basic gist behind it.

What exactly is a graphical parameter? Basically, the idea is that there are some characteristics of a plot which are pretty universal: for instance, regardless of what kind of graph you're drawing, you probably need to specify what colour to use for the plot, right? So you'd expect there to be something like a `col` argument to every single graphics function in R? Well, sort of. In order to avoid having hundreds of arguments for every single function, what R does is refer to a bunch of these “graphical parameters” which are pretty general purpose. Graphical parameters can be changed directly by using the low-level `par()` function, which I discuss briefly in Section ?? though not in a lot of detail. If you look at the help files for graphical parameters (i.e., type `?par`) you'll see that there's *lots* of them. Fortunately, (a) the default settings are generally pretty good so you can ignore the majority of the parameters, and (b) as you'll see as we go through this chapter, you very rarely need to use `par()` directly, because you can “pretend” that graphical parameters are just additional arguments to your high-level function (e.g. `plot.default()`). In short... yes, R does have these wacky “graphical parameters” which can be quite confusing. But in most basic uses of the plotting functions, you can act as if they were just undocumented additional arguments to your function.

6.2.2 Customising the title and the axis labels

One of the first things that you'll find yourself wanting to do when customising your plot is to label it better. You might want to specify more appropriate axis labels, add a title or add a subtitle. The arguments that you need to specify to make this happen are:

- `main`. A character string containing the title.
- `sub`. A character string containing the subtitle.
- `xlab`. A character string containing the x-axis label.
- `ylab`. A character string containing the y-axis label.

These aren't graphical parameters, they're arguments to the high-level function. However, because the high-level functions all rely on the same low-level function to do the drawing⁵ the names of these arguments are identical for pretty much every high-level function I've come across. Let's have a look at what happens when we make use of all these arguments. Here's the command...

⁵The low-level function that does this is called `title()` in case you ever need to know, and you can type `?title` to find out a bit more detail about what these arguments do.

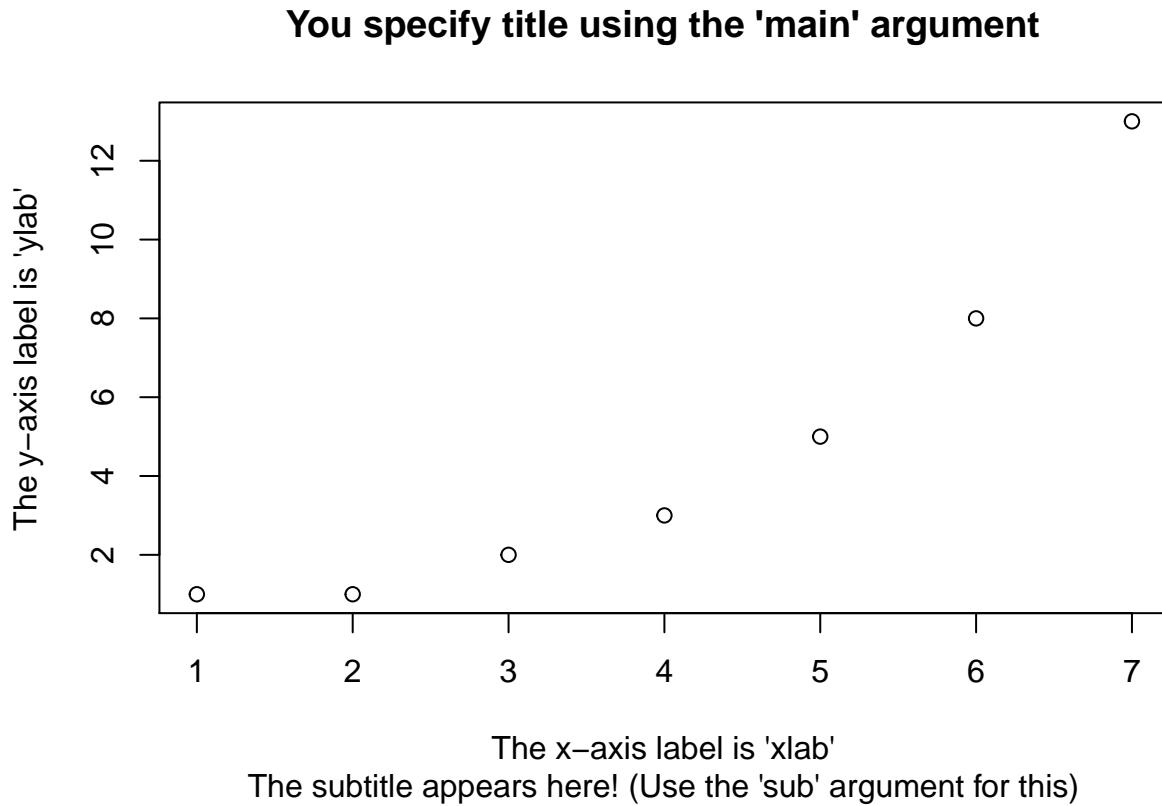


Figure 6.3: How to add your own title, subtitle, x-axis label and y-axis label to the plot.

```
> plot( x = Fibonacci,
+       main = "You specify title using the 'main' argument",
+       sub = "The subtitle appears here! (Use the 'sub' argument for this)",
+       xlab = "The x-axis label is 'xlab'",
+       ylab = "The y-axis label is 'ylab'"
+ )
```

The picture that this draws is shown in Figure 6.3.

It's more or less as you'd expect. The plot itself is identical to the one we drew in Figure 6.2, except for the fact that we've changed the axis labels, and added a title and a subtitle. Even so, there's a couple of interesting features worth calling your attention to. Firstly, notice that the subtitle is drawn below the plot, which I personally find annoying; as a consequence I almost never use subtitles. You may have a different opinion, of course, but the important thing is that you remember where the subtitle actually goes. Secondly, notice that R has decided to use boldface text and a larger font size for the title. This is one of my most hated default settings in R graphics, since I feel that it draws too much attention to the title. Generally, while I do want my reader to look at the title, I find that the R defaults are a bit overpowering, so I often like to change the settings. To that end, there are a bunch of graphical parameters that you can use to customise the font style:

- *Font styles:* `font.main`, `font.sub`, `font.lab`, `font.axis`. These four parameters control the font style used for the plot title (`font.main`), the subtitle (`font.sub`), the axis labels (`font.lab`: note that you can't specify separate styles for the x-axis and y-axis without using low level commands), and

the numbers next to the tick marks on the axis (`font.axis`). Somewhat irritatingly, these arguments are numbers instead of meaningful names: a value of 1 corresponds to plain text, 2 means boldface, 3 means italic and 4 means bold italic.

- *Font colours:* `col.main`, `col.sub`, `col.lab`, `col.axis`. These parameters do pretty much what the name says: each one specifies a **colour** in which to type each of the different bits of text. Conveniently, R has a very large number of named colours (type `colours()` to see a list of over 650 colour names that R knows), so you can use the English language name of the colour to select it.⁶ Thus, the parameter value here string like "`red`", "`gray25`" or "`springgreen4`" (yes, R really does recognise four different shades of "spring green").
- *Font size:* `cex.main`, `cex.sub`, `cex.lab`, `cex.axis`. Font size is handled in a slightly curious way in R. The "cex" part here is short for "character expansion", and it's essentially a magnification value. By default, all of these are set to a value of 1, except for the font title: `cex.main` has a default magnification of 1.2, which is why the title font is 20% bigger than the others.
- *Font family:* `family`. This argument specifies a font family to use: the simplest way to use it is to set it to "`sans`", "`serif`", or "`mono`", corresponding to a san serif font, a serif font, or a monospaced font. If you want to, you can give the name of a specific font, but keep in mind that different operating systems use different fonts, so it's probably safest to keep it simple. Better yet, unless you have some deep objections to the R defaults, just ignore this parameter entirely. That's what I usually do.

To give you a sense of how you can use these parameters to customise your titles, the following command can be used to draw Figure 6.4:

```
> plot( x = Fibonacci,
+       main = "The first 7 Fibonacci numbers",      # the title
+       xlab = "Position in the sequence",           # x-axis label
+       ylab = "The Fibonacci number",               # y-axis label
+       font.main = 1,                                # plain text for title
+       cex.main = 1,                                # normal size for title
+       font.axis = 2,                                # bold text for numbering
+       col.lab = "gray50"                            # grey colour for labels
+ )
```

Although this command is quite long, it's not complicated: all it does is override a bunch of the default parameter values. The only difficult aspect to this is that you have to remember what each of these parameters is called, and what all the different values are. And in practice I never remember: I have to look up the help documentation every time, or else look it up in this book.

6.2.3 Changing the plot type

Adding and customising the titles associated with the plot is one way in which you can play around with what your picture looks like. Another thing that you'll want to do is customise the appearance of the actual plot! To start with, let's look at the single most important options that the `plot()` function (or, recalling that we're dealing with a generic function, in this case the `plot.default()` function, since that's the one doing all the work) provides for you to use, which is the `type` argument. The `type` argument specifies the visual style of the plot. The possible values for this are:

- `type = "p"`. Draw the **points** only.
- `type = "l"`. Draw a **line** through the points.
- `type = "o"`. Draw the **line over the top of the points**.
- `type = "b"`. Draw **both** points and lines, but don't overplot.

⁶On the off chance that this isn't enough freedom for you, you can select a colour directly as a "red, green, blue" specification using the `rgb()` function, or as a "hue, saturation, value" specification using the `hsv()` function.

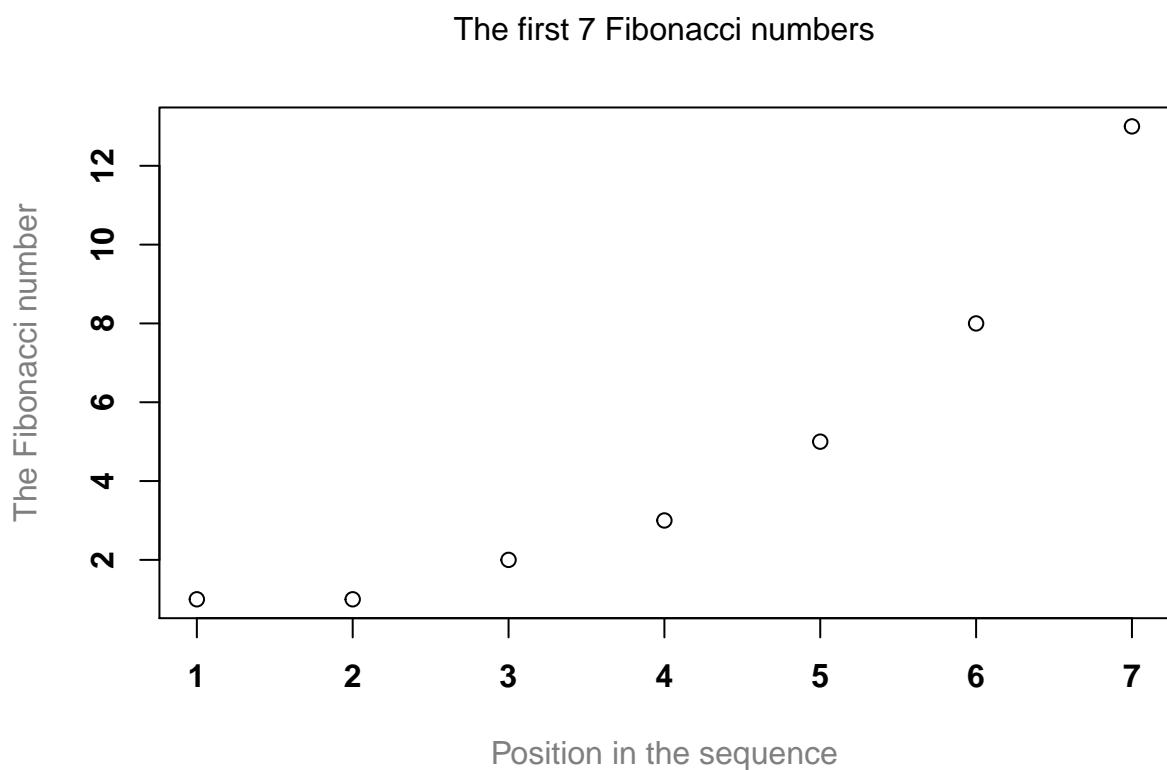


Figure 6.4: How to customise the appearance of the titles and labels.

- `type = "h"`. Draw “histogram-like” vertical bars.
- `type = "s"`. Draw a staircase, going horizontally then vertically.
- `type = "S"`. Draw a Staircase, going vertically then horizontally.
- `type = "c"`. Draw only the connecting lines from the “b” version.
- `type = "n"`. Draw nothing. (Apparently this is useful sometimes?)

The simplest way to illustrate what each of these really looks like is just to draw them. To that end, Figure 6.5 shows the same Fibonacci data, drawn using six different `types` of plot. As you can see, by altering the `type` argument you can get a qualitatively different appearance to your plot. In other words, as far as R is concerned, the only difference between a scatterplot (like the ones we drew in Section 5.7 and a line plot is that you draw a scatterplot by setting `type = "p"` and you draw a line plot by setting `type = "l"`. However, that doesn’t imply that *you* should think of them as begin equivalent to each other. As you can see by looking at Figure 6.5, a line plot implies that there is some notion of continuity from one point to the next, whereas a scatterplot does not.

6.2.4 Changing other features of the plot

In Section ?? we talked about a group of graphical parameters that are related to the formatting of titles, axis labels etc. The second group of parameters I want to discuss are those related to the formatting of the plot itself:

- *Colour of the plot*: `col`. As we saw with the previous colour-related parameters, the simplest way to specify this parameter is using a character string: e.g., `col = "blue"`. It’s a pretty straightforward parameter to specify: the only real subtlety is that every high-level function tends to draw a different “thing” as it’s output, and so this parameter gets interpreted a little differently by different functions. However, for the `plot.default()` function it’s pretty simple: the `col` argument refers to the colour of the points and/or lines that get drawn!
- *Character used to plot points*: `pch`. The `plot` character parameter is a number, usually between 1 and 25. What it does is tell R what symbol to use to draw the points that it plots. The simplest way to illustrate what the different values do is with a picture. Figure 6.6a shows the first 25 plotting characters. The default plotting character is a hollow circle (i.e., `pch = 1`).
- *Plot size*: `cex`. This parameter describes a character expansion factor (i.e., magnification) for the plotted characters. By default `cex=1`, but if you want bigger symbols in your graph you should specify a larger value.
- *Line type*: `lty`. The line type parameter describes the kind of line that R draws. It has seven values which you can specify using a number between 0 and 7, or using a meaningful character string: “`blank`”, “`solid`”, “`dashed`”, “`dotted`”, “`dotdash`”, “`longdash`”, or “`twodash`”. Note that the “`blank`” version (value 0) just means that R doesn’t draw the lines at all. The other six versions are shown in Figure 6.6b.
- *Line width*: `lwd`. The last graphical parameter in this category that I want to mention is the line width parameter, which is just a number specifying the width of the line. The default value is 1. Not surprisingly, larger values produce thicker lines and smaller values produce thinner lines. Try playing around with different values of `lwd` to see what happens.

To illustrate what you can do by altering these parameters, let’s try the following command:

```
> plot( x = Fibonacci,    # the data set
+       type = "b",        # plot both points and lines
+       col = "blue",       # change the plot colour to blue
+       pch = 19,          # plotting character is a solid circle
+       cex = 5,           # plot it at 5x the usual size
+       lty = 2,           # change line type to dashed
```

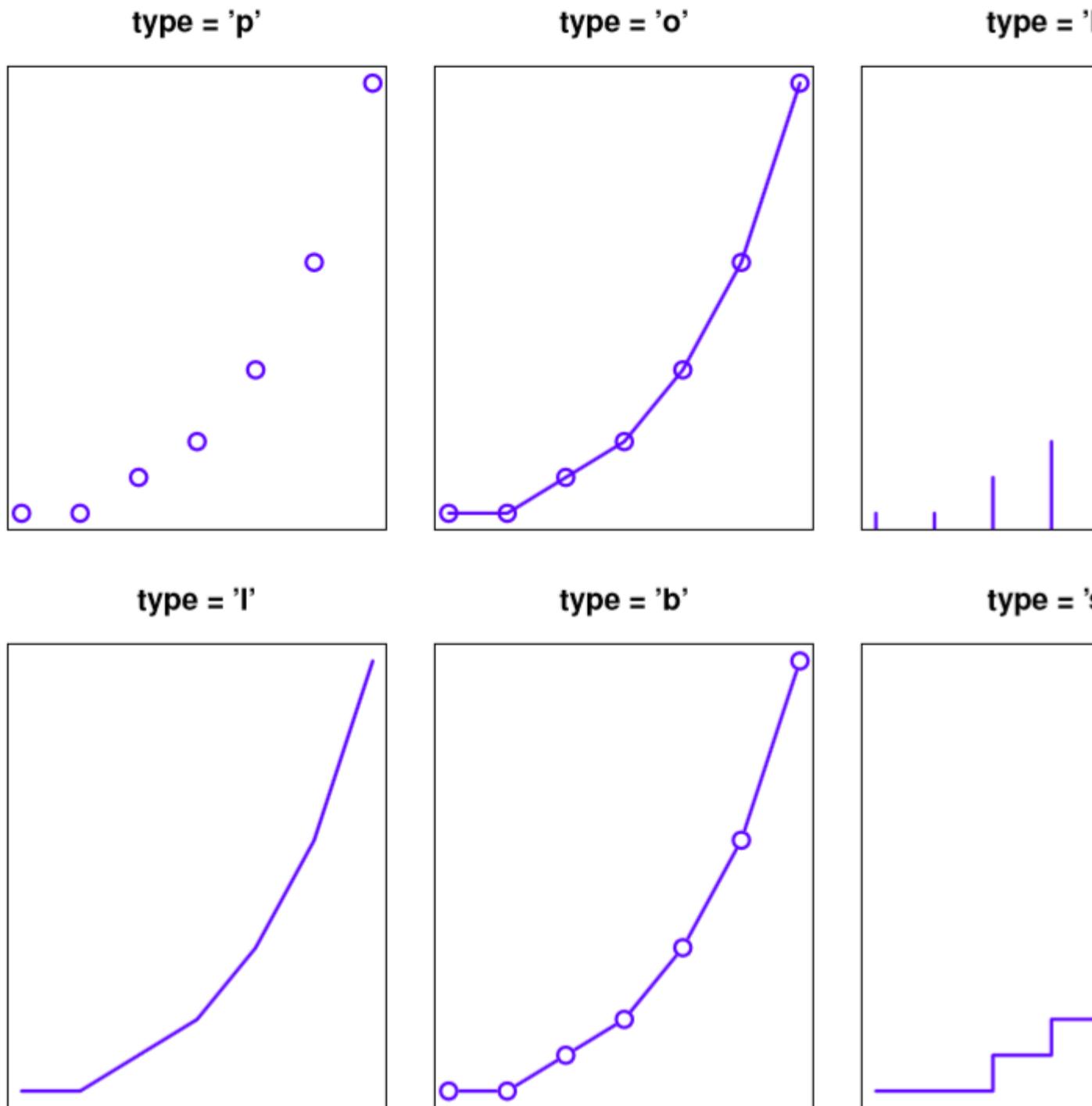


Figure 6.5: Changing the ‘type’ of the plot.

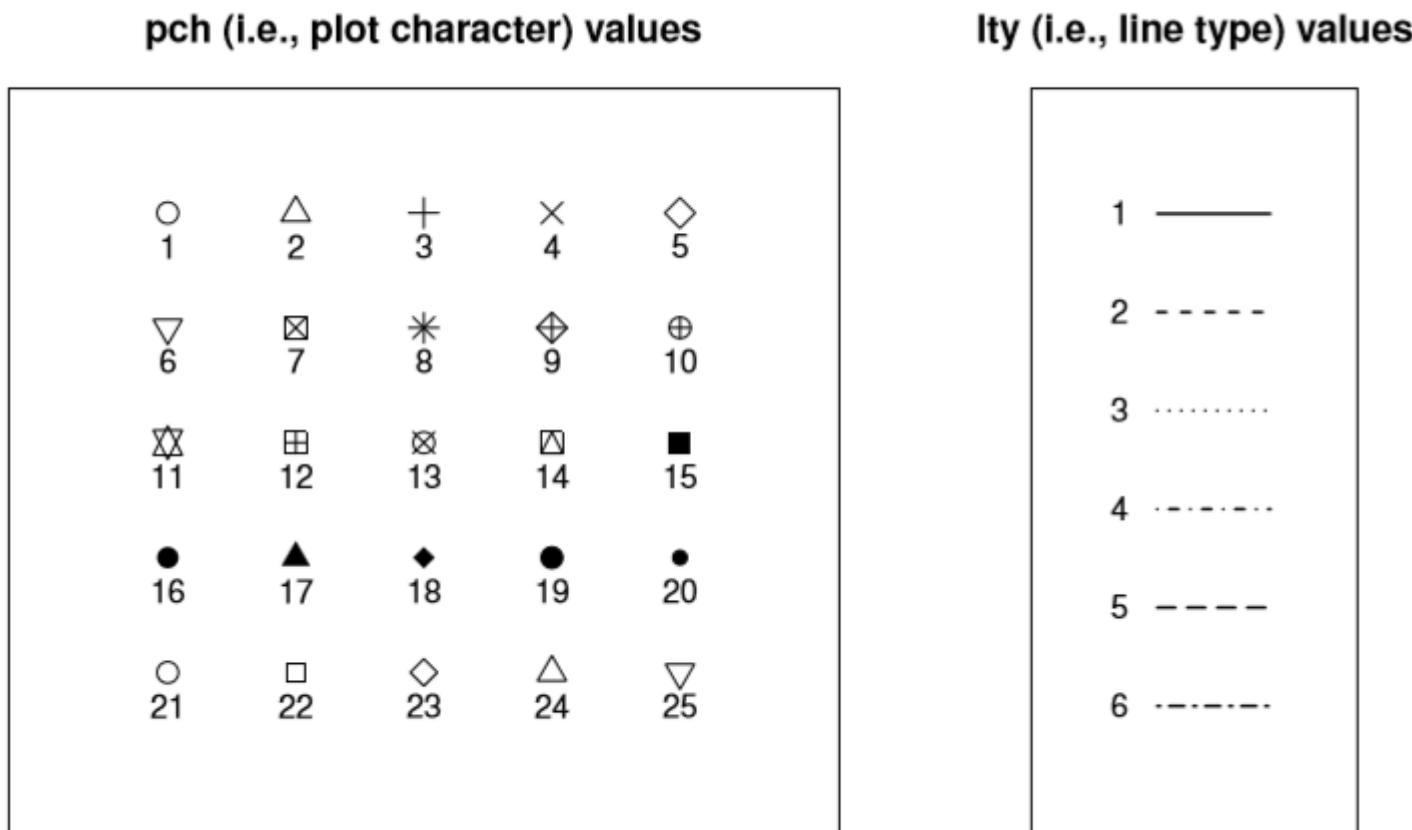


Figure 6.6: Changing the line and plotted characters of the plot.

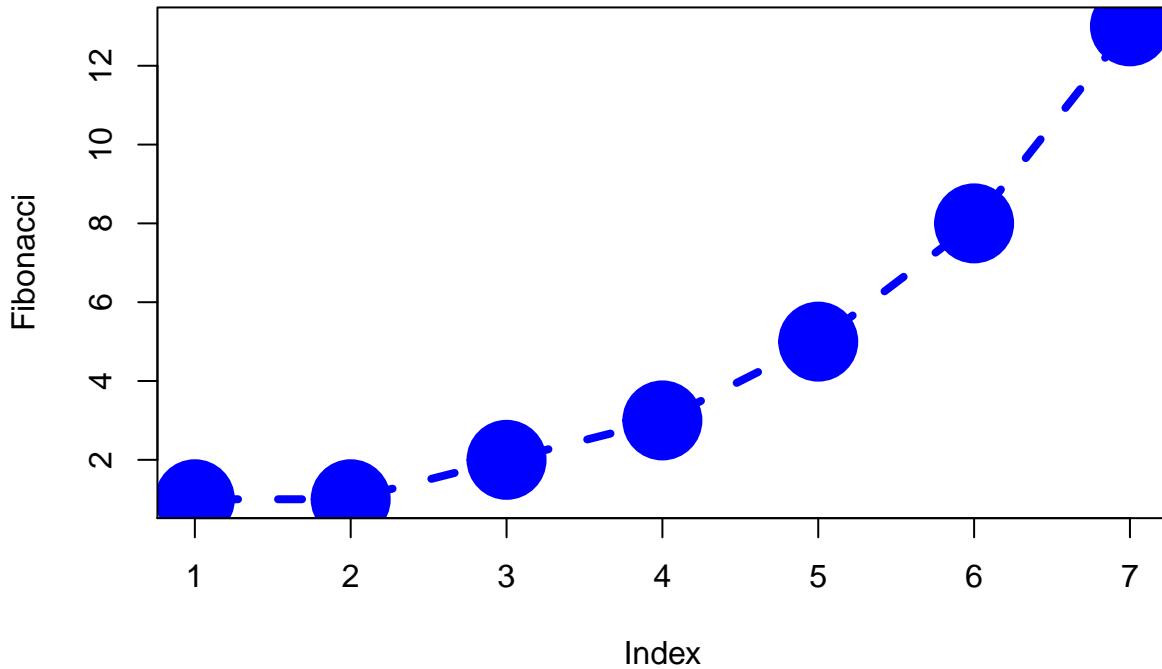


Figure 6.7: Customising various aspects to the plot itself.

```
+      lwd = 4           # change line width to 4x the usual
+ )
```

The output is shown in Figure 6.7.

```
plot( x = Fibonacci,
      type = "b",
      col = "blue",
      pch = 19,
      cex=5,
      lty=2,
      lwd=4)
```

6.2.5 Changing the appearance of the axes

There are several other possibilities worth discussing. Ignoring graphical parameters for the moment, there's a few other arguments to the `plot.default()` function that you might want to use. As before, many of these are standard arguments that are used by a lot of high level graphics functions:

- *Changing the axis scales:* `xlim`, `ylim`. Generally R does a pretty good job of figuring out where to set the edges of the plot. However, you can override its choices by setting the `xlim` and `ylim` arguments.

For instance, if I decide I want the vertical scale of the plot to run from 0 to 100, then I'd set `ylim = c(0, 100)`.

- *Suppress labelling:* `ann`. This is a logical-valued argument that you can use if you don't want R to include any text for a title, subtitle or axis label. To do so, set `ann = FALSE`. This will stop R from including any text that would normally appear in those places. Note that this will override any of your manual titles. For example, if you try to add a title using the `main` argument, but you also specify `ann = FALSE`, no title will appear.
- *Suppress axis drawing:* `axes`. Again, this is a logical valued argument. Suppose you don't want R to draw any axes at all. To suppress the axes, all you have to do is add `axes = FALSE`. This will remove the axes and the numbering, but not the axis labels (i.e. the `xlab` and `ylab` text). Note that you can get finer grain control over this by specifying the `xaxt` and `yaxt` graphical parameters instead (see below).
- *Include a framing box:* `frame.plot`. Suppose you've removed the axes by setting `axes = FALSE`, but you still want to have a simple box drawn around the plot; that is, you only wanted to get rid of the numbering and the tick marks, but you want to keep the box. To do that, you set `frame.plot = TRUE`.

Note that this list isn't exhaustive. There are a few other arguments to the `plot.default` function that you can play with if you want to, but those are the ones you are probably most likely to want to use. As always, however, if these aren't enough options for you, there's also a number of other graphical parameters that you might want to play with as well. That's the focus of the next section. In the meantime, here's a command that makes use of all these different options:

```
> plot( x = Fibonacci,
+       xlim = c(0, 15),      # expand the x-scale
+       ylim = c(0, 15),      # expand the y-scale
+       ann = FALSE,          # delete all annotations
+       axes = FALSE,          # delete the axes
+       frame.plot = TRUE    # but include a framing box
+ )
```

The output is shown in Figure 6.8, and it's pretty much exactly as you'd expect. The axis scales on both the horizontal and vertical dimensions have been expanded, the axes have been suppressed as have the annotations, but I've kept a box around the plot.

```
plot( x = Fibonacci,
      xlim = c(0, 15),
      ylim = c(0, 15),
      ann = FALSE,
      axes = FALSE,
      frame.plot = TRUE)
```

Before moving on, I should point out that there are several graphical parameters relating to the axes, the box, and the general appearance of the plot which allow finer grain control over the appearance of the axes and the annotations.

- *Suppressing the axes individually:* `xaxt`, `yaxt`. These graphical parameters are basically just fancier versions of the `axes` argument we discussed earlier. If you want to stop R from drawing the vertical axis but you'd like it to keep the horizontal axis, set `yaxt = "n"`. I trust that you can figure out how to keep the vertical axis and suppress the horizontal one!
- *Box type:* `bty`. In the same way that `xaxt`, `yaxt` are just fancy versions of `axes`, the `box type` parameter is really just a fancier version of the `frame.plot` argument, allowing you to specify exactly which out of the four borders you want to keep. The way we specify this parameter is a bit stupid, in my opinion: the possible values are "`"o`" (the default), "`"l`", "`"7`", "`"c`", "`"u`", or "`"]`", each of which will

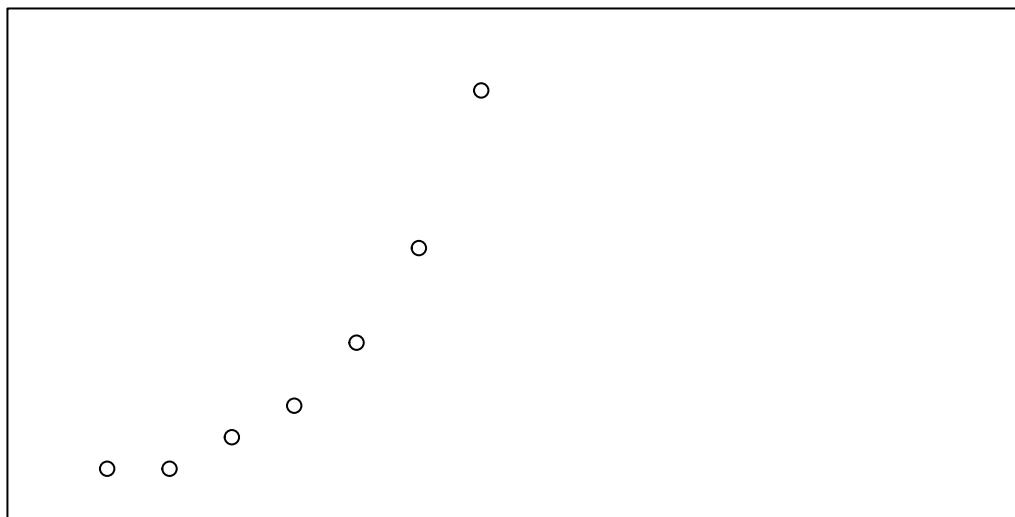


Figure 6.8: Altering the scale and appearance of the plot axes.

draw only those edges that the corresponding character suggests. That is, the letter "c" has a top, a bottom and a left, but is blank on the right hand side, whereas "7" has a top and a right, but is blank on the left and the bottom. Alternatively a value of "n" means that no box will be drawn.

- *Orientation of the axis labels* `las`. I presume that the name of this parameter is an acronym of `label` style or something along those lines; but what it actually does is govern the orientation of the text used to label the individual tick marks (i.e., the numbering, not the `xlab` and `ylab` axis labels). There are four possible values for `las`: A value of 0 means that the labels of both axes are printed parallel to the axis itself (the default). A value of 1 means that the text is always horizontal. A value of 2 means that the labelling text is printed at right angles to the axis. Finally, a value of 3 means that the text is always vertical.

Again, these aren't the only possibilities. There are a few other graphical parameters that I haven't mentioned that you could use to customise the appearance of the axes,⁷ but that's probably enough (or more than enough) for now. To give a sense of how you could use these parameters, let's try the following command:

```
> plot( x = Fibonacci,    # the data
+       xaxt = "n",        # don't draw the x-axis
+       bty = "]",          # keep bottom, right and top of box only
+       las = 1             # rotate the text
+ )
```

The output is shown in Figure 6.9. As you can see, this isn't a very useful plot at all. However, it does illustrate the graphical parameters we're talking about, so I suppose it serves its purpose.

```
plot( x = Fibonacci,
      xaxt = "n",
      bty = "]",
      las = 1 )
```

6.2.6 Don't panic

At this point, a lot of readers will be probably be thinking something along the lines of, “if there’s this much detail just for drawing a simple plot, how horrible is it going to get when we start looking at more complicated things?” Perhaps, contrary to my earlier pleas for mercy, you’ve found a brick to hurl and are right now leafing through an Adelaide phone book trying to find my address. Well, fear not! And please, put the brick down. In a lot of ways, we’ve gone through the hardest part: we’ve already covered vast majority of the plot customisations that you might want to do. As you’ll see, each of the other high level plotting commands we’ll talk about will only have a smallish number of additional options. Better yet, even though I’ve told you about a billion different ways of tweaking your plot, you don’t usually need them. So in practice, now that you’ve read over it once to get the gist, the majority of the content of this section is stuff you can safely forget: just remember to come back to this section later on when you want to tweak your plot.

6.3 Histograms

Now that we’ve tamed (or possibly fled from) the beast that is R graphical parameters, let’s talk more seriously about some real life graphics that you’ll want to draw. We begin with the humble **histogram**. Histograms are one of the simplest and most useful ways of visualising data. They make most sense when

⁷Also, there’s a low level function called `axis()` that allows a lot more control over the appearance of the axes.

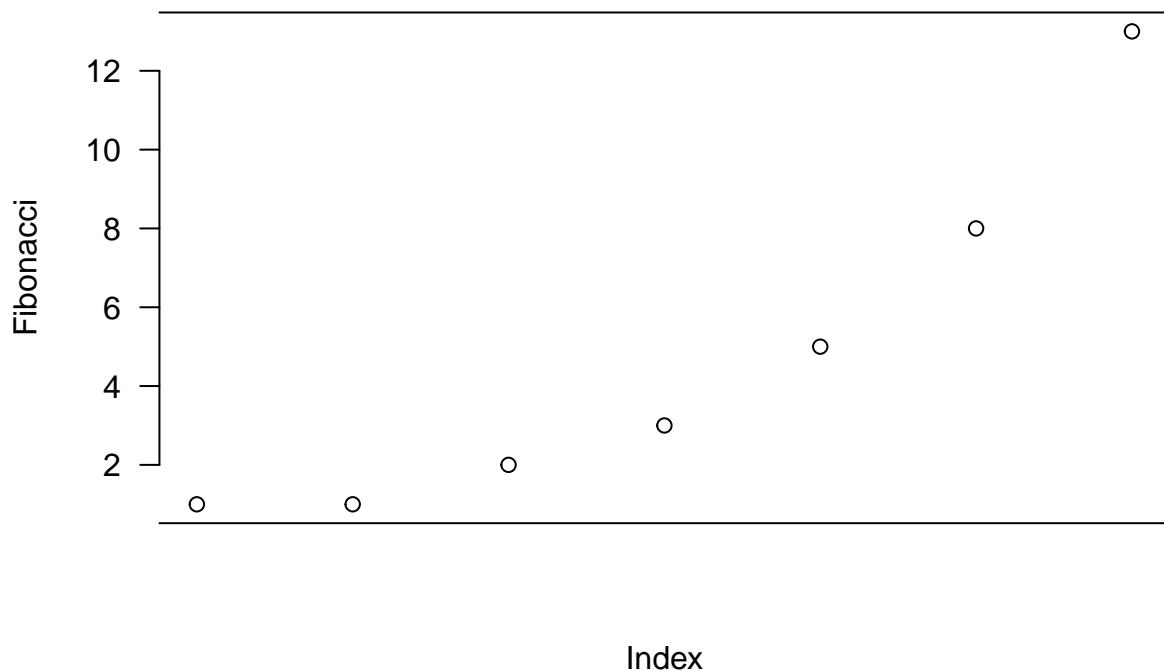


Figure 6.9: Other ways to customise the axes

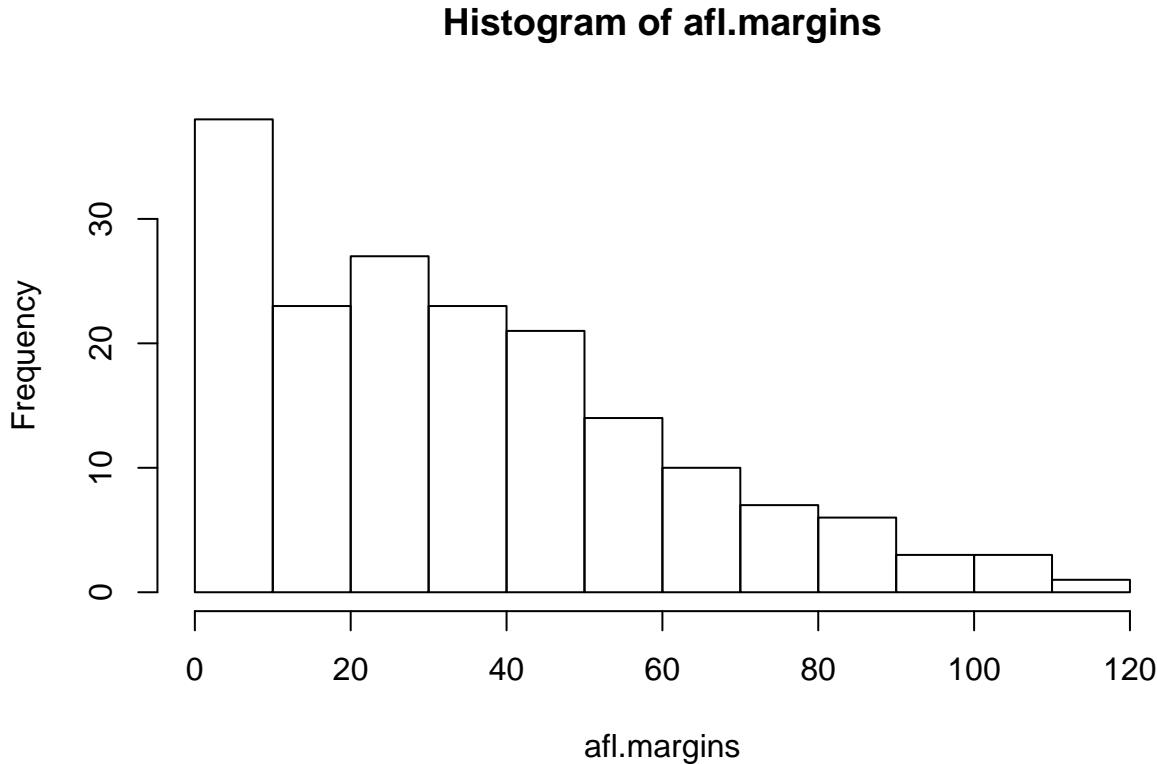


Figure 6.10: The default histogram that R produces

you have an interval or ratio scale (e.g., the `afl.margins` data from Chapter 5 and what you want to do is get an overall impression of the data. Most of you probably know how histograms work, since they're so widely used, but for the sake of completeness I'll describe them. All you do is divide up the possible values into *bins*, and then count the number of observations that fall within each bin. This count is referred to as the frequency of the bin, and is displayed as a bar: in the AFL winning margins data, there are 33 games in which the winning margin was less than 10 points, and it is this fact that is represented by the height of the leftmost bar in Figure 6.10. Drawing this histogram in R is pretty straightforward. The function you need to use is called `hist()`, and it has pretty reasonable default settings. In fact, Figure 6.10 is exactly what you get if you just type this:

```
> hist( afl.margins )    # panel a

load("./rbook-master/data/aflsmall.Rdata")
hist(afl.margins)    # panel a
```

Although this image would need a lot of cleaning up in order to make a good presentation graphic (i.e., one you'd include in a report), it nevertheless does a pretty good job of describing the data. In fact, the big strength of a histogram is that (properly used) it does show the entire spread of the data, so you can get a pretty good sense about what it looks like. The downside to histograms is that they aren't very compact: unlike some of the other plots I'll talk about it's hard to cram 20-30 histograms into a single image without overwhelming the viewer. And of course, if your data are nominal scale (e.g., the `afl.finalists` data) then histograms are useless.

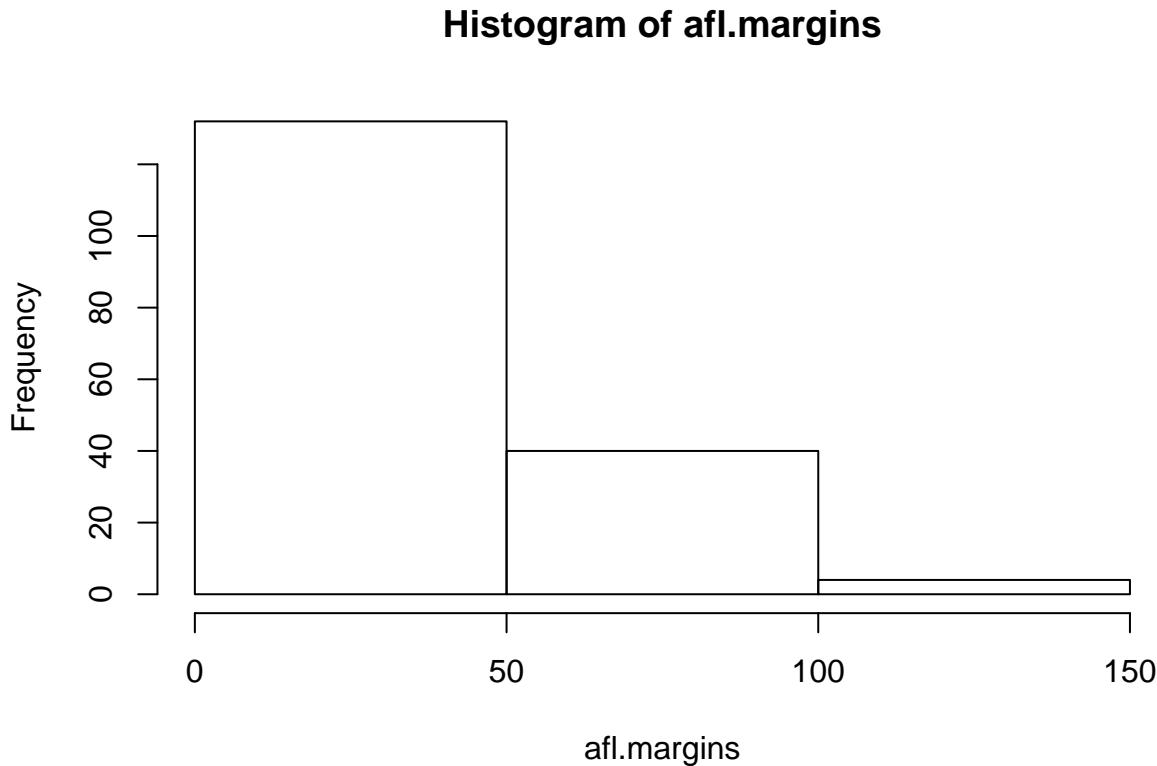


Figure 6.11: A histogram with too few bins

The main subtlety that you need to be aware of when drawing histograms is determining where the `breaks` that separate bins should be located, and (relatedly) how many breaks there should be. In Figure 6.10, you can see that R has made pretty sensible choices all by itself: the breaks are located at 0, 10, 20, ... 120, which is exactly what I would have done had I been forced to make a choice myself. On the other hand, consider the two histograms in Figure 6.11 and 6.12, which I produced using the following two commands:

```
hist( x = afl.margins, breaks = 3 )      # panel b
```

```
hist( x = afl.margins, breaks = 0:116 )  # panel c
```

In Figure 6.12, the bins are only 1 point wide. As a result, although the plot is very informative (it displays the entire data set with no loss of information at all!) the plot is very hard to interpret, and feels quite cluttered. On the other hand, the plot in Figure 6.11 has a bin width of 50 points, and has the opposite problem: it's very easy to “read” this plot, but it doesn't convey a lot of information. One gets the sense that this histogram is hiding too much. In short, the way in which you specify the breaks has a big effect on what the histogram looks like, so it's important to make sure you choose the breaks sensibly. In general R does a pretty good job of selecting the breaks on its own, since it makes use of some quite clever tricks that statisticians have devised for automatically selecting the right bins for a histogram, but nevertheless it's usually a good idea to play around with the breaks a bit to see what happens.

There is one fairly important thing to add regarding how the `breaks` argument works. There are two different ways you can specify the breaks. You can either specify *how many* breaks you want (which is what I did

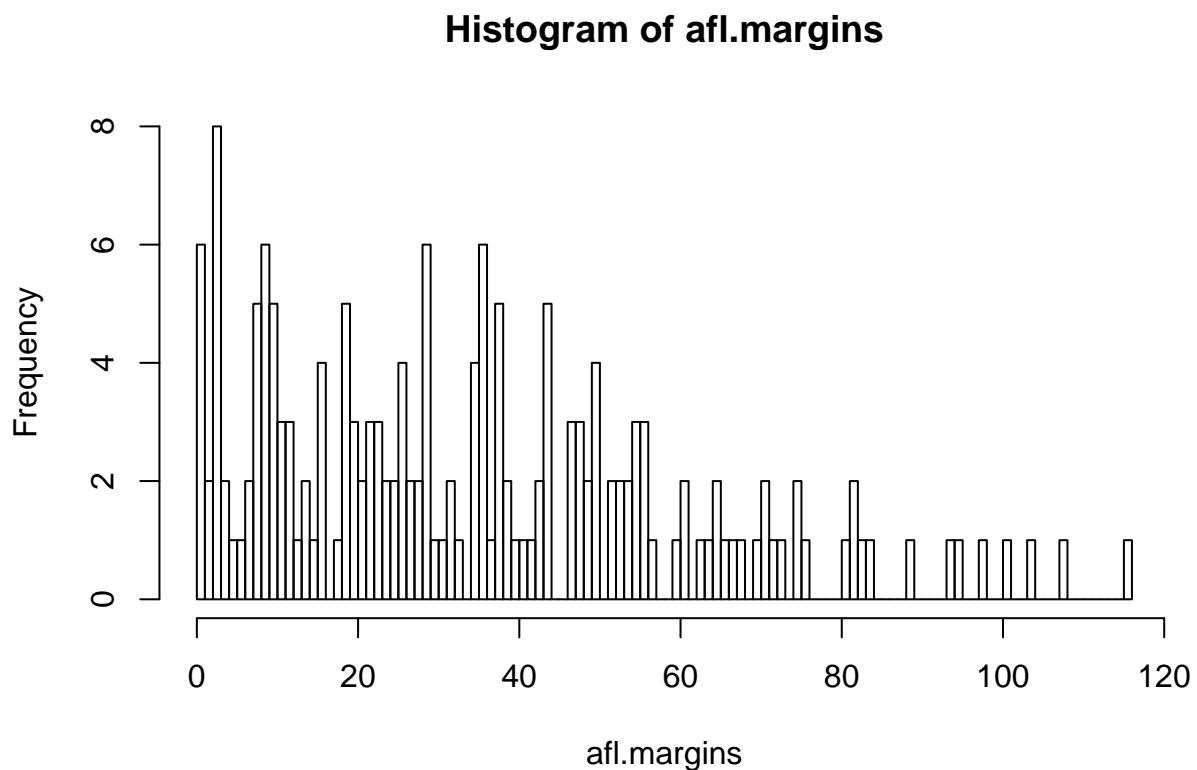


Figure 6.12: A histogram with too many bins

for panel b when I typed `breaks = 3`) and let R figure out where they should go, or you can provide a vector that tells R exactly where the breaks should be placed (which is what I did for panel c when I typed `breaks = 0:116`). The behaviour of the `hist()` function is slightly different depending on which version you use. If all you do is tell it *how many* breaks you want, R treats it as a “suggestion” not as a demand. It assumes you want “approximately 3” breaks, but if it doesn’t think that this would look very pretty on screen, it picks a different (but similar) number. It does this for a sensible reason – it tries to make sure that the breaks are located at sensible values (like 10) rather than stupid ones (like 7.224414). And most of the time R is right: usually, when a human researcher says “give me 3 breaks”, he or she really does mean “give me approximately 3 breaks, and don’t put them in stupid places”. However, sometimes R is dead wrong. Sometimes you really do mean “exactly 3 breaks”, and you know precisely where you want them to go. So you need to invoke “real person privilege”, and order R to do what it’s bloody well told. In order to do that, you *have* to input the full vector that tells R exactly where you want the breaks. If you do that, R will go back to behaving like the nice little obedient calculator that it’s supposed to be.

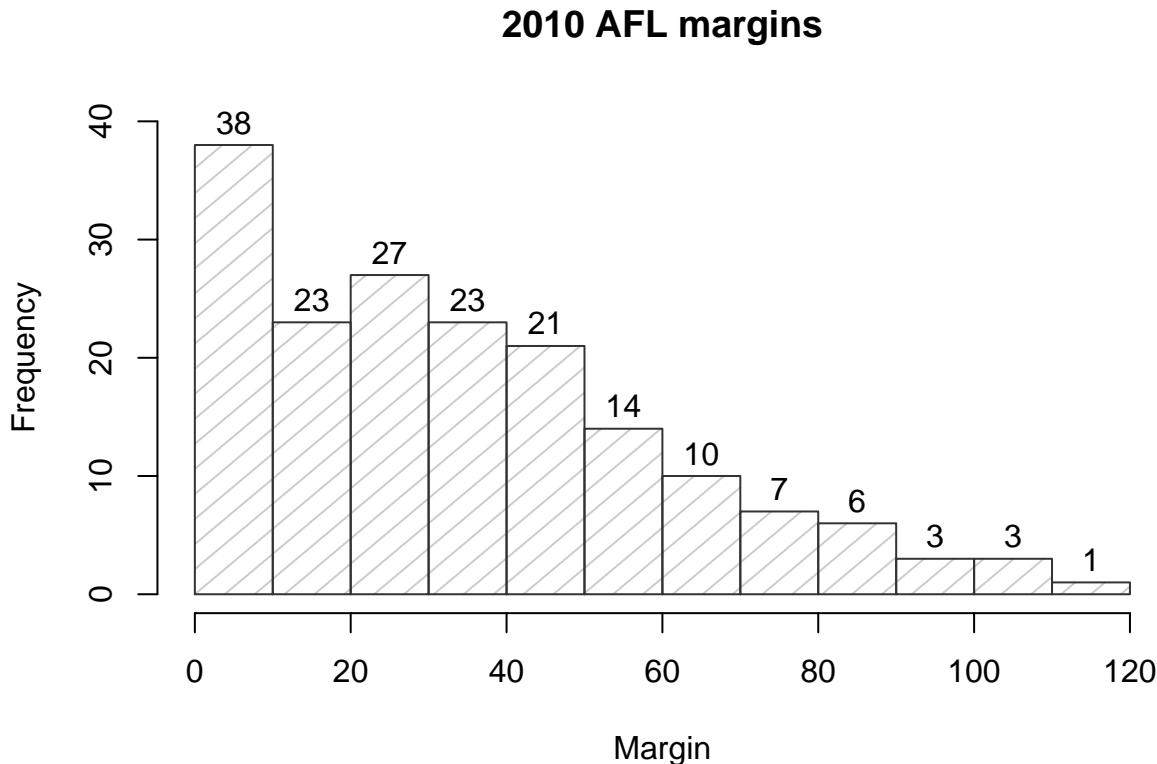
6.3.1 Visual style of your histogram

Okay, so at this point we can draw a basic histogram, and we can alter the number and even the location of the `breaks`. However, the visual style of the histograms shown in Figure @ref(fig:hist1a; hist1b; hist1c) could stand to be improved. We can fix this by making use of some of the other arguments to the `hist()` function. Most of the things you might want to try doing have already been covered in Section 6.2, but there’s a few new things:

- *Shading lines:* `density`, `angle`. You can add diagonal lines to shade the bars: the `density` value is a number indicating how many lines per inch R should draw (the default value of `NULL` means no lines), and the `angle` is a number indicating how many degrees from horizontal the lines should be drawn at (default is `angle = 45` degrees).
- *Specifics regarding colours:* `col`, `border`. You can also change the colours: in this instance the `col` parameter sets the colour of the shading (either the shading lines if there are any, or else the colour of the interior of the bars if there are not), and the `border` argument sets the colour of the edges of the bars.
- *Labelling the bars:* `labels`. You can also attach labels to each of the bars using the `labels` argument. The simplest way to do this is to set `labels = TRUE`, in which case R will add a number just above each bar, that number being the exact number of observations in the bin. Alternatively, you can choose the labels yourself, by inputting a vector of strings, e.g., `labels = c("label 1", "label 2", "etc")`

Not surprisingly, this doesn’t exhaust the possibilities. If you type `help("hist")` or `?hist` and have a look at the help documentation for histograms, you’ll see a few more options. A histogram that makes use of the histogram-specific customisations as well as several of the options we discussed in Section ?? is shown in Figure ???. The R command that I used to draw it is this:

```
hist( x = afl.margins,
      main = "2010 AFL margins", # title of the plot
      xlab = "Margin",           # set the x-axis label
      density = 10,              # draw shading lines: 10 per inch
      angle = 40,                # set the angle of the shading lines is 40 degrees
      border = "gray20",          # set the colour of the borders of the bars
      col = "gray80",             # set the colour of the shading lines
      labels = TRUE,              # add frequency labels to each bar
      ylim = c(0,40)              # change the scale of the y-axis
)
```



Overall, this is a much nicer histogram than the default ones.

6.4 Stem and leaf plots

Histograms are one of the most widely used methods for displaying the observed values for a variable. They're simple, pretty, and very informative. However, they do take a little bit of effort to draw. Sometimes it can be quite useful to make use of simpler, if less visually appealing, options. One such alternative is the *stem and leaf plot*. To a first approximation you can think of a stem and leaf plot as a kind of text-based histogram. Stem and leaf plots aren't used as widely these days as they were 30 years ago, since it's now just as easy to draw a histogram as it is to draw a stem and leaf plot. Not only that, they don't work very well for larger data sets. As a consequence you probably won't have as much of a need to use them yourself, though you may run into them in older publications. These days, the only real world situation where I use them is if I have a small data set with 20-30 data points and I don't have a computer handy, because it's pretty easy to quickly sketch a stem and leaf plot by hand.

With all that as background, lets have a look at stem and leaf plots. The AFL margins data contains 176 observations, which is at the upper end for what you can realistically plot this way. The function in R for drawing stem and leaf plots is called `stem()` and if we ask for a stem and leaf plot of the `afl.margins` data, here's what we get:

```
stem( afl.margins )
## 
##   The decimal point is 1 digit(s) to the right of the |
##
```

```
##   0 | 00111122333333344567788888999999
##   1 | 0000011122234456666899999
##   2 | 0001122233445566667788999999
##   3 | 0122355556666678888899
##   4 | 0123344444777888899
##   5 | 00002233445556667
##   6 | 0113455678
##   7 | 01123556
##   8 | 122349
##   9 | 458
##  10 | 148
##  11 | 6
```

The values to the left of the | are called *stems* and the values to the right are called *leaves*. If you just look at the shape that the leaves make, you can see something that looks a lot like a histogram made out of numbers, just rotated by 90 degrees. But if you know how to read the plot, there's quite a lot of additional information here. In fact, it's also giving you the actual values of *all* of the observations in the data set. To illustrate, let's have a look at the last line in the stem and leaf plot, namely 11 | 6. Specifically, let's compare this to the largest values of the `afl.margins` data set:

```
> max( afl.margins )
[1] 116
```

Hm... 11 | 6 versus 116. Obviously the stem and leaf plot is trying to tell us that the largest value in the data set is 116. Similarly, when we look at the line that reads 10 | 148, the way we interpret it to note that the stem and leaf plot is telling us that the data set contains observations with values 101, 104 and 108. Finally, when we see something like 5 | 00002233445556667 the four 0s in the the stem and leaf plot are telling us that there are four observations with value 50.

I won't talk about them in a lot of detail, but I should point out that some customisation options are available for stem and leaf plots in R. The two arguments that you can use to do this are:

- **scale**. Changing the `scale` of the plot (default value is 1), which is analogous to changing the number of breaks in a histogram. Reducing the scale causes R to reduce the number of stem values (i.e., the number of breaks, if this were a histogram) that the plot uses.
- **width**. The second way that to can customise a stem and leaf plot is to alter the `width` (default value is 80). Changing the width alters the maximum number of leaf values that can be displayed for any given stem.

However, since stem and leaf plots aren't as important as they used to be, I'll leave it to the interested reader to investigate these options. Try the following two commands to see what happens:

```
> stem( x = afl.margins, scale = .25 )
> stem( x = afl.margins, width = 20 )
```

The only other thing to note about stem and leaf plots is the line in which R tells you where the decimal point is. If our data set had included only the numbers .11, .15, .23, .35 and .59 and we'd drawn a stem and leaf plot of these data, then R would move the decimal point: the stem values would be 1,2,3,4 and 5, but R would tell you that the decimal point has moved to the left of the | symbol. If you want to see this in action, try the following command:

```
> stem( x = afl.margins / 1000 )
```

The stem and leaf plot itself will look identical to the original one we drew, except for the fact that R will tell you that the decimal point has moved.

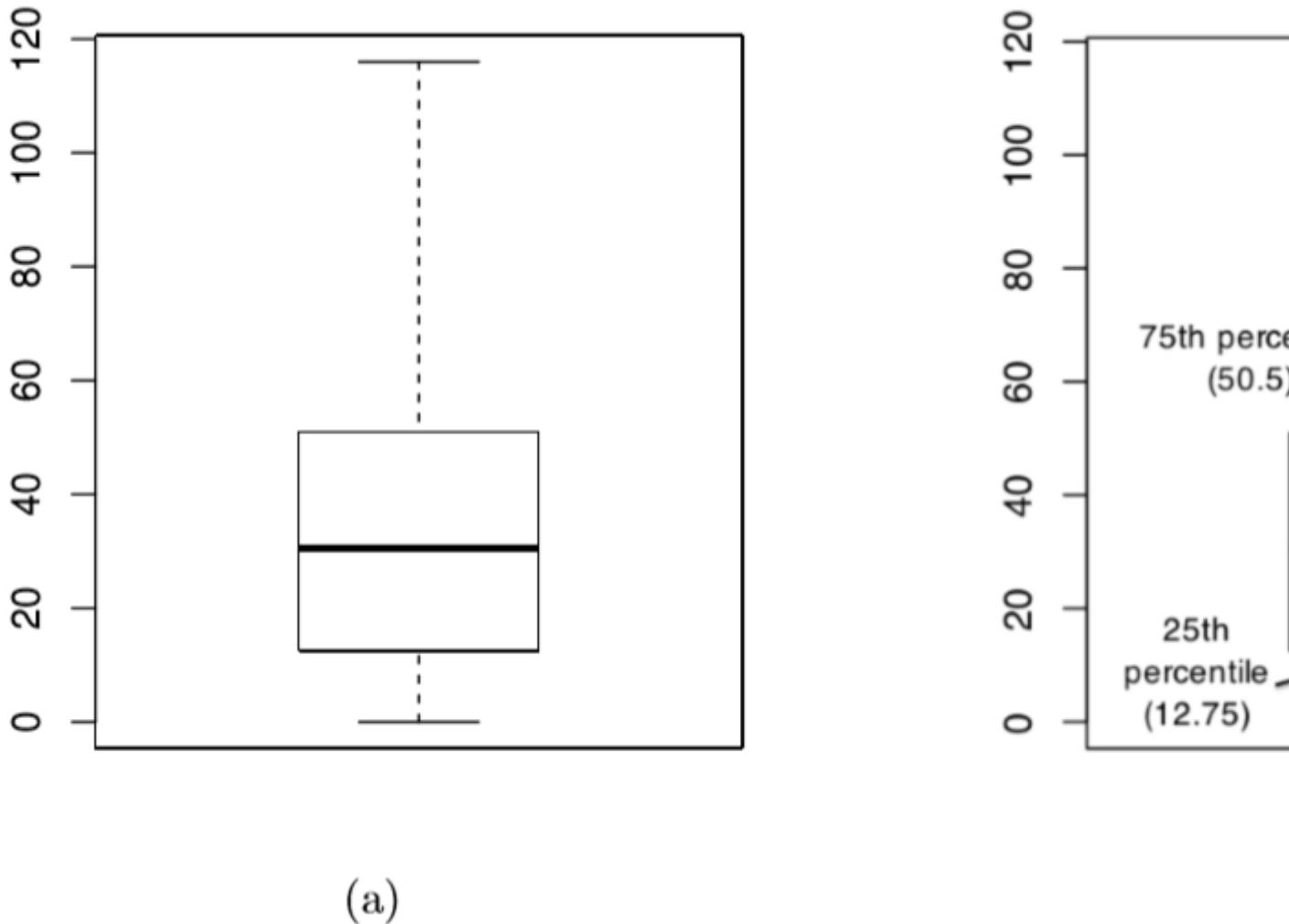


Figure 6.13: A basic boxplot (panel a), plus the same plot with annotations added to explain what aspect of the data set each part of the boxplot corresponds to (panel b).

6.5 Boxplots

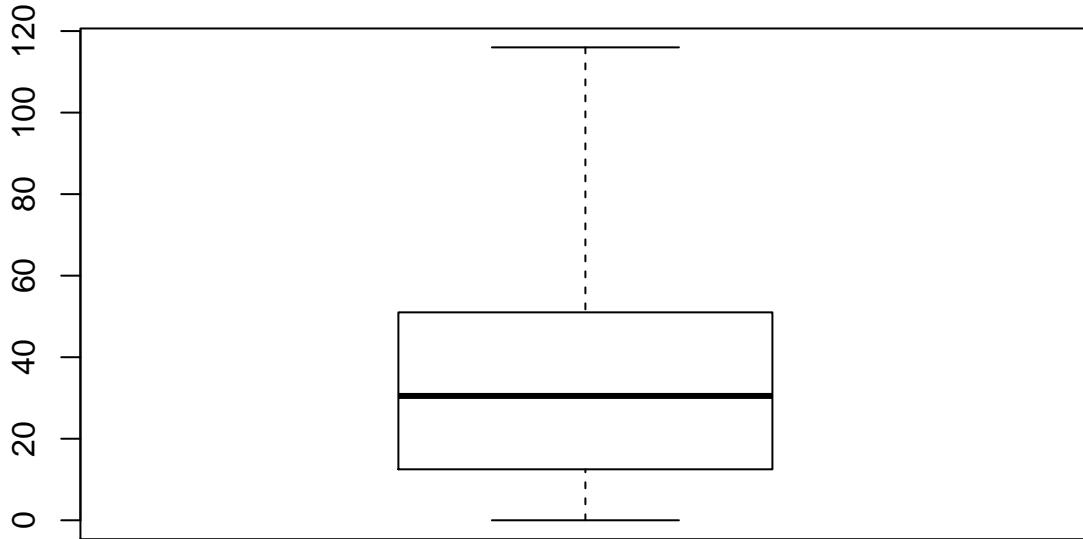
Another alternative to histograms is a ***boxplot***, sometimes called a “box and whiskers” plot. Like histograms, they’re most suited to interval or ratio scale data. The idea behind a boxplot is to provide a simple visual depiction of the median, the interquartile range, and the range of the data. And because they do so in a fairly compact way, boxplots have become a very popular statistical graphic, especially during the exploratory stage of data analysis when you’re trying to understand the data yourself. Let’s have a look at how they work, again using the `afl.margins` data as our example. Firstly, let’s actually calculate these numbers ourselves using the `summary()` function:⁸

```
> summary( afl.margins )
   Min. 1st Qu. Median     Mean 3rd Qu.    Max.
0.00    12.75   30.50   35.30   50.50  116.00
```

⁸R being what it is, it’s no great surprise that there’s also a `fivenum()` function that does much the same thing.

So how does a boxplot capture these numbers? The easiest way to describe what a boxplot looks like is just to draw one. The function for doing this in R is (surprise, surprise) `boxplot()`. As always there's a lot of optional arguments that you can specify if you want, but for the most part you can just let R choose the defaults for you. That said, I'm going to override one of the defaults to start with by specifying the `range` option, but for the most part you won't want to do this (I'll explain why in a minute). With that as preamble, let's try the following command:

```
boxplot( x = afl.margins, range = 100 )
```



What R draws is shown in Figure ??, the most basic boxplot possible. When you look at this plot, this is how you should interpret it: the thick line in the middle of the box is the median; the box itself spans the range from the 25th percentile to the 75th percentile; and the “whiskers” cover the full range from the minimum value to the maximum value. This is summarised in the annotated plot in Figure ??.

In practice, this isn't quite how boxplots usually work. In most applications, the “whiskers” don't cover the full range from minimum to maximum. Instead, they actually go out to the most extreme data point that doesn't exceed a certain bound. By default, this value is 1.5 times the interquartile range, corresponding to a `range` value of 1.5. Any observation whose value falls outside this range is plotted as a circle instead of being covered by the whiskers, and is commonly referred to as an *outlier*. For our AFL margins data, there is one observation (a game with a margin of 116 points) that falls outside this range. As a consequence, the upper whisker is pulled back to the next largest observation (a value of 108), and the observation at 116 is plotted as a circle. This is illustrated in Figure @ref(fig:boxplot2a). Since the default value is `range = 1.5` we can draw this plot using the simple command

```
boxplot( afl.margins )
```

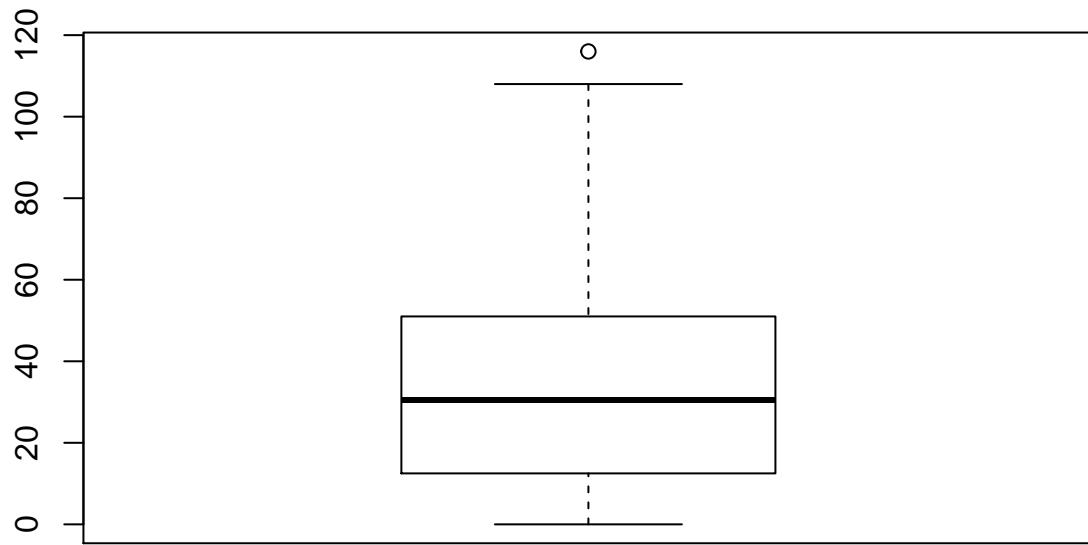
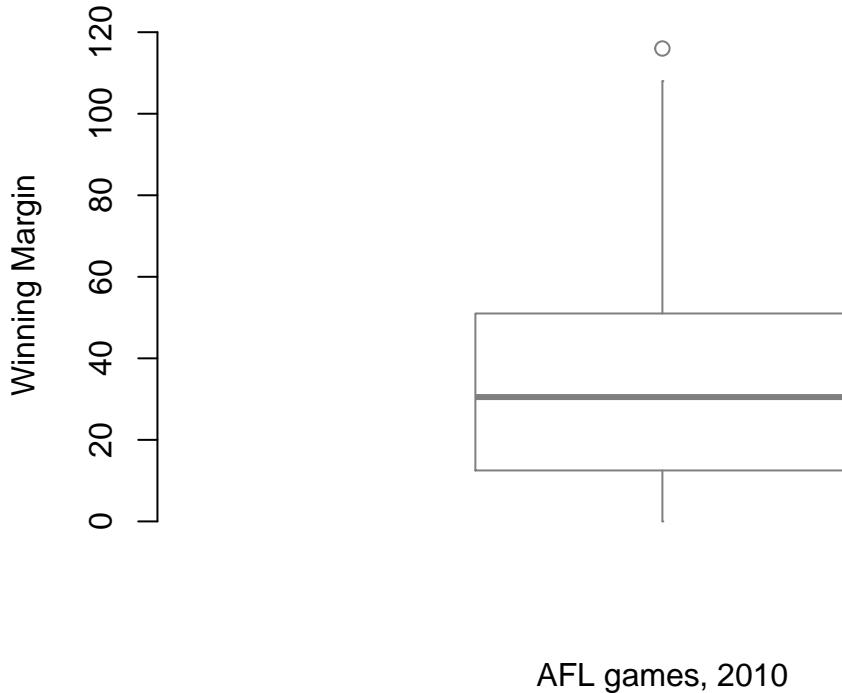


Figure 6.14: By default, R will only extent the whiskers a distance of 1.5 times the interquartile range, and will plot any points that fall outside that range separately

6.5.1 Visual style of your boxplot

I'll talk a little more about the relationship between boxplots and outliers in the Section ??, but before I do let's take the time to clean this figure up. Boxplots in R are extremely customisable. In addition to the usual range of graphical parameters that you can tweak to make the plot look nice, you can also exercise nearly complete control over every element to the plot. Consider the boxplot in Figure ??: in this version of the plot, not only have I added labels (`xlab`, `ylab`) and removed the stupid border (`frame.plot`), I've also dimmed all of the graphical elements of the boxplot except the central bar that plots the median (`border`) so as to draw more attention to the median rather than the rest of the boxplot.



You've seen all these options in previous sections in this chapter, so hopefully those customisations won't need any further explanation. However, I've done two new things as well: I've deleted the cross-bars at the top and bottom of the whiskers (known as the "staples" of the plot), and converted the whiskers themselves to solid lines. The arguments that I used to do this are called by the ridiculous names of `staplewex` and `whisklty`,⁹ and I'll explain these in a moment.

But first, here's the actual command I used to draw this figure:

```
> boxplot( x = afl.margins,           # the data
+           xlab = "AFL games, 2010", # x-axis label
+           ylab = "Winning Margin", # y-axis label
+           border = "grey50",       # dim the border of the box
+           frame.plot = FALSE,     # don't draw a frame
+           staplewex = 0,          # don't draw staples
```

⁹I realise there's a kind of logic to the way R names are constructed, but they still sound dumb. When I typed this sentence, all I could think was that it sounded like the name of a kids movie if it had been written by Lewis Carroll: "The frabjous gambolles of Staplewex and Whisklty" or something along those lines.

```
+      whisklty = 1          # solid line for whisker
+ )
```

Overall, I think the resulting boxplot is a huge improvement in visual design over the default version. In my opinion at least, there's a fairly minimalist aesthetic that governs good statistical graphics. Ideally, every visual element that you add to a plot should convey part of the message. If your plot includes things that don't actually help the reader learn anything new, you should consider removing them. Personally, I can't see the point of the cross-bars on a standard boxplot, so I've deleted them.

Okay, what commands can we use to customise the boxplot? If you type `?boxplot` and flick through the help documentation, you'll notice that it does mention `staplewex` as an argument, but there's no mention of `whisklty`. The reason for this is that the function that handles the drawing is called `bxp()`, so if you type `?bxp` all the gory details appear. Here's the short summary. In order to understand why these arguments have such stupid names, you need to recognise that they're put together from two components. The first part of the argument name specifies one part of the box plot: `staple` refers to the staples of the plot (i.e., the cross-bars), and `whisk` refers to the whiskers. The second part of the name specifies a graphical parameter: `wex` is a width parameter, and `lty` is a line type parameter. The parts of the plot you can customise are:

- `box`. The box that covers the interquartile range.
- `med`. The line used to show the median.
- `whisk`. The vertical lines used to draw the whiskers.
- `staple`. The cross bars at the ends of the whiskers.
- `out`. The points used to show the outliers.

The actual graphical parameters that you might want to specify are slightly different for each visual element, just because they're different shapes from each other. As a consequence, the following options are available:

- *Width expansion*: `boxwex`, `staplewex`, `outwex`. These are scaling factors that govern the width of various parts of the plot. The default scaling factor is (usually) 0.8 for the box, and 0.5 for the other two. Note that in the case of the outliers this parameter is meaningless unless you decide to draw lines plotting the outliers rather than use points.
- *Line type*: `boxlty`, `medlty`, `whisklty`, `staplelty`, `outlty`. These govern the line type for the relevant elements. The values for this are exactly the same as those used for the regular `lty` parameter, with two exceptions. There's an additional option where you can set `medlty = "blank"` to suppress the median line completely (useful if you want to draw a point for the median rather than plot a line). Similarly, by default the outlier line type is set to `outlty = "blank"`, because the default behaviour is to draw outliers as points instead of lines.
- *Line width*: `boxlwd`, `medlwd`, `whisklwd`, `staplelwd`, `outlwd`. These govern the line widths for the relevant elements, and behave the same way as the regular `lwd` parameter. The only thing to note is that the default value for `medlwd` value is three times the value of the others.
- *Line colour*: `boxcol`, `medcol`, `whiskcol`, `staplecol`, `outcol`. These govern the colour of the lines used to draw the relevant elements. Specify a colour in the same way that you usually do.
- *Fill colour*: `boxfill`. What colour should we use to fill the box?
- *Point character*: `medpch`, `outpch`. These behave like the regular `pch` parameter used to select the plot character. Note that you can set `outpch = NA` to stop R from plotting the outliers at all, and you can also set `medpch = NA` to stop it from drawing a character for the median (this is the default!).
- *Point expansion*: `medcex`, `outcex`. Size parameters for the points used to plot medians and outliers. These are only meaningful if the corresponding points are actually plotted. So for the default boxplot, which includes outlier points but uses a line rather than a point to draw the median, only the `outcex` parameter is meaningful.
- *Background colours*: `medbg`, `outbg`. Again, the background colours are only meaningful if the points are actually plotted.

Taken as a group, these parameters allow you almost complete freedom to select the graphical style for your boxplot that you feel is most appropriate to the data set you're trying to describe. That said, when you're first starting out there's no shame in using the default settings! But if you want to master the art of designing beautiful figures, it helps to try playing around with these parameters to see what works and what doesn't. Finally, I should mention a few other arguments that you might want to make use of:

- **horizontal**. Set this to TRUE to display the plot horizontally rather than vertically.
- **varwidth**. Set this to TRUE to get R to scale the width of each box so that the areas are proportional to the number of observations that contribute to the boxplot. This is only useful if you're drawing multiple boxplots at once (see Section 6.5.3).
- **show.names**. Set this to TRUE to get R to attach labels to the boxplots.
- **notch**. If you set **notch** = TRUE, R will draw little notches in the sides of each box. If the notches of two boxplots don't overlap, then there is a "statistically significant" difference between the corresponding medians. If you haven't read Chapter 11, ignore this argument – we haven't discussed statistical significance, so this doesn't mean much to you. I'm mentioning it only because you might want to come back to the topic later on. (see also the **notch.frac** option when you type `?bxp`).

6.5.2 Using box plots to detect outliers

Because the boxplot automatically (unless you change the **range** argument) separates out those observations that lie within a certain range, people often use them as an informal method for detecting **outliers**: observations that are "suspiciously" distant from the rest of the data. Here's an example. Suppose that I'd drawn the boxplot for the AFL margins data, and it came up looking like Figure 6.15.

It's pretty clear that something funny is going on with one of the observations. Apparently, there was one game in which the margin was over 300 points! That doesn't sound right to me. Now that I've become suspicious, it's time to look a bit more closely at the data. One function that can be handy for this is the **which()** function; it takes as input a vector of logicals, and outputs the indices of the TRUE cases. This is particularly useful in the current context because it lets me do this:

```
> suspicious.cases <- afl.margins > 300
> which( suspicious.cases )
[1] 137
```

although in real life I probably wouldn't bother creating the **suspicious.cases** variable: I'd just cut out the middle man and use a command like `which(afl.margins > 300)`. In any case, what this has done is shown me that the outlier corresponds to game 137. Then, I find the recorded margin for that game:

```
> afl.margins[137]
[1] 333
```

Hm. That definitely doesn't sound right. So then I go back to the original data source (the internet!) and I discover that the actual margin of that game was 33 points. Now it's pretty clear what happened. Someone must have typed in the wrong number. Easily fixed, just by typing `afl.margins[137] <- 33`. While this might seem like a silly example, I should stress that this kind of thing actually happens a lot. Real world data sets are often riddled with stupid errors, especially when someone had to type something into a computer at some point. In fact, there's actually a name for this phase of data analysis, since in practice it can waste a huge chunk of our time: **data cleaning**. It involves searching for typos, missing data and all sorts of other obnoxious errors in raw data files.¹⁰

¹⁰Sometimes it's convenient to have the boxplot automatically label the outliers for you. The original `boxplot()` function doesn't allow you to do this; however, the `Boxplot()` function in the `car` package does. The design of the `Boxplot()` function is very similar to `boxplot()`. It just adds a few new arguments that allow you to tweak the labelling scheme. I'll leave it to the reader to check this out.

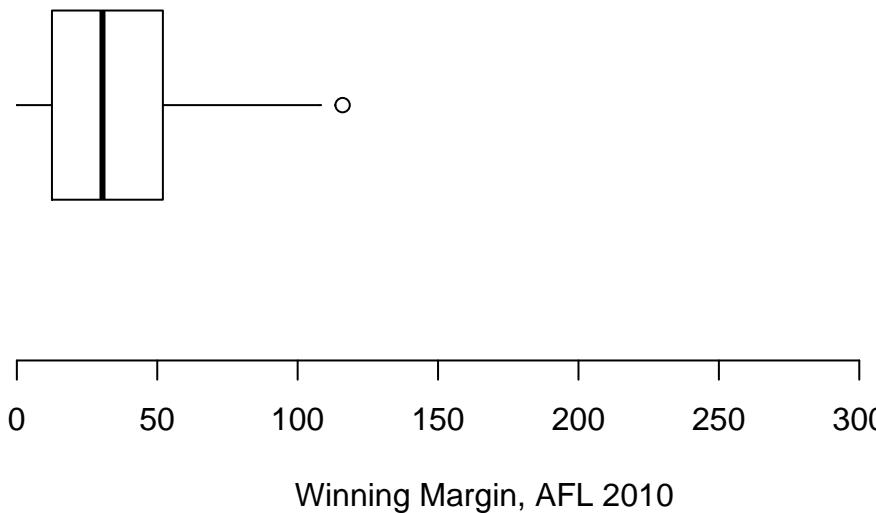


Figure 6.15: A boxplot showing one very suspicious outlier! I've drawn this plot in a similar, minimalist style to the one in Figure ??, but I've used the `horizontal` argument to draw it sideways in order to save space.

What about the real data? Does the value of 116 constitute a funny observation not? Possibly. As it turns out the game in question was Fremantle v Hawthorn, and was played in round 21 (the second last home and away round of the season). Fremantle had already qualified for the final series and for them the outcome of the game was irrelevant; and the team decided to rest several of their star players. As a consequence, Fremantle went into the game severely underpowered. In contrast, Hawthorn had started the season very poorly but had ended on a massive winning streak, and for them a win could secure a place in the finals. With the game played on Hawthorn's home turf¹¹ and with so many unusual factors at play, it is perhaps no surprise that Hawthorn annihilated Fremantle by 116 points. Two weeks later, however, the two teams met again in an elimination final on Fremantle's home ground, and Fremantle won comfortably by 30 points.¹²

So, should we exclude the game from subsequent analyses? If this were a psychology experiment rather than an AFL season, I'd be quite tempted to exclude it because there's pretty strong evidence that Fremantle weren't really trying very hard: and to the extent that my research question is based on an assumption that participants are genuinely trying to do the task. On the other hand, in a lot of studies we're actually interested in seeing the full range of possible behaviour, and that includes situations where people decide not to try very hard: so excluding that observation would be a bad idea. In the context of the AFL data, a similar distinction applies. If I'd been trying to make tips about who would perform well in the finals, I would have (and in fact did) disregard the Round 21 massacre, because it's way too misleading. On the other hand, if my interest is solely in the home and away season itself, I think it would be a shame to throw away information pertaining to one of the most distinctive (if boring) games of the year. In other words, the decision about whether to include outliers or exclude them depends heavily on *why* you think the data look they way they do, and what you want to use the data *for*. Statistical tools can provide an automatic method for suggesting candidates for deletion, but you really need to exercise good judgment here. As I've said before, R is a mindless automaton. It doesn't watch the footy, so it lacks the broader context to make an informed decision. You are *not* a mindless automaton, so you should exercise judgment: if the outlier looks legitimate to you, then keep it. In any case, I'll return to the topic again in Section ??, so let's return to our discussion of how to draw boxplots.

6.5.3 Drawing multiple boxplots

One last thing. What if you want to draw multiple boxplots at once? Suppose, for instance, I wanted separate boxplots showing the AFL margins not just for 2010, but for every year between 1987 and 2010. To do that, the first thing we'll have to do is find the data. These are stored in the `aflsmall12.Rdata` file. So let's load it and take a quick peek at what's inside:

```
> load( "aflsmall12.Rdata" )
> who( TRUE )
-- Name --  -- Class --  -- Size --
afl2        data.frame    4296 x 2
$margin     numeric       4296
$year       numeric       4296
```

Notice that `afl2` data frame is pretty big. It contains 4296 games, which is far more than I want to see printed out on my computer screen. To that end, R provides you with a few useful functions to print out only a few of the row in the data frame. The first of these is `head()` which prints out the first 6 rows, of the data frame, like this:

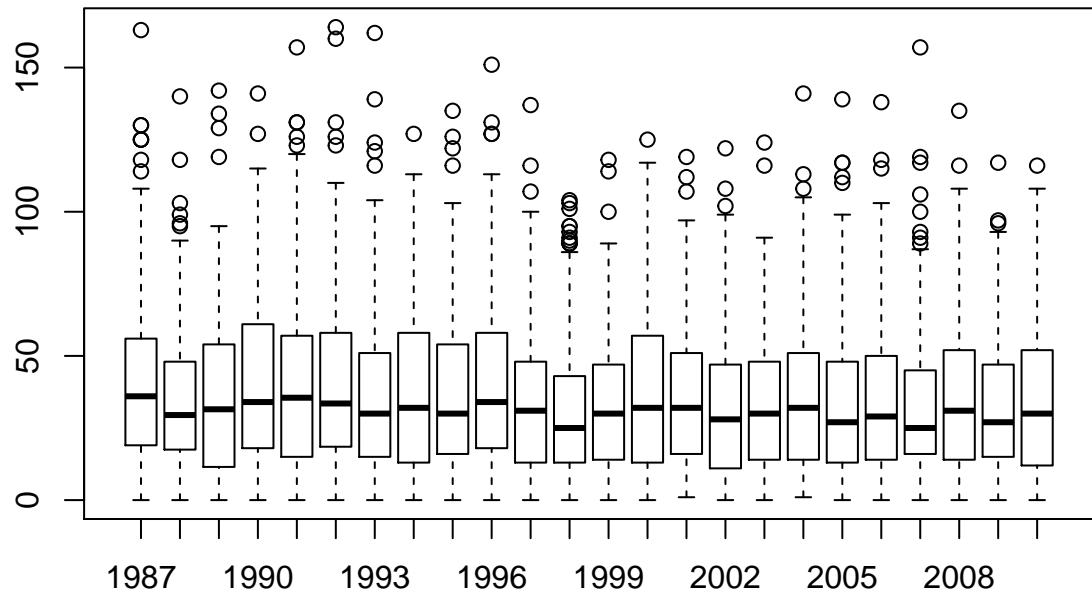
¹¹Sort of. The game was played in Launceston, which is a de facto home away from home for Hawthorn.

¹²Contrast this situation with the next largest winning margin in the data set, which was Geelong's 108 point demolition of Richmond in round 6 at their home ground, Kardinia Park. Geelong have been one of the most dominant teams over the last several years, a period during which they strung together an incredible 29-game winning streak at Kardinia Park. Richmond have been useless for several years. This is in no meaningful sense an outlier. Geelong have been winning by these margins (and Richmond losing by them) for quite some time. Frankly I'm surprised that the result wasn't more lopsided: as happened to Melbourne in 2011 when Geelong won by a modest 186 points.

```
> head( afl2 )
  margin year
1      33 1987
2      59 1987
3      45 1987
4      91 1987
5      39 1987
6      1 1987
```

You can also use the `tail()` function to print out the last 6 rows. The `car` package also provides a handy little function called `some()` which prints out a random subset of the rows.

In any case, the important thing is that we have the `afl2` data frame which contains the variables that we're interested in. What we want to do is have R draw boxplots for the `margin` variable, plotted separately for each separate `year`. The way to do this using the `boxplot()` function is to input a `formula` rather than a variable as the input. In this case, the formula we want is `margin ~ year`. So our boxplot command now looks like



this:

The result is shown in Figure ??.¹³ Even this, the default version of the plot, gives a sense of why it's sometimes useful to choose boxplots instead of histograms. Even before taking the time to turn this basic output into something more readable, it's possible to get a good sense of what the data look like from year to year without getting overwhelmed with too much detail. Now imagine what would have happened if I'd tried to cram 24 histograms into this space: no chance at all that the reader is going to learn anything useful.

¹³Actually, there's other ways to do this. If the input argument `x` is a list object (see Section 4.9, the `boxplot()` function will draw a separate boxplot for each variable in that list. Relatedly, since the `plot()` function – which we'll discuss shortly – is a generic (see Section 4.11, you might not be surprised to learn that one of its special cases is a boxplot: specifically, if you use `plot()` where the first argument `x` is a factor and the second argument `y` is numeric, then the result will be a boxplot, showing the values in `y`, with a separate boxplot for each level. For instance, something like `plot(x = afl2$year, y = afl2$margin)` would work.

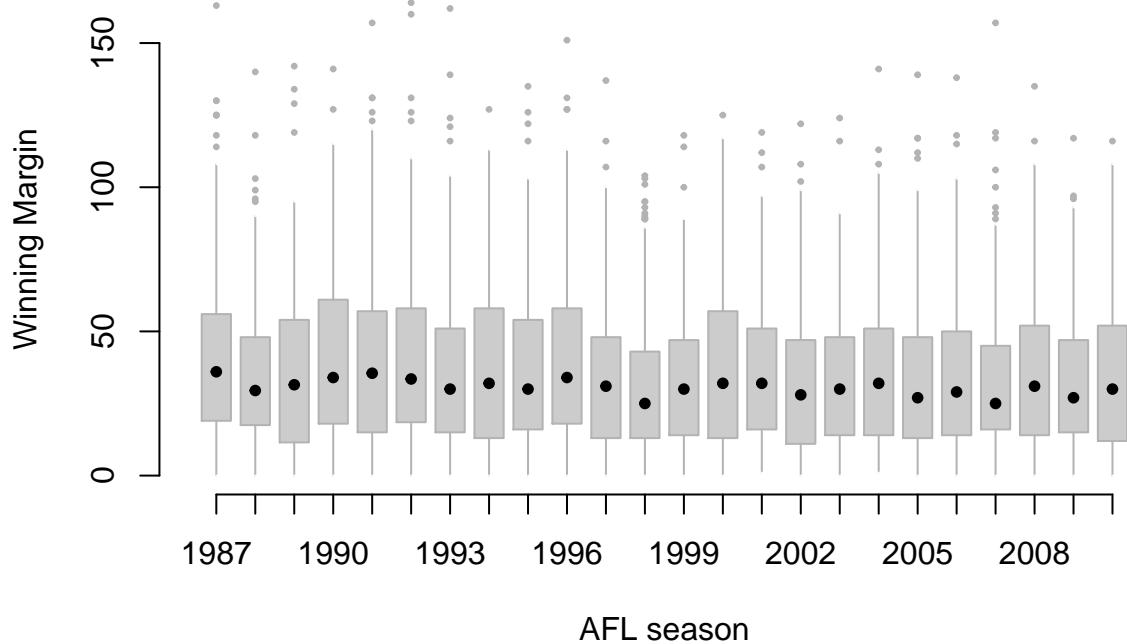


Figure 6.16: A cleaned up version of Figure ???. Notice that I've used a very minimalist design for the boxplots, so as to focus the eye on the medians. I've also converted the medians to solid dots, to convey a sense that year to year variation in the median should be thought of as a single coherent plot (similar to what we did when plotting the Fibonacci variable earlier). The size of outliers has been shrunk, because they aren't actually very interesting. In contrast, I've added a fill colour to the boxes, to make it easier to look at the trend in the interquartile range across years.

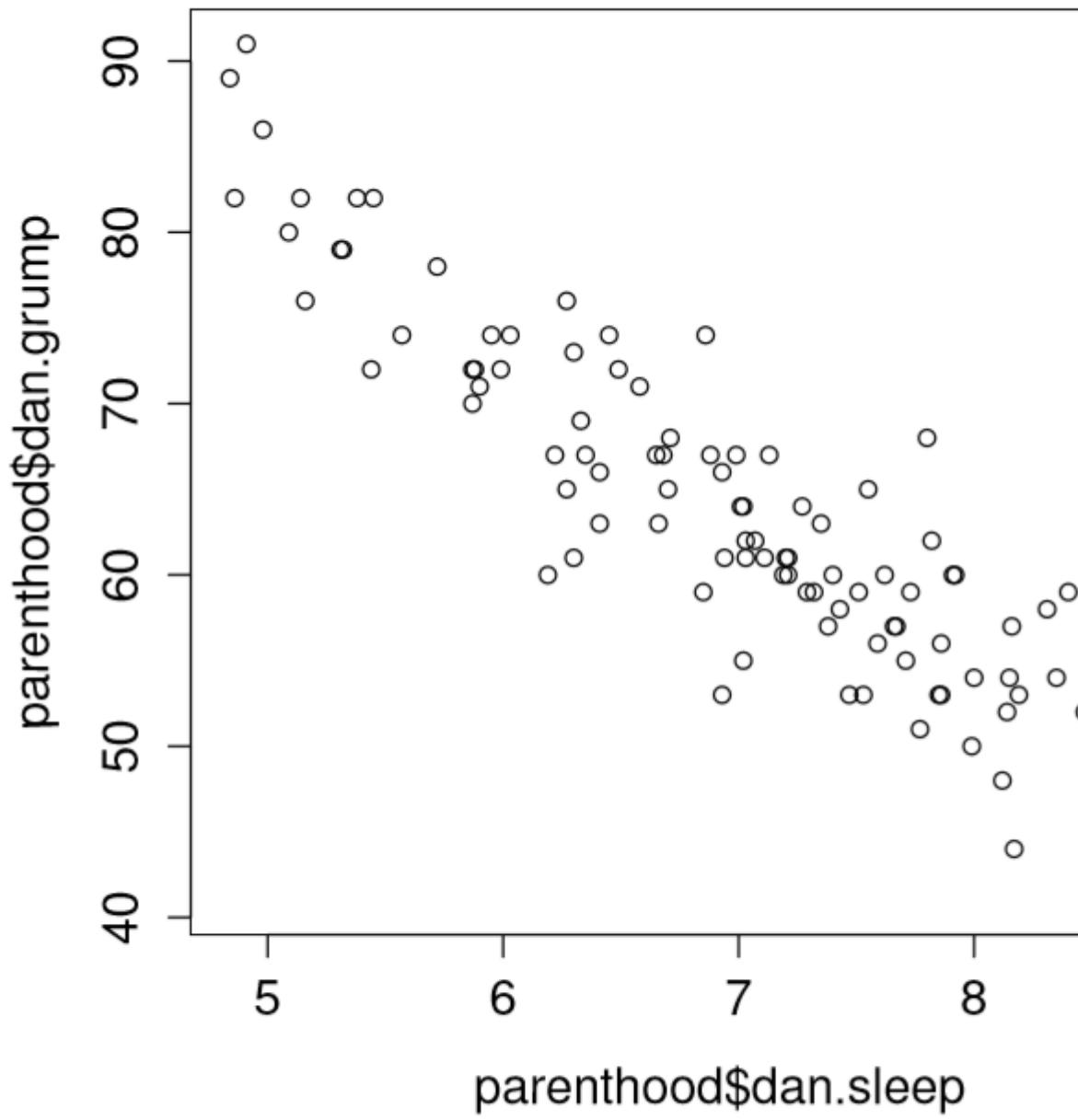
That being said, the default boxplot leaves a great deal to be desired in terms of visual clarity. The outliers are too visually prominent, the dotted lines look messy, and the interesting content (i.e., the behaviour of the median and the interquartile range across years) gets a little obscured. Fortunately, this is easy to fix, since we've already covered a lot of tools you can use to customise your output. After playing around with several different versions of the plot, the one I settled on is shown in Figure 6.16. The command I used to produce it is long, but not complicated:

```
> boxplot( formula = margin ~ year,      # the formula
+          data = afl2,                  # the data set
+          xlab = "AFL season",        # x axis label
+          ylab = "Winning Margin",    # y axis label
+          frame.plot = FALSE,        # don't draw a frame
+          staplewex = 0,              # don't draw staples
+          staplecol = "white",       # (fixes a tiny display issue)
+          boxwex = .75,               # narrow the boxes slightly
+          boxfill = "grey80",         # lightly shade the boxes
+          whisklty = 1,               # solid line for whiskers
+          whiskcol = "grey70",        # dim the whiskers
+          boxcol = "grey70",          # dim the box borders
+          outcol = "grey70",          # dim the outliers
+          outpch = 20,                # outliers as solid dots
+          outcex = .5,                # shrink the outliers
+          medlty = "blank",           # no line for the medians
+          medpch = 20,                # instead, draw solid dots
+          medlwd = 1.5,               # make them larger
+ )
```

Of course, given that the command is that long, you might have guessed that I didn't spend ages typing all that rubbish in over and over again. Instead, I wrote a script, which I kept tweaking until it produced the figure that I wanted. We'll talk about scripts later in Section 8.1, but given the length of the command I thought I'd remind you that there's an easier way of trying out different commands than typing them all in over and over.

6.6 Scatterplots

Scatterplots are a simple but effective tool for visualising data. We've already seen scatterplots in this chapter, when using the `plot()` function to draw the `Fibonacci` variable as a collection of dots (Section 6.2). However, for the purposes of this section I have a slightly different notion in mind. Instead of just plotting one variable, what I want to do with my scatterplot is display the relationship between *two* variables, like we saw with the figures in the section on correlation (Section 5.7). It's this latter application that we usually have in mind when we use the term "scatterplot". In this kind of plot, each observation corresponds to one dot: the horizontal location of the dot plots the value of the observation on one variable, and the vertical location displays its value on the other variable. In many situations you don't really have a clear opinions about what the *causal* relationship is (e.g., does A cause B, or does B cause A, or does some other variable C control both A and B). If that's the case, it doesn't really matter which variable you plot on the x-axis and which one you plot on the y-axis. However, in many situations you do have a pretty strong idea which variable you think is most likely to be causal, or at least you have some suspicions in that direction. If so, then it's conventional to plot the cause variable on the x-axis, and the effect variable on the y-axis. With that in mind, let's look at how to draw scatterplots in R, using the same `parenthood` data set (i.e. `parenthood.Rdata`) that I used when introducing the idea of correlations.



(a)

\begin{figure}

\caption{{Two different scatterplots: (a) the default scatterplot that R produces, (b) one that makes use of several options for fancier display.}} \end{figure}

Suppose my goal is to draw a scatterplot displaying the relationship between the amount of sleep that I get (`dan.sleep`) and how grumpy I am the next day (`dan.grump`). As you might expect given our earlier use of `plot()` to display the `Fibonacci` data, the function that we use is the `plot()` function, but because it's a generic function all the hard work is still being done by the `plot.default()` function. In any case, there

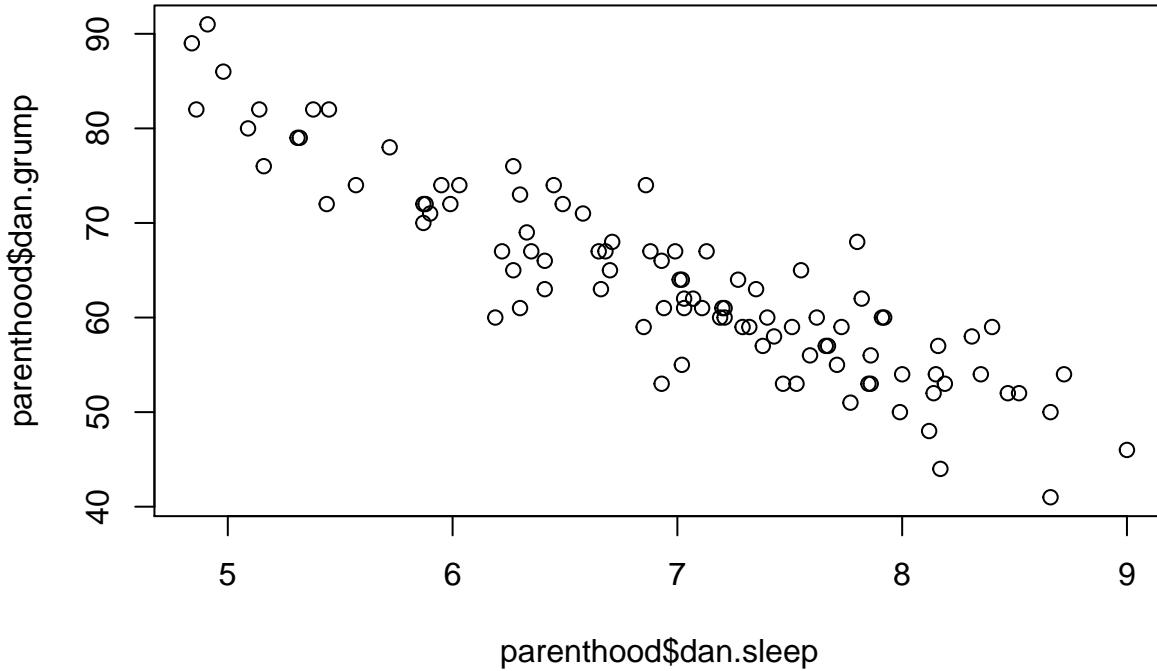


Figure 6.17: the default scatterplot that R produces

are two different ways in which we can get the plot that we're after. The first way is to specify the name of the variable to be plotted on the x axis and the variable to be plotted on the y axis. When we do it this way, the command looks like this:

```
plot( x = parenthood$dan.sleep,    # data on the x-axis
      y = parenthood$dan.grump    # data on the y-axis
)
```

The second way do to it is to use a “formula and data frame” format, but I’m going to avoid using it.¹⁴ For now, let’s just stick with the x and y version. If we do this, the result is the very basic scatterplot shown in Figure 6.17. This serves fairly well, but there’s a few customisations that we probably want to make in order to have this work properly. As usual, we want to add some labels, but there’s a few other things we might want to do as well. Firstly, it’s sometimes useful to rescale the plots. In Figure 6.17 R has selected the scales so that the data fall neatly in the middle. But, in this case, we happen to know that the grumpiness measure falls on a scale from 0 to 100, and the hours slept falls on a natural scale between 0 hours and about 12 or so hours (the longest I can sleep in real life). So the command I might use to draw this is:

¹⁴The reason is that there’s an annoying design flaw in the way the `plot()` function handles this situation. The problem is that the `plot.formula()` function uses different names to for the arguments than the `plot()` function expects. As a consequence, you can’t specify the formula argument by name. If you just specify a formula as the first argument without using the name it works fine, because the `plot()` function thinks the formula corresponds to the `x` argument, and the `plot.formula()` function thinks it corresponds to the `formula` argument; and surprisingly, everything works nicely. But the moment that you, the user, tries to be unambiguous about the name, one of those two functions is going to cry.

```

plot( x = parenthood$dan.sleep,           # data on the x-axis
      y = parenthood$dan.grump,          # data on the y-axis
      xlab = "My sleep (hours)",        # x-axis label
      ylab = "My grumpiness (0-100)",   # y-axis label
      xlim = c(0,12),                  # scale the x-axis
      ylim = c(0,100),                  # scale the y-axis
      pch = 20,                        # change the plot type
      col = "gray50",                  # dim the dots slightly
      frame.plot = FALSE              # don't draw a box
)

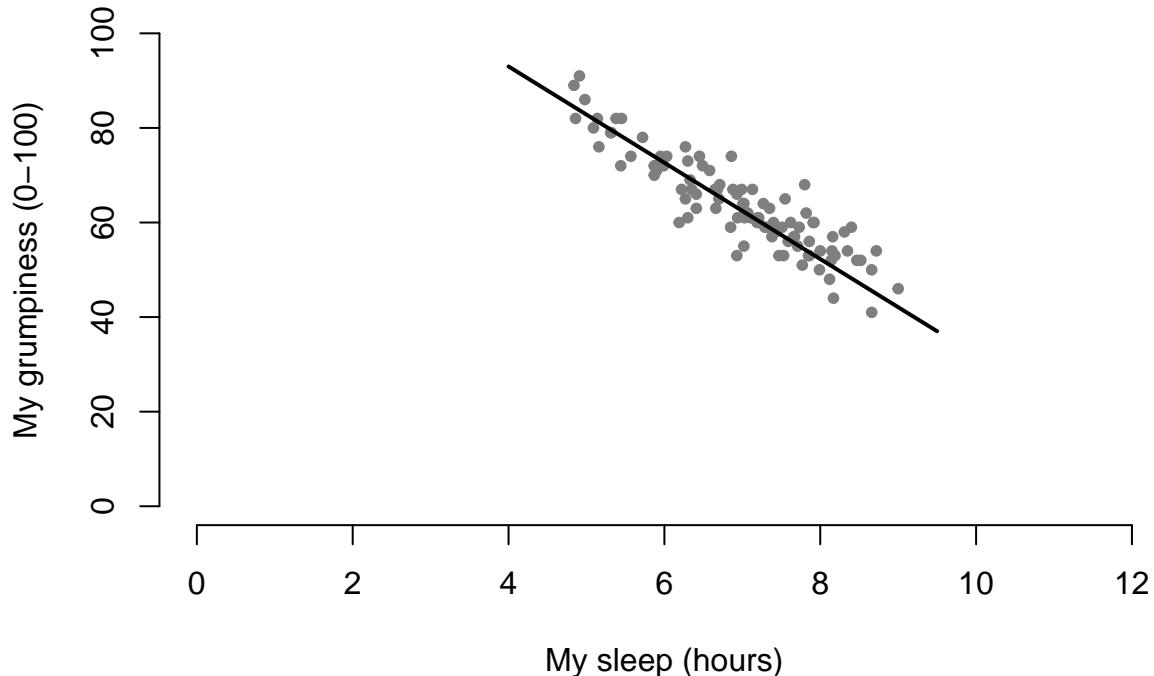
```

This command produces the scatterplot in Figure ??, or at least very nearly. What it doesn't do is draw the line through the middle of the points. Sometimes it can be very useful to do this, and I can do so using `lines()`, which is a low level plotting function. Better yet, the arguments that I need to specify are pretty much the exact same ones that I use when calling the `plot()` function. That is, suppose that I want to draw a line that goes from the point (4,93) to the point (9.5,37). Then the x locations can be specified by the vector `c(4,9.5)` and the y locations correspond to the vector `c(93,37)`. In other words, I use this command:

```

plot( x = parenthood$dan.sleep,           # data on the x-axis
      y = parenthood$dan.grump,          # data on the y-axis
      xlab = "My sleep (hours)",        # x-axis label
      ylab = "My grumpiness (0-100)",   # y-axis label
      xlim = c(0,12),                  # scale the x-axis
      ylim = c(0,100),                  # scale the y-axis
      pch = 20,                        # change the plot type
      col = "gray50",                  # dim the dots slightly
      frame.plot = FALSE              # don't draw a box
)
lines( x = c(4,9.5),    # the horizontal locations
       y = c(93,37),    # the vertical locations
       lwd = 2          # line width
)

```



And when I do so, R plots the line over the top of the plot that I drew using the previous command. In most realistic data analysis situations you absolutely don't want to just guess where the line through the points goes, since there's about a billion different ways in which you can get R to do a better job. However, it does at least illustrate the basic idea.

One possibility, if you do want to get R to draw nice clean lines through the data for you, is to use the `scatterplot()` function in the `car` package. Before we can use `scatterplot()` we need to load the package:

```
> library( car )
```

Having done so, we can now use the function. The command we need is this one:

```
> scatterplot( dan.grump ~ dan.sleep,
+               data = parenthood,
+               smooth = FALSE
+ )
```

The first two arguments should be familiar: the first input is a formula `dan.grump ~ dan.sleep` telling R what variables to plot,¹⁵ and the second specifies a `data` frame. The third argument `smooth` I've set to `FALSE` to stop the `scatterplot()` function from drawing a fancy "smoothed" trendline (since it's a bit

¹⁵You might be wondering why I haven't specified the argument name for the formula. The reason is that there's a bug in how the `scatterplot()` function is written: under the hood there's one function that expects the argument to be named `x` and another one that expects it to be called `formula`. I don't know why the function was written this way, but it's not an isolated problem: this particular kind of bug repeats itself in a couple of other functions (you'll see it again in Chapter ??). The solution in such cases is to omit the argument name: that way, one function "thinks" that you've specified `x` and the other one "thinks" you've specified `formula` and everything works the way it's supposed to. It's not a great state of affairs, I'll admit, but it sort of works.

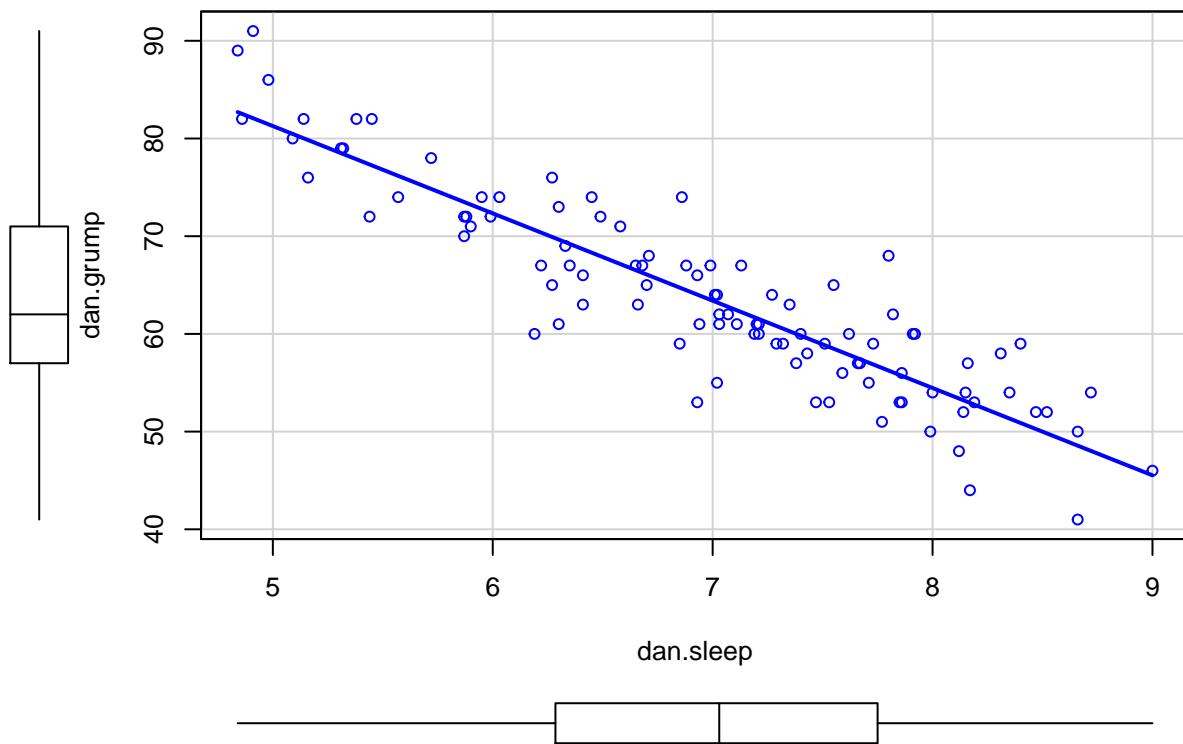


Figure 6.18: A fancy scatterplot drawn using the `scatterplot()` function in the `car` package.

confusing to beginners). The scatterplot itself is shown in Figure 6.18. As you can see, it's not only drawn the scatterplot, but its also drawn boxplots for each of the two variables, as well as a simple line of best fit showing the relationship between the two variables.

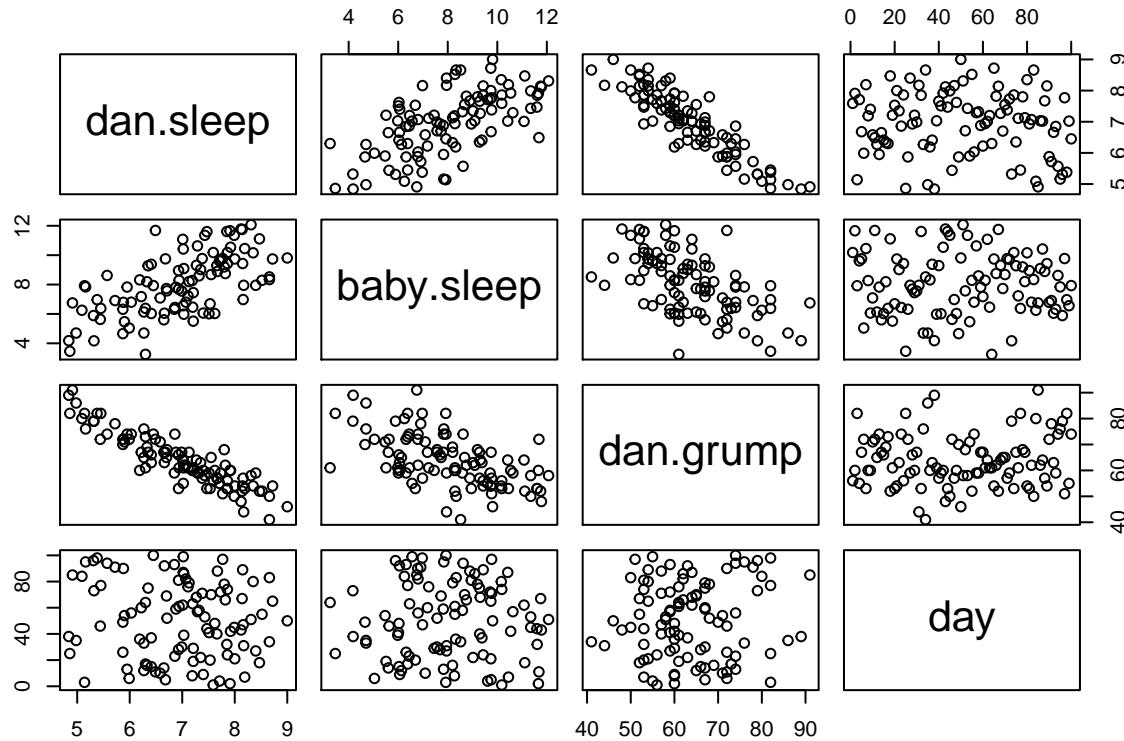
6.6.1 More elaborate options

Often you find yourself wanting to look at the relationships between several variables at once. One useful tool for doing so is to produce a *scatterplot matrix*, analogous to the correlation matrix.

```
> cor( x = parenthood ) # calculate correlation matrix
      dan.sleep  baby.sleep  dan.grump      day
dan.sleep    1.00000000  0.62794934 -0.90338404 -0.09840768
baby.sleep   0.62794934  1.00000000 -0.56596373 -0.01043394
dan.grump   -0.90338404 -0.56596373  1.00000000  0.07647926
day        -0.09840768 -0.01043394  0.07647926  1.00000000
```

We can get a the corresponding scatterplot matrix by using the `pairs()` function:¹⁶

```
pairs( x = parenthood ) # draw corresponding scatterplot matrix
```



The output of the `pairs()` command is shown in Figure ???. An alternative way of calling the `pairs()` function, which can be useful in some situations, is to specify the variables to include using a one-sided formula. For instance, this

¹⁶ Yet again, we could have produced this output using the `plot()` function: when the `x` argument is a data frame containing numeric variables only, then the output is a scatterplot matrix. So, once again, what I could have done is just type `plot(parenthood)`.

```
> pairs( formula = ~ dan.sleep + baby.sleep + dan.grump,
+         data = parenthood
+ )
```

would produce a 3×3 scatterplot matrix that only compare `dan.sleep`, `dan.grump` and `baby.sleep`. Obviously, the first version is much easier, but there are cases where you really only want to look at a few of the variables, so it's nice to use the formula interface.

6.7 Bar graphs

Another form of graph that you often want to plot is the ***bar graph***. The main function that you can use in R to draw them is the `barplot()` function.¹⁷ And to illustrate the use of the function, I'll use the `finalists` variable that I introduced in Section 5.1.7. What I want to do is draw a bar graph that displays the number of finals that each team has played in over the time spanned by the `afl` data set. So, let's start by creating a vector that contains this information. I'll use the `tabulate()` function to do this (which will be discussed properly in Section ??, since it creates a simple numeric vector:

```
> freq <- tabulate( afl.finalists )
> print( freq )
[1] 26 25 26 28 32  0  6 39 27 28 28 17  6 24 26 39 24
```

This isn't exactly the prettiest of frequency tables, of course. I'm only doing it this way so that you can see the `barplot()` function in its "purest" form: when the input is just an ordinary numeric vector. That being said, I'm obviously going to need the team names to create some labels, so let's create a variable with those. I'll do this using the `levels()` function, which outputs the names of all the levels of a factor (see Section 4.7):

```
> teams <- levels( afl.finalists )
> print( teams )
[1] "Adelaide"          "Brisbane"           "Carlton"            "Collingwood"
[5] "Essendon"            "Fitzroy"             "Fremantle"          "Geelong"
[9] "Hawthorn"            "Melbourne"           "North Melbourne"    "Port Adelaide"
[13] "Richmond"           "St Kilda"            "Sydney"             "West Coast"
[17] "Western Bulldogs"
```

Okay, so now that we have the information we need, let's draw our bar graph. The main argument that you need to specify for a bar graph is the `height` of the bars, which in our case correspond to the values stored in the `freq` variable:

```
> barplot( height = freq ) # specifying the argument name (panel a)
> barplot( freq )        # the lazier version (panel a)
```

Either of these two commands will produce the simple bar graph shown in Figure 6.19.

As you can see, R has drawn a pretty minimal plot. It doesn't have any labels, obviously, because we didn't actually tell the `barplot()` function what the labels are! To do this, we need to specify the `names.arg` argument. The `names.arg` argument needs to be a vector of character strings containing the text that needs to be used as the label for each of the items. In this case, the `teams` vector is exactly what we need, so the command we're looking for is:

¹⁷Once again, it's worth noting the link to the generic `plot()` function. If the `x` argument to `plot()` is a factor (and no `y` argument is given), the result is a bar graph. So you could use `plot(afl.finalists)` and get the same output as `barplot(afl.finalists)`.

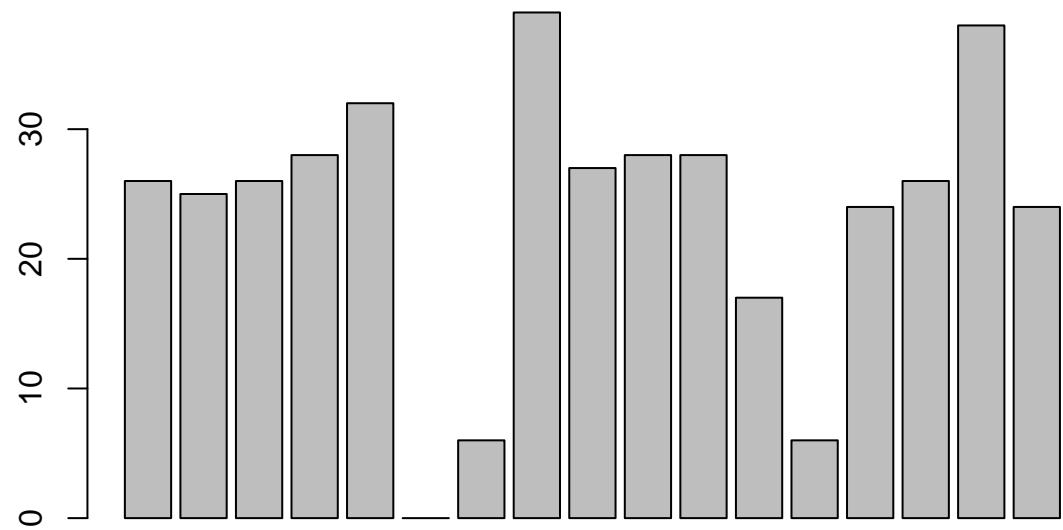


Figure 6.19: the simplest version of a bargraph, containing the data but no labels

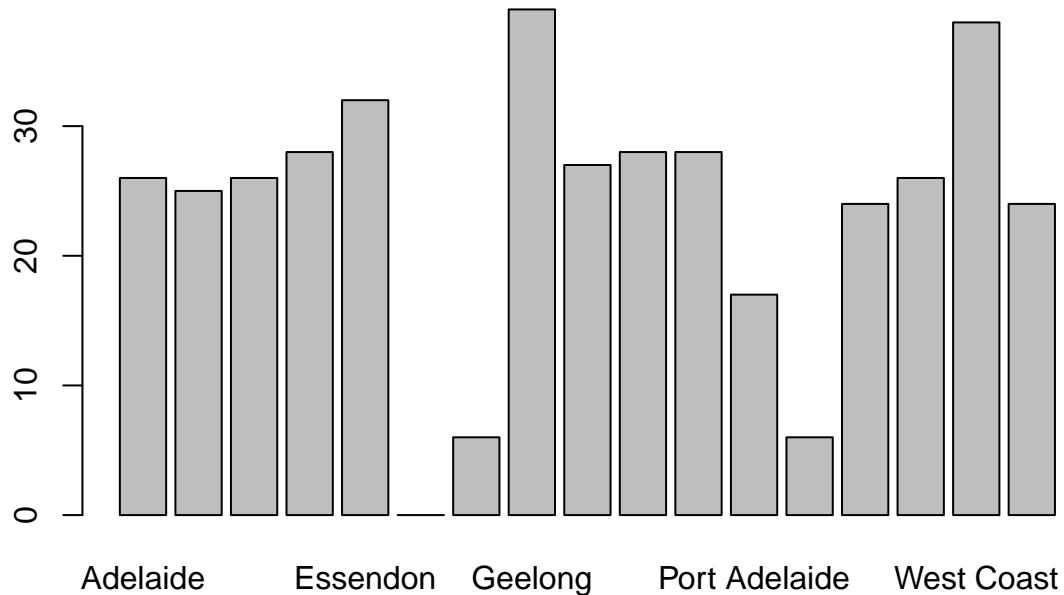


Figure 6.20: we've added the labels, but because the text runs horizontally R only includes a few of them

```
barplot( height = freq, names.arg = teams )
```

This is an improvement, but not much of an improvement. R has only included a few of the labels, because it can't fit them in the plot. This is the same behaviour we saw earlier with the multiple-boxplot graph in Figure ???. However, in Figure ??? it wasn't an issue: it's pretty obvious from inspection that the two unlabelled plots in between 1987 and 1990 must correspond to the data from 1988 and 1989. However, the fact that `barplot()` has omitted the names of every team in between Adelaide and Fitzroy is a lot more problematic.

The simplest way to fix this is to rotate the labels, so that the text runs vertically not horizontally. To do this, we need to alter set the `las` parameter, which I discussed briefly in Section ???. What I'll do is tell R to rotate the text so that it's always perpendicular to the axes (i.e., I'll set `las = 2`). When I do that, as per the following command...

```
barplot(height = freq, # the frequencies
        names.arg = teams, # the label
        las = 2)           # rotate the labels
```

... the result is the bar graph shown in Figure 6.21. We've fixed the problem, but we've created a new one: the axis labels don't quite fit anymore. To fix this, we have to be a bit cleverer again. A simple fix would be to use shorter names rather than the full name of all teams, and in many situations that's probably the right thing to do. However, at other times you really do need to create a bit more space to add your labels, so I'll show you how to do that.

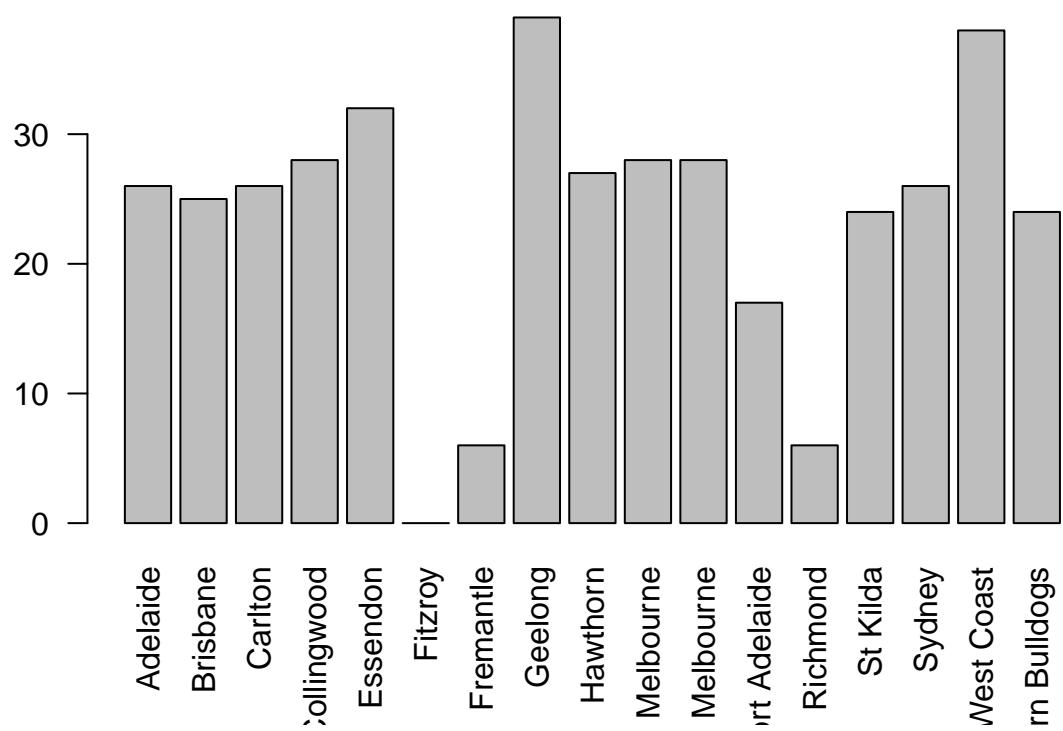


Figure 6.21: we've rotated the labels, but now the text is too long to fit

6.7.1 Changing global settings using `par()`

Altering the margins to the plot is actually a somewhat more complicated exercise than you might think. In principle it's a very simple thing to do: the size of the margins is governed by a graphical parameter called `mar`, so all we need to do is alter this parameter. First, let's look at what the `mar` argument specifies. The `mar` argument is a vector containing four numbers: specifying the amount of space at the bottom, the left, the top and then the right. The units are “number of lines”’. The default value `formarisc(5.1, 4.1, 4.1, 2.1)'`, meaning that R leaves 5.1”lines” empty at the bottom, 4.1 lines on the left and the bottom, and only 2.1 lines on the right. In order to make more room at the bottom, what I need to do is change the first of these numbers. A value of 10.1 should do the trick.

So far this doesn't seem any different to the other graphical parameters that we've talked about. However, because of the way that the traditional graphics system in R works, you need to specify what the margins will be *before* calling your high-level plotting function. Unlike the other cases we've seen, you can't treat `mar` as if it were just another argument in your plotting function. Instead, you have to use the `par()` function to change the graphical parameters beforehand, and only then try to draw your figure. In other words, the first thing I would do is this:

```
> par( mar = c( 10.1, 4.1, 4.1, 2.1) )
```

There's no visible output here, but behind the scenes R has changed the graphical parameters associated with the current device (remember, in R terminology all graphics are drawn onto a “device”). Now that this is done, we could use the exact same command as before, but this time you'd see that the labels all fit, because R now leaves twice as much room for the labels at the bottom. However, since I've now figured out how to get the labels to display properly, I might as well play around with some of the other options, all of which are things you've seen before:

```
barplot( height = freq,
         names.arg = teams,
         las=2,
         ylab = "Number of Finals",
         main = "Finals Played, 1987-2010",
         density = 10,
         angle = 20)
```

However, one thing to remember about the `par()` function is that it doesn't just change the graphical parameters for the current *plot*. Rather, the changes pertain to any subsequent plot that you draw onto the same *device*. This might be exactly what you want, in which case there's no problem. But if not, you need to reset the graphical parameters to their original settings. To do this, you can either close the device (e.g., close the window, or click the “Clear All” button in the Plots panel in Rstudio) or you can reset the graphical parameters to their original values, using a command like this:

```
> par( mar = c(5.1, 4.1, 4.1, 2.1) )
```

6.8 Saving image files using R and Rstudio

Hold on, you might be thinking. What's the good of being able to draw pretty pictures in R if I can't save them and send them to friends to brag about how awesome my data is? How do I save the picture? This is another one of those situations where the easiest thing to do is to use the RStudio tools.

If you're running R through Rstudio, then the easiest way to save your image is to click on the “Export” button in the Plot panel (i.e., the area in Rstudio where all the plots have been appearing). When you

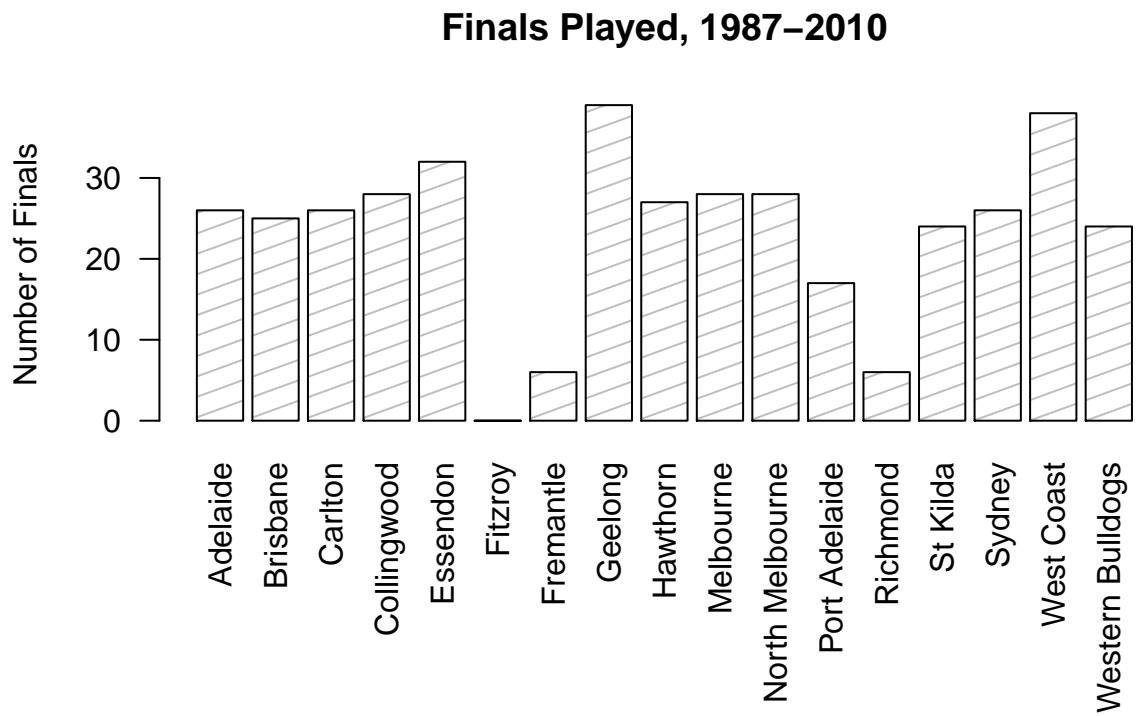


Figure 6.22: we fix this by expanding the margin at the bottom, and add several other customisations to make the chart a bit nicer

do that you'll see a menu that contains the options "Save Plot as PDF" and "Save Plot as Image". Either version works. Both will bring up dialog boxes that give you a few options that you can play with, but besides that it's pretty simple.

This works pretty nicely for most situations. So, unless you're filled with a burning desire to learn the low level details, feel free to skip the rest of this section.

6.8.1 The ugly details (advanced)

As I say, the menu-based options should be good enough for most people most of the time. However, one day you might want to be a bit more sophisticated, and make use of R's image writing capabilities at a lower level. In this section I'll give you a very basic introduction to this. In all honesty, this barely scratches the surface, but it will help a little bit in getting you started if you want to learn the details.

Okay, as I hinted earlier, whenever you're drawing pictures in R you're deemed to be drawing *to* a device of some kind. There are devices that correspond to a figure drawn on screen, and there are devices that correspond to graphics files that R will produce for you. For the purposes of this section I'll assume that you're using the default application in either Windows or Mac OS, not Rstudio. The reason for this is that my experience with the graphical device provided by Rstudio has led me to suspect that it still has a bunch on non-standard (or possibly just undocumented) features, and so I don't quite trust that it always does what I expect. I've no doubt they'll smooth it out later, but I can honestly say that I don't quite get what's going on with the `RStudioGD` device. In any case, we can ask R to list all of the graphics devices that currently exist, simply by using the command `dev.list()`. If there are no figure windows open, then you'll see this:

```
> dev.list()
NULL
```

which just means that R doesn't have any graphics devices open. However, suppose if you've just drawn a histogram and you type the same command, R will now give you a different answer. For instance, if you're using Windows:

```
> hist( afl.margins )
> dev.list()
windows
2
```

What this means is that there is one graphics device (device 2) that is currently open, and it's a figure window. If you did the same thing on a Mac, you get basically the same answer, except that the name of the device would be `quartz` rather than `windows`. If you had several graphics windows open (which, incidentally, you can do by using the `dev.new()` command) then you'd see something like this:

```
> dev.list()
windows windows windows
2      3      4
```

Okay, so that's the basic idea behind graphics devices. The key idea here is that graphics files (like JPEG images etc) are *also* graphics devices as far as R is concerned. So what you want to do is to *copy* the contents of one graphics device to another one. There's a command called `dev.copy()` that does this, but what I'll explain to you is a simpler one called `dev.print()`. It's pretty simple:

```
> dev.print( device = jpeg,           # what are we printing to?
+           filename = "thisfile.jpg", # name of the image file
+           width = 480,            # how many pixels wide should it be
+           height = 300            # how many pixels high should it be
+ )
```

This takes the “active” figure window, copies it to a jpeg file (which R treats as a device) and then closes that device. The `filename = "thisfile.jpg"` part tells R what to name the graphics file, and the `width = 480` and `height = 300` arguments tell R to draw an image that is 300 pixels high and 480 pixels wide. If you want a different kind of file, just change the device argument from `jpeg` to something else. R has devices for `png`, `tiff` and `bmp` that all work in exactly the same way as the `jpeg` command, but produce different kinds of files. Actually, for simple cartoonish graphics like this histogram, you’d be better advised to use PNG or TIFF over JPEG. The JPEG format is very good for natural images, but is wasteful for simple line drawings. The information above probably covers most things you might want to. However, if you want more information about what kinds of options you can specify using R, have a look at the help documentation by typing `?jpeg` or `?tiff` or whatever.

6.9 Summary

Perhaps I’m a simple minded person, but I love pictures. Every time I write a new scientific paper, one of the first things I do is sit down and think about what the pictures will be. In my head, an article is really just a sequence of pictures, linked together by a story. All the rest of it is just window dressing. What I’m really trying to say here is that the human visual system is a very powerful data analysis tool. Give it the right kind of information and it will supply a human reader with a massive amount of knowledge very quickly. Not for nothing do we have the saying “a picture is worth a thousand words”. With that in mind, I think that this is one of the most important chapters in the book. The topics covered were:

- *Basic overview to R graphics.* In Section 6.1 we talked about how graphics in R are organised, and then moved on to the basics of how they’re drawn in Section 6.2.
- *Common plots.* Much of the chapter was focused on standard graphs that statisticians like to produce: histograms (Section 6.3), stem and leaf plots (Section 6.4), boxplots (Section 6.5), scatterplots (Section 6.6) and bar graphs (Section 6.7).
- *Saving image files.* The last part of the chapter talked about how to export your pictures (Section 6.8)

One final thing to point out. At the start of the chapter I mentioned that R has several completely distinct systems for drawing figures. In this chapter I’ve focused on the *traditional* graphics system. It’s the easiest one to get started with: you can draw a histogram with a command as simple as `hist(x)`. However, it’s not the most powerful tool for the job, and after a while most R users start looking to shift to fancier systems. One of the most popular graphics systems is provided by the `ggplot2` package (see <http://ggplot2.org/>), which is loosely based on “The grammar of graphics” @[Wilkinson2006]. It’s not for novices: you need to have a pretty good grasp of R before you can start using it, and even then it takes a while to really get the hang of it. But when you’re finally at that stage, it’s worth taking the time to teach yourself, because it’s a much cleaner system.

Chapter 7

Pragmatic matters

The garden of life never seems to confine itself to the plots philosophers have laid out for its convenience. Maybe a few more tractors would do the trick.

–Roger Zelazny¹

This is a somewhat strange chapter, even by my standards. My goal in this chapter is to talk a bit more honestly about the realities of working with data than you'll see anywhere else in the book. The problem with real world data sets is that they are *messy*. Very often the data file that you start out with doesn't have the variables stored in the right format for the analysis you want to do. Sometimes might be a lot of missing values in your data set. Sometimes you only want to analyse a subset of the data. Et cetera. In other words, there's a lot of **data manipulation** that you need to do, just to get all your data set into the format that you need it. The purpose of this chapter is to provide a basic introduction to all these pragmatic topics. Although the chapter is motivated by the kinds of practical issues that arise when manipulating real data, I'll stick with the practice that I've adopted through most of the book and rely on very small, toy data sets that illustrate the underlying issue. Because this chapter is essentially a collection of "tricks" and doesn't tell a single coherent story, it may be useful to start with a list of topics:

- Section 7.1. Tabulating data.
- Section 7.2. Transforming or recoding a variable.
- Section 7.3. Some useful mathematical functions.
- Section 7.4. Extracting a subset of a vector.
- Section 7.5. Extracting a subset of a data frame.
- Section 7.6. Sorting, flipping or merging data sets.
- Section 7.7. Reshaping a data frame.
- Section 7.8. Manipulating text.
- Section 7.9. Opening data from different file types.
- Section 7.10. Coercing data from one type to another.
- Section 7.11. Other important data types.
- Section 7.12. Miscellaneous topics.

As you can see, the list of topics that the chapter covers is pretty broad, and there's a *lot* of content there. Even though this is one of the longest and hardest chapters in the book, I'm really only scratching the surface of several fairly different and important topics. My advice, as usual, is to read through the chapter once and try to follow as much of it as you can. Don't worry too much if you can't grasp it all at once, especially the later sections. The rest of the book is only lightly reliant on this chapter, so you can get away with just understanding the basics. However, what you'll probably find is that later on you'll need to flick back to this chapter in order to understand some of the concepts that I refer to here.

¹The quote comes from *Home is the Hangman*, published in 1975.

7.1 Tabulating and cross-tabulating data

A very common task when analysing data is the construction of frequency tables, or cross-tabulation of one variable against another. There are several functions that you can use in R for that purpose. In this section I'll illustrate the use of three functions – `table()`, `xtabs()` and `tabulate()` – though there are other options (e.g., `ftable()`) available.

7.1.1 Creating tables from vectors

Let's start with a simple example. As the father of a small child, I naturally spend a lot of time watching TV shows like *In the Night Garden*. In the `nightgarden.Rdata` file, I've transcribed a short section of the dialogue. The file contains two variables, `speaker` and `utterance`, and when we take a look at the data, it becomes very clear what happened to my sanity.

```
library(lsr)
load("./rbook-master/data/nightgarden.Rdata")
who()

##   -- Name --      -- Class --   -- Size --
##   afl.finalists    factor      400
##   afl.margins     numeric     176
##   afl.margins_out numeric     176
##   afl2            data.frame 4296 x 2
##   colour          logical      1
##   d.cor           numeric      1
##   describeImg     list         0
##   effort          data.frame 10 x 2
##   emphCol         character    1
##   emphColLight    character    1
##   emphGrey        character    1
##   eps             logical      1
##   Fibonacci       numeric      7
##   freq            integer     17
##   height          numeric      1
##   old              list        66
##   oneCorPlot      function
##   out.0           data.frame 100 x 2
##   out.1           data.frame 100 x 2
##   out.2           data.frame 100 x 2
##   parenthood      data.frame 100 x 4
##   plotOne         function
##   speaker          character    10
##   teams            character    17
##   utterance        character    10
##   width            numeric      1
##   X1               numeric     11
##   X2               numeric     11
##   X3               numeric     11
##   X4               numeric     11
##   Y1               numeric     11
##   Y2               numeric     11
##   Y3               numeric     11
##   Y4               numeric     11
```

```
print( speaker )

## [1] "upsy-daisy"  "upsy-daisy"  "upsy-daisy"  "upsy-daisy"  "tombliboo"
## [6] "tombliboo"   "makka-pakka" "makka-pakka" "makka-pakka" "makka-pakka"

print( utterance )

## [1] "pip" "pip" "onk" "onk" "ee"   "oo"   "pip" "pip" "onk" "onk"
```

With these as my data, one task I might find myself needing to do is construct a frequency count of the number of words each character speaks during the show. The `table()` function provides a simple way to do this. The basic usage of the `table()` function is as follows:

```
table(speaker)

## speaker
## makka-pakka    tombliboo  upsy-daisy
##        4           2           4
```

The output here tells us on the first line that what we're looking at is a tabulation of the `speaker` variable. On the second line it lists all the different speakers that exist in the data, and on the third line it tells you how many times that speaker appears in the data. In other words, it's a frequency table² Notice that in the command above I didn't name the argument, since `table()` is another function that makes use of unnamed arguments. You just type in a list of the variables that you want R to tabulate, and it tabulates them. For instance, if I type in the name of two variables, what I get as the output is a cross-tabulation:

```
table(speaker, utterance)

##          utterance
## speaker      ee onk oo pip
##   makka-pakka 0  2  0  2
##   tombliboo   1  0  1  0
##   upsy-daisy  0  2  0  2
```

When interpreting this table, remember that these are counts: so the fact that the first row and second column corresponds to a value of 2 indicates that Makka-Pakka (row 1) says “onk” (column 2) twice in this data set. As you'd expect, you can produce three way or higher order cross tabulations just by adding more objects to the list of inputs. However, I won't discuss that in this section.

7.1.2 Creating tables from data frames

Most of the time your data are stored in a data frame, not kept as separate variables in the workspace. Let's create one:

²As usual, you can assign this output to a variable. If you type `speaker.freq <- table(speaker)` at the command prompt R will store the table as a variable. If you then type `class(speaker.freq)` you'll see that the output is actually of class `table`. The key thing to note about a table object is that it's basically a matrix (see Section 7.11.1).

```
itng <- data.frame( speaker, utterance )
itng
```

```
##           speaker utterance
## 1    upsy-daisy      pip
## 2    upsy-daisy      pip
## 3    upsy-daisy     onk
## 4    upsy-daisy     onk
## 5   tombliboo       ee
## 6   tombliboo       oo
## 7 makka-pakka      pip
## 8 makka-pakka      pip
## 9 makka-pakka     onk
## 10 makka-pakka    onk
```

There's a couple of options under these circumstances. Firstly, if you just want to cross-tabulate all of the variables in the data frame, then it's really easy:

```
table(itng)
```

```
##           utterance
## speaker      ee onk oo pip
## makka-pakka 0   2   0   2
## tombliboo   1   0   1   0
## upsy-daisy  0   2   0   2
```

However, it's often the case that you want to select particular variables from the data frame to tabulate. This is where the `xtabs()` function is useful. In this function, you input a one sided `formula` in order to list all the variables you want to cross-tabulate, and the name of the `data` frame that stores the data:

```
xtabs( formula = ~ speaker + utterance, data = itng )
```

```
##           utterance
## speaker      ee onk oo pip
## makka-pakka 0   2   0   2
## tombliboo   1   0   1   0
## upsy-daisy  0   2   0   2
```

Clearly, this is a totally unnecessary command in the context of the `itng` data frame, but in most situations when you're analysing real data this is actually extremely useful, since your data set will almost certainly contain lots of variables and you'll only want to tabulate a few of them at a time.

7.1.3 Converting a table of counts to a table of proportions

The tabulation commands discussed so far all construct a table of raw frequencies: that is, a count of the total number of cases that satisfy certain conditions. However, often you want your data to be organised in terms of proportions rather than counts. This is where the `prop.table()` function comes in handy. It has two arguments:

- `x`. The frequency table that you want to convert.

- `margin`. Which “dimension” do you want to calculate proportions for. By default, R assumes you want the proportion to be expressed as a fraction of all possible events. See examples for details.

To see how this works:

```
itng.table <- table(itng) # create the table, and assign it to a variable
itng.table # display the table again, as a reminder
```

```
##          utterance
## speaker      ee onk oo pip
##   makka-pakka 0  2  0  2
##   tombliboo   1  0  1  0
##   upsy-daisy  0  2  0  2

prop.table( x = itng.table ) # express as proportion:

##          utterance
## speaker      ee onk oo pip
##   makka-pakka 0.0 0.2 0.0 0.2
##   tombliboo   0.1 0.0 0.1 0.0
##   upsy-daisy  0.0 0.2 0.0 0.2
```

Notice that there were 10 observations in our original data set, so all that R has done here is divide all our raw frequencies by 10. That’s a sensible default, but more often you actually want to calculate the proportions separately by row (`margin = 1`) or by column (`margin = 2`). Again, this is most clearly seen by looking at examples:

```
prop.table( x = itng.table, margin = 1)
```

```
##          utterance
## speaker      ee onk oo pip
##   makka-pakka 0.0 0.5 0.0 0.5
##   tombliboo   0.5 0.0 0.5 0.0
##   upsy-daisy  0.0 0.5 0.0 0.5
```

Notice that each row now sums to 1, but that’s not true for each column. What we’re looking at here is the proportions of utterances made by each character. In other words, 50% of Makka-Pakka’s utterances are “pip”, and the other 50% are “onk”. Let’s contrast this with the following command:

```
prop.table( x = itng.table, margin = 2)
```

```
##          utterance
## speaker      ee onk oo pip
##   makka-pakka 0.0 0.5 0.0 0.5
##   tombliboo   1.0 0.0 1.0 0.0
##   upsy-daisy  0.0 0.5 0.0 0.5
```

Now the columns all sum to 1 but the rows don’t. In this version, what we’re seeing is the proportion of characters associated with each utterance. For instance, whenever the utterance “ee” is made (in this data set), 100% of the time it’s a Tombliboo saying it.

7.1.4 Low level tabulation

One final function I want to mention is the `tabulate()` function, since this is actually the low-level function that does most of the hard work. It takes a numeric vector as input, and outputs frequencies as outputs:

```
some.data <- c(1,2,3,1,1,3,1,1,2,8,3,1,2,4,2,3,5,2)
tabulate(some.data)

## [1] 6 5 4 1 1 0 0 1
```

7.2 Transforming and recoding a variable

It's not uncommon in real world data analysis to find that one of your variables isn't quite equivalent to the variable that you really want. For instance, it's often convenient to take a continuous-valued variable (e.g., age) and break it up into a smallish number of categories (e.g., younger, middle, older). At other times, you may need to convert a numeric variable into a different numeric variable (e.g., you may want to analyse at the absolute value of the original variable). In this section I'll describe a few key tricks that you can make use of to do this.

7.2.1 Creating a transformed variable

The first trick to discuss is the idea of *transforming* a variable. Taken literally, *anything* you do to a variable is a transformation, but in practice what it usually means is that you apply a relatively simple mathematical function to the original variable, in order to create new variable that either (a) provides a better way of describing the thing you're actually interested in or (b) is more closely in agreement with the assumptions of the statistical tests you want to do. Since – at this stage – I haven't talked about statistical tests or their assumptions, I'll show you an example based on the first case.

To keep the explanation simple, the variable we'll try to transform (`likert.raw`) isn't inside a data frame, though in real life it almost certainly would be. However, I think it's useful to start with an example that doesn't use data frames because it illustrates the fact that you already know how to do variable transformations. To see this, let's go through an example. Suppose I've run a short study in which I ask 10 people a single question:

On a scale of 1 (strongly disagree) to 7 (strongly agree), to what extent do you agree with the proposition that "Dinosaurs are awesome"?

Now let's load and look at the data. The data file `likert.Rdata` contains a single variable that contains the raw Likert-scale responses:

```
load("./rbook-master/data/likert.Rdata")
likert.raw

## [1] 1 7 3 4 4 4 2 6 5 5
```

However, if you think about it, this isn't the best way to represent these responses. Because of the fairly symmetric way that we set up the response scale, there's a sense in which the midpoint of the scale should have been coded as 0 (no opinion), and the two endpoints should be +3 (strong agree) and -3 (strong disagree). By recoding the data in this way, it's a bit more reflective of how we really think about the responses. The recoding here is trivially easy: we just subtract 4 from the raw scores:

```
likert centred <- likert raw - 4
likert centred
```

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

One reason why it might be useful to have the data in this format is that there are a lot of situations where you might prefer to analyse the *strength* of the opinion separately from the *direction* of the opinion. We can do two different transformations on this `likert.centred` variable in order to distinguish between these two different concepts. Firstly, to compute an `opinion.strength` variable, we want to take the absolute value of the centred data (using the `abs()` function that we've seen previously), like so:

```
opinion.strength <- abs( likert.centred )
opinion.strength
```

```
## [1] 3 3 1 0 0 0 2 2 1 1
```

Secondly, to compute a variable that contains only the direction of the opinion and ignores the strength, we can use the `sign()` function to do this. If you type `?sign` you'll see that this function is really simple: all negative numbers are converted to -1 , all positive numbers are converted to 1 and zero stays as 0 . So, when we apply the `sign()` function we obtain the following:

```
opinion.dir <- sign( likert.centred )
opinion.dir
```

```
## [1] -1 1 -1 0 0 0 -1 1 1 1
```

And we're done. We now have three shiny new variables, all of which are useful transformations of the original `likert.raw` data. All of this should seem pretty familiar to you. The tools that you use to do regular calculations in R (e.g., Chapters 3 and 4) are very much the same ones that you use to transform your variables! To that end, in Section 7.3 I'll revisit the topic of doing calculations in R because there's a lot of other functions and operations that are worth knowing about.

Before moving on, you might be curious to see what these calculations look like if the data had started out in a data frame. To that end, it may help to note that the following example does all of the calculations using variables inside a data frame, and stores the variables created inside it:

```
df <- data.frame( likert.raw )                      # create data frame
df$likert.centred <- df$likert.raw - 4            # create centred data
df$opinion.strength <- abs( df$likert.centred )    # create strength variable
df$opinion.dir <- sign( df$likert.centred )        # create direction variable
df                                         # print the final data frame:
```

	likert.raw	likert.centred	opinion.strength	opinion.dir
## 1	1	-3	3	-1
## 2	7	3	3	1
## 3	3	-1	1	-1
## 4	4	0	0	0
## 5	4	0	0	0
## 6	4	0	0	0
## 7	2	-2	2	-1
## 8	6	2	2	1
## 9	5	1	1	1
## 10	5	1	1	1

In other words, the commands you use are basically ones as before: it's just that every time you want to read a variable from the data frame or write to the data frame, you use the `$` operator. That's the easiest way to do it, though I should make note of the fact that people sometimes make use of the `within()` function to do the same thing. However, since (a) I don't use the `within()` function anywhere else in this book, and (b) the `$` operator works just fine, I won't discuss it any further.

7.2.2 Cutting a numeric variable into categories

One pragmatic task that arises more often than you'd think is the problem of cutting a numeric variable up into discrete categories. For instance, suppose I'm interested in looking at the age distribution of people at a social gathering:

```
age <- c( 60, 58, 24, 26, 34, 42, 31, 30, 33, 2, 9 )
```

In some situations it can be quite helpful to group these into a smallish number of categories. For example, we could group the data into three broad categories: young (0-20), adult (21-40) and older (41-60). This is a quite coarse-grained classification, and the labels that I've attached only make sense in the context of this data set (e.g., viewed more generally, a 42 year old wouldn't consider themselves as "older"). We can slice this variable up quite easily using the `cut()` function.³ To make things a little cleaner, I'll start by creating a variable that defines the boundaries for the categories:

```
age.breaks <- seq( from = 0, to = 60, by = 20 )
age.breaks
```

```
## [1] 0 20 40 60
```

and another one for the labels:

```
age.labels <- c( "young", "adult", "older" )
age.labels
```

```
## [1] "young" "adult" "older"
```

Note that there are four numbers in the `age.breaks` variable, but only three labels in the `age.labels` variable; I've done this because the `cut()` function requires that you specify the *edges* of the categories rather than the mid-points. In any case, now that we've done this, we can use the `cut()` function to assign each observation to one of these three categories. There are several arguments to the `cut()` function, but the three that we need to care about are:

- `x`. The variable that needs to be categorised.
- `breaks`. This is either a vector containing the locations of the breaks separating the categories, or a number indicating how many categories you want.
- `labels`. The labels attached to the categories. This is optional: if you don't specify this R will attach a boring label showing the range associated with each category.

Since we've already created variables corresponding to the breaks and the labels, the command we need is just:

³It's worth noting that there's also a more powerful function called `recode()` function in the `car` package that I won't discuss in this book but is worth looking into if you're looking for a bit more flexibility.

```
age.group <- cut( x = age,                      # the variable to be categorised
                  breaks = age.breaks,    # the edges of the categories
                  labels = age.labels ) # the labels for the categories
```

Note that the output variable here is a factor. In order to see what this command has actually done, we could just print out the `age.group` variable, but I think it's actually more helpful to create a data frame that includes both the original variable and the categorised one, so that you can see the two side by side:

```
data.frame(age, age.group)
```

```
##   age age.group
## 1  60    older
## 2  58    older
## 3  24   adult
## 4  26   adult
## 5  34   adult
## 6  42    older
## 7  31   adult
## 8  30   adult
## 9  33   adult
## 10  2    young
## 11  9    young
```

It can also be useful to tabulate the output, just to see if you've got a nice even division of the sample:

```
table( age.group )
```

```
## age.group
## young adult older
##      2     6     3
```

In the example above, I made all the decisions myself. Much like the `hist()` function that we saw in Chapter 6, if you want to you can delegate a lot of the choices to R. For instance, if you want you can specify the *number* of categories you want, rather than giving explicit ranges for them, and you can allow R to come up with some labels for the categories. To give you a sense of how this works, have a look at the following example:

```
age.group2 <- cut( x = age, breaks = 3 )
```

With this command, I've asked for three categories, but let R make the choices for where the boundaries should be. I won't bother to print out the `age.group2` variable, because it's not terribly pretty or very interesting. Instead, all of the important information can be extracted by looking at the tabulated data:

```
table( age.group2 )
```

```
## age.group2
## (1.94,21.3] (21.3,40.7] (40.7,60.1]
##      2         6         3
```

This output takes a little bit of interpretation, but it's not complicated. What R has done is determined that the lowest age category should run from 1.94 years up to 21.3 years, the second category should run from 21.3 years to 40.7 years, and so on. The formatting on those labels might look a bit funny to those of you who haven't studied a lot of maths, but it's pretty simple. When R describes the first category as corresponding to the range (1.94, 21.3] what it's saying is that the range consists of those numbers that are larger than 1.94 but less than *or equal to* 21.3. In other words, the weird asymmetric brackets is R's way of telling you that if there happens to be a value that is exactly equal to 21.3, then it belongs to the first category, not the second one. Obviously, this isn't actually possible since I've only specified the ages to the nearest whole number, but R doesn't know this and so it's trying to be precise just in case. This notation is actually pretty standard, but I suspect not everyone reading the book will have seen it before. In any case, those labels are pretty ugly, so it's usually a good idea to specify your own, meaningful labels to the categories.

Before moving on, I should take a moment to talk a little about the mechanics of the `cut()` function. Notice that R has tried to divide the `age` variable into three roughly equal sized bins. Unless you specify the particular breaks you want, that's what it will do. But suppose you want to divide the `age` variable into three categories of different size, but with approximately identical numbers of people. How would you do that? Well, if that's the case, then what you want to do is have the breaks correspond to the 0th, 33rd, 66th and 100th percentiles of the data. One way to do this would be to calculate those values using the `quantiles()` function and then use those quantiles as input to the `cut()` function. That's pretty easy to do, but it does take a couple of lines to type. So instead, the `lsr` package has a function called `quantileCut()` that does exactly this:

```
age.group3 <- quantileCut( x = age, n = 3 )
table( age.group3 )
```

```
## age.group3
## (1.94,27.3] (27.3,33.7] (33.7,60.1]
##          4            3            4
```

Notice the difference in the boundaries that the `quantileCut()` function selects. The first and third categories now span an age range of about 25 years each, whereas the middle category has shrunk to a span of only 6 years. There are some situations where this is genuinely what you want (that's why I wrote the function!), but in general you should be careful. Usually the numeric variable that you're trying to cut into categories is already expressed in meaningful units (i.e., it's interval scale), but if you cut it into unequal bin sizes then it's often very difficult to attach meaningful interpretations to the resulting categories.

More generally, regardless of whether you're using the original `cut()` function or the `quantileCut()` version, it's important to take the time to figure out whether or not the resulting categories make any sense at all in terms of your research project. If they don't make any sense to you as meaningful categories, then any data analysis that uses those categories is likely to be just as meaningless. More generally, in practice I've noticed that people have a very strong desire to carve their (continuous and messy) data into a few (discrete and simple) categories; and then run analysis using the categorised data instead of the original one.⁴ I wouldn't go so far as to say that this is an inherently bad idea, but it does have some fairly serious drawbacks at times, so I would advise some caution if you are thinking about doing it.

⁴If you've read further into the book, and are re-reading this section, then a good example of this would be someone choosing to do an ANOVA using `age.group3` as the grouping variable, instead of running a regression using `age` as a predictor. There are sometimes good reasons for do this: for instance, if the relationship between `age` and your outcome variable is highly non-linear, and you aren't comfortable with trying to run non-linear regression! However, unless you really do have a good rationale for doing this, it's best not to. It tends to introduce all sorts of other problems (e.g., the data will probably violate the normality assumption), and you can lose a lot of power.

Table 7.1: Some of the mathematical functions available in R.

mathematical.function	R.function	example.input	answer
square root	<code>sqrt()</code>	<code>sqrt(25)</code>	5
absolute value	<code>abs()</code>	<code>abs(-23)</code>	23
logarithm (base 10)	<code>log10()</code>	<code>log10(1000)</code>	3
logarithm (base e)	<code>log()</code>	<code>log(1000)</code>	6.908
exponentiation	<code>exp()</code>	<code>exp(6.908)</code>	1000.245
rounding to nearest	<code>round()</code>	<code>round(1.32)</code>	1
rounding down	<code>floor()</code>	<code>floor(1.32)</code>	1
rounding up	<code>ceiling()</code>	<code>ceiling(1.32)</code>	2

7.3 A few more mathematical functions and operations

In Section 7.2 I discussed the ideas behind variable transformations, and showed that a lot of the transformations that you might want to apply to your data are based on fairly simple mathematical functions and operations, of the kind that we discussed in Chapter 3. In this section I want to return to that discussion, and mention several other mathematical functions and arithmetic operations that I didn't bother to mention when introducing you to R, but are actually quite useful for a lot of real world data analysis. Table 7.1 gives a brief overview of the various mathematical functions I want to talk about (and some that I already have talked about). Obviously this doesn't even come close to cataloging the range of possibilities available in R, but it does cover a very wide range of functions that are used in day to day data analysis.

7.3.1 Rounding a number

One very simple transformation that crops up surprisingly often is the need to round a number to the nearest whole number, or to a certain number of significant digits. To start with, let's assume that we want to round to a whole number. To that end, there are three useful functions in R you want to know about: `round()`, `floor()` and `ceiling()`. The `round()` function just rounds to the *nearest* whole number. So if you round the number 4.3, it "rounds down" to 4, like so:

```
round( x = 4.3 )
```

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

In contrast, if we want to round the number 4.7, we would round upwards to 5. In everyday life, when someone talks about "rounding", they usually mean "round to nearest", so this is the function we use most of the time. However sometimes you have reasons to want to always round up or always round down. If you want to always round down, use the `floor()` function instead; and if you want to force R to round up, then use `ceiling()`. That's the only difference between the three functions. What if you want to round to a certain number of digits? Let's suppose you want to round to a fixed number of decimal places, say 2 decimal places. If so, what you need to do is specify the `digits` argument to the `round()` function. That's pretty straightforward:

```
round( x = 0.0123, digits = 2 )
```

```
## [1] 0.01
```

The only subtlety that you need to keep in mind is that sometimes what you want to do is round to 2 *significant digits* and not to two decimal places. The difference is that, when determining the number of

Table 7.2: Two more arithmetic operations that sometimes come in handy

operation	operator	example.input	answer
integer division	%/%	42 %/% 10	4
modulus	%%	42 %% 10	2

significant digits, zeros don't count. To see this, let's apply the `signif()` function instead of the `round()` function:

```
signif( x = 0.0123, digits = 2 )
```

```
## [1] 0.012
```

This time around, we get an answer of 0.012 because the zeros don't count as significant digits. Quite often scientific journals will ask you to report numbers to two or three significant digits, so it's useful to remember the distinction.

7.3.2 Modulus and integer division

Since we're on the topic of simple calculations, there are two other arithmetic operations that I should mention, since they can come in handy when working with real data. These operations are calculating a modulus and doing integer division. They don't come up anywhere else in this book, but they are worth knowing about. First, let's consider *integer division*. Suppose I have \$42 in my wallet, and want to buy some sandwiches, which are selling for \$10 each. How many sandwiches can I afford⁵ to buy? The answer is of course 4. Note that it's not 4.2, since no shop will sell me one-fifth of a sandwich. That's integer division. In R we perform integer division by using the `%/%` operator:

```
42 %/% 10
```

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

Okay, that's easy enough. What about the *modulus*? Basically, a modulus is the remainder after integer division, and it's calculated using the `%%` operator. For the sake of argument, let's suppose I buy four overpriced \$10 sandwiches. If I started out with \$42, how much money do I have left? The answer, as both R and common sense tells us, is \$2:

```
42 %% 10
```

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

So that's also pretty easy. There is, however, one subtlety that I need to mention, and this relates to how negative numbers are handled. Firstly, what would happen if I tried to do integer division with a negative number? Let's have a look:

```
-42 %/% 10
```

```
## [1] -5
```

⁵The real answer is 0: \$10 for a sandwich is a total ripoff so I should go next door and buy noodles.

This might strike you as counterintuitive: why does `42 %% 10` produce an answer of 4, but `-42 %% 10` gives us an answer of -5? Intuitively you might think that the answer to the second one should be -4. The way to think about it is like this. Suppose I *owe* the sandwich shop \$42, but I don't have any money. How many sandwiches would *I* have to give *them* in order to stop them from calling security? The answer⁶ here is 5, not 4. If I handed them 4 sandwiches, I'd still owe them \$2, right? So I actually have to give them 5 sandwiches. And since it's *me* giving them the sandwiches, the answer to `-42 %% 10` is -5. As you might expect, the behaviour of the modulus operator has a similar pattern. If I've handed 5 sandwiches over to the shop in order to pay off my debt of \$42, then *they* now owe me \$8. So the modulus is now:

```
-42 %% 10
```

```
## [1] 8
```

7.3.3 Logarithms and exponentials

As I've mentioned earlier, R has an incredible range of mathematical functions built into it, and there really wouldn't be much point in trying to describe or even list all of them. For the most part, I've focused only on those functions that are strictly necessary for this book. However I do want to make an exception for logarithms and exponentials. Although they aren't needed anywhere else in this book, they are *everywhere* in statistics more broadly, and not only that, there are a *lot* of situations in which it is convenient to analyse the logarithm of a variable (i.e., to take a "log-transform" of the variable). I suspect that many (maybe most) readers of this book will have encountered logarithms and exponentials before, but from past experience I know that there's a substantial proportion of students who take a social science statistics class who haven't touched logarithms since high school, and would appreciate a bit of a refresher.

In order to understand logarithms and exponentials, the easiest thing to do is to actually calculate them and see how they relate to other simple calculations. There are three R functions in particular that I want to talk about, namely `log()`, `log10()` and `exp()`. To start with, let's consider `log10()`, which is known as the "logarithm in base 10". The trick to understanding a **logarithm** is to understand that it's basically the "opposite" of taking a power. Specifically, the logarithm in base 10 is closely related to the powers of 10. So let's start by noting that 10-cubed is 1000. Mathematically, we would write this:

$$10^3 = 1000$$

and in R we'd calculate it by using the command `10^3`. The trick to understanding a logarithm is to recognise that the statement that "10 to the power of 3 is equal to 1000" is equivalent to the statement that "the logarithm (in base 10) of 1000 is equal to 3". Mathematically, we write this as follows,

$$\log_{10}(1000) = 3$$

and if we wanted to do the calculation in R we would type this:

```
log10( 1000 )
```

```
## [1] 3
```

Obviously, since you already know that $10^3 = 1000$ there's really no point in getting R to tell you that the base-10 logarithm of 1000 is 3. However, most of the time you probably don't know what right answer is. For instance, I can honestly say that I didn't know that $10^{2.69897} = 500$, so it's rather convenient for me that I can use R to calculate the base-10 logarithm of 500:

⁶Again, I doubt that's the right "real world" answer. I suspect that most sandwich shops won't allow you to pay off your debts to them in sandwiches. But you get the idea.

```
log10( 500 )
```

```
## [1] 2.69897
```

Or at least it would be convenient if I had a pressing need to know the base-10 logarithm of 500.

Okay, since the `log10()` function is related to the powers of 10, you might expect that there are other logarithms (in bases other than 10) that are related to other powers too. And of course that's true: there's not really anything mathematically special about the number 10. You and I happen to find it useful because decimal numbers are built around the number 10, but the big bad world of mathematics scoffs at our decimal numbers. Sadly, the universe doesn't actually care how we write down numbers. Anyway, the consequence of this cosmic indifference is that there's nothing particularly special about calculating logarithms in base 10. You could, for instance, calculate your logarithms in base 2, and in fact R does provide a function for doing that, which is (not surprisingly) called `log2()`. Since we know that $2^3 = 2 \times 2 \times 2 = 8$, it's not surprising to see that

```
log2( 8 )
```

```
## [1] 3
```

Alternatively, a third type of logarithm – and one we see a lot more of in statistics than either base 10 or base 2 – is called the *natural logarithm*, and corresponds to the logarithm in base e . Since you might one day run into it, I'd better explain what e is. The number e , known as *Euler's number*, is one of those annoying “irrational” numbers whose decimal expansion is infinitely long, and is considered one of the most important numbers in mathematics. The first few digits of e are:

$$e = 2.718282$$

There are quite a few situations in statistics that require us to calculate powers of e , though none of them appear in this book. Raising e to the power x is called the *exponential* of x , and so it's very common to see e^x written as `exp(x)`. And so it's no surprise that R has a function that calculates exponentials, called `exp()`. For instance, suppose I wanted to calculate e^3 . I could try typing in the value of e manually, like this:

```
2.718282 ^ 3
```

```
## [1] 20.08554
```

but it's much easier to do the same thing using the `exp()` function:

```
exp( 3 )
```

```
## [1] 20.08554
```

Anyway, because the number e crops up so often in statistics, the natural logarithm (i.e., logarithm in base e) also tends to turn up. Mathematicians often write it as $\log_e(x)$ or $\ln(x)$, or sometimes even just $\log(x)$. In fact, R works the same way: the `log()` function corresponds to the natural logarithm⁷ Anyway, as a quick check, let's calculate the natural logarithm of 20.08554 using R:

⁷Actually, that's a bit of a lie: the `log()` function is more flexible than that, and can be used to calculate logarithms in *any* base. The `log()` function has a `base` argument that you can specify, which has a default value of e . Thus `log10(1000)` is actually equivalent to `log(x = 1000, base = 10)`.

```
log( 20.08554 )
```

```
## [1] 3
```

And with that, I think we've had quite enough exponentials and logarithms for this book!

7.4 Extracting a subset of a vector

One very important kind of data handling is being able to extract a particular subset of the data. For instance, you might be interested only in analysing the data from one experimental condition, or you may want to look closely at the data from people over 50 years in age. To do this, the first step is getting R to extract the subset of the data corresponding to the observations that you're interested in. In this section I'll talk about subsetting as it applies to vectors, extending the discussion from Chapters 3 and 4. In Section 7.5 I'll go on to talk about how this discussion extends to data frames.

7.4.1 Refresher

This section returns to the `nightgarden.Rdata` data set. If you're reading this whole chapter in one sitting, then you should already have this data set loaded. If not, don't forget to use the `load("nightgarden.Rdata")` command. For this section, let's ignore the `itng` data frame that we created earlier, and focus instead on the two vectors `speaker` and `utterance` (see Section 7.1 if you've forgotten what those vectors look like). Suppose that what I want to do is pull out only those utterances that were made by Makka-Pakka. To that end, I could first use the equality operator to have R tell me which cases correspond to Makka-Pakka speaking:

```
is.MP.speaking <- speaker == "makka-pakka"
is.MP.speaking
```

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

and then use logical indexing to get R to print out those elements of `utterance` for which `is.MP.speaking` is true, like so:

```
utterance[ is.MP.speaking ]
```

```
## [1] "pip" "pip" "onk" "onk"
```

Or, since I'm lazy, I could collapse it to a single command like so:

```
utterance[ speaker == "makka-pakka" ]
```

```
## [1] "pip" "pip" "onk" "onk"
```

7.4.2 Using %in% to match multiple cases

A second useful trick to be aware of is the `%in%` operator⁸. It's actually very similar to the `==` operator, except that you can supply a collection of acceptable values. For instance, suppose I wanted to keep only those cases when the utterance is either "pip" or "oo". One simple way do to this is:

```
utterance %in% c("pip", "oo")
## [1] TRUE TRUE FALSE FALSE TRUE TRUE TRUE FALSE FALSE
```

What this does if return `TRUE` for those elements of `utterance` that are either "pip" or "oo" and returns `FALSE` for all the others. What that means is that if I want a list of all those instances of characters speaking either of these two words, I could do this:

```
speaker[ utterance %in% c("pip", "oo") ]
## [1] "upsy-daisy"  "upsy-daisy"  "tombliboo"   "makka-pakka" "makka-pakka"
```

7.4.3 Using negative indices to drop elements

Before moving onto data frames, there's a couple of other tricks worth mentioning. The first of these is to use negative values as indices. Recall from Section 3.10 that we can use a vector of numbers to extract a set of elements that we would like to keep. For instance, suppose I want to keep only elements 2 and 3 from `utterance`. I could do so like this:

```
utterance[2:3]
## [1] "pip" "onk"
```

But suppose, on the other hand, that I have discovered that observations 2 and 3 are untrustworthy, and I want to keep everything *except* those two elements. To that end, R lets you use negative numbers to remove specific values, like so:

```
utterance[ -(2:3) ]
## [1] "pip" "onk" "ee"  "oo"  "pip" "pip" "onk" "onk"
```

The output here corresponds to element 1 of the original vector, followed by elements 4, 5, and so on. When all you want to do is remove a few cases, this is a very handy convention.

7.4.4 Splitting a vector by group

One particular example of subsetting that is especially common is the problem of splitting one variable up into several different variables, one corresponding to each group. For instance, in our *In the Night Garden* example, I might want to create subsets of the `utterance` variable for every character. One way to do this would be to just repeat the exercise that I went through earlier separately for each character, but that quickly gets annoying. A faster way do it is to use the `split()` function. The arguments are:

⁸It's also worth checking out the `match()` function

- `x`. The variable that needs to be split into groups.
- `f`. The grouping variable.

What this function does is output a list (Section 4.9), containing one variable for each group. For instance, I could split up the `utterance` variable by `speaker` using the following command:

```
speech.by.char <- split( x = utterance, f = speaker )
speech.by.char
```

```
## `$`makka-pakka`
## [1] "pip" "pip" "onk" "onk"
##
## $tomboliboo
## [1] "ee" "oo"
##
## `$`upsy-daisy`
## [1] "pip" "pip" "onk" "onk"
```

Once you're starting to become comfortable working with lists and data frames, this output is all you need, since you can work with this list in much the same way that you would work with a data frame. For instance, if you want the first utterance made by Makka-Pakka, all you need to do is type this:

```
speech.by.char$`makka-pakka`[1]
```

```
## [1] "pip"
```

Just remember that R does need you to add the quoting characters (i.e. `'`). Otherwise, there's nothing particularly new or difficult here.

However, sometimes – especially when you're just starting out – it can be convenient to pull these variables out of the list, and into the workspace. This isn't too difficult to do, though it can be a little daunting to novices. To that end, I've included a function called `importList()` in the `lsr` package that does this.⁹ First, here's what you'd have if you had wiped the workspace before the start of this section:

```
who()
```

```
## -- Name --      -- Class --      -- Size --
## afl.finalists   factor        400
## afl.margins     numeric       176
## afl.margins_out numeric       176
## af12            data.frame  4296 x 2
## age              numeric       11
## age.breaks      numeric        4
## age.group       factor        11
## age.group2      factor        11
## age.group3      factor        11
## age.labels      character      3
## colour          logical        1
```

⁹It also works on data frames if you ever feel the need to import all of your variables from the data frame into the workspace. This can be useful at times, though it's not a good idea if you have large data sets or if you're working with multiple data sets at once. In particular, if you do this, never forget that you now have *two* copies of all your variables, one in the workspace and another in the data frame.

```

##   d.cor      numeric    1
##   describeImg  list      0
##   df          data.frame 10 x 4
##   effort      data.frame 10 x 2
##   emphCol     character  1
##   emphColLight character  1
##   emphGrey    character  1
##   eps         logical   1
##   Fibonacci   numeric   7
##   freq        integer   17
##   height      numeric   1
##   is_MP.speaking logical  10
##   itng        data.frame 10 x 2
##   itng.table   table     3 x 4
##   likert centred numeric  10
##   likert.raw   numeric  10
##   old         list      66
##   oneCorPlot   function
##   opinion.dir  numeric  10
##   opinion.strength numeric 10
##   out.0        data.frame 100 x 2
##   out.1        data.frame 100 x 2
##   out.2        data.frame 100 x 2
##   parenthood   data.frame 100 x 4
##   plotOne     function
##   some.data    numeric  18
##   speaker      character 10
##   speech.by.char list      3
##   teams        character 17
##   utterance    character 10
##   width        numeric  1
##   X1           numeric  11
##   X2           numeric  11
##   X3           numeric  11
##   X4           numeric  11
##   Y1           numeric  11
##   Y2           numeric  11
##   Y3           numeric  11
##   Y4           numeric  11

```

Now we use the `importList()` function to copy all of the variables within the `speech.by.char` list:

```
importList( speech.by.char, ask = FALSE)
```

Because the `importList()` function is attempting to create new variables based on the names of the elements of the list, it pauses to check that you're okay with the variable names. The reason it does this is that, if one of the to-be-created variables has the same name as a variable that you already have in your workspace, that variable will end up being overwritten, so it's a good idea to check. Assuming that you type `y`, it will go on to create the variables. Nothing *appears* to have happened, but if we look at our workspace now:

```
who()
```

```

##   -- Name --      -- Class --      -- Size --

```

```
##   afl.finalists    factor      400
##   afl.margins     numeric      176
##   afl.margins_out numeric      176
##   afl2            data.frame 4296 x 2
##   age             numeric      11
##   age.breaks     numeric       4
##   age.group      factor       11
##   age.group2     factor       11
##   age.group3     factor       11
##   age.labels     character     3
##   colour          logical      1
##   d.cor           numeric      1
##   describeImg    list         0
##   df              data.frame 10 x 4
##   effort          data.frame 10 x 2
##   emphCol         character     1
##   emphColLight   character     1
##   emphGrey        character     1
##   eps              logical      1
##   Fibonacci       numeric      7
##   freq             integer      17
##   height           numeric      1
##   is.MP.speaking  logical      10
##   itng             data.frame 10 x 2
##   itng.table      table        3 x 4
##   likert centred numeric      10
##   likert.raw      numeric      10
##   makka.pakka    character     4
##   old              list         66
##   oneCorPlot      function
##   opinion.dir     numeric      10
##   opinion.strength numeric      10
##   out.0            data.frame 100 x 2
##   out.1            data.frame 100 x 2
##   out.2            data.frame 100 x 2
##   parenthood      data.frame 100 x 4
##   plotOne          function
##   some.data        numeric      18
##   speaker          character     10
##   speech.by.char  list         3
##   teams            character     17
##   tombliboo       character     2
##   upsy.daisy      character     4
##   utterance        character     10
##   width            numeric      1
##   X1               numeric      11
##   X2               numeric      11
##   X3               numeric      11
##   X4               numeric      11
##   Y1               numeric      11
##   Y2               numeric      11
##   Y3               numeric      11
##   Y4               numeric      11
```

we see that there are three new variables, called `makka.pakka`, `tombliboo` and `upsy.daisy`. Notice that the `importList()` function has converted the original character strings into valid R variable names, so the variable corresponding to "makka-pakka" is actually `makka.pakka`.¹⁰ Nevertheless, even though the names can change, note that each of these variables contains the exact same information as the original elements of the list did. For example:

```
> makka.pakka
[1] "pip" "pip" "onk" "onk"
```

7.5 Extracting a subset of a data frame

In this section we turn to the question of how to subset a data frame rather than a vector. To that end, the first thing I should point out is that, if all you want to do is subset *one* of the variables inside the data frame, then as usual the `$` operator is your friend. For instance, suppose I'm working with the `itng` data frame, and what I want to do is create the `speech.by.char` list. I can use the exact same tricks that I used last time, since what I really want to do is `split()` the `itng$utterance` vector, using the `itng$speaker` vector as the grouping variable. However, most of the time what you actually want to do is select several different variables within the data frame (i.e., keep only some of the columns), or maybe a subset of cases (i.e., keep only some of the rows). In order to understand how this works, we need to talk more specifically about data frames and how to subset them.

7.5.1 Using the `subset()` function

There are several different ways to subset a data frame in R, some easier than others. I'll start by discussing the `subset()` function, which is probably the conceptually simplest way to do it. For our purposes there are three different arguments that you'll be most interested in:

- `x`. The data frame that you want to subset.
- `subset`. A vector of logical values indicating which cases (rows) of the data frame you want to keep. By default, all cases will be retained.
- `select`. This argument indicates which variables (columns) in the data frame you want to keep. This can either be a list of variable names, or a logical vector indicating which ones to keep, or even just a numeric vector containing the relevant column numbers. By default, all variables will be retained.

Let's start with an example in which I use all three of these arguments. Suppose that I want to subset the `itng` data frame, keeping only the utterances made by Makka-Pakka. What that means is that I need to use the `select` argument to pick out the `utterance` variable, and I also need to use the `subset` variable, to pick out the cases when Makka-Pakka is speaking (i.e., `speaker == "makka-pakka"`). Therefore, the command I need to use is this:

```
df <- subset( x = itng,
               subset = speaker == "makka-pakka",
               select = utterance )                                # data frame is itng
                                                       # keep only Makka-Pakkas speech
                                                       # keep only the utterance variable
print( df )

##      utterance
## 7      pip
```

¹⁰You can do this yourself using the `make.names()` function. In fact, this is itself a handy thing to know about. For example, if you want to convert the names of the variables in the `speech.by.char` list into valid R variable names, you could use a command like this: `names(speech.by.char) <- make.names(names(speech.by.char))`. However, I won't go into details here.

```
## 8      pip
## 9      onk
## 10     onk
```

The variable `df` here is still a data frame, but it only contains one variable (called `utterance`) and four cases. Notice that the row numbers are actually the same ones from the original data frame. It's worth taking a moment to briefly explain this. The reason that this happens is that these "row numbers" are actually row *names*. When you create a new data frame from scratch R will assign each row a fairly boring row name, which is identical to the row number. However, when you subset the data frame, each row keeps its original row name. This can be quite useful, since – as in the current example – it provides you a visual reminder of what each row in the new data frame corresponds to in the original data frame. However, if it annoys you, you can change the row names using the `rownames()` function.¹¹

In any case, let's return to the `subset()` function, and look at what happens when we don't use all three of the arguments. Firstly, suppose that I didn't bother to specify the `select` argument. Let's see what happens:

```
subset( x = itng,
        subset = speaker == "makka-pakka" )
```

```
##           speaker utterance
## 7  makka-pakka      pip
## 8  makka-pakka      pip
## 9  makka-pakka      onk
## 10 makka-pakka      onk
```

Not surprisingly, R has kept the same cases from the original data set (i.e., rows 7 through 10), but this time it has kept all of the variables from the data frame. Equally unsurprisingly, if I don't specify the `subset` argument, what we find is that R keeps all of the cases:

```
subset( x = itng,
        select = utterance )
```

```
##   utterance
## 1      pip
## 2      pip
## 3      onk
## 4      onk
## 5      ee
## 6      oo
## 7      pip
## 8      pip
## 9      onk
## 10     onk
```

Again, it's important to note that this output is still a data frame: it's just a data frame with only a single variable.

¹¹ Conveniently, if you type `rownames(df) <- NULL` R will renumber all the rows from scratch. For the `df` data frame, the labels that currently run from 7 to 10 will be changed to go from 1 to 4.

7.5.2 Using square brackets: I. Rows and columns

Throughout the book so far, whenever I've been subsetting a vector I've tended use the square brackets `[]` to do so. But in the previous section when I started talking about subsetting a data frame I used the `subset()` function. As a consequence, you might be wondering whether it is possible to use the square brackets to subset a data frame. The answer, of course, is yes. Not only *can* you use square brackets for this purpose, as you become more familiar with R you'll find that this is actually much more convenient than using `subset()`. Unfortunately, the use of square brackets for this purpose is somewhat complicated, and can be very confusing to novices. So be warned: this section is more complicated than it feels like it "should" be. With that warning in place, I'll try to walk you through it slowly. For this section, I'll use a slightly different data set, namely the `garden` data frame that is stored in the "`nightgarden2.Rdata`" file.

```
load("./rbook-master/data/nightgarden2.Rdata")
garden
```

```
##           speaker utterance line
## case.1    upsy-daisy      pip    1
## case.2    upsy-daisy      pip    2
## case.3    tombliboo       ee     5
## case.4    makka-pakka    pip    7
## case.5    makka-pakka    onk    9
```

As you can see, the `garden` data frame contains 3 variables and 5 cases, and this time around I've used the `rownames()` function to attach slightly verbose labels to each of the cases. Moreover, let's assume that what we want to do is to pick out rows 4 and 5 (the two cases when Makka-Pakka is speaking), and columns 1 and 2 (variables `speaker` and `utterance`).

How shall we do this? As usual, there's more than one way. The first way is based on the observation that, since a data frame is basically a table, every element in the data frame has a row number and a column number. So, if we want to pick out a single element, we have to specify the row number *and* a column number within the square brackets. By convention, the row number comes first. So, for the data frame above, which has 5 rows and 3 columns, the numerical indexing scheme looks like this:

```
knitr::kable(data.frame(stringsAsFactors=FALSE, row = c("1","2","3", "4", "5"), col1 = c("[1,1]", "[2,1]")))
```

row	col1	col2	col3
1	[1,1]	[1,2]	[1,3]
2	[2,1]	[2,2]	[2,3]
3	[3,1]	[3,2]	[3,3]
4	[4,1]	[4,2]	[4,3]
5	[5,1]	[5,2]	[5,3]

If I want the 3rd case of the 2nd variable, what I would type is `garden[3,2]`, and R would print out some output showing that, this element corresponds to the utterance "`ee`". However, let's hold off from actually doing that for a moment, because there's something slightly counterintuitive about the specifics of what R does under those circumstances (see Section 7.5.4). Instead, let's aim to solve our original problem, which is to pull out two rows (4 and 5) and two columns (1 and 2). This is fairly simple to do, since R allows us to specify multiple rows and multiple columns. So let's try that:

```
garden[ 4:5, 1:2 ]
```

```
##           speaker utterance
## case.4    makka-pakka    pip
## case.5    makka-pakka    onk
```

Clearly, that's exactly what we asked for: the output here is a data frame containing two variables and two cases. Note that I could have gotten the same answer if I'd used the `c()` function to produce my vectors rather than the `:` operator. That is, the following command is equivalent to the last one:

```
garden[ c(4,5), c(1,2) ]
```

```
##           speaker utterance
## case.4 makka-pakka      pip
## case.5 makka-pakka      onk
```

It's just not as pretty. However, if the columns and rows that you want to keep don't happen to be next to each other in the original data frame, then you might find that you have to resort to using commands like `garden[c(2,4,5), c(1,3)]` to extract them.

A second way to do the same thing is to use the names of the rows and columns. That is, instead of using the row numbers and column numbers, you use the character strings that are used as the labels for the rows and columns. To apply this idea to our `garden` data frame, we would use a command like this:

```
garden[ c("case.4", "case.5"), c("speaker", "utterance") ]
```

```
##           speaker utterance
## case.4 makka-pakka      pip
## case.5 makka-pakka      onk
```

Once again, this produces exactly the same output, so I haven't bothered to show it. Note that, although this version is more annoying to *type* than the previous version, it's a bit easier to *read*, because it's often more meaningful to refer to the elements by their names rather than their numbers. Also note that you don't have to use the same convention for the rows and columns. For instance, I often find that the variable names are meaningful and so I sometimes refer to them by name, whereas the row names are pretty arbitrary so it's easier to refer to them by number. In fact, that's more or less exactly what's happening with the `garden` data frame, so it probably makes more sense to use this as the command:

```
garden[ 4:5, c("speaker", "utterance") ]
```

```
##           speaker utterance
## case.4 makka-pakka      pip
## case.5 makka-pakka      onk
```

Again, the output is identical.

Finally, both the rows and columns can be indexed using logical vectors as well. For example, although I *claimed* earlier that my goal was to extract cases 4 and 5, it's pretty obvious that what I really wanted to do was select the cases where Makka-Pakka is speaking. So what I could have done is create a logical vector that indicates which cases correspond to Makka-Pakka speaking:

```
is.MP.speaking <- garden$speaker == "makka-pakka"
is.MP.speaking
```

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

As you can see, the 4th and 5th elements of this vector are `TRUE` while the others are `FALSE`. Now that I've constructed this "indicator" variable, what I can do is use this vector to select the rows that I want to keep:

```
garden[ is.MP.speaking, c("speaker", "utterance") ]
```

```
##           speaker utterance
## case.4 makka-pakka      pip
## case.5 makka-pakka      onk
```

And of course the output is, yet again, the same.

7.5.3 Using square brackets: II. Some elaborations

There are two fairly useful elaborations on this “rows and columns” approach that I should point out. Firstly, what if you want to keep all of the rows, or all of the columns? To do this, all we have to do is leave the corresponding entry blank, but it is crucial to remember to **keep the comma**! For instance, suppose I want to keep all the rows in the `garden` data, but I only want to retain the first two columns. The easiest way do this is to use a command like this:

```
garden[ , 1:2 ]
```

```
##           speaker utterance
## case.1  upsy-daisy      pip
## case.2  upsy-daisy      pip
## case.3  tombliboo       ee
## case.4 makka-pakka      pip
## case.5 makka-pakka      onk
```

Alternatively, if I want to keep all the columns but only want the last two rows, I use the same trick, but this time I leave the second index blank. So my command becomes:

```
garden[ 4:5, ]
```

```
##           speaker utterance line
## case.4 makka-pakka      pip    7
## case.5 makka-pakka      onk    9
```

The second elaboration I should note is that it’s still okay to use negative indexes as a way of telling R to delete certain rows or columns. For instance, if I want to delete the 3rd column, then I use this command:

```
garden[ , -3 ]
```

```
##           speaker utterance
## case.1  upsy-daisy      pip
## case.2  upsy-daisy      pip
## case.3  tombliboo       ee
## case.4 makka-pakka      pip
## case.5 makka-pakka      onk
```

whereas if I want to delete the 3rd row, then I’d use this one:

```
garden[ -3, ]
```

```
##           speaker utterance line
## case.1    upsy-daisy      pip    1
## case.2    upsy-daisy      pip    2
## case.4    makka-pakka    pip    7
## case.5    makka-pakka    onk    9
```

So that's nice.

7.5.4 Using square brackets: III. Understanding “dropping”

At this point some of you might be wondering why I've been so terribly careful to choose my examples in such a way as to ensure that the output always has multiple rows and multiple columns. The reason for this is that I've been trying to hide the somewhat curious “dropping” behaviour that R produces when the output only has a single column. I'll start by showing you what happens, and then I'll try to explain it. Firstly, let's have a look at what happens when the output contains only a single *row*:

```
garden[ 5, ]
```

```
##           speaker utterance line
## case.5    makka-pakka    onk    9
```

This is exactly what you'd expect to see: a data frame containing three variables, and only one case per variable. Okay, no problems so far. What happens when you ask for a single *column*? Suppose, for instance, I try this as a command:

```
garden[ , 3 ]
```

Based on everything that I've shown you so far, you would be well within your rights to expect to see R produce a data frame containing a single variable (i.e., `line`) and five cases. After all, that *is* what the `subset()` command does in this situation, and it's pretty consistent with everything else that I've shown you so far about how square brackets work. In other words, you should expect to see this:

```
line
case.1    1
case.2    2
case.3    5
case.4    7
case.5    9
```

However, that is emphatically not what happens. What you actually get is this:

```
garden[ , 3 ]
```

```
## [1] 1 2 5 7 9
```

That output is not a data frame at all! That's just an ordinary numeric vector containing 5 elements. What's going on here is that R has "noticed" that the output that we've asked for doesn't really "need" to be wrapped up in a data frame at all, because it only corresponds to a single variable. So what it does is "drop" the output from a data frame *containing* a single variable, "down" to a simpler output that corresponds to that variable. This behaviour is actually very convenient for day to day usage once you've become familiar with it – and I suppose that's the real reason why R does this – but there's no escaping the fact that it is *deeply* confusing to novices. It's especially confusing because the behaviour appears only for a very specific case: (a) it only works for columns and not for rows, because the columns correspond to variables and the rows do not, and (b) it only applies to the "rows and columns" version of the square brackets, and not to the `subset()` function,¹² or to the "just columns" use of the square brackets (next section). As I say, it's very confusing when you're just starting out. For what it's worth, you can suppress this behaviour if you want, by setting `drop = FALSE` when you construct your bracketed expression. That is, you could do something like this:

```
garden[ , 3, drop = FALSE ]
```

```
##      line
## case.1    1
## case.2    2
## case.3    5
## case.4    7
## case.5    9
```

I suppose that helps a little bit, in that it gives you some control over the dropping behaviour, but I'm not sure it helps to make things any easier to understand. Anyway, that's the "dropping" special case. Fun, isn't it?

7.5.5 Using square brackets: IV. Columns only

As if the weird "dropping" behaviour wasn't annoying enough, R actually provides a completely different way of using square brackets to index a data frame. Specifically, if you *only* give a single index, R will assume you want the corresponding columns, not the rows. Do not be fooled by the fact that this second method also uses square brackets: it behaves differently to the "rows and columns" method that I've discussed in the last few sections. Again, what I'll do is show you *what* happens first, and then I'll try to explain *why* it happens afterwards. To that end, let's start with the following command:

```
garden[ 1:2 ]
```

```
##      speaker utterance
## case.1  upsy-daisy     pip
## case.2  upsy-daisy     pip
## case.3  tombliboo      ee
## case.4  makka-pakka    pip
## case.5  makka-pakka    onk
```

As you can see, the output gives me the first two columns, much as if I'd typed `garden[,1:2]`. It doesn't give me the first two rows, which is what I'd have gotten if I'd used a command like `garden[1:2,]`. Not only that, if I ask for a *single* column, R does not drop the output:

¹²Actually, you can make the `subset()` function behave this way by using the optional `drop` argument, but by default `subset()` does not drop, which is probably more sensible and more intuitive to novice users.

```
garden[3]
```

```
##      line
## case.1    1
## case.2    2
## case.3    5
## case.4    7
## case.5    9
```

As I said earlier, the *only* case where dropping occurs by default is when you use the “row and columns” version of the square brackets, and the output happens to correspond to a single column. However, if you really want to force R to drop the output, you can do so using the “double brackets” notation:

```
garden[[3]]
```

```
## [1] 1 2 5 7 9
```

Note that R will only allow you to ask for one column at a time using the double brackets. If you try to ask for multiple columns in this way, you get completely different behaviour,¹³ which may or may not produce an error, but definitely won’t give you the output you’re expecting. The only reason I’m mentioning it at all is that you might run into double brackets when doing further reading, and a lot of books don’t explicitly point out the difference between `[` and `[[`. However, I promise that I won’t be using `[[` anywhere else in this book.

Okay, for those few readers that have persevered with this section long enough to get here without having set fire to the book, I should explain *why* R has these two different systems for subsetting a data frame (i.e., “row and column” versus “just columns”), and why they behave so differently to each other. I’m not 100% sure about this since I’m still reading through some of the old references that describe the early development of R, but I think the answer relates to the fact that data frames are actually a very strange hybrid of two different kinds of thing. At a low level, a data frame is a list (Section 4.9). I can demonstrate this to you by overriding the normal `print()` function¹⁴ and forcing R to print out the `garden` data frame using the default print method rather than the special one that is defined only for data frames. Here’s what we get:

```
print.default( garden )
```

```
## $speaker
## [1] upsy-daisy upsy-daisy tombliboo makka-pakka makka-pakka
## Levels: makka-pakka tombliboo upsy-daisy
##
## $utterance
## [1] pip pip ee pip onk
## Levels: ee onk oo pip
##
## $line
## [1] 1 2 5 7 9
##
## attr(),"class")
## [1] "data.frame"
```

¹³Specifically, recursive indexing, a handy tool in some contexts but not something that I want to discuss here.

¹⁴Remember, `print()` is generic: see Section 4.11.

Apart from the weird part of the output right at the bottom, this is *identical* to the print out that you get when you print out a list (see Section 4.9). In other words, a data frame is a list. View from this “list based” perspective, it’s clear what `garden[1]` is: it’s the first variable stored in the list, namely `speaker`. In other words, when you use the “just columns” way of indexing a data frame, using only a single index, R assumes that you’re thinking about the data frame as if it were a *list of variables*. In fact, when you use the `$` operator you’re taking advantage of the fact that the data frame is secretly a list.

However, a data frame is more than just a list. It’s a very special kind of list where all the variables are of the same length, and the first element in each variable happens to correspond to the first “case” in the data set. That’s why no-one ever wants to see a data frame printed out in the default “list-like” way that I’ve shown in the extract above. In terms of the deeper *meaning* behind what a data frame is used for, a data frame really does have this rectangular shape to it:

```
print( garden )
```

```
##           speaker utterance line
## case.1    upsy-daisy      pip    1
## case.2    upsy-daisy      pip    2
## case.3    tombliboo      ee     5
## case.4    makka-pakka    pip    7
## case.5    makka-pakka    onk    9
```

Because of the fact that a data frame is basically a table of data, R provides a second “row and column” method for interacting with the data frame (see Section 7.11.1 for a related example). This method makes much more sense in terms of the high-level *table of data* interpretation of what a data frame is, and so for the most part it’s this method that people tend to prefer. In fact, throughout the rest of the book I will be sticking to the “row and column” approach (though I will use `$` a lot), and never again referring to the “just columns” approach. However, it does get used a lot in practice, so I think it’s important that this book explain what’s going on.

And now let us never speak of this again.

7.6 Sorting, flipping and merging data

In this section I discuss a few useful operations that I feel are loosely related to one another: sorting a vector, sorting a data frame, binding two or more vectors together into a data frame (or matrix), and flipping a data frame (or matrix) on its side. They’re all fairly straightforward tasks, at least in comparison to some of the more obnoxious data handling problems that turn up in real life.

7.6.1 Sorting a numeric or character vector

One thing that you often want to do is sort a variable. If it’s a numeric variable you might want to sort in increasing or decreasing order. If it’s a character vector you might want to sort alphabetically, etc. The `sort()` function provides this capability.

```
numbers <- c(2,4,3)
sort( x = numbers )
```

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

You can ask for R to sort in decreasing order rather than increasing:

```
sort( x = numbers, decreasing = TRUE )
## [1] 4 3 2
```

And you can ask it to sort text data in alphabetical order:

```
text <- c("aardvark", "zebra", "swing")
sort( text )

## [1] "aardvark" "swing"      "zebra"
```

That's pretty straightforward. That being said, it's important to note that I'm glossing over something here. When you apply `sort()` to a character vector it doesn't strictly sort into alphabetical order. R actually has a slightly different notion of how characters are ordered (see Section 7.8.5 and Table 7.3), which is more closely related to how computers store text data than to how letters are ordered in the alphabet. However, that's a topic we'll discuss later. For now, the only thing I should note is that the `sort()` function doesn't alter the original variable. Rather, it creates a new, sorted variable as the output. So if I inspect my original `text` variable:

```
text

## [1] "aardvark" "zebra"     "swing"
```

I can see that it has remained unchanged.

7.6.2 Sorting a factor

You can also sort factors, but the story here is slightly more subtle because there's two different ways you can sort a factor: alphabetically (by label) or by factor level. The `sort()` function uses the latter. To illustrate, let's look at the two different examples. First, let's create a factor in the usual way:

```
fac <- factor( text )
fac

## [1] aardvark zebra    swing
## Levels: aardvark swing zebra
```

Now let's sort it:

```
sort(fac)

## [1] aardvark swing    zebra
## Levels: aardvark swing zebra
```

This *looks* like it's sorted things into alphabetical order, but that's only because the factor levels themselves happen to be alphabetically ordered. Suppose I deliberately define the factor levels in a non-alphabetical order:

```
fac <- factor( text, levels = c("zebra","swing","aardvark") )
fac
```

```
## [1] aardvark zebra    swing
## Levels: zebra swing aardvark
```

Now what happens when we try to sort `fac` this time? The answer:

```
sort(fac)
```

```
## [1] zebra    swing    aardvark
## Levels: zebra swing aardvark
```

It sorts the data into the numerical order implied by the factor levels, not the alphabetical order implied by the labels attached to those levels. Normally you never notice the distinction, because by default the factor levels are assigned in alphabetical order, but it's important to know the difference:

7.6.3 Sorting a data frame

The `sort()` function doesn't work properly with data frames. If you want to sort a data frame the standard advice that you'll find online is to use the `order()` function (not described in this book) to determine what order the rows should be sorted, and then use square brackets to do the shuffling. There's nothing inherently wrong with this advice, I just find it tedious. To that end, the `lsr` package includes a function called `sortFrame()` that you can use to do the sorting. The first argument to the function is named `(x)`, and should correspond to the data frame that you want sorted. After that, all you do is type a list of the names of the variables that you want to use to do the sorting. For instance, if I type this:

```
sortFrame( garden, speaker, line)
```

```
##           speaker utterance line
## case.4 makka-pakka      pip    7
## case.5 makka-pakka      onk    9
## case.3 tombliboo        ee     5
## case.1 upsy-daisy       pip    1
## case.2 upsy-daisy       pip    2
```

what R does is first sort by `speaker` (factor level order). Any ties (i.e., data from the same speaker) are then sorted in order of `line` (increasing numerical order). You can use the minus sign to indicate that numerical variables should be sorted in reverse order:

```
sortFrame( garden, speaker, -line)
```

```
##           speaker utterance line
## case.5 makka-pakka      onk    9
## case.4 makka-pakka      pip    7
## case.3 tombliboo        ee     5
## case.2 upsy-daisy       pip    2
## case.1 upsy-daisy       pip    1
```

As of the current writing, the `sortFrame()` function is under development. I've started introducing functionality to allow you to use the `-` sign to non-numeric variables or to make a distinction between sorting factors alphabetically or by factor level. The idea is that you should be able to type in something like this:

```
sortFrame( garden, -speaker)
```

and have the output correspond to a sort of the `garden` data frame in *reverse* alphabetical order (or reverse factor level order) of `speaker`. As things stand right now, this will actually work, and it will produce sensible output:

```
sortFrame( garden, -speaker)
```

```
##           speaker utterance line
## case.1    upsy-daisy      pip    1
## case.2    upsy-daisy      pip    2
## case.3    tombliboo       ee     5
## case.4    makka-pakka    pip    7
## case.5    makka-pakka    onk    9
```

However, I'm not completely convinced that I've set this up in the ideal fashion, so this may change a little bit in the future.

7.6.4 Binding vectors together

A not-uncommon task that you might find yourself needing to undertake is to combine several vectors. For instance, let's suppose we have the following two numeric vectors:

```
cake.1 <- c(100, 80, 0, 0, 0)
cake.2 <- c(100, 100, 90, 30, 10)
```

The numbers here might represent the amount of each of the two cakes that are left at five different time points. Apparently the first cake is tastier, since that one gets devoured faster. We've already seen one method for combining these vectors: we could use the `data.frame()` function to convert them into a data frame with two variables, like so:

```
cake.df <- data.frame( cake.1, cake.2 )
cake.df
```

```
##   cake.1 cake.2
## 1    100    100
## 2     80    100
## 3      0     90
## 4      0     30
## 5      0     10
```

Two other methods that I want to briefly refer to are the `rbind()` and `cbind()` functions, which will convert the vectors into a matrix. I'll discuss matrices properly in Section 7.11.1 but the details don't matter too much for our current purposes. The `cbind()` function ("column bind") produces a very similar looking output to the data frame example:

```
cake.mat1 <- cbind( cake.1, cake.2 )
cake.mat1
```

```
##      cake.1 cake.2
## [1,]    100    100
## [2,]     80    100
## [3,]      0     90
## [4,]      0     30
## [5,]      0     10
```

but nevertheless it's important to keep in mind that `cake.mat1` is a matrix rather than a data frame, and so has a few differences from the `cake.df` variable. The `rbind()` function ("row bind") produces a somewhat different output: it binds the vectors together row-wise rather than column-wise, so the output now looks like this:

```
cake.mat2 <- rbind( cake.1, cake.2 )
cake.mat2
```

```
##      [,1] [,2] [,3] [,4] [,5]
## cake.1 100   80    0    0    0
## cake.2 100   100   90   30   10
```

You can add names to a matrix by using the `rownames()` and `colnames()` functions, and I should also point out that there's a fancier function in R called `merge()` that supports more complicated "database like" merging of vectors and data frames, but I won't go into details here.

7.6.5 Binding multiple copies of the same vector together

It is sometimes very useful to bind together multiple copies of the same vector. You could do this using the `rbind` and `cbind` functions, using commands like this one

```
fibonacci <- c( 1,1,2,3,5,8 )
rbind( fibonacci, fibonacci, fibonacci )
```

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

but that can be pretty annoying, especially if you need lots of copies. To make this a little easier, the `lsr` package has two additional functions `rowCopy` and `colCopy` that do the same job, but all you have to do is specify the number of copies that you want, instead of typing the name in over and over again. The two arguments you need to specify are `x`, the vector to be copied, and `times`, indicating how many copies should be created:¹⁵

¹⁵Note for advanced users: both of these functions are just wrappers to the `matrix()` function, which is pretty flexible in terms of the ability to convert vectors into matrices. Also, while I'm on this topic, I'll briefly mention the fact that if you're a Matlab user and looking for an equivalent of Matlab's `repmat()` function, I'd suggest checking out the `matlab` package which contains R versions of a lot of handy Matlab functions.

```
rowCopy( x = fibonacci, times = 3 )

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

Of course, in practice you don't need to name the arguments all the time. For instance, here's an example using the `colCopy()` function with the argument names omitted:

```
colCopy( fibonacci, 3 )

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

7.6.6 Transposing a matrix or data frame

One of the main reasons that I wanted to discuss the `rbind()` and `cbind()` functions in the same section as the `data.frame()` function is that it immediately raises the question of how to “flip” or *transpose* a matrix or data frame. Notice that in the last section I was able to produce two different matrices, `cake.mat1` and `cake.mat2` that were basically mirror images of one another. A natural question to ask is whether you can directly transform one into another. The transpose function `t()` allows us to do this in a straightforward fashion. To start with, I'll show you how to transpose a matrix, and then I'll move onto talk about data frames. Firstly, let's load a matrix I prepared earlier, from the `cakes.Rdata` file:

```
load("./rbook-master/data/cakes.Rdata")
cakes

##      time.1 time.2 time.3 time.4 time.5
## cake.1    100     80      0      0      0
## cake.2    100    100     90     30     10
## cake.3    100     20     20     20     20
## cake.4    100    100    100    100    100
```

And just to make sure you believe me that this is actually a matrix:

```
class( cakes )

## [1] "matrix"
```

Okay, now let's transpose the matrix:

```
cakes.flipped <- t( cakes )
cakes.flipped
```

```
##      cake.1 cake.2 cake.3 cake.4
## time.1    100    100    100    100
## time.2     80    100     20    100
## time.3      0     90     20    100
## time.4      0     30     20    100
## time.5      0     10     20    100
```

The output here is still a matrix:

```
class( cakes.flipped )

## [1] "matrix"
```

At this point you should have two questions: (1) how do we do the same thing for data frames? and (2) why should we care about this? Let's start with the how question. First, I should note that you can transpose a data frame just fine using the `t()` function, but that has the slightly awkward consequence of converting the output from a data frame to a matrix, which isn't usually what you want. It's quite easy to convert the output back again, of course,¹⁶ but I hate typing two commands when I can do it with one. To that end, the `lsr` package has a simple “convenience” function called `tFrame()` which does exactly the same thing as `t()` but converts the output to a data frame for you. To illustrate this, let's transpose the `itng` data frame that we used earlier. Here's the original data frame:

```
itng
```

```
##      speaker utterance
## 1  upsy-daisy      pip
## 2  upsy-daisy      pip
## 3  upsy-daisy     onk
## 4  upsy-daisy     onk
## 5  tombliboo      ee
## 6  tombliboo      oo
## 7 makka-pakka      pip
## 8 makka-pakka      pip
## 9 makka-pakka     onk
## 10 makka-pakka    onk
```

and here's what happens when you transpose it using `tFrame()`:

```
tFrame( itng )
```

```
##          V1          V2          V3          V4          V5          V6
## speaker  upsy-daisy  upsy-daisy  upsy-daisy  upsy-daisy  tombliboo  tombliboo
## utterance      pip        pip        onk        onk        ee         oo
##                  V7          V8          V9          V10
## speaker  makka-pakka  makka-pakka  makka-pakka  makka-pakka
## utterance      pip        pip        onk        onk
```

An important point to recognise is that transposing a data frame is not always a sensible thing to do: in fact, I'd go so far as to argue that it's usually *not* sensible. It depends a lot on whether the “cases” from your original data frame would make sense as variables, and to think of each of your original “variables” as

¹⁶The function you need for that is called `as.data.frame()`.

cases. I think that's emphatically *not* true for our `itng` data frame, so I wouldn't advise doing it in this situation.

That being said, sometimes it really is true. For instance, had we originally stored our `cakes` variable as a data frame instead of a matrix, then it would absolutely be sensible to flip the data frame!¹⁷ There are some situations where it is useful to flip your data frame, so it's nice to know that you can do it. Indeed, that's the main reason why I have spent so much time talking about this topic. A lot of statistical tools make the assumption that the rows of your data frame (or matrix) correspond to observations, and the columns correspond to the variables. That's not unreasonable, of course, since that is a pretty standard convention. However, think about our `cakes` example here. This is a situation where you might want to do an analysis of the different cakes (i.e. cakes as variables, time points as cases), but equally you might want to do an analysis where you think of the times as being the things of interest (i.e., times as variables, cakes as cases). If so, then it's useful to know how to flip a matrix or data frame around.

7.7 Reshaping a data frame

One of the most annoying tasks that you need to undertake on a regular basis is that of reshaping a data frame. Framed in the most general way, reshaping the data means taking the data in whatever format it's given to you, and converting it to the format you need it. Of course, if we're going to characterise the problem that broadly, then about half of this chapter can probably be thought of as a kind of reshaping. So we're going to have to narrow things down a little bit. To that end, I'll talk about a few different tools that you can use for a few different tasks. In particular, I'll discuss a couple of easy to use (but limited) functions that I've included in the `lsr` package. In future versions of the book I plan to expand this discussion to include some of the more powerful tools that are available in R, but I haven't had the time to do so yet.

7.7.1 Long form and wide form data

The most common format in which you might obtain data is as a “case by variable” layout, commonly known as the *wide form* of the data.

```
load("./rbook-master/data/repeated.Rdata")
```

```
who()
```

```
## -- Name --      -- Class --      -- Size --
##   afl.finalists    factor      400
##   afl.margins     numeric     176
##   afl.margins_out  numeric     176
##   afl2            data.frame 4296 x 2
##   age             numeric      11
##   age.breaks      numeric      4
##   age.group       factor      11
##   age.group2      factor      11
##   age.group3      factor      11
##   age.labels      character     3
##   cake.1          numeric      5
##   cake.2          numeric      5
##   cake.df         data.frame  5 x 2
##   cake.mat1       matrix      5 x 2
```

¹⁷In truth, I suspect that most of the cases when you can sensibly flip a data frame occur when all of the original variables are measurements of the same type (e.g., all variables are response times), and if so you could easily have chosen to encode your data as a matrix instead of as a data frame. But since people do sometimes prefer to work with data frames, I've written the `tFrame()` function for the sake of convenience. I don't really think it's something that is needed very often.

```

##   cake.mat2      matrix      2 x 5
##   cakes         matrix      4 x 5
##   cakes.flipped matrix      5 x 4
##   choice        data.frame 4 x 10
##   colour        logical     1
##   d.cor         numeric     1
##   describeImg   list       0
##   df            data.frame 4 x 1
##   drugs         data.frame 10 x 8
##   effort        data.frame 10 x 2
##   emphCol       character   1
##   emphColLight  character   1
##   emphGrey      character   1
##   eps           logical     1
##   fac           factor      3
##   fibonacci    numeric     6
##   Fibonacci    numeric     7
##   freq          integer     17
##   garden        data.frame 5 x 3
##   height        numeric     1
##   is.MP.speaking logical     5
##   itng          data.frame 10 x 2
##   itng.table    table      3 x 4
##   likert centred numeric     10
##   likert.raw    numeric     10
##   makka.pakka   character   4
##   numbers       numeric     3
##   old           list       66
##   oneCorPlot   function
##   opinion.dir   numeric     10
##   opinion.strength numeric     10
##   out.0          data.frame 100 x 2
##   out.1          data.frame 100 x 2
##   out.2          data.frame 100 x 2
##   parenthood    data.frame 100 x 4
##   plotOne       function
##   some.data     numeric     18
##   speaker        character   10
##   speech.by.char list       3
##   teams          character   17
##   text           character   3
##   tombliboo     character   2
##   upsy.daisy    character   4
##   utterance     character   10
##   width          numeric     1
##   X1            numeric     11
##   X2            numeric     11
##   X3            numeric     11
##   X4            numeric     11
##   Y1            numeric     11
##   Y2            numeric     11
##   Y3            numeric     11
##   Y4            numeric     11

```

To get a sense of what I'm talking about, consider an experiment in which we are interested in the different effects that alcohol and caffeine have on people's working memory capacity (WMC) and reaction times (RT). We recruit 10 participants, and measure their WMC and RT under three different conditions: a "no drug" condition, in which they are not under the influence of either caffeine or alcohol, a "caffeine" condition, in which they are under the influence of caffeine, and an "alcohol" condition, in which... well, you can probably guess. Ideally, I suppose, there would be a fourth condition in which both drugs are administered, but for the sake of simplicity let's ignore that. The `drugs` data frame gives you a sense of what kind of data you might observe in an experiment like this:

```
drugs
```

```
##   id gender WMC_alcohol WMC_caffeine WMC_no.drug RT_alcohol RT_caffeine
## 1  1 female      3.7       3.7      3.9      488      236
## 2  2 female      6.4       7.3      7.9      607      376
## 3  3 female      4.6       7.4      7.3      643      226
## 4  4 male        6.4       7.8      8.2      684      206
## 5  5 female      4.9       5.2      7.0      593      262
## 6  6 male        5.4       6.6      7.2      492      230
## 7  7 male        7.9       7.9      8.9      690      259
## 8  8 male        4.1       5.9      4.5      486      230
## 9  9 female      5.2       6.2      7.2      686      273
## 10 10 female     6.2       7.4      7.8      645      240
##   RT_no.drug
## 1      371
## 2      349
## 3      412
## 4      252
## 5      439
## 6      464
## 7      327
## 8      305
## 9      327
## 10     498
```

This is a data set in "wide form", in which each participant corresponds to a single row. We have two variables that are characteristics of the subject (i.e., their `id` number and their `gender`) and six variables that refer to one of the two measured variables (WMC or RT) in one of the three testing conditions (alcohol, caffeine or no drug). Because all of the testing conditions (i.e., the three drug types) are applied to all participants, drug type is an example of a *within-subject factor*.

7.7.2 Reshaping data using `wideToLong()`

The "wide form" of this data set is useful for some situations: it is often very useful to have each row correspond to a single subject. However, it is not the only way in which you might want to organise this data. For instance, you might want to have a separate row for each "testing occasion". That is, "participant 1 under the influence of alcohol" would be one row, and "participant 1 under the influence of caffeine" would be another row. This way of organising the data is generally referred to as the *long form* of the data. It's not too difficult to switch between wide and long form, and I'll explain how it works in a moment; for now, let's just have a look at what the long form of this data set looks like:

```
drugs.2 <- wideToLong( data = drugs, within = "drug" )
head(drugs.2)
```

```
##   id gender    drug WMC  RT
## 1  1 female alcohol 3.7 488
## 2  2 female alcohol 6.4 607
## 3  3 female alcohol 4.6 643
## 4  4   male alcohol 6.4 684
## 5  5 female alcohol 4.9 593
## 6  6   male alcohol 5.4 492
```

The `drugs.2` data frame that we just created has 30 rows: each of the 10 participants appears in three separate rows, one corresponding to each of the three testing conditions. And instead of having a variable like `WMC_caffeine` that indicates that we were measuring “WMC” in the “caffeine” condition, this information is now recorded in two separate variables, one called `drug` and another called `WMC`. Obviously, the long and wide forms of the data contain the same information, but they represent quite different ways of organising that information. Sometimes you find yourself needing to analyse data in wide form, and sometimes you find that you need long form. So it’s really useful to know how to switch between the two.

In the example I gave above, I used a function called `wideToLong()` to do the transformation. The `wideToLong()` function is part of the `lsr` package. The key to understanding this function is that it relies on the *variable names* to do all the work. Notice that the variable names in the `drugs` data frame follow a very clear scheme. Whenever you have a variable with a name like `WMC_caffeine` you know that the variable being measured is “WMC”, and that the specific condition in which it is being measured is the “caffeine” condition. Similarly, you know that `RT_no.drug` refers to the “RT” variable measured in the “no drug” condition. The measured variable comes first (e.g., `WMC`), followed by a separator character (in this case the separator is an underscore, `_`), and then the name of the condition in which it is being measured (e.g., `caffeine`). There are two different prefixes (i.e., the strings before the separator, `WMC`, `RT`) which means that there are two separate variables being measured. There are three different suffixes (i.e., the strings after the separator, `caffeine`, `alcohol`, `no.drug`) meaning that there are three different levels of the within-subject factor. Finally, notice that the separator string (i.e., `_`) does not appear anywhere in two of the variables (`id`, `gender`), indicating that these are ***between-subject*** variables, namely variables that do not vary within participant (e.g., a person’s `gender` is the same regardless of whether they’re under the influence of alcohol, caffeine etc.).

Because of the fact that the variable naming scheme here is so informative, it’s quite possible to reshape the data frame without any additional input from the user. For example, in this particular case, you could just type the following:

```
wideToLong( drugs )
```

```
##   id gender    within WMC  RT
## 1  1 female   alcohol 3.7 488
## 2  2 female   alcohol 6.4 607
## 3  3 female   alcohol 4.6 643
## 4  4   male   alcohol 6.4 684
## 5  5 female   alcohol 4.9 593
## 6  6   male   alcohol 5.4 492
## 7  7   male   alcohol 7.9 690
## 8  8   male   alcohol 4.1 486
## 9  9 female   alcohol 5.2 686
## 10 10 female   alcohol 6.2 645
## 11 1  female  caffeine 3.7 236
## 12 2 female  caffeine 7.3 376
## 13 3 female  caffeine 7.4 226
## 14 4   male  caffeine 7.8 206
## 15 5 female  caffeine 5.2 262
```

```
## 16 6 male caffeine 6.6 230
## 17 7 male caffeine 7.9 259
## 18 8 male caffeine 5.9 230
## 19 9 female caffeine 6.2 273
## 20 10 female caffeine 7.4 240
## 21 1 female no.drug 3.9 371
## 22 2 female no.drug 7.9 349
## 23 3 female no.drug 7.3 412
## 24 4 male no.drug 8.2 252
## 25 5 female no.drug 7.0 439
## 26 6 male no.drug 7.2 464
## 27 7 male no.drug 8.9 327
## 28 8 male no.drug 4.5 305
## 29 9 female no.drug 7.2 327
## 30 10 female no.drug 7.8 498
```

This is pretty good, actually. The only think it has gotten wrong here is that it doesn't know what name to assign to the within-subject factor, so instead of calling it something sensible like `drug`, it has used the unimaginative name `within`. If you want to ensure that the `wideToLong()` function applies a sensible name, you have to specify the `within` argument, which is just a character string that specifies the name of the within-subject factor. So when I used this command earlier,

```
drugs.2 <- wideToLong( data = drugs, within = "drug" )
```

all I was doing was telling R to use `drug` as the name of the within subject factor.

Now, as I was hinting earlier, the `wideToLong()` function is very inflexible. It *requires* that the variable names all follow this naming scheme that I outlined earlier. If you don't follow this naming scheme it won't work.¹⁸ The only flexibility that I've included here is that you can change the separator character by specifying the `sep` argument. For instance, if you were using variable names of the form `WMC/caffeine`, for instance, you could specify that `sep="/"`, using a command like this

```
drugs.2 <- wideToLong( data = drugs, within = "drug", sep = "/" )
```

and it would still work.

7.7.3 Reshaping data using `longToWide()`

To convert data from long form to wide form, the `lsr` package also includes a function called `longToWide()`. Recall from earlier that the long form of the data (i.e., the `drugs.2` data frame) contains variables named `id`, `gender`, `drug`, `WMC` and `RT`. In order to convert from long form to wide form, all you need to do is indicate which of these variables are measured separately for each condition (i.e., `WMC` and `RT`), and which variable is the within-subject factor that specifies the condition (i.e., `drug`). You do this via a two-sided formula, in which the measured variables are on the left hand side, and the within-subject factor is on the right hand side. In this case, the formula would be `WMC + RT ~ drug`. So the command that we would use might look like this:

```
longToWide( data=drugs.2, formula= WMC+RT ~ drug )
```

¹⁸This limitation is deliberate, by the way: if you're getting to the point where you want to do something more complicated, you should probably start learning how to use `reshape()`, `cast()` and `melt()` or some of other the more advanced tools. The `wideToLong()` and `longToWide()` functions are included only to help you out when you're first starting to use R.

```
##   id gender WMC_alcohol RT_alcohol WMC_caffeine RT_caffeine WMC_no.drug
## 1  1 female     3.7      488      3.7      236      3.9
## 2  2 female     6.4      607      7.3      376      7.9
## 3  3 female     4.6      643      7.4      226      7.3
## 4  4 male       6.4      684      7.8      206      8.2
## 5  5 female     4.9      593      5.2      262      7.0
## 6  6 male       5.4      492      6.6      230      7.2
## 7  7 male       7.9      690      7.9      259      8.9
## 8  8 male       4.1      486      5.9      230      4.5
## 9  9 female     5.2      686      6.2      273      7.2
## 10 10 female    6.2      645      7.4      240      7.8
##   RT_no.drug
## 1      371
## 2      349
## 3      412
## 4      252
## 5      439
## 6      464
## 7      327
## 8      305
## 9      327
## 10     498
```

or, if we chose to omit argument names, we could simplify it to this:

```
longToWide( drugs.2, WMC+RT ~ drug )
```

```
##   id gender WMC_alcohol RT_alcohol WMC_caffeine RT_caffeine WMC_no.drug
## 1  1 female     3.7      488      3.7      236      3.9
## 2  2 female     6.4      607      7.3      376      7.9
## 3  3 female     4.6      643      7.4      226      7.3
## 4  4 male       6.4      684      7.8      206      8.2
## 5  5 female     4.9      593      5.2      262      7.0
## 6  6 male       5.4      492      6.6      230      7.2
## 7  7 male       7.9      690      7.9      259      8.9
## 8  8 male       4.1      486      5.9      230      4.5
## 9  9 female     5.2      686      6.2      273      7.2
## 10 10 female    6.2      645      7.4      240      7.8
##   RT_no.drug
## 1      371
## 2      349
## 3      412
## 4      252
## 5      439
## 6      464
## 7      327
## 8      305
## 9      327
## 10     498
```

Note that, just like the `wideToLong()` function, the `longToWide()` function allows you to override the default separator character. For instance, if the command I used had been

```
longToWide( drugs.2, WMC+RT ~ drug, sep="/" )

##      id gender WMC/alcohol RT/alcohol WMC/caffeine RT/caffeine WMC/no.drug
## 1    1 female     3.7       488      3.7       236      3.9
## 2    2 female     6.4       607      7.3       376      7.9
## 3    3 female     4.6       643      7.4       226      7.3
## 4    4 male       6.4       684      7.8       206      8.2
## 5    5 female     4.9       593      5.2       262      7.0
## 6    6 male       5.4       492      6.6       230      7.2
## 7    7 male       7.9       690      7.9       259      8.9
## 8    8 male       4.1       486      5.9       230      4.5
## 9    9 female     5.2       686      6.2       273      7.2
## 10   10 female    6.2       645      7.4       240      7.8
##      RT/no.drug
## 1    371
## 2    349
## 3    412
## 4    252
## 5    439
## 6    464
## 7    327
## 8    305
## 9    327
## 10   498
```

the output would contain variables with names like `RT/alcohol` instead of `RT_alcohol`.

7.7.4 Reshaping with multiple within-subject factors

As I mentioned above, the `wideToLong()` and `longToWide()` functions are quite limited in terms of what they can do. However, they do handle a broader range of situations than the one outlined above. Consider the following, fairly simple psychological experiment. I'm interested in the effects of practice on some simple decision making problem. It doesn't really matter what the problem is, other than to note that I'm interested in two distinct outcome variables. Firstly, I care about people's accuracy, measured by the proportion of decisions that people make correctly, denoted `PC`. Secondly, I care about people's speed, measured by the mean response time taken to make those decisions, denoted `MRT`. That's standard in psychological experiments: the speed-accuracy trade-off is pretty ubiquitous, so we generally need to care about both variables.

To look at the effects of practice over the long term, I test each participant on two days, `day1` and `day2`, where for the sake of argument I'll assume that `day1` and `day2` are about a week apart. To look at the effects of practice over the short term, the testing during each day is broken into two "blocks", `block1` and `block2`, which are about 20 minutes apart. This isn't the world's most complicated experiment, but it's still a fair bit more complicated than the last one. This time around we have two within-subject factors (i.e., `day` and `block`) and we have two measured variables for each condition (i.e., `PC` and `MRT`). The `choice` data frame shows what the wide form of this kind of data might look like:

```
choice
```

```
##      id gender MRT/block1/day1 MRT/block1/day2 MRT/block2/day1
## 1    1 male      415          400          455
## 2    2 male      500          490          532
```

```

## 3 3 female      478      468      499
## 4 4 female      550      502      602
##   MRT/block2/day2 PC/block1/day1 PC/block1/day2 PC/block2/day1
## 1      450       79       88       82
## 2      518       83       92       86
## 3      474       91       98       90
## 4      588       75       89       78
##   PC/block2/day2
## 1      93
## 2      97
## 3     100
## 4      95

```

Notice that this time around we have variable names of the form `MRT/block1/day2`. As before, the first part of the name refers to the measured variable (response time), but there are now two suffixes, one indicating that the testing took place in block 1, and the other indicating that it took place on day 2. And just to complicate matters, it uses `/` as the separator character rather than `_`. Even so, reshaping this data set is pretty easy. The command to do it is,

```
choice.2 <- wideToLong( choice, within=c("block", "day"), sep="/" )
```

which is pretty much the exact same command we used last time. The only difference here is that, because there are two within-subject factors, the `within` argument is a vector that contains two names. When we look at the long form data frame that this creates, we get this:

```
choice.2
```

```

##   id gender MRT  PC  block  day
## 1  1   male  415  79 block1 day1
## 2  2   male  500  83 block1 day1
## 3  3 female  478  91 block1 day1
## 4  4 female  550  75 block1 day1
## 5  1   male  400  88 block1 day2
## 6  2   male  490  92 block1 day2
## 7  3 female  468  98 block1 day2
## 8  4 female  502  89 block1 day2
## 9  1   male  455  82 block2 day1
## 10 2   male  532  86 block2 day1
## 11 3 female  499  90 block2 day1
## 12 4 female  602  78 block2 day1
## 13 1   male  450  93 block2 day2
## 14 2   male  518  97 block2 day2
## 15 3 female  474 100 block2 day2
## 16 4 female  588  95 block2 day2

```

In this long form data frame we have two between-subject variables (`id` and `gender`), two variables that define our within-subject manipulations (`block` and `day`), and two more contain the measurements we took (`MRT` and `PC`).

To convert this back to wide form is equally straightforward. We use the `longToWide()` function, but this time around we need to alter the formula in order to tell it that we have two within-subject factors. The command is now

```
longToWide( choice.2, MRT+PC ~ block+day, sep="/" )

##   id gender MRT/block1/day1 PC/block1/day1 MRT/block1/day2 PC/block1/day2
## 1  1    male      415          79      400          88
## 2  2    male      500          83      490          92
## 3  3  female     478          91      468          98
## 4  4  female     550          75      502          89
##   MRT/block2/day1 PC/block2/day1 MRT/block2/day2 PC/block2/day2
## 1      455          82      450          93
## 2      532          86      518          97
## 3      499          90      474         100
## 4      602          78      588          95
```

and this produces a wide form data set containing the same variables as the original `choice` data frame.

7.7.5 What other options are there?

The advantage to the approach described in the previous section is that it solves a quite specific problem (but a commonly encountered one) with a minimum of fuss. The disadvantage is that the tools are quite limited in scope. They allow you to switch your data back and forth between two different formats that are very common in everyday data analysis. However, there are a number of other tools that you can use if need be. Just within the core packages distributed with R there is the `reshape()` function, as well as the `stack()` and `unstack()` functions, all of which can be useful under certain circumstances. And there are of course thousands of packages on CRAN that you can use to help you with different tasks. One popular package for this purpose is the `reshape` package, written by Hadley Wickham ?, for details see Wickham2007. There are two key functions in this package, called `melt()` and `cast()` that are pretty useful for solving a lot of reshaping problems. In a future version of this book I intend to discuss `melt()` and `cast()` in a fair amount of detail.

7.8 Working with text

Sometimes your data set is quite text heavy. This can be for a lot of different reasons. Maybe the raw data are actually taken from text sources (e.g., newspaper articles), or maybe your data set contains a lot of free responses to survey questions, in which people can write whatever text they like in response to some query. Or maybe you just need to rejig some of the text used to describe nominal scale variables. Regardless of what the reason is, you'll probably want to know a little bit about how to handle text in R. Some things you already know how to do: I've discussed the use of `nchar()` to calculate the number of characters in a string (Section 3.8.1), and a lot of the general purpose tools that I've discussed elsewhere (e.g., the `==` operator) have been applied to text data as well as to numeric data. However, because text data is quite rich, and generally not as well structured as numeric data, R provides a lot of additional tools that are quite specific to text. In this section I discuss only those tools that come as part of the base packages, but there are other possibilities out there: the `stringr` package provides a powerful alternative that is a lot more coherent than the basic tools, and is well worth looking into.

7.8.1 Shortening a string

The first task I want to talk about is how to shorten a character string. For example, suppose that I have a vector that contains the names of several different animals:

```
animals <- c( "cat", "dog", "kangaroo", "whale" )
```

It might be useful in some contexts to extract the first three letters of each word. This is often useful when annotating figures, or when creating variable labels: it's often very inconvenient to use the full name, so you want to shorten it to a short code for space reasons. The `strtrim()` function can be used for this purpose. It has two arguments: `x` is a vector containing the text to be shortened and `width` specifies the number of characters to keep. When applied to the `animals` data, here's what we get:

```
strtrim( x = animals, width = 3 )
```

```
## [1] "cat" "dog" "kan" "wha"
```

Note that the only thing that `strtrim()` does is chop off excess characters at the end of a string. It doesn't insert any whitespace characters to fill them out if the original string is shorter than the `width` argument. For example, if I trim the `animals` data to 4 characters, here's what I get:

```
strtrim( x = animals, width = 4 )
```

```
## [1] "cat" "dog" "kang" "whal"
```

The "cat" and "dog" strings still only use 3 characters. Okay, but what if you don't want to start from the first letter? Suppose, for instance, I only wanted to keep the second and third letter of each word. That doesn't happen quite as often, but there are some situations where you need to do something like that. If that does happen, then the function you need is `substr()`, in which you specify a `start` point and a `stop` point instead of specifying the width. For instance, to keep only the 2nd and 3rd letters of the various `animals`, I can do the following:

```
substr( x = animals, start = 2, stop = 3 )
```

```
## [1] "at" "og" "an" "ha"
```

7.8.2 Pasting strings together

Much more commonly, you will need either to glue several character strings together or to pull them apart. To glue several strings together, the `paste()` function is very useful. There are three arguments to the `paste()` function:

- ... As usual, the dots “match” up against any number of inputs. In this case, the inputs should be the various different strings you want to paste together.
- `sep`. This argument should be a string, indicating what characters R should use as separators, in order to keep each of the original strings separate from each other in the pasted output. By default the value is a single space, `sep = " "`. This is made a little clearer when we look at the examples.
- `collapse`. This is an argument indicating whether the `paste()` function should interpret vector inputs as things to be collapsed, or whether a vector of inputs should be converted into a vector of outputs. The default value is `collapse = NULL` which is interpreted as meaning that vectors should not be collapsed. If you want to collapse vectors into a single string, then you should specify a value for `collapse`. Specifically, the value of `collapse` should correspond to the separator character that you want to use for the collapsed inputs. Again, see the examples below for more details.

That probably doesn't make much sense yet, so let's start with a simple example. First, let's try to paste two words together, like this:

```
paste( "hello", "world" )
```

```
## [1] "hello world"
```

Notice that R has inserted a space between the "hello" and "world". Suppose that's not what I wanted. Instead, I might want to use . as the separator character, or to use no separator at all. To do either of those, I would need to specify `sep = "."` or `sep = ""`.¹⁹ For instance:

```
paste( "hello", "world", sep = "." )
```

```
## [1] "hello.world"
```

Now let's consider a slightly more complicated example. Suppose I have two vectors that I want to `paste()` together. Let's say something like this:

```
hw <- c( "hello", "world" )
ng <- c( "nasty", "government" )
```

And suppose I want to paste these together. However, if you think about it, this statement is kind of ambiguous. It could mean that I want to do an “element wise” paste, in which all of the first elements get pasted together (“hello nasty”) and all the second elements get pasted together (“world government”). Or, alternatively, I might intend to collapse everything into one big string (“hello nasty world government”). By default, the `paste()` function assumes that you want to do an element-wise paste:

```
paste( hw, ng )
```

```
## [1] "hello nasty"      "world government"
```

However, there's nothing stopping you from overriding this default. All you have to do is specify a value for the `collapse` argument, and R will chuck everything into one dirty big string. To give you a sense of exactly how this works, what I'll do in this next example is specify *different* values for `sep` and `collapse`:

```
paste( hw, ng, sep = ".", collapse = ":::" )
```

```
## [1] "hello.nasty:::world.government"
```

7.8.3 Splitting strings

At other times you have the opposite problem to the one in the last section: you have a whole lot of text bundled together into a single string that needs to be pulled apart and stored as several different variables. For instance, the data set that you get sent might include a single variable containing someone's full name, and you need to separate it into first names and last names. To do this in R you can use the `strsplit()` function, and for the sake of argument, let's assume that the string you want to split up is the following string:

¹⁹To be honest, it does bother me a little that the default value of `sep` is a space. Normally when I want to paste strings together I don't want any separator character, so I'd prefer it if the default were `sep=""`. To that end, it's worth noting that there's also a `paste0()` function, which is identical to `paste()` except that it always assumes that `sep=""`. Type `?paste` for more information about this.

```
monkey <- "It was the best of times. It was the blurst of times."
```

To use the `strsplit()` function to break this apart, there are three arguments that you need to pay particular attention to:

- `x`. A vector of character strings containing the data that you want to split.
- `split`. Depending on the value of the `fixed` argument, this is either a fixed string that specifies a delimiter, or a regular expression that matches against one or more possible delimiters. If you don't know what regular expressions are (probably most readers of this book), don't use this option. Just specify a separator string, just like you would for the `paste()` function.
- `fixed`. Set `fixed = TRUE` if you want to use a fixed delimiter. As noted above, unless you understand regular expressions this is definitely what you want. However, the default value is `fixed = FALSE`, so you have to set it explicitly.

Let's look at a simple example:

```
monkey.1 <- strsplit( x = monkey, split = " ", fixed = TRUE )
monkey.1
```

```
## [[1]]
## [1] "It"      "was"     "the"     "best"    "of"      "times." "It"
## [8] "was"     "the"     "blurst"  "of"      "times."
```

One thing to note in passing is that the output here is a list (you can tell from the `[[1]]` part of the output), whose first and only element is a character vector. This is useful in a lot of ways, since it means that you can input a character vector for `x` and then have the `strsplit()` function split all of them, but it's kind of annoying when you only have a single input. To that end, it's useful to know that you can `unlist()` the output:

```
unlist( monkey.1 )
```

```
## [1] "It"      "was"     "the"     "best"    "of"      "times." "It"
## [8] "was"     "the"     "blurst"  "of"      "times."
```

To understand why it's important to remember to use the `fixed = TRUE` argument, suppose we wanted to split this into two separate sentences. That is, we want to use `split = "."` as our delimiter string. As long as we tell R to remember to treat this as a *fixed* separator character, then we get the right answer:

```
strsplit( x = monkey, split = ".", fixed = TRUE )
```

```
## [[1]]
## [1] "It was the best of times"    " It was the blurst of times"
```

However, if we don't do this, then R will assume that when you typed `split = "."` you were trying to construct a “regular expression”, and as it happens the character `.` has a special meaning within a regular expression. As a consequence, if you forget to include the `fixed = TRUE` part, you won't get the answers you're looking for.

7.8.4 Making simple conversions

A slightly different task that comes up quite often is making transformations to text. A simple example of this would be converting text to lower case or upper case, which you can do using the `toupper()` and `tolower()` functions. Both of these functions have a single argument `x` which contains the text that needs to be converted. An example of this is shown below:

```
text <- c("lIfe", "Impact")
tolower(x = text)
```

```
## [1] "life"    "impact"
```

A slightly more powerful way of doing text transformations is to use the `chartr()` function, which allows you to specify a “character by character” substitution. This function contains three arguments, `old`, `new` and `x`. As usual `x` specifies the text that needs to be transformed. The `old` and `new` arguments are strings of the same length, and they specify how `x` is to be converted. Every instance of the first character in `old` is converted to the first character in `new` and so on. For instance, suppose I wanted to convert `"albino"` to `"libido"`. To do this, I need to convert all of the `"a"` characters (all 1 of them) in `"albino"` into `"l"` characters (i.e., `a → l`). Additionally, I need to make the substitutions `l → i` and `n → d`. To do so, I would use the following command:

```
old.text <- "albino"
chartr(old = "aln", new = "lid", x = old.text)
```

```
## [1] "libido"
```

7.8.5 Applying logical operations to text

In Section 3.9.5 we discussed a very basic text processing tool, namely the ability to use the equality operator `==` to test to see if two strings are identical to each other. However, you can also use other logical operators too. For instance R also allows you to use the `<` and `>` operators to determine which of two strings comes first, alphabetically speaking. Sort of. Actually, it’s a bit more complicated than that, but let’s start with a simple example:

```
"cat" < "dog"
```

```
## [1] TRUE
```

In this case, we see that `"cat"` does come before `"dog"` alphabetically, so R judges the statement to be true. However, if we ask R to tell us if `"cat"` comes before `"anteater"`,

```
"cat" < "anteater"
```

```
## [1] FALSE
```

It tells us that the statement is false. So far, so good. But text data is a bit more complicated than the dictionary suggests. What about `"cat"` and `"CAT"`? Which of these comes first? Let’s try it and find out:

Table 7.3: The ordering of various text characters used by the `<` and `>` operators, as well as by the `sort()` function. Not shown is the “space” character, which actually comes <U+FB01>rst on the list.

Characters
! " # \$ % & ' () * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ? @ A B C D E F G H I J K L M N O P Q R S T U V W X Y

```
"CAT" < "cat"
```

```
## [1] FALSE
```

In other words, R assumes that uppercase letters come before lowercase ones. Fair enough. No-one is likely to be surprised by that. What you might find surprising is that R assumes that *all* uppercase letters come before *all* lowercase ones. That is, while `"anteater" < "zebra"` is a true statement, and the uppercase equivalent `"ANTEATER" < "ZEBRA"` is also true, it is *not* true to say that `"anteater" < "ZEBRA"`, as the following extract illustrates:

```
"anteater" < "ZEBRA"
```

```
## [1] TRUE
```

This may seem slightly counterintuitive. With that in mind, it may help to have a quick look Table 7.3, which lists various text characters in the order that R uses.

7.8.6 Concatenating and printing with `cat()`

One function that I want to make a point of talking about, even though it’s not quite on topic, is the `cat()` function. The `cat()` function is a mixture of `paste()` and `print()`. That is, what it does is concatenate strings and then print them out. In your own work you can probably survive without it, since `print()` and `paste()` will actually do what you need, but the `cat()` function is so widely used that I think it’s a good idea to talk about it here. The basic idea behind `cat()` is straightforward. Like `paste()`, it takes several arguments as inputs, which it converts to strings, collapses (using a separator character specified using the `sep` argument), and prints on screen. If you want, you can use the `file` argument to tell R to print the output into a file rather than on screen (I won’t do that here). However, it’s important to note that the `cat()` function collapses vectors first, and *then* concatenates them. That is, notice that when I use `cat()` to combine `hw` and `ng`, I get a different result than if I’d used `paste()`

```
cat( hw, ng )
```

```
## hello world nasty government
```

```
paste( hw, ng, collapse = " " )
```

```
## [1] "hello nasty world government"
```

Notice the difference in the ordering of words. There’s a few additional details that I need to mention about `cat()`. Firstly, `cat()` really is a function for *printing*, and not for creating text strings to store for later. You can’t assign the output to a variable, as the following example illustrates:

```
x <- cat( hw, ng )

## hello world nasty government

x

## NULL
```

Despite my attempt to store the output as a variable, `cat()` printed the results on screen anyway, and it turns out that the variable I created doesn't contain anything at all.²⁰ Secondly, the `cat()` function makes use of a number of "special" characters. I'll talk more about these in the next section, but I'll illustrate the basic point now, using the example of "`\n`" which is interpreted as a "new line" character. For instance, compare the behaviour of `print()` and `cat()` when asked to print the string "`hello\nworld`":

```
print( "hello\nworld" )  # print literally:

## [1] "hello\nworld"

cat( "hello\nworld" )  # interpret as newline

## hello
## world
```

In fact, this behaviour is important enough that it deserves a section of its very own...

7.8.7 Using escape characters in text

The previous section brings us quite naturally to a fairly fundamental issue when dealing with strings, namely the issue of delimiters and escape characters. Reduced to its most basic form, the problem we have is that R commands are written using text characters, and our strings also consist of text characters. So, suppose I want to type in the word "hello", and have R encode it as a string. If I were to just type `hello`, R will think that I'm referring to a variable or a function called `hello` rather than interpret it as a string. The solution that R adopts is to require you to enclose your string by *delimiter* characters, which can be either double quotes or single quotes. So, when I type "`hello`" or '`hello`' then R knows that it should treat the text in between the quote marks as a character string. However, this isn't a complete solution to the problem: after all, " and ' are themselves perfectly legitimate text characters, and so we might want to include those in our string as well. For instance, suppose I wanted to encode the name "O'Rourke" as a string. It's *not* legitimate for me to type '`O'rourke`' because R is too stupid to realise that "O'Rourke" is a real word. So it will interpret the '`O`' part as a complete string, and then will get confused when it reaches the `Rourke`' part. As a consequence, what you get is an error message:

```
'O'Rourke'
Error: unexpected symbol in "'O'Rourke"
```

To some extent, R offers us a cheap fix to the problem because of the fact that it allows us to use either " or ' as the delimiter character. Although '`O'rourke`' will make R cry, it is perfectly happy with "`O'Rourke`":

²⁰Note that you can capture the output from `cat()` if you want to, but you have to be sneaky and use the `capture.output()` function. For example, the command `x <- capture.output(cat(hw,ng))` would work just fine.

Table 7.4: Standard escape characters that are evaluated by some text processing commands, including ‘cat()’. This convention dates back to the development of the C programming language in the 1970s, and as a consequence a lot of these characters make most sense if you pretend that R is actually a typewriter, as explained in the main text. Type ‘?Quotes’ for the corresponding R help file.

Escape.sequence	Interpretation
‘\n’	Newline
‘\t’	Horizontal Tab
‘\v’	Vertical Tab
‘\b’	Backspace
‘\r’	Carriage Return
‘\f’	Form feed
‘\a’	Alert sound
‘\\’	Backslash
‘\’	Single quote
‘”	Double quote

```
"O'Rourke"
```

```
## [1] "O'Rourke"
```

This is a real advantage to having two different delimiter characters. Unfortunately, anyone with even the slightest bit of deviousness to them can see the problem with this. Suppose I’m reading a book that contains the following passage,

P.J. O’Rourke says, “Yay, money!”. It’s a joke, but no-one laughs.

and I want to enter this as a string. Neither the ‘ or ” delimiters will solve the problem here, since this string contains both a single quote character and a double quote character. To encode strings like this one, we have to do something a little bit clever.

The solution to the problem is to designate an *escape character*, which in this case is \, the humble backslash. The escape character is a bit of a sacrificial lamb: if you include a backslash character in your string, R will *not* treat it as a literal character at all. It’s actually used as a way of inserting “special” characters into your string. For instance, if you want to force R to insert actual quote marks into the string, then what you actually type is \‘ or \” (these are called *escape sequences*). So, in order to encode the string discussed earlier, here’s a command I could use:

```
PJ <- "P.J. O\'Rourke says, \"Yay, money!\". It\'s a joke, but no-one laughs."
```

Notice that I’ve included the backslashes for both the single quotes and double quotes. That’s actually overkill: since I’ve used ” as my delimiter, I only needed to do this for the double quotes. Nevertheless, the command has worked, since I didn’t get an error message. Now let’s see what happens when I print it out:

```
print( PJ )
```

```
## [1] "P.J. O'Rourke says, \"Yay, money!\". It's a joke, but no-one laughs."
```

Hm. Why has R printed out the string using \”? For the exact same reason that *I* needed to insert the backslash in the first place. That is, when R prints out the PJ string, it has enclosed it with delimiter characters, and it wants to unambiguously show us which of the double quotes are delimiters and which ones are actually part of the string. Fortunately, if this bugs you, you can make it go away by using the `print.noquote()` function, which will just print out the literal string that you encoded in the first place:

```
print.noquote( PJ )
```

Typing `cat(PJ)` will produce a similar output.

Introducing the escape character solves a lot of problems, since it provides a mechanism by which we can insert all sorts of characters that aren't on the keyboard. For instance, as far as a computer is concerned, "new line" is actually a text character. It's the character that is printed whenever you hit the "return" key on your keyboard. If you want to insert a new line character into your string, you can actually do this by including the escape sequence `\n`. Or, if you want to insert a backslash character, then you can use `\\\`. A list of the standard escape sequences recognised by R is shown in Table 7.4. A lot of these actually date back to the days of the typewriter (e.g., carriage return), so they might seem a bit counterintuitive to people who've never used one. In order to get a sense for what the various escape sequences do, we'll have to use the `cat()` function, because it's the only function "dumb" enough to literally print them out:

```
cat( "xxxx\bboo" ) # \b is a backspace, so it deletes the preceding x
cat( "xxxx\ttoo" ) # \t is a tab, so it inserts a tab space
cat( "xxxx\nnoo" ) # \n is a newline character
cat( "xxxx\rroo" ) # \r returns you to the beginning of the line
```

And that's pretty much it. There are a few other escape sequence that R recognises, which you can use to insert arbitrary ASCII or Unicode characters into your string (type `?Quotes` for more details) but I won't go into details here.

7.8.8 Matching and substituting text

Another task that we often want to solve is find all strings that match a certain criterion, and possibly even to make alterations to the text on that basis. There are several functions in R that allow you to do this, three of which I'll talk about briefly here: `grep()`, `gsub()` and `sub()`. Much like the `substr()` function that I talked about earlier, all three of these functions are intended to be used in conjunction with regular expressions (see Section 7.8.9 but you can also use them in a simpler fashion, since they all allow you to set `fixed = TRUE`, which means we can ignore all this regular expression rubbish and just use simple text matching).

So, how do these functions work? Let's start with the `grep()` function. The purpose of this function is to input a vector of character strings `x`, and to extract all those strings that fit a certain pattern. In our examples, I'll assume that the `pattern` in question is a literal sequence of characters that the string must contain (that's what `fixed = TRUE` does). To illustrate this, let's start with a simple data set, a vector that contains the names of three beers. Something like this:

```
beers <- c( "little creatures", "sierra nevada", "coopers pale" )
```

Next, let's use `grep()` to find out which of these strings contains the substring "`er`". That is, the `pattern` that we need to match is the fixed string "`er`", so the command we need to use is:

```
grep( pattern = "er", x = beers, fixed = TRUE )
```

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

What the output here is telling us is that the second and third elements of `beers` both contain the substring "`er`". Alternatively, however, we might prefer it if `grep()` returned the actual strings themselves. We can do this by specifying `value = TRUE` in our function call. That is, we'd use a command like this:

```
grep( pattern = "er", x = beers, fixed = TRUE, value = TRUE )

## [1] "sierra nevada" "coopers pale"
```

The other two functions that I wanted to mention in this section are `gsub()` and `sub()`. These are both similar in spirit to `grep()` insofar as what they do is search through the input strings (`x`) and find all of the strings that match a `pattern`. However, what these two functions do is *replace* the pattern with a `replacement` string. The `gsub()` function will replace *all* instances of the pattern, whereas the `sub()` function just replaces the first instance of it in each string. To illustrate how this works, suppose I want to replace all instances of the letter "a" with the string "BLAH". I can do this to the `beers` data using the `gsub()` function:

```
gsub( pattern = "a", replacement = "BLAH", x = beers, fixed = TRUE )

## [1] "little creBLAHtures"      "sierrBLAH nevBLAHdBLAH"
## [3] "coopers pBLAHle"
```

Notice that all three of the "a"s in "sierra nevada" have been replaced. In contrast, let's see what happens when we use the exact same command, but this time using the `sub()` function instead:

```
sub( pattern = "a", replacement = "BLAH", x = beers, fixed = TRUE )

## [1] "little creBLAHtures" "sierrBLAH nevada"    "coopers pBLAHle"
```

Only the first "a" is changed.

7.8.9 Regular expressions (not really)

There's one last thing I want to talk about regarding text manipulation, and that's the concept of a *regular expression*. Throughout this section we've often needed to specify `fixed = TRUE` in order to force R to treat some of our strings as actual strings, rather than as regular expressions. So, before moving on, I want to very briefly explain what regular expressions are. I'm *not* going to talk at all about how they work or how you specify them, because they're genuinely complicated and not at all relevant to this book. However, they are extremely powerful tools and they're quite widely used by people who have to work with lots of text data (e.g., people who work with natural language data), and so it's handy to at least have a vague idea about what they are. The basic idea is quite simple. Suppose I want to extract all strings in my `beers` vector that contain a vowel followed immediately by the letter "s". That is, I want to finds the beer names that contain either "as", "es", "is", "os" or "us". One possibility would be to manually specify all of these possibilities and then match against these as fixed strings one at a time, but that's tedious. The alternative is to try to write out a single "regular" expression that matches all of these. The regular expression that does this²¹ is "[aeiou]s", and you can kind of see what the syntax is doing here. The bracketed expression means "any of the things in the middle", so the expression as a whole means "any of the things in the middle" (i.e. vowels) followed by the letter "s". When applied to our beer names we get this:

```
grep( pattern = "[aeiou]s", x = beers, value = TRUE )

## [1] "little creatures"
```

²¹Sigh. For advanced users: R actually supports two different ways of specifying regular expressions. One is the POSIX standard, the other is to use Perl-style regular expressions. The default is generally POSIX. If you understand regular expressions, that probably made sense to you. If not, don't worry. It's not important.

So it turns out that only "little creatures" contains a vowel followed by the letter "s". But of course, had the data contained a beer like "fosters", that would have matched as well because it contains the string "os". However, I deliberately chose not to include it because Fosters is not – in my opinion – a proper beer.²² As you can tell from this example, regular expressions are a neat tool for specifying *patterns* in text: in this case, "vowel then s". So they are definitely things worth knowing about if you ever find yourself needing to work with a large body of text. However, since they are fairly complex and not necessary for any of the applications discussed in this book, I won't talk about them any further.

7.9 Reading unusual data files

In this section I'm going to switch topics (again!) and turn to the question of how you can load data from a range of different sources. Throughout this book I've assumed that your data are stored as an `.Rdata` file or as a "properly" formatted CSV file. And if so, then the basic tools that I discussed in Section 4.5 should be quite sufficient. However, in real life that's not a terribly plausible assumption to make, so I'd better talk about some of the other possibilities that you might run into.

7.9.1 Loading data from text files

The first thing I should point out is that if your data are saved as a text file but aren't *quite* in the proper CSV format, then there's still a pretty good chance that the `read.csv()` function (or equivalently, `read.table()`) will be able to open it. You just need to specify a few more of the optional arguments to the function. If you type `?read.csv` you'll see that the `read.csv()` function actually has several arguments that you can specify. Obviously you need to specify the `file` that you want it to load, but the others all have sensible default values. Nevertheless, you will sometimes need to change them. The ones that I've often found myself needing to change are:

- `header`. A lot of the time when you're storing data as a CSV file, the first row actually contains the column names and not data. If that's not true, you need to set `header = FALSE`.
- `sep`. As the name "comma separated value" indicates, the values in a row of a CSV file are usually separated by commas. This isn't universal, however. In Europe the decimal point is typically written as , instead of . and as a consequence it would be somewhat awkward to use , as the separator. Therefore it is not unusual to use ; over there. At other times, I've seen a TAB character used. To handle these cases, we'd need to set `sep = ";"` or `sep = "\t"`.
- `quote`. It's conventional in CSV files to include a quoting character for textual data. As you can see by looking at the `booksales.csv` file, this is usually a double quote character, ". But sometimes there is no quoting character at all, or you might see a single quote mark ' used instead. In those cases you'd need to specify `quote = ""` or `quote = "'"`.
- `skip`. It's actually very common to receive CSV files in which the first few rows have nothing to do with the actual data. Instead, they provide a human readable summary of where the data came from, or maybe they include some technical info that doesn't relate to the data. To tell R to ignore the first (say) three lines, you'd need to set `skip = 3`
- `na.strings`. Often you'll get given data with missing values. For one reason or another, some entries in the table are missing. The data file needs to include a "special" string to indicate that the entry is missing. By default R assumes that this string is `NA`, since that's what *it* would do, but there's no universal agreement on what to use in this situation. If the file uses `???` instead, then you'll need to set `na.strings = "???"`.

It's kind of nice to be able to have all these options that you can tinker with. For instance, have a look at the data file shown pictured in Figure 7.1. This file contains almost the same data as the last file (except it

²²I thank Amy Perfors for this example.

doesn't have a header), and it uses a bunch of wacky features that you don't normally see in CSV files. In fact, it just so happens that I'm going to have to change all five of those arguments listed above in order to load this file. Here's how I would do it:

```
data <- read.csv( file = "./rbook-master/data/booksales2.csv", # specify the name of the file
                  header = FALSE,           # variable names in the file?
                  skip = 8,                 # ignore the first 8 lines
                  quote = "*",              # what indicates text data?
                  sep = "\t",               # what separates different entries?
                  na.strings = "NFI" )      # what is the code for missing data?
```

If I now have a look at the data I've loaded, I see that this is what I've got:

```
head( data )
```

```
##          V1  V2  V3  V4
## 1 January 31   0 high
## 2 February 28 100 high
## 3 March   31 200 low
## 4 April   30  50 out
## 5 May     31   NA out
## 6 June    30   0 high
```

Because I told R to expect * to be used as the quoting character instead of "; to look for tabs (which we write like this: \t) instead of commas, and to skip the first 8 lines of the file, it's basically loaded the right data. However, since `booksales2.csv` doesn't contain the column names, R has made them up. Showing the kind of imagination I expect from insentient software, R decided to call them V1, V2, V3 and V4. Finally, because I told it that the file uses "NFI" to denote missing data, R correctly figures out that the sales data for May are actually missing.

In real life you'll rarely see data this stupidly formatted.²³

7.9.2 Loading data from SPSS (and other statistics packages)

The commands listed above are the main ones we'll need for data files in this book. But in real life we have many more possibilities. For example, you might want to read data files in from other statistics programs. Since SPSS is probably the most widely used statistics package in psychology, it's worth briefly showing how to open SPSS data files (file extension `.sav`). It's surprisingly easy. The extract below should illustrate how to do so:

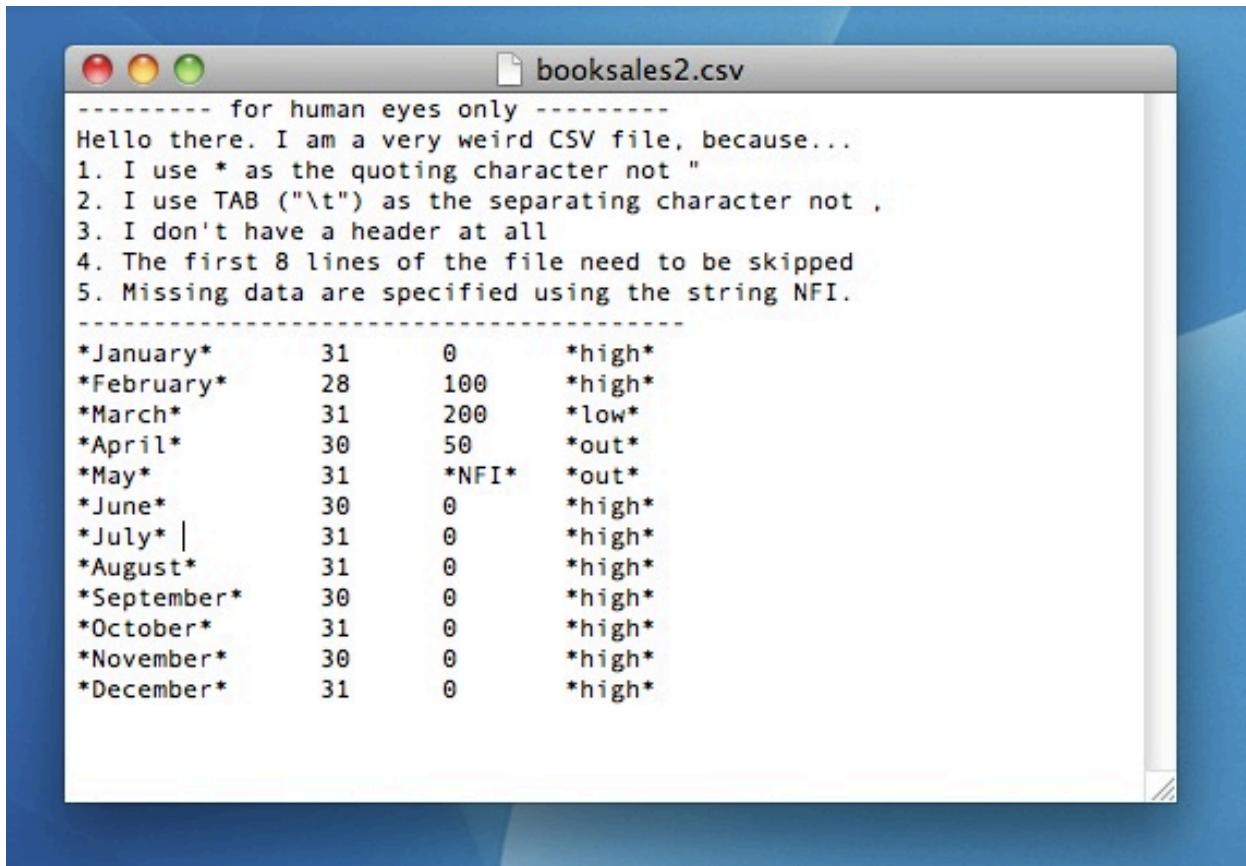
```
library( foreign )                      # load the package
X <- read.spss( "./rbook-master/data/datafile.sav" )  # create a list containing the data
X <- as.data.frame( X )                # convert to data frame
```

If you wanted to import from an SPSS file to a data frame directly, instead of importing a list and then converting the list to a data frame, you can do that too:

```
X <- read.spss( file = "datafile.sav", to.data.frame = TRUE )
```

And that's pretty much it, at least as far as SPSS goes. As far as other statistical software goes, the `foreign` package provides a wealth of possibilities. To open SAS files, check out the `read.ssd()` and `read.xport()` functions. To open data from Minitab, the `read.mtp()` function is what you're looking for. For Stata, the `read.dta()` function is what you want. For Systat, the `read.systat()` function is what you're after.

²³If you're lucky.



The screenshot shows a terminal window with a white background and a blue title bar. The title bar has three colored window control buttons (red, yellow, green) on the left and the text "booksales2.csv" in the center. The main area of the terminal contains the following text:

```
----- for human eyes only -----
Hello there. I am a very weird CSV file, because...
1. I use * as the quoting character not "
2. I use TAB ("\t") as the separating character not ,
3. I don't have a header at all
4. The first 8 lines of the file need to be skipped
5. Missing data are specified using the string NFI.

-----
*January*      31      0      *high*
*February*    28     100      *high*
*March*        31     200      *low*
*April*        30      50      *out*
*May*          31      *NFI*    *out*
*June*         30      0      *high*
*July* |       31      0      *high*
*August*       31      0      *high*
*September*   30      0      *high*
*October*     31      0      *high*
*November*   30      0      *high*
*December*   31      0      *high*
```

Figure 7.1: The `booksales2.csv` data file. It contains more or less the same data as the original `booksales.csv` data file, but has a lot of very quirky features.

7.9.3 Loading Excel files

A different problem is posed by Excel files. Despite years of yelling at people for sending data to me encoded in a proprietary data format, I get sent a lot of Excel files. In general R does a pretty good job of opening them, but it's bit finicky because Microsoft don't seem to be terribly fond of people using non-Microsoft products, and go to some lengths to make it tricky. If you get an Excel file, my suggestion would be to open it up in Excel (or better yet, OpenOffice, since that's free software) and then save the spreadsheet as a CSV file. Once you've got the data in that format, you can open it using `read.csv()`. However, if for some reason you're desperate to open the `.xls` or `.xlsx` file directly, then you can use the `read.xls()` function in the `gdata` package:

```
library( gdata )          # load the package
X <- read.xls( "datafile.xlsx" )  # create a data frame
```

This usually works. And if it doesn't, you're probably justified in "suggesting" to the person that sent you the file that they should send you a nice clean CSV file instead.

7.9.4 Loading Matlab (& Octave) files

A lot of scientific labs use Matlab as their default platform for scientific computing; or Octave as a free alternative. Opening Matlab data files (file extension `.mat`) slightly more complicated, and if it wasn't for the fact that Matlab is so very widespread and is an extremely good platform, I wouldn't mention it. However, since Matlab is so widely used, I think it's worth discussing briefly how to get Matlab and R to play nicely together. The way to do this is to install the `R.matlab` package (don't forget to install the dependencies too). Once you've installed and loaded the package, you have access to the `readMat()` function. As any Matlab user will know, the `.mat` files that Matlab produces are workspace files, very much like the `.Rdata` files that R produces. So you can't import a `.mat` file as a data frame. However, you can import it as a list. So, when we do this:

```
library( R.matlab )      # load the package
data <- readMat( "matlabfile.mat" )  # read the data file to a list
```

The `data` object that gets created will be a list, containing one variable for every variable stored in the Matlab file. It's fairly straightforward, though there are some subtleties that I'm ignoring. In particular, note that if you don't have the `Rcompression` package, you can't open Matlab files above the version 6 format. So, if like me you've got a recent version of Matlab, and don't have the `Rcompression` package, you'll need to save your files using the `-v6` flag otherwise R can't open them.

Oh, and Octave users? The `foreign` package contains a `read.octave()` command. Just this once, the world makes life easier for you folks than it does for all those cashed-up swanky Matlab bastards.

7.9.5 Saving other kinds of data

Given that I talked extensively about how to load data from non-R files, it might be worth briefly mentioning that R is also pretty good at writing data into other file formats besides its own native ones. I won't discuss them in this book, but the `write.csv()` function can write CSV files, and the `write.foreign()` function (in the `foreign` package) can write SPSS, Stata and SAS files. There are also a lot of low level commands that you can use to write very specific information to a file, so if you really, really needed to you could create your own `write.obscurFileType()` function, but that's also a long way beyond the scope of this book. For now, all that I want you to recognise is that this capability is there if you need it.

7.9.6 Are we done yet?

Of course not. If I've learned nothing else about R it's that you're *never bloody done*. This listing doesn't even come close to exhausting the possibilities. Databases are supported by the `RODBC`, `DBI`, and `RMySQL` packages among others. You can open webpages using the `RCurl` package. Reading and writing JSON objects is supported through the `rjson` package. And so on. In a sense, the right question is not so much "can R do this?" so much as "whereabouts in the wilds of CRAN *is* the damn package that does it?"

7.10 Coercing data from one class to another

Sometimes you want to change the variable class. This can happen for all sorts of reasons. Sometimes when you import data from files, it can come to you in the wrong format: numbers sometimes get imported as text, dates usually get imported as text, and many other possibilities besides. Regardless of how you've ended up in this situation, there's a very good chance that sometimes you'll want to convert a variable from one class into another one. Or, to use the correct term, you want to *coerce* the variable from one class into another. Coercion is a little tricky, and so I'll only discuss the very basics here, using a few simple examples.

Firstly, let's suppose we have a variable `x` that is *supposed* to be representing a number, but the data file that you've been given has encoded it as text. Let's imagine that the variable is something like this:

```
x <- "100" # the variable
class(x) # what class is it?

## [1] "character"
```

Obviously, if I want to do calculations using `x` in its current state, R is going to get very annoyed at me. It thinks that `x` is text, so it's not going to allow me to try to do mathematics using it! Obviously, we need to coerce `x` from character to numeric. We can do that in a straightforward way by using the `as.numeric()` function:

```
x <- as.numeric(x) # coerce the variable
class(x) # what class is it?

## [1] "numeric"

x + 1 # hey, addition works!

## [1] 101
```

Not surprisingly, we can also convert it back again if we need to. The function that we use to do this is the `as.character()` function:

```
x <- as.character(x) # coerce back to text
class(x) # check the class:

## [1] "character"
```

However, there's some fairly obvious limitations: you can't coerce the string "`hello world`" into a number because, well, there's isn't a number that corresponds to it. Or, at least, you can't do anything useful:

```
as.numeric( "hello world" ) # this isn't going to work.

## Warning: NAs introduced by coercion

## [1] NA
```

In this case R doesn't give you an error message; it just gives you a warning, and then says that the data is missing (see Section 4.6.1 for the interpretation of NA).

That gives you a feel for how to change between numeric and character data. What about logical data? To cover this briefly, coercing text to logical data is pretty intuitive: you use the `as.logical()` function, and the character strings "T", "TRUE", "True" and "true" all convert to the logical value of TRUE. Similarly "F", "FALSE", "False", and "false" all become FALSE. All other strings convert to NA. When you go back the other way using `as.character()`, TRUE converts to "TRUE" and FALSE converts to "FALSE". Converting numbers to logicals – again using `as.logical()` – is straightforward. Following the convention in the study of logic, the number 0 converts to FALSE. Everything else is TRUE. Going back using `as.numeric()`, FALSE converts to 0 and TRUE converts to 1.

7.11 Other useful data structures

Up to this point we have encountered several different kinds of variables. At the simplest level, we've seen numeric data, logical data and character data. However, we've also encountered some more complicated kinds of variables, namely factors, formulas, data frames and lists. We'll see a few more specialised data structures later on in this book, but there's a few more generic ones that I want to talk about in passing. None of them are central to the rest of the book (and in fact, the only one we'll even see anywhere else is the matrix), but they do crop up a fair bit in real life.

7.11.1 Matrices

In various different places in this chapter I've made reference to an R data structure called a *matrix*, and mentioned that I'd talk a bit more about matrices later on. That time has come. Much like a data frame, a matrix is basically a big rectangular table of data, and in fact there are quite a few similarities between the two. However, there are also some key differences, so it's important to talk about matrices in a little detail. Let's start by using `rbind()` to create a small matrix:²⁴

```
row.1 <- c( 2,3,1 )           # create data for row 1
row.2 <- c( 5,6,7 )           # create data for row 2
M <- rbind( row.1, row.2 )    # row bind them into a matrix
print( M )                   # and print it out...

##      [,1] [,2] [,3]
## row.1    2     3     1
## row.2    5     6     7
```

The variable `M` is a matrix, which we can confirm by using the `class()` function. Notice that, when we bound the two vectors together, R retained the names of the original variables as row names. We could delete these if we wanted by typing `rownames(M)<-NULL`, but I generally prefer having meaningful names attached to my variables, so I'll keep them. In fact, let's also add some highly unimaginative column names as well:

²⁴You can also use the `matrix()` command itself, but I think the “binding” approach is a little more intuitive.

Table 7.5: The row and column version, which is identical to the corresponding indexing scheme for a data frame of the same size.

Row	Col.1	Col.2	Col.3
Row 1	[1,1]	[1,2]	[1,3]
Row 2	[2,1]	[2,2]	[2,3]

Table 7.6: The single-index version, which is quite different to what we would get with a data frame.

Row	Col.1	Col.2	Col.3
Row 1	1	3	5
Row 2	2	4	6

```
colnames(M) <- c( "col.1", "col.2", "col.3" )
print(M)
```

```
##      col.1 col.2 col.3
## row.1     2     3     1
## row.2     5     6     7
```

You can use square brackets to subset a matrix in much the same way that you can for data frames, again specifying a row index and then a column index. For instance, $M[2,3]$ pulls out the entry in the 2nd row and 3rd column of the matrix (i.e., 7), whereas $M[2,]$ pulls out the entire 2nd row, and $M[,3]$ pulls out the entire 3rd column. However, it's worth noting that when you pull out a column, R will print the results horizontally, not vertically. The reason for this relates to how matrices (and arrays generally) are implemented. The original matrix M is treated as a two-dimensional objects, containing 2 rows and 3 columns. However, whenever you pull out a single row or a single column, the result is considered to be one-dimensional. As far as R is concerned there's no real reason to distinguish between a one-dimensional object printed vertically (a column) and a one-dimensional object printed horizontally (a row), and it prints them all out horizontally.²⁵ There is also a way of using only a single index, but due to the internal structure to how R defines a matrix, it works very differently to what we saw previously with data frames.

The single-index approach is illustrated in Table 7.6 but I don't really want to focus on it since we'll never really need it for this book, and matrices don't play anywhere near as large a role in this book as data frames do. The reason for these differences is that for this is that, for both data frames and matrices, the "row and column" version exists to allow the human user to interact with the object in the psychologically meaningful way: since both data frames and matrices are basically just tables of data, it's the same in each case. However, the single-index version is really a method for you to interact with the object in terms of its internal structure, and the internals for data frames and matrices are quite different.

The critical difference between a data frame and a matrix is that, in a data frame, we have this notion that each of the columns corresponds to a different variable: as a consequence, the columns in a data frame can be of different data types. The first column could be numeric, and the second column could contain character

²⁵This has some interesting implications for how matrix algebra is implemented in R (which I'll admit I initially found odd), but that's a little beyond the scope of this book. However, since there will be a small proportion of readers that do care, I'll quickly outline the basic thing you need to get used to: when multiplying a matrix by a vector (or one-dimensional array) using the $\%*\%$ operator R will attempt to interpret the vector (or 1D array) as either a row-vector or column-vector, depending on whichever one makes the multiplication work. That is, suppose M is the 2×3 matrix, and v is a 1×3 row vector. It is impossible to multiply Mv , since the dimensions don't conform, but you *can* multiply by the corresponding column vector, Mv^t . So, if I set $v <- M[2,]$ and then try to calculate $M \%*\% v$, which you'd think would fail, it actually works because R treats the one dimensional array as if it were a column vector for the purposes of matrix multiplication. Note that if both objects are one dimensional arrays/vectors, this leads to ambiguity since vv^t (inner product) and v^tv (outer product) yield different answers. In this situation, the $\%*\%$ operator returns the inner product not the outer product. To understand all the details, check out the help documentation.

strings, and the third column could be logical data. In that sense, there is a fundamental asymmetry built into a data frame, because of the fact that columns represent variables (which can be qualitatively different to each other) and rows represent cases (which cannot). Matrices are intended to be thought of in a different way. At a fundamental level, a matrix really is just *one* variable: it just happens that this one variable is formatted into rows and columns. If you want a matrix of numeric data, every single element in the matrix *must* be a number. If you want a matrix of character strings, every single element in the matrix *must* be a character string. If you try to mix data of different types together, then R will either spit out an error, or quietly coerce the underlying data into a list. If you want to find out what class R secretly thinks the data within the matrix is, you need to do something like this:

```
class( M[1] )
```

```
## [1] "numeric"
```

You can't type `class(M)`, because all that will happen is R will tell you that `M` is a matrix: we're not interested in the class of the matrix itself, we want to know what class the underlying data is assumed to be. Anyway, to give you a sense of how R enforces this, let's try to change one of the elements of our numeric matrix into a character string:

```
M[1,2] <- "text"
M
```

```
##      col.1 col.2 col.3
## row.1 "2"   "text" "1"
## row.2 "5"   "6"    "7"
```

It looks as if R has coerced all of the data in our matrix into character strings. And in fact, if we now typed in `class(M[1])` we'd see that this is exactly what has happened. If you alter the contents of one element in a matrix, R will change the underlying data type as necessary.

There's only one more thing I want to talk about regarding matrices. The concept behind a matrix is very much a mathematical one, and in mathematics a matrix is a most definitely a two-dimensional object. However, when doing data analysis, we often have reasons to want to use higher dimensional tables (e.g., sometimes you need to cross-tabulate three variables against each other). You can't do this with matrices, but you can do it with *arrays*. An array is just like a matrix, except it can have more than two dimensions if you need it to. In fact, as far as R is concerned a matrix is just a special kind of array, in much the same way that a data frame is a special kind of list. I don't want to talk about arrays too much, but I will very briefly show you an example of what a 3D array looks like. To that end, let's cross tabulate the `speaker` and `utterance` variables from the `nightgarden.Rdata` data file, but we'll add a third variable to the cross-tabs this time, a logical variable which indicates whether or not I was still awake at this point in the show:

```
dan.awake <- c( TRUE, TRUE, TRUE, TRUE, TRUE, FALSE, FALSE, FALSE, FALSE )
```

Now that we've got all three variables in the workspace (assuming you loaded the `nightgarden.Rdata` data earlier in the chapter) we can construct our three way cross-tabulation, using the `table()` function.

```
xtab.3d <- table( speaker, utterance, dan.awake )
xtab.3d
```

```
## , , dan.awake = FALSE
##
##          utterance
```

```

## speaker      ee onk oo pip
## makka-pakka 0   2   0   2
## tombliboo   0   0   1   0
## upsy-daisy  0   0   0   0
##
## , , dan.awake = TRUE
##
##          utterance
## speaker      ee onk oo pip
## makka-pakka 0   0   0   0
## tombliboo   1   0   0   0
## upsy-daisy  0   2   0   2

```

Hopefully this output is fairly straightforward: because R can't print out text in three dimensions, what it does is show a sequence of 2D slices through the 3D table. That is, the `, , dan.awake = FALSE` part indicates that the 2D table that follows below shows the 2D cross-tabulation of `speaker` against `utterance` only for the `dan.awake = FALSE` instances, and so on.²⁶

7.11.2 Ordered factors

One topic that I neglected to mention when discussing factors previously (Section 4.7) is that there are actually two different types of factor in R, unordered factors and ordered factors. An unordered factor corresponds to a nominal scale variable, and all of the factors we've discussed so far in this book have been unordered (as will all the factors used anywhere else except in this section). However, it's often very useful to explicitly tell R that your variable is *ordinal scale*, and if so you need to declare it to be an *ordered factor*. For instance, earlier in this chapter we made use of a variable consisting of Likert scale data, which we represented as the `likert.raw` variable:

```
likert.raw
```

```
## [1] 1 7 3 4 4 4 2 6 5 5
```

We can declare this to be an ordered factor in by using the `factor()` function, and setting `ordered = TRUE`. To illustrate how this works, let's create an ordered factor called `likert.ordinal` and have a look at it:

```

likert.ordinal <- factor( x = likert.raw,           # the raw data
                           levels = seq(7,1,-1),  # strongest agreement is 1, weakest is 7
                           ordered = TRUE )       # and it's ordered
print( likert.ordinal )

## [1] 1 7 3 4 4 4 2 6 5 5
## Levels: 7 < 6 < 5 < 4 < 3 < 2 < 1

```

Notice that when we print out the ordered factor, R explicitly tells us what order the levels come in. Because I wanted to order my levels in terms of *increasing* strength of agreement, and because a response of 1 corresponded to the strongest agreement and 7 to the strongest disagreement, it was important that I tell R to encode 7 as the lowest value and 1 as the largest. Always check this when creating an ordered

²⁶I should note that if you type `class(xtab.3d)` you'll discover that this is a "table" object rather than an "array" object. However, this labelling is only skin deep. The underlying data structure here is actually an array. Advanced users may wish to check this using the command `class(unclass(xtab.3d))`, but it's not important for our purposes. All I really want to do in this section is show you what the output looks like when you encounter a 3D array.

factor: it's very easy to accidentally encode your data "upside down" if you're not paying attention. In any case, note that we can (and should) attach meaningful names to these factor levels by using the `levels()` function, like this:

```
levels( likert.ordinal ) <- c( "strong.disagree", "disagree", "weak.disagree",
                                "neutral", "weak.agree", "agree", "strong.agree" )
print( likert.ordinal )

## [1] strong.agree      strong.disagree  weak.agree       neutral
## [5] neutral          neutral        agree           disagree
## [9] weak.disagree    weak.disagree
## 7 Levels: strong.disagree < disagree < weak.disagree < ... < strong.agree
```

One nice thing about using ordered factors is that there are a lot of analyses for which R automatically treats ordered factors differently from unordered factors, and generally in a way that is more appropriate for ordinal data. However, since I don't discuss that in this book, I won't go into details. Like so many things in this chapter, my main goal here is to make you aware that R has this capability built into it; so if you ever need to start thinking about ordinal scale variables in more detail, you have at least some idea where to start looking!

7.11.3 Dates and times

Times and dates are very annoying types of data. To a first approximation we can say that there are 365 days in a year, 24 hours in a day, 60 minutes in an hour and 60 seconds in a minute, but that's not quite correct. The length of the solar day is not exactly 24 hours, and the length of solar year is not exactly 365 days, so we have a complicated system of corrections that have to be made to keep the time and date system working. On top of that, the measurement of time is usually taken relative to a local time zone, and most (but not all) time zones have both a standard time and a daylight savings time, though the date at which the switch occurs is not at all standardised. So, as a form of data, times and dates *suck*. Unfortunately, they're also important. Sometimes it's possible to avoid having to use any complicated system for dealing with times and dates. Often you just want to know what year something happened in, so you can just use numeric data: in quite a lot of situations something as simple as `this.year <- 2011` works just fine. If you can get away with that for your application, this is probably the best thing to do. However, sometimes you really do need to know the actual date. Or, even worse, the actual time. In this section, I'll very briefly introduce you to the basics of how R deals with date and time data. As with a lot of things in this chapter, I won't go into details because I don't use this kind of data anywhere else in the book. The goal here is to show you the basics of what you need to do if you ever encounter this kind of data in real life. And then we'll all agree never to speak of it again.

To start with, let's talk about the date. As it happens, modern operating systems are very good at keeping track of the time and date, and can even handle all those annoying timezone issues and daylight savings pretty well. So R takes the quite sensible view that it can just ask the operating system what the date is. We can pull the date using the `Sys.Date()` function:

```
today <- Sys.Date()  # ask the operating system for the date
print(today)          # display the date

## [1] "2018-12-29"
```

Okay, that seems straightforward. But, it does rather look like `today` is just a character string, doesn't it? That would be a problem, because dates really do have a numeric character to them, and it would be nice to be able to do basic addition and subtraction to them. Well, fear not. If you type in `class(today)`, R

will tell you that the class of the `today` variable is "Date". What this means is that, hidden underneath this text string that prints out an actual date, R actually has a numeric representation.²⁷ What that means is that you actually can add and subtract days. For instance, if we add 1 to `today`, R will print out the date for tomorrow:

```
today + 1
```

```
## [1] "2018-12-30"
```

Let's see what happens when we add 365 days:

```
today + 365
```

```
## [1] "2019-12-29"
```

This is particularly handy if you forget that a year is a leap year since in that case you'd probably get it wrong is doing this in your head. R provides a number of functions for working with dates, but I don't want to talk about them in any detail. I will, however, make passing mention of the `weekdays()` function which will tell you what day of the week a particular date corresponded to, which is extremely convenient in some situations:

```
weekdays( today )
```

```
## [1] "Saturday"
```

I'll also point out that you can use the `as.Date()` to convert various different kinds of data into dates. If the data happen to be strings formatted exactly according to the international standard notation (i.e., `yyyy-mm-dd`) then the conversion is straightforward, because that's the format that R expects to see by default. You can convert dates from other formats too, but it's slightly trickier, and beyond the scope of this book.

What about times? Well, times are even more annoying, so much so that I don't intend to talk about them at all in this book, other than to point you in the direction of some vaguely useful things. R itself does provide you with some tools for handling time data, and in fact there are two separate classes of data that are used to represent times, known by the odd names `POSIXct` and `POSIXlt`. You can use these to work with times if you want to, but for most applications you would probably be better off downloading the `chron` package, which provides some much more user friendly tools for working with times and dates.

7.12 Miscellaneous topics

To finish this chapter, I have a few topics to discuss that don't really fit in with any of the other things in this chapter. They're all kind of useful things to know about, but they are really just "odd topics" that don't fit with the other examples. Here goes:

²⁷Date objects are coded as the number of days that have passed since January 1, 1970.

7.12.1 The problems with floating point arithmetic

If I've learned nothing else about transfinite arithmetic (and I haven't) it's that infinity is a tedious and inconvenient concept. Not only is it annoying and counterintuitive at times, but it has nasty practical consequences. As we were all taught in high school, there are some numbers that *cannot* be represented as a decimal number of finite length, nor can they be represented as any kind of fraction between two whole numbers; $\sqrt{2}$, π and e , for instance. In everyday life we mostly don't care about this. I'm perfectly happy to approximate π as 3.14, quite frankly. Sure, this does produce some rounding errors from time to time, and if I'd used a more detailed approximation like 3.1415926535 I'd be less likely to run into those issues, but in all honesty I've never needed my calculations to be *that* precise. In other words, although our pencil and paper calculations cannot represent the number π exactly as a decimal number, we humans are smart enough to realise that we don't care. Computers, unfortunately, are dumb ... and you don't have to dig too deep in order to run into some very weird issues that arise because they can't represent numbers perfectly. Here is my favourite example:

```
0.1 + 0.2 == 0.3
```

```
## [1] FALSE
```

Obviously, R has made a mistake here, because this is definitely the wrong answer. Your first thought might be that R is broken, and you might be considering switching to some other language. But you can reproduce the same error in dozens of different programming languages, so the issue isn't specific to R. Your next thought might be that it's something in the hardware, but you can get the same mistake on any machine. It's something deeper than that.

The fundamental issue at hand is ***floating point arithmetic***, which is a fancy way of saying that computers will *always* round a number to fixed number of significant digits. The exact number of significant digits that the computer stores isn't important to us:²⁸ what matters is that whenever the number that the computer is trying to store is very long, you get rounding errors. That's actually what's happening with our example above. There are teeny tiny rounding errors that have appeared in the computer's storage of the numbers, and these rounding errors have in turn caused the internal storage of $0.1 + 0.2$ to be a tiny bit different from the internal storage of 0.3. How big are these differences? Let's ask R:

```
0.1 + 0.2 - 0.3
```

```
## [1] 5.551115e-17
```

Very tiny indeed. No sane person would care about differences that small. But R is not a sane person, and the equality operator `==` is very literal minded. It returns a value of `TRUE` only when the two values that it is given are absolutely identical to each other. And in this case they are not. However, this only answers half of the question. The other half of the question is, why are we getting these rounding errors when we're only using nice simple numbers like 0.1, 0.2 and 0.3? This seems a little counterintuitive. The answer is that, like most programming languages, R doesn't store numbers using their *decimal* expansion (i.e., base 10: using digits 0, 1, 2 ..., 9). We humans like to write our numbers in base 10 because we have 10 fingers. But computers don't have fingers, they have transistors; and transistors are built to store 2 numbers not 10. So you can see where this is going: the internal storage of a number in R is based on its *binary* expansion (i.e., base 2: using digits 0 and 1). And unfortunately, here's what the binary expansion of 0.1 looks like:

$$.1(\text{decimal}) = .00011001100110011\dots(\text{binary})$$

and the pattern continues forever. In other words, from the perspective of your computer, which likes to encode numbers in binary,²⁹ 0.1 is not a simple number at all. To a computer, 0.1 is actually an infinitely long binary number! As a consequence, the computer can make minor errors when doing calculations here.

²⁸For advanced users: type `?double` for more information.

²⁹Or at least, that's the default. If all your numbers are integers (whole numbers), then you can explicitly tell R to store them as integers by adding an L suffix at the end of the number. That is, an assignment like `x <- 2L` tells R to assign `x` a value of 2, and to store it as an integer rather than as a binary expansion. Type `?integer` for more details.

With any luck you now understand the problem, which ultimately comes down to the twin fact that (1) we usually think in decimal numbers and computers usually compute with binary numbers, and (2) computers are finite machines and can't store infinitely long numbers. The only questions that remain are when you should care and what you should do about it. Thankfully, you don't have to care very often: because the rounding errors are small, the only practical situation that I've seen this issue arise for is when you want to test whether an arithmetic fact holds exactly (e.g., is someone's response time equal to *exactly* 2×0.33 seconds?) This is pretty rare in real world data analysis, but just in case it does occur, it's better to use a test that allows for a small *tolerance*. That is, if the difference between the two numbers is below a certain threshold value, we deem them to be equal for all practical purposes. For instance, you could do something like this, which asks whether the difference between the two numbers is less than a tolerance of 10^{-10}

```
abs( 0.1 + 0.2 - 0.3 ) < 10^-10
```

```
## [1] TRUE
```

To deal with this problem, there is a function called `all.equal()` that lets you test for equality but allows a small tolerance for rounding errors:

```
all.equal( 0.1 + 0.2, 0.3 )
```

```
## [1] TRUE
```

7.12.2 The recycling rule

There's one thing that I haven't mentioned about how vector arithmetic works in R, and that's the *recycling rule*. The easiest way to explain it is to give a simple example. Suppose I have two vectors of different length, `x` and `y`, and I want to add them together. It's not obvious what that actually means, so let's have a look at what R does:

```
x <- c( 1,1,1,1,1,1 ) # x is length 6
y <- c( 0,1 )           # y is length 2
x + y                  # now add them:
```

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

As you can see from looking at this output, what R has done is "recycle" the value of the shorter vector (in this case `y`) several times. That is, the first element of `x` is added to the first element of `y`, and the second element of `x` is added to the second element of `y`. However, when R reaches the third element of `x` there isn't any corresponding element in `y`, so it returns to the beginning: thus, the third element of `x` is added to the *first* element of `y`. This process continues until R reaches the last element of `x`. And that's all there is to it really. The same recycling rule also applies for subtraction, multiplication and division. The only other thing I should note is that, if the length of the longer vector isn't an exact multiple of the length of the shorter one, R still does it, but also gives you a warning message:

```
x <- c( 1,1,1,1,1 ) # x is length 5
y <- c( 0,1 )         # y is length 2
x + y                # now add them:
```

```
## Warning in x + y: longer object length is not a multiple of shorter object
## length
```

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

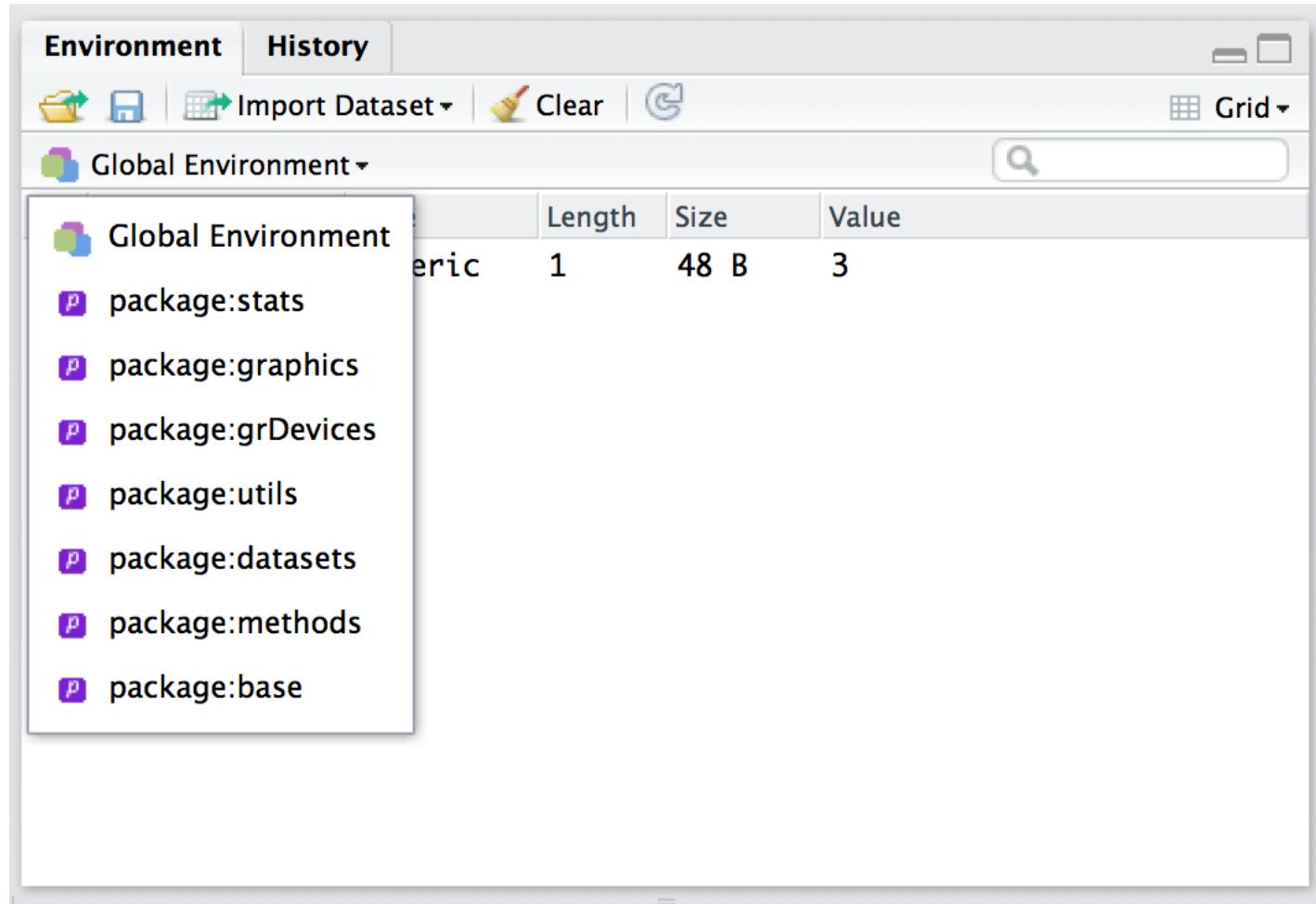


Figure 7.2: The environment panel in Rstudio can actually show you the contents of any loaded package: each package defines a separate environment, so you can select the one you want to look at in this panel.

7.12.3 An introduction to environments

In this section I want to ask a slightly different question: what *is* the workspace exactly? This question seems simple, but there's a fair bit to it. This section can be skipped if you're not really interested in the technical details. In the description I gave earlier, I talked about the workspace as an abstract location in which R variables are stored. That's basically true, but it hides a couple of key details. For example, any time you have R open, it has to store *lots* of things in the computer's memory, not just your variables. For example, the `who()` function that I wrote has to be stored in memory somewhere, right? If it weren't I wouldn't be able to use it. That's pretty obvious. But equally obviously it's not in the workspace either, otherwise you should have seen it! Here's what's happening. R needs to keep track of a lot of different things, so what it does is organise them into *environments*, each of which can contain lots of different variables and functions. Your workspace is one such environment. Every package that you have loaded is another environment. And every time you call a function, R briefly creates a temporary environment in which the function itself can work, which is then deleted after the calculations are complete. So, when I type in `search()` at the command line

```
search()
```

```
## [1] ".GlobalEnv"      "package:sciplot"    "package:HistData"
```

```
## [4] "package:MASS"      "package:lsr"       "package:psych"
## [7] "package:car"        "package:carData"   "package:stats"
## [10] "package:graphics"   "package:grDevices" "package:utils"
## [13] "package:datasets"   "package:methods"  "Autoloads"
## [16] "package:base"
```

what I'm actually looking at is a *sequence of environments*. The first one, ".GlobalEnv" is the technically-correct name for your workspace. No-one really calls it that: it's either called the workspace or the global environment. And so when you type in `objects()` or `who()` what you're really doing is listing the contents of ".GlobalEnv". But there's no reason why we can't look up the contents of these other environments using the `objects()` function (currently `who()` doesn't support this). You just have to be a bit more explicit in your command. If I wanted to find out what is in the `package:stats` environment (i.e., the environment into which the contents of the `stats` package have been loaded), here's what I'd get

```
head(objects("package:stats"))
```

```
## [1] "acf"          "acf2AR"        "add.scope"     "add1"         "addmargins"
## [6] "aggregate"
```

where this time I've used `head()` to hide a lot of output because the `stats` package contains about 500 functions. In fact, you can actually use the environment panel in Rstudio to browse any of your loaded packages (just click on the text that says "Global Environment" and you'll see a dropdown menu like the one shown in Figure 7.2). The key thing to understand then, is that you can access any of the R variables and functions that are stored in one of these environments, precisely because those are the environments that you have loaded.³⁰

7.12.4 Attaching a data frame

The last thing I want to mention in this section is the `attach()` function, which you often see referred to in introductory R books. Whenever it is introduced, the author of the book usually mentions that the `attach()` function can be used to "attach" the data frame to the search path, so you don't have to use the `$` operator. That is, if I use the command `attach(df)` to attach my data frame, I no longer need to type `df$variable`, and instead I can just type `variable`. This is true as far as it goes, but it's very misleading and novice users often get led astray by this description, because it hides a lot of critical details.

Here is the very abridged description: when you use the `attach()` function, what R does is create an entirely new *environment* in the search path, just like when you load a package. Then, what it does is *copy* all of the variables in your data frame into this new environment. When you do this, however, you end up with two completely different versions of all your variables: one in the original data frame, and one in the new environment. Whenever you make a statement like `df$variable` you're working with the variable inside the data frame; but when you just type `variable` you're working with the copy in the new environment. And here's the part that really upsets new users: *changes to one version are not reflected in the other version*. As a consequence, it's really easy for R to end up with different value stored in the two different locations, and you end up really confused as a result.

To be fair to the writers of the `attach()` function, the help documentation does actually state all this quite explicitly, and they even give some examples of how this can cause confusion at the bottom of the help page. And I can actually see how it can be very useful to create copies of your data in a separate location (e.g., it lets you make all kinds of modifications and deletions to the data without having to touch the original

³⁰For advanced users: that's a little over simplistic in two respects. First, it's a terribly imprecise way of talking about scoping. Second, it might give you the impression that all the variables in question are actually loaded into memory. That's not quite true, since that would be very wasteful of memory. Instead R has a "lazy loading" mechanism, in which what R actually does is create a "promise" to load those objects if they're actually needed. For details, check out the `delayedAssign()` function.

data frame). However, I don't think it's helpful for new users, since it means you have to be very careful to keep track of which copy you're talking about. As a consequence of all this, for the purpose of this book I've decided not to use the `attach()` function. It's something that you can investigate yourself once you're feeling a little more confident with R, but I won't do it here.

7.13 Summary

Obviously, there's no real coherence to this chapter. It's just a grab bag of topics and tricks that can be handy to know about, so the best wrap up I can give here is just to repeat this list:

- Section 7.1. Tabulating data.
- Section 7.2. Transforming or recoding a variable.
- Section 7.3. Some useful mathematical functions.
- Section 7.4. Extracting a subset of a vector.
- Section 7.5. Extracting a subset of a data frame.
- Section 7.6. Sorting, flipping or merging data sets.
- Section 7.7. Reshaping a data frame.
- Section 7.8. Manipulating text.
- Section 7.9. Opening data from different file types.
- Section 7.10. Coercing data from one type to another.
- Section 7.11. Other important data types.
- Section 7.12. Miscellaneous topics.

There are a number of books out there that extend this discussion. A couple of my favourites are Spector (2008) "Data Manipulation with R" and Teator (2011) "R Cookbook".

Chapter 8

Basic programming

Machine dreams hold a special vertigo.

—William Gibson¹

Up to this point in the book I've tried hard to avoid using the word "programming" too much because – at least in my experience – it's a word that can cause a lot of fear. For one reason or another, programming (like mathematics and statistics) is often perceived by people on the "outside" as a black art, a magical skill that can be learned only by some kind of super-nerd. I think this is a shame. It's certainly true that advanced programming is a very specialised skill: several different skills actually, since there's quite a lot of different kinds of programming out there. However, the *basics* of programming aren't all that hard, and you can accomplish a lot of very impressive things just using those basics.

With that in mind, the goal of this chapter is to discuss a few basic programming concepts and how to apply them in R. However, before I do, I want to make one further attempt to point out just how non-magical programming really is, via one very simple observation: *you already know how to do it*. Stripped to its essentials, programming is nothing more (and nothing less) than the process of writing out a bunch of instructions that a computer can understand. To phrase this slightly differently, when you write a computer program, you need to write it in a *programming language* that the computer knows how to interpret. R is one such language. Although I've been having you type all your commands at the command prompt, and all the commands in this book so far have been shown as if that's what I were doing, it's also quite possible (and as you'll see shortly, shockingly easy) to write a program using these R commands. In other words, if this is the first time reading this book, then you're only one short chapter away from being able to legitimately claim that you can program in R, albeit at a beginner's level.

8.1 Scripts

Computer programs come in quite a few different forms: the kind of program that we're mostly interested in from the perspective of everyday data analysis using R is known as a *script*. The idea behind a script is that, instead of typing your commands into the R console one at a time, instead you write them all in a text file. Then, once you've finished writing them and saved the text file, you can get R to execute all the commands in your file by using the `source()` function. In a moment I'll show you exactly how this is done, but first I'd better explain why you should care.

¹The quote comes from *Count Zero* (1986)

8.1.1 Why use scripts?

Before discussing scripting and programming concepts in any more detail, it's worth stopping to ask why you should bother. After all, if you look at the R commands that I've used everywhere else this book, you'll notice that they're all formatted as if I were typing them at the command line. Outside this chapter you won't actually see any scripts. Do not be fooled by this. The reason that I've done it that way is purely for pedagogical reasons. My goal in this book is to teach statistics and to teach R. To that end, what *I've* needed to do is chop everything up into tiny little slices: each section tends to focus on one kind of statistical concept, and only a smallish number of R functions. As much as possible, I want you to see what each function does in isolation, one command at a time. By forcing myself to write everything as if it were being typed at the command line, it imposes a kind of discipline on me: it *prevents* me from piecing together lots of commands into one big script. From a teaching (and learning) perspective I think that's the right thing to do... but from a *data analysis* perspective, it is not. When you start analysing real world data sets, you will rapidly find yourself needing to write scripts.

To understand why scripts are so very useful, it may be helpful to consider the drawbacks to typing commands directly at the command prompt. The approach that we've been adopting so far, in which you type commands one at a time, and R sits there patiently in between commands, is referred to as the *interactive* style. Doing your data analysis this way is rather like having a conversation ... a very annoying conversation between you and your data set, in which you and the data aren't directly speaking to each other, and so you have to rely on R to pass messages back and forth. This approach makes a lot of sense when you're just trying out a few ideas: maybe you're trying to figure out what analyses are sensible for your data, or maybe just you're trying to remember how the various R functions work, so you're just typing in a few commands until you get the one you want. In other words, the interactive style is very useful as a tool for exploring your data. However, it has a number of drawbacks:

- *It's hard to save your work effectively.* You can save the workspace, so that later on you can load any variables you created. You can save your plots as images. And you can even save the history or copy the contents of the R console to a file. Taken together, all these things let you create a reasonably decent record of what you did. But it does leave a lot to be desired. It seems like you ought to be able to save a single file that R could use (in conjunction with your raw data files) and reproduce everything (or at least, everything interesting) that you did during your data analysis.
- *It's annoying to have to go back to the beginning when you make a mistake.* Suppose you've just spent the last two hours typing in commands. Over the course of this time you've created lots of new variables and run lots of analyses. Then suddenly you realise that there was a nasty typo in the first command you typed, so all of your later numbers are wrong. Now you have to fix that first command, and then spend another hour or so combing through the R history to try and recreate what you did.
- *You can't leave notes for yourself.* Sure, you can scribble down some notes on a piece of paper, or even save a Word document that summarises what you did. But what you really want to be able to do is write down an English translation of your R commands, preferably right "next to" the commands themselves. That way, you can look back at what you've done and actually remember what you were doing. In the simple exercises we've engaged in so far, it hasn't been all that hard to remember what you were doing or why you were doing it, but only because everything we've done could be done using only a few commands, and you've never been asked to reproduce your analysis six months after you originally did it! When your data analysis starts involving hundreds of variables, and requires quite complicated commands to work, then you really, really need to leave yourself some notes to explain your analysis to, well, yourself.
- *It's nearly impossible to reuse your analyses later, or adapt them to similar problems.* Suppose that, sometime in January, you are handed a difficult data analysis problem. After working on it for ages, you figure out some really clever tricks that can be used to solve it. Then, in September, you get handed a really similar problem. You can sort of remember what you did, but not very well. You'd like to have a clean record of what you did last time, how you did it, and why you did it the way you did. Something like that would really help you solve this new problem.

- *It's hard to do anything except the basics.* There's a nasty side effect of these problems. Typos are inevitable. Even the best data analyst in the world makes a lot of mistakes. So the chance that you'll be able to string together dozens of correct R commands in a row are very small. So unless you have some way around this problem, you'll never really be able to do anything other than simple analyses.
- *It's difficult to share your work other people.* Because you don't have this nice clean record of what R commands were involved in your analysis, it's not easy to share your work with other people. Sure, you can send them all the data files you've saved, and your history and console logs, and even the little notes you wrote to yourself, but odds are pretty good that no-one else will really understand what's going on (trust me on this: I've been handed lots of random bits of output from people who've been analysing their data, and it makes very little sense unless you've got the original person who did the work sitting right next to you explaining what you're looking at)

Ideally, what you'd like to be able to do is something like this... Suppose you start out with a data set `myrawdata.csv`. What you want is a single document – let's call it `mydataanalysis.R` – that stores all of the commands that you've used in order to do your data analysis. Kind of similar to the R history but much more focused. It would only include the commands that you want to keep for later. Then, later on, instead of typing in all those commands again, you'd just tell R to run all of the commands that are stored in `mydataanalysis.R`. Also, in order to help you make sense of all those commands, what you'd want is the ability to add some notes or *comments* within the file, so that anyone reading the document for themselves would be able to understand what each of the commands actually does. But these comments wouldn't get in the way: when you try to get R to run `mydataanalysis.R` it would be smart enough to recognise that these comments are for the benefit of humans, and so it would ignore them. Later on you could tweak a few of the commands inside the file (maybe in a new file called `mynewdatanalaysis.R`) so that you can adapt an old analysis to be able to handle a new problem. And you could email your friends and colleagues a copy of this file so that they can reproduce your analysis themselves.

In other words, what you want is a *script*.

8.1.2 Our first script

Okay then. Since scripts are so terribly awesome, let's write one. To do this, open up a simple text editing program, like TextEdit (on a Mac) or Notebook (on Windows). Don't use a fancy word processing program like Microsoft Word or OpenOffice: use the simplest program you can find. Open a new text document, and type some R commands, hitting enter after each command. Let's try using `x <- "hello world"` and `print(x)` as our commands. Then save the document as `hello.R`, and remember to save it as a plain text file: don't save it as a word document or a rich text file. Just a boring old plain text file. Also, when it asks you *where* to save the file, save it to whatever folder you're using as your working directory in R. At this point, you should be looking at something like Figure 8.1. And if so, you have now successfully written your first R program. Because I don't want to take screenshots for every single script, I'm going to present scripts using extracts formatted as follows:

```
## --- hello.R
x <- "hello world"
print(x)
```

The line at the top is the filename, and not part of the script itself. Below that, you can see the two R commands that make up the script itself. Next to each command I've included the line numbers. You don't actually type these into your script, but a lot of text editors (including the one built into Rstudio that I'll show you in a moment) will show line numbers, since it's a very useful convention that allows you to say things like “line 1 of the script creates a new variable, and line 2 prints it out”.

So how do we run the script? Assuming that the `hello.R` file has been saved to your working directory, then you can run the script using the following command:

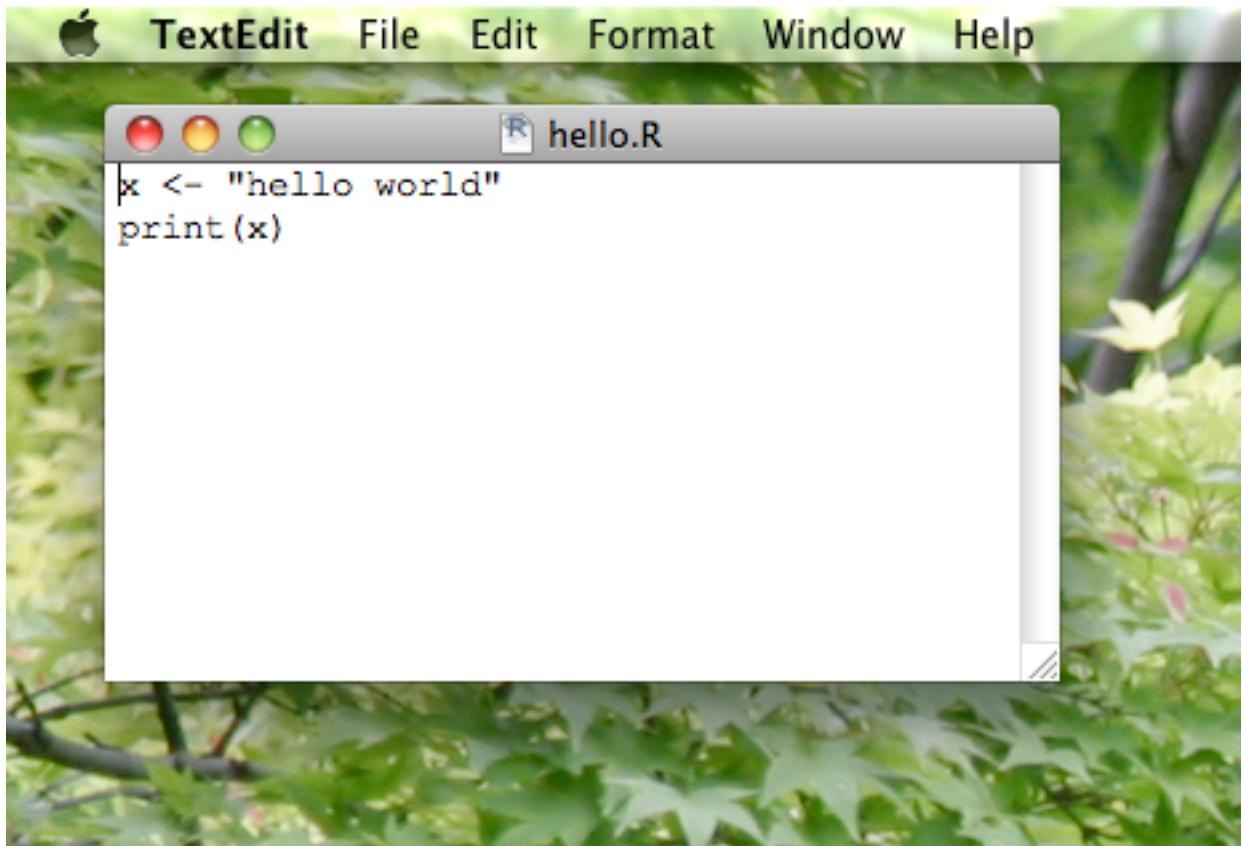


Figure 8.1: A screenshot showing the ‘hello.R’ script if you open it using the default text editor (TextEdit) on a Mac. Using a simple text editor like TextEdit on a Mac or Notepad on Windows isn’t actually the best way to write your scripts, but it is the simplest. More to the point, it highlights the fact that a script really is just an ordinary text file.

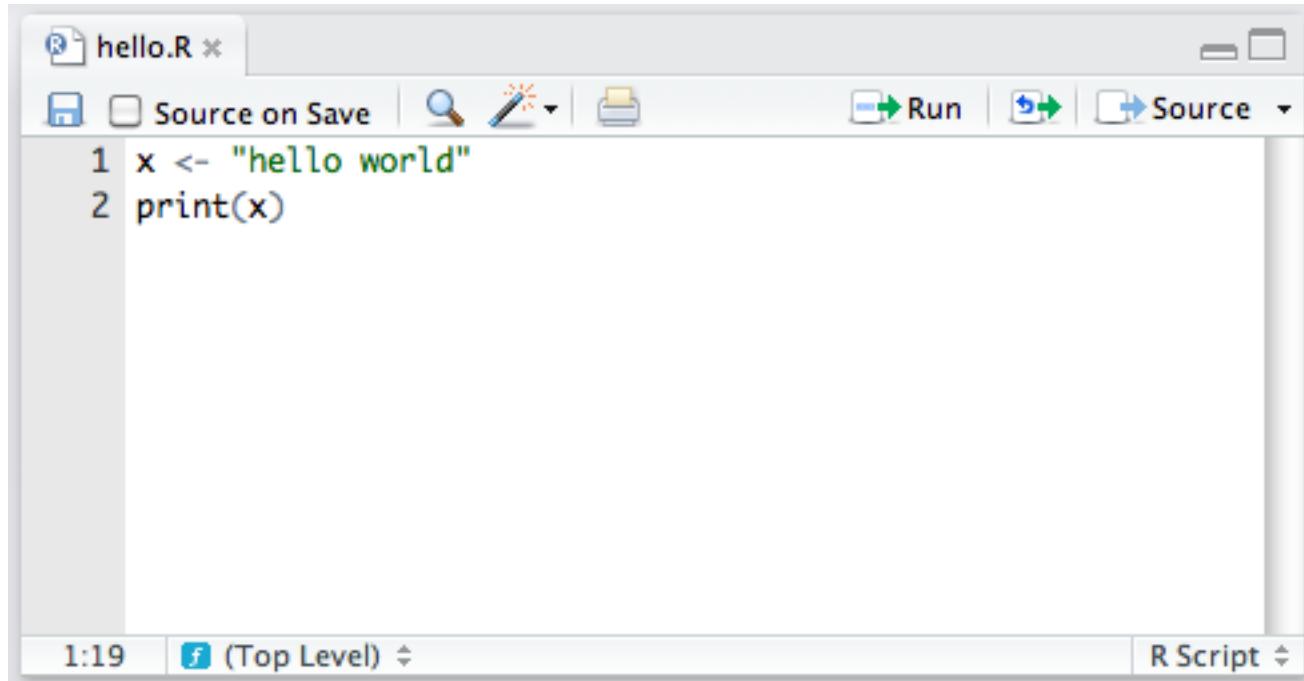


Figure 8.2: A screenshot showing the ‘hello.R’ script open in Rstudio. Assuming that you’re looking at this document in colour, you’ll notice that the “hello world” text is shown in green. This isn’t something that you do yourself: that’s Rstudio being helpful. Because the text editor in Rstudio “knows” something about how R commands work, it will highlight different parts of your script in different colours. This is useful, but it’s not actually part of the script itself.

```
source( "hello.R" )
```

If the script file is saved in a different directory, then you need to specify the path to the file, in exactly the same way that you would have to when loading a data file using `load()`. In any case, when you type this command, R opens up the script file: it then reads each command in the file in the same order that they appear in the file, and executes those commands in that order. The simple script that I’ve shown above contains two commands. The first one creates a variable `x` and the second one prints it on screen. So, when we run the script, this is what we see on screen:

```
source("./rbook-master/scripts/hello.R")
```

```
## [1] "hello world"
```

If we inspect the workspace using a command like `who()` or `objects()`, we discover that R has created the new variable `x` within the workspace, and not surprisingly `x` is a character string containing the text “`hello world`”. And just like that, you’ve written your first program R. It really is that simple.

8.1.3 Using Rstudio to write scripts

In the example above I assumed that you were writing your scripts using a simple text editor. However, it’s usually more convenient to use a text editor that is specifically designed to help you write scripts. There’s a lot of these out there, and experienced programmers will all have their own personal favourites. For our purposes, however, we can just use the one built into R studio. To create new script file in R studio, go to the

“File” menu, select the “New” option, and then click on “R script”. This will open a new window within the “source” panel. Then you can type the commands you want (or **code** as it is generally called when you’re typing the commands into a script file) and save it when you’re done. The nice thing about using Rstudio to do this is that it automatically changes the colour of the text to indicate which parts of the code are comments and which are parts are actual R commands (these colours are called *syntax highlighting*, but they’re not actually part of the file – it’s just Rstudio trying to be helpful). To see an example of this, let’s open up our `hello.R` script in Rstudio. To do this, go to the “File” menu again, and select “Open...”. Once you’ve opened the file, you should be looking at something like Figure 8.2. As you can see (if you’re looking at this book in colour) the character string “hello world” is highlighted in green.

Using Rstudio for your text editor is convenient for other reasons too. Notice in the top right hand corner of Figure 8.2 there’s a little button that reads “Source”? If you click on that, Rstudio will construct the relevant `source()` command for you, and send it straight to the R console. So you don’t even have to type in the `source()` command, which actually I think is a great thing, because it really bugs me having to type all those extra keystrokes every time I want to run my script. Anyway, Rstudio provide several other convenient little tools to help make scripting easier, but I won’t discuss them here.²

8.1.4 Commenting your script

When writing up your data analysis as a script, one thing that is generally a good idea is to include a lot of comments in the code. That way, if someone else tries to read it (or if you come back to it several days, weeks, months or years later) they can figure out what’s going on. As a beginner, I think it’s especially useful to comment thoroughly, partly because it gets you into the habit of commenting the code, and partly because the simple act of typing in an explanation of what the code does will help you keep it clear in your own mind what you’re trying to achieve. To illustrate this idea, consider the following script:

```
## --- itngscript.R
# A script to analyse nightgarden.Rdata_
# author: Dan Navarro_
# date: 22/11/2011_

# Load the data, and tell the user that this is what we're
# doing.
cat( "loading data from nightgarden.Rdata...\n" )
load( "./rbook-master/data/nightgarden.Rdata" )

# Create a cross tabulation and print it out:
cat( "tabulating data...\n" )
itng.table <- table( speaker, utterance )
print( itng.table )
```

You’ll notice that I’ve gone a bit overboard with my commenting: at the top of the script I’ve explained the purpose of the script, who wrote it, and when it was written. Then, throughout the script file itself I’ve added a lot of comments explaining what each section of the code actually does. In real life people don’t tend to comment this thoroughly, but the basic idea is a very good one: you really do want your script to explain itself. Nevertheless, as you’d expect R completely ignores all of the commented parts. When we run this script, this is what we see on screen:

²Okay, I lied. Sue me. One of the coolest features of Rstudio is the support for *R Markdown*, which lets you embed R code inside a Markdown document, and you can automatically publish your R Markdown to the web on Rstudio’s servers. If you’re the kind of nerd interested in this sort of thing, it’s really nice. And, yes, since I’m also that kind of nerd, of course I’m aware that iPython notebooks do the same thing and that R just nicked their idea. So what? It’s still cool. And anyway, this book isn’t called *Learning Statistics with Python* now, is it? Hm. Maybe I should write a Python version...

```

## --- itngscript.R
# A script to analyse nightgarden.Rdata
# author: Dan Navarro
# date: 22/11/2011

# Load the data, and tell the user that this is what we're
# doing.
cat( "loading data from nightgarden.Rdata...\n" )

## loading data from nightgarden.Rdata...

load( "./rbook-master/data/nightgarden.Rdata" )

# Create a cross tabulation and print it out:
cat( "tabulating data...\n" )

## tabulating data...

itng.table <- table( speaker, utterance )
print( itng.table )

##          utterance
## speaker      ee onk oo pip
##   makka-pakka  0   2   0   2
##   tombliboo    1   0   1   0
##   upsy-daisy   0   2   0   2

```

Even here, notice that the script announces its behaviour. The first two lines of the output tell us a lot about what the script is actually doing behind the scenes (the code do to this corresponds to the two `cat()` commands on lines 8 and 12 of the script). It's usually a pretty good idea to do this, since it helps ensure that the output makes sense when the script is executed.

8.1.5 Differences between scripts and the command line

For the most part, commands that you insert into a script behave in exactly the same way as they would if you typed the same thing in at the command line. The one major exception to this is that if you want a variable to be printed on screen, you need to explicitly tell R to print it. You can't just type the name of the variable. For example, our original `hello.R` script produced visible output. The following script does not:

```

## --- silenthello.R
x <- "hello world"
x

```

It *does* still create the variable `x` when you `source()` the script, but it won't print anything on screen.

However, apart from the fact that scripts don't use “auto-printing” as it's called, there aren't a lot of differences in the underlying mechanics. There are a few stylistic differences though. For instance, if you want to load a package at the command line, you would generally use the `library()` function. If you want do to it from a script, it's conventional to use `require()` instead. The two commands are basically identical, the only difference being that if the package doesn't exist, `require()` produces a warning whereas `library()` gives you an error. Stylistically, what this means is that if the `require()` command fails in your script, R will boldly continue on and try to execute the rest of the script. Often that's what you'd like to see happen, so it's better to use `require()`. Clearly, however, you can get by just fine using the `library()` command for everyday usage.

8.1.6 Done!

At this point, you've learned the basics of scripting. You are now officially allowed to say that you can program in R, though you probably shouldn't say it too loudly. There's a *lot* more to learn, but nevertheless, if you can write scripts like these then what you are doing is in fact basic programming. The rest of this chapter is devoted to introducing some of the key commands that you need in order to make your programs more powerful; and to help you get used to thinking in terms of scripts, for the rest of this chapter I'll write up most of my extracts as scripts.

8.2 Loops

The description I gave earlier for how a script works was a tiny bit of a lie. Specifically, it's not necessarily the case that R starts at the top of the file and runs straight through to the end of the file. For all the scripts that we've seen so far that's exactly what happens, and unless you insert some commands to explicitly alter how the script runs, that is what will *always* happen. However, you actually have quite a lot of flexibility in this respect. Depending on how you write the script, you can have R repeat several commands, or skip over different commands, and so on. This topic is referred to as *flow control*, and the first concept to discuss in this respect is the idea of a *loop*. The basic idea is very simple: a loop is a block of code (i.e., a sequence of commands) that R will execute over and over again until some termination criterion is met. Looping is a very powerful idea. There are three different ways to construct a loop in R, based on the `while`, `for` and `repeat` functions. I'll only discuss the first two in this book.

8.2.1 The `while` loop

A `while` loop is a simple thing. The basic format of the loop looks like this:

```
while ( CONDITION ) {
  STATEMENT1
  STATEMENT2
  ETC
}
```

The code corresponding to CONDITION needs to produce a logical value, either `TRUE` or `FALSE`. Whenever R encounters a `while` statement, it checks to see if the CONDITION is `TRUE`. If it is, then R goes on to execute all of the commands inside the curly brackets, proceeding from top to bottom as usual. However, when it gets to the bottom of those statements, it moves back up to the `while` statement. Then, like the mindless automaton it is, it checks to see if the CONDITION is `TRUE`. If it is, then R goes on to execute all ... well, you get the idea. This continues endlessly until at some point the CONDITION turns out to be `FALSE`. Once that happens, R jumps to the bottom of the loop (i.e., to the `}` character), and then continues on with whatever commands appear next in the script.

To start with, let's keep things simple, and use a `while` loop to calculate the smallest multiple of 17 that is greater than or equal to 1000. This is a very silly example since you can actually calculate it using simple arithmetic operations, but the point here isn't to do something novel. The point is to show how to write a `while` loop. Here's the script:

```
## --- whileexample.R
x <- 0
while ( x < 1000 ) {
  x <- x + 17
}
print( x )
```

When we run this script, R starts at the top and creates a new variable called `x` and assigns it a value of 0. It then moves down to the loop, and “notices” that the condition here is `x < 1000`. Since the current value of `x` is zero, the condition is true, so it enters the body of the loop (inside the curly braces). There’s only one command here³ which instructs R to increase the value of `x` by 17. R then returns to the top of the loop, and rechecks the condition. The value of `x` is now 17, but that’s still less than 1000, so the loop continues. This cycle will continue for a total of 59 iterations, until finally `x` reaches a value of 1003 (i.e., $59 \times 17 = 1003$). At this point, the loop stops, and R finally reaches line 5 of the script, prints out the value of `x` on screen, and then halts. Let’s watch:

```
source( "./rbook-master/scripts/whileexample.R" )
```

```
## [1] 1003
```

Truly fascinating stuff.

8.2.2 The `for` loop

The `for` loop is also pretty simple, though not quite as simple as the `while` loop. The basic format of this loop goes like this:

```
for ( VAR in VECTOR ) {
    STATEMENT1
    STATEMENT2
    ETC
}
```

In a `for` loop, R runs a fixed number of iterations. We have a `VECTOR` which has several elements, each one corresponding to a possible value of the variable `VAR`. In the first iteration of the loop, `VAR` is given a value corresponding to the first element of `VECTOR`; in the second iteration of the loop `VAR` gets a value corresponding to the second value in `VECTOR`; and so on. Once we’ve exhausted all of the values in `VECTOR`, the loop terminates and the flow of the program continues down the script.

Once again, let’s use some very simple examples. Firstly, here is a program that just prints out the word “hello” three times and then stops:

```
## --- forexample.R
for ( i in 1:3 ) {
    print( "hello" )
}
```

This is the simplest example of a `for` loop. The vector of possible values for the `i` variable just corresponds to the numbers from 1 to 3. Not only that, the body of the loop doesn’t actually depend on `i` at all. Not surprisingly, here’s what happens when we run it:

```
source( "./rbook-master/scripts/forexample.R" )
```

```
## [1] "hello"
## [1] "hello"
## [1] "hello"
```

³As an aside: if there’s only a single command that you want to include inside your loop, then you don’t actually need to bother including the curly braces at all. However, until you’re comfortable programming in R I’d advise *always* using them, even when you don’t have to.

However, there's nothing that stops you from using something non-numeric as the vector of possible values, as the following example illustrates. This time around, we'll use a character vector to control our loop, which in this case will be a vector of words. And what we'll do in the loop is get R to convert the word to upper case letters, calculate the length of the word, and print it out. Here's the script:

```
## --- forexample2.R

#the words_
words <- c("it", "was", "the", "dirty", "end", "of", "winter")

#loop over the words_
for ( w in words ) {

  w.length <- nchar( w )      # calculate the number of letters_
  W <- toupper( w )           # convert the word to upper case letters_
  msg <- paste( W, "has", w.length, "letters" )  # a message to print_
  print( msg )                # print it_

}
```

And here's the output:

```
source( "./rbook-master/scripts/forexample2.R" )

## [1] "IT has 2 letters"
## [1] "WAS has 3 letters"
## [1] "THE has 3 letters"
## [1] "DIRTY has 5 letters"
## [1] "END has 3 letters"
## [1] "OF has 2 letters"
## [1] "WINTER has 6 letters"
```

Again, pretty straightforward I hope.

8.2.3 A more realistic example of a loop

To give you a sense of how you can use a loop in a more complex situation, let's write a simple script to simulate the progression of a mortgage. Suppose we have a nice young couple who borrow \$300000 from the bank, at an annual interest rate of 5%. The mortgage is a 30 year loan, so they need to pay it off within 360 months total. Our happy couple decide to set their monthly mortgage payment at \$1600 per month. Will they pay off the loan in time or not? Only time will tell.⁴ Or, alternatively, we could simulate the whole process and get R to tell us. The script to run this is a fair bit more complicated.

```
## --- mortgage.R

# set up
month <- 0      # count the number of months
balance <- 300000 # initial mortgage balance
```

⁴Okay, fine. This example is still a bit ridiculous, in three respects. Firstly, the bank absolutely will not let the couple pay less than the amount required to terminate the loan in 30 years. Secondly, a constant interest rate of 30 years is hilarious. Thirdly, you can solve this much more efficiently than through brute force simulation. However, we're not exactly in the business of being realistic or efficient here.

```

payments <- 1600 # monthly payments
interest <- 0.05 # 5% interest rate per year
total.paid <- 0 # track what you've paid the bank

# convert annual interest to a monthly multiplier
monthly.multiplier <- (1+interest) ^ (1/12)

# keep looping until the loan is paid off...
while ( balance > 0 ) {

    # do the calculations for this month
    month <- month + 1 # one more month
    balance <- balance * monthly.multiplier # add the interest
    balance <- balance - payments # make the payments
    total.paid <- total.paid + payments # track the total paid

    # print the results on screen
    cat( "month", month, ": balance", round(balance), "\n" )

} # end of loop

# print the total payments at the end
cat("total payments made", total.paid, "\n" )

```

To explain what's going on, let's go through it carefully. In the first block of code (under `#set up`) all we're doing is specifying all the variables that define the problem. The loan starts with a `balance` of \$300,000 owed to the bank on `month` zero, and at that point in time the `total.paid` money is nothing. The couple is making monthly `payments` of \$1600, at an annual `interest` rate of 5%. Next, we convert the annual percentage interest into a monthly multiplier. That is, the number that you have to multiply the current balance by each month in order to produce an annual interest rate of 5%. An annual interest rate of 5% implies that, if no payments were made over 12 months the balance would end up being 1.05 times what it was originally, so the *annual* multiplier is 1.05. To calculate the monthly multiplier, we need to calculate the 12th root of 1.05 (i.e., raise 1.05 to the power of 1/12). We store this value in as the `monthly.multiplier` variable, which as it happens corresponds to a value of about 1.004. All of which is a rather long winded way of saying that the *annual* interest rate of 5% corresponds to a *monthly* interest rate of about 0.4%.

Anyway... all of that is really just setting the stage. It's not the interesting part of the script. The interesting part (such as it is) is the loop. The `while` statement on tells R that it needs to keep looping until the `balance` reaches zero (or less, since it might be that the final payment of \$1600 pushes the balance below zero). Then, inside the body of the loop, we have two different blocks of code. In the first bit, we do all the number crunching. Firstly we increase the value `month` by 1. Next, the bank charges the interest, so the `balance` goes up. Then, the couple makes their monthly payment and the `balance` goes down. Finally, we keep track of the total amount of money that the couple has paid so far, by adding the `payments` to the running tally. After having done all this number crunching, we tell R to issue the couple with a very terse monthly statement, which just indicates how many months they've been paying the loan and how much money they still owe the bank. Which is rather rude of us really. I've grown attached to this couple and I really feel they deserve better than that. But, that's banks for you.

In any case, the key thing here is the tension between the increase in `balance` on and the decrease. As long as the decrease is bigger, then the balance will eventually drop to zero and the loop will eventually terminate. If not, the loop will continue forever! This is actually very bad programming on my part: I really should have included something to force R to stop if this goes on too long. However, I haven't shown you how to evaluate "if" statements yet, so we'll just have to hope that the author of the book has rigged the example so that the code actually runs. Hm. I wonder what the odds of that are? Anyway, assuming that the loop

does eventually terminate, there's one last line of code that prints out the total amount of money that the couple handed over to the bank over the lifetime of the loan.

Now that I've explained everything in the script in tedious detail, let's run it and see what happens:

```
source( "./rbook-master/scripts/mortgage.R" )
```

```
## month 1 : balance 299622
## month 2 : balance 299243
## month 3 : balance 298862
## month 4 : balance 298480
## month 5 : balance 298096
## month 6 : balance 297710
## month 7 : balance 297323
## month 8 : balance 296934
## month 9 : balance 296544
## month 10 : balance 296152
## month 11 : balance 295759
## month 12 : balance 295364
## month 13 : balance 294967
## month 14 : balance 294569
## month 15 : balance 294169
## month 16 : balance 293768
## month 17 : balance 293364
## month 18 : balance 292960
## month 19 : balance 292553
## month 20 : balance 292145
## month 21 : balance 291735
## month 22 : balance 291324
## month 23 : balance 290911
## month 24 : balance 290496
## month 25 : balance 290079
## month 26 : balance 289661
## month 27 : balance 289241
## month 28 : balance 288820
## month 29 : balance 288396
## month 30 : balance 287971
## month 31 : balance 287545
## month 32 : balance 287116
## month 33 : balance 286686
## month 34 : balance 286254
## month 35 : balance 285820
## month 36 : balance 285385
## month 37 : balance 284947
## month 38 : balance 284508
## month 39 : balance 284067
## month 40 : balance 283625
## month 41 : balance 283180
## month 42 : balance 282734
## month 43 : balance 282286
## month 44 : balance 281836
## month 45 : balance 281384
## month 46 : balance 280930
## month 47 : balance 280475
```

```
## month 48 : balance 280018
## month 49 : balance 279559
## month 50 : balance 279098
## month 51 : balance 278635
## month 52 : balance 278170
## month 53 : balance 277703
## month 54 : balance 277234
## month 55 : balance 276764
## month 56 : balance 276292
## month 57 : balance 275817
## month 58 : balance 275341
## month 59 : balance 274863
## month 60 : balance 274382
## month 61 : balance 273900
## month 62 : balance 273416
## month 63 : balance 272930
## month 64 : balance 272442
## month 65 : balance 271952
## month 66 : balance 271460
## month 67 : balance 270966
## month 68 : balance 270470
## month 69 : balance 269972
## month 70 : balance 269472
## month 71 : balance 268970
## month 72 : balance 268465
## month 73 : balance 267959
## month 74 : balance 267451
## month 75 : balance 266941
## month 76 : balance 266428
## month 77 : balance 265914
## month 78 : balance 265397
## month 79 : balance 264878
## month 80 : balance 264357
## month 81 : balance 263834
## month 82 : balance 263309
## month 83 : balance 262782
## month 84 : balance 262253
## month 85 : balance 261721
## month 86 : balance 261187
## month 87 : balance 260651
## month 88 : balance 260113
## month 89 : balance 259573
## month 90 : balance 259031
## month 91 : balance 258486
## month 92 : balance 257939
## month 93 : balance 257390
## month 94 : balance 256839
## month 95 : balance 256285
## month 96 : balance 255729
## month 97 : balance 255171
## month 98 : balance 254611
## month 99 : balance 254048
## month 100 : balance 253483
## month 101 : balance 252916
```

```
## month 102 : balance 252346
## month 103 : balance 251774
## month 104 : balance 251200
## month 105 : balance 250623
## month 106 : balance 250044
## month 107 : balance 249463
## month 108 : balance 248879
## month 109 : balance 248293
## month 110 : balance 247705
## month 111 : balance 247114
## month 112 : balance 246521
## month 113 : balance 245925
## month 114 : balance 245327
## month 115 : balance 244727
## month 116 : balance 244124
## month 117 : balance 243518
## month 118 : balance 242911
## month 119 : balance 242300
## month 120 : balance 241687
## month 121 : balance 241072
## month 122 : balance 240454
## month 123 : balance 239834
## month 124 : balance 239211
## month 125 : balance 238585
## month 126 : balance 237958
## month 127 : balance 237327
## month 128 : balance 236694
## month 129 : balance 236058
## month 130 : balance 235420
## month 131 : balance 234779
## month 132 : balance 234136
## month 133 : balance 233489
## month 134 : balance 232841
## month 135 : balance 232189
## month 136 : balance 231535
## month 137 : balance 230879
## month 138 : balance 230219
## month 139 : balance 229557
## month 140 : balance 228892
## month 141 : balance 228225
## month 142 : balance 227555
## month 143 : balance 226882
## month 144 : balance 226206
## month 145 : balance 225528
## month 146 : balance 224847
## month 147 : balance 224163
## month 148 : balance 223476
## month 149 : balance 222786
## month 150 : balance 222094
## month 151 : balance 221399
## month 152 : balance 220701
## month 153 : balance 220000
## month 154 : balance 219296
## month 155 : balance 218590
```

```
## month 156 : balance 217880
## month 157 : balance 217168
## month 158 : balance 216453
## month 159 : balance 215735
## month 160 : balance 215014
## month 161 : balance 214290
## month 162 : balance 213563
## month 163 : balance 212833
## month 164 : balance 212100
## month 165 : balance 211364
## month 166 : balance 210625
## month 167 : balance 209883
## month 168 : balance 209138
## month 169 : balance 208390
## month 170 : balance 207639
## month 171 : balance 206885
## month 172 : balance 206128
## month 173 : balance 205368
## month 174 : balance 204605
## month 175 : balance 203838
## month 176 : balance 203069
## month 177 : balance 202296
## month 178 : balance 201520
## month 179 : balance 200741
## month 180 : balance 199959
## month 181 : balance 199174
## month 182 : balance 198385
## month 183 : balance 197593
## month 184 : balance 196798
## month 185 : balance 196000
## month 186 : balance 195199
## month 187 : balance 194394
## month 188 : balance 193586
## month 189 : balance 192775
## month 190 : balance 191960
## month 191 : balance 191142
## month 192 : balance 190321
## month 193 : balance 189496
## month 194 : balance 188668
## month 195 : balance 187837
## month 196 : balance 187002
## month 197 : balance 186164
## month 198 : balance 185323
## month 199 : balance 184478
## month 200 : balance 183629
## month 201 : balance 182777
## month 202 : balance 181922
## month 203 : balance 181063
## month 204 : balance 180201
## month 205 : balance 179335
## month 206 : balance 178466
## month 207 : balance 177593
## month 208 : balance 176716
## month 209 : balance 175836
```

```
## month 210 : balance 174953
## month 211 : balance 174065
## month 212 : balance 173175
## month 213 : balance 172280
## month 214 : balance 171382
## month 215 : balance 170480
## month 216 : balance 169575
## month 217 : balance 168666
## month 218 : balance 167753
## month 219 : balance 166836
## month 220 : balance 165916
## month 221 : balance 164992
## month 222 : balance 164064
## month 223 : balance 163133
## month 224 : balance 162197
## month 225 : balance 161258
## month 226 : balance 160315
## month 227 : balance 159368
## month 228 : balance 158417
## month 229 : balance 157463
## month 230 : balance 156504
## month 231 : balance 155542
## month 232 : balance 154576
## month 233 : balance 153605
## month 234 : balance 152631
## month 235 : balance 151653
## month 236 : balance 150671
## month 237 : balance 149685
## month 238 : balance 148695
## month 239 : balance 147700
## month 240 : balance 146702
## month 241 : balance 145700
## month 242 : balance 144693
## month 243 : balance 143683
## month 244 : balance 142668
## month 245 : balance 141650
## month 246 : balance 140627
## month 247 : balance 139600
## month 248 : balance 138568
## month 249 : balance 137533
## month 250 : balance 136493
## month 251 : balance 135449
## month 252 : balance 134401
## month 253 : balance 133349
## month 254 : balance 132292
## month 255 : balance 131231
## month 256 : balance 130166
## month 257 : balance 129096
## month 258 : balance 128022
## month 259 : balance 126943
## month 260 : balance 125861
## month 261 : balance 124773
## month 262 : balance 123682
## month 263 : balance 122586
```

```
## month 264 : balance 121485
## month 265 : balance 120380
## month 266 : balance 119270
## month 267 : balance 118156
## month 268 : balance 117038
## month 269 : balance 115915
## month 270 : balance 114787
## month 271 : balance 113654
## month 272 : balance 112518
## month 273 : balance 111376
## month 274 : balance 110230
## month 275 : balance 109079
## month 276 : balance 107923
## month 277 : balance 106763
## month 278 : balance 105598
## month 279 : balance 104428
## month 280 : balance 103254
## month 281 : balance 102074
## month 282 : balance 100890
## month 283 : balance 99701
## month 284 : balance 98507
## month 285 : balance 97309
## month 286 : balance 96105
## month 287 : balance 94897
## month 288 : balance 93683
## month 289 : balance 92465
## month 290 : balance 91242
## month 291 : balance 90013
## month 292 : balance 88780
## month 293 : balance 87542
## month 294 : balance 86298
## month 295 : balance 85050
## month 296 : balance 83797
## month 297 : balance 82538
## month 298 : balance 81274
## month 299 : balance 80005
## month 300 : balance 78731
## month 301 : balance 77452
## month 302 : balance 76168
## month 303 : balance 74878
## month 304 : balance 73583
## month 305 : balance 72283
## month 306 : balance 70977
## month 307 : balance 69666
## month 308 : balance 68350
## month 309 : balance 67029
## month 310 : balance 65702
## month 311 : balance 64369
## month 312 : balance 63032
## month 313 : balance 61688
## month 314 : balance 60340
## month 315 : balance 58986
## month 316 : balance 57626
## month 317 : balance 56261
```

```

## month 318 : balance 54890
## month 319 : balance 53514
## month 320 : balance 52132
## month 321 : balance 50744
## month 322 : balance 49351
## month 323 : balance 47952
## month 324 : balance 46547
## month 325 : balance 45137
## month 326 : balance 43721
## month 327 : balance 42299
## month 328 : balance 40871
## month 329 : balance 39438
## month 330 : balance 37998
## month 331 : balance 36553
## month 332 : balance 35102
## month 333 : balance 33645
## month 334 : balance 32182
## month 335 : balance 30713
## month 336 : balance 29238
## month 337 : balance 27758
## month 338 : balance 26271
## month 339 : balance 24778
## month 340 : balance 23279
## month 341 : balance 21773
## month 342 : balance 20262
## month 343 : balance 18745
## month 344 : balance 17221
## month 345 : balance 15691
## month 346 : balance 14155
## month 347 : balance 12613
## month 348 : balance 11064
## month 349 : balance 9509
## month 350 : balance 7948
## month 351 : balance 6380
## month 352 : balance 4806
## month 353 : balance 3226
## month 354 : balance 1639
## month 355 : balance 46
## month 356 : balance -1554
## total payments made 569600

```

So our nice young couple have paid off their \$300,000 loan in just 4 months shy of the 30 year term of their loan, at a bargain basement price of \$568,046 (since $569600 - 1554 = 568046$). A happy ending!

8.3 Conditional statements

A second kind of flow control that programming languages provide is the ability to evaluate ***conditional statements***. Unlike loops, which can repeat over and over again, a conditional statement only executes once, but it can switch between different possible commands depending on a CONDITION that is specified by the programmer. The power of these commands is that they allow the program itself to make choices, and in particular, to make different choices depending on the context in which the program is run. The most prominent example of a conditional statement is the ***if*** statement, and the accompanying ***else*** statement. The basic format of an ***if*** statement in R is as follows:

```
if ( CONDITION ) {
  STATEMENT1
  STATEMENT2
  ETC
}
```

And the execution of the statement is pretty straightforward. If the CONDITION is true, then R will execute the statements contained in the curly braces. If the CONDITION is false, then it does not. If you want to, you can extend the `if` statement to include an `else` statement as well, leading to the following syntax:

```
if ( CONDITION ) {
  STATEMENT1
  STATEMENT2
  ETC
} else {
  STATEMENT3
  STATEMENT4
  ETC
}
```

As you'd expect, the interpretation of this version is similar. If the CONDITION is true, then the contents of the first block of code (i.e., STATEMENT1, STATEMENT2, ETC) are executed; but if it is false, then the contents of the second block of code (i.e., STATEMENT3, STATEMENT4, ETC) are executed instead.

To give you a feel for how you can use `if` and `else` to do something useful, the example that I'll show you is a script that prints out a different message depending on what day of the week you run it. We can do this making use of some of the tools that we discussed in Section 7.11.3. Here's the script:

```
## --- ifelseexample.R
# find out what day it is...
today <- Sys.Date()          # pull the date from the system clock
day <- weekdays( today )    # what day of the week it is_

# now make a choice depending on the day...
if ( day == "Monday" ) {
  print( "I don't like Mondays" )
} else {
  print( "I'm a happy little automaton" )
}

## [1] "I'm a happy little automaton"
```

Since today happens to be a Saturday, when I run the script here's what happens:

```
source( "./rbook-master/scripts/ifelseexample.R" )

## [1] "I'm a happy little automaton"
```

There are other ways of making conditional statements in R. In particular, the `ifelse()` function and the `switch()` functions can be very useful in different contexts. However, my main aim in this chapter is to briefly cover the very basics, so I'll move on.

8.4 Writing functions

In this section I want to talk about functions again. Functions were introduced in Section 3.5, but you've learned a lot about R since then, so we can talk about them in more detail. In particular, I want to show you how to create your own. To stick with the same basic framework that I used to describe loops and conditionals, here's the syntax that you use to create a function:

```
FNAME <- function ( ARG1, ARG2, ETC ) {
  STATEMENT1
  STATEMENT2
  ETC
  return( VALUE )
}
```

What this does is create a function with the name FNAME, which has arguments ARG1, ARG2 and so forth. Whenever the function is called, R executes the statements in the curly braces, and then outputs the contents of VALUE to the user. Note, however, that R does not execute the commands inside the function in the workspace. Instead, what it does is create a temporary local environment: all the internal statements in the body of the function are executed there, so they remain invisible to the user. Only the final results in the VALUE are returned to the workspace.

To give a simple example of this, let's create a function called `quadruple()` which multiplies its inputs by four. In keeping with the approach taken in the rest of the chapter, I'll use a script to do this:

```
## --- functionexample.R
quadruple <- function(x) {
  y <- x*4
  return(y)
}
```

When we run this script, as follows

```
source( "./rbook-master/scripts/functionexample.R" )
```

nothing appears to have happened, but there is a new object created in the workspace called `quadruple`. Not surprisingly, if we ask R to tell us what kind of object it is, it tells us that it is a function:

```
class( quadruple )
```

```
## [1] "function"
```

And now that we've created the `quadruple()` function, we can call it just like any other function. And if I want to store the output as a variable, I can do this:

```
my.var <- quadruple(10)
print(my.var)
```

```
## [1] 40
```

An important thing to recognise here is that the two internal variables that the `quadruple()` function makes use of, `x` and `y`, stay internal. That is, if we inspect the contents of the workspace,

```
library(lsrr)
who()

##   -- Name --      -- Class --      -- Size --
##   afl.finalists    factor        400
##   afl.margins      numeric       176
##   afl.margins_out  numeric       176
##   afl2            data.frame  4296 x 2
##   age             numeric       11
##   age.breaks      numeric        4
##   age.group       factor        11
##   age.group2      factor        11
##   age.group3      factor        11
##   age.labels      character     3
##   animals          character     4
##   balance          numeric       1
##   beers            character     3
##   cake.1           numeric       5
##   cake.2           numeric       5
##   cake.df          data.frame   5 x 2
##   cake.mat1        matrix        5 x 2
##   cake.mat2        matrix        2 x 5
##   cakes            matrix        4 x 5
##   cakes.flipped   matrix        5 x 4
##   choice           data.frame   4 x 10
##   choice.2         data.frame   16 x 6
##   colour           logical       1
##   d.cor            numeric       1
##   dan.awake        logical       10
##   data             data.frame   12 x 4
##   day              character     1
##   describeImg     list          0
##   df               data.frame   4 x 1
##   drugs            data.frame   10 x 8
##   drugs.2          data.frame   30 x 5
##   effort           data.frame   10 x 2
##   emphCol          character     1
##   emphColLight    character     1
##   emphGrey         character     1
##   eps              logical       1
##   fac              factor        3
##   fibonacci        numeric       6
##   Fibonacci        numeric       7
##   freq             integer       17
##   garden           data.frame   5 x 3
##   height           numeric       1
##   hw               character     2
##   i                integer       1
##   interest          numeric       1
##   is.MP.speaking   logical       5
##   itng             data.frame   10 x 2
##   itng.table       table         3 x 4
##   likert centred  numeric      10
```

```
## likert.ordinal      ordered      10
## likert.raw          numeric      10
## M                  matrix      2 x 3
## makka.pakka         character     4
## monkey              character     1
## monkey.1            list         1
## month               numeric      1
## monthly.multiplier numeric      1
## msg                 character     1
## my.var              numeric      1
## ng                  character     2
## numbers             numeric      3
## old                list         66
## old.text            character     1
## oneCorPlot          function
## opinion.dir         numeric      10
## opinion.strength   numeric      10
## out.0               data.frame  100 x 2
## out.1               data.frame  100 x 2
## out.2               data.frame  100 x 2
## parenthood          data.frame  100 x 4
## payments            numeric      1
## PJ                 character     1
## plotOne             function
## quadruple           function
## row.1               numeric      3
## row.2               numeric      3
## some.data            numeric      18
## speaker              character     10
## speech.by.char      list         3
## teams               character     17
## text                character     2
## today               Date         1
## tombliboo            character     2
## total.paid           numeric      1
## upsy.daisy           character     4
## utterance            character     10
## w                   character     1
## W                   character     1
## w.length             integer       1
## width               numeric      1
## words               character     7
## x                   numeric      1
## X1                 numeric      11
## X2                 numeric      11
## X3                 numeric      11
## X4                 numeric      11
## xtab.3d              table        3 x 4 x 2
## y                   numeric      2
## Y1                 numeric      11
## Y2                 numeric      11
## Y3                 numeric      11
## Y4                 numeric      11
```

we see everything in our workspace from this chapter including the `quadruple()` function itself, as well as the `my.var` variable that we just created.

Now that we know how to create our own functions in R, it's probably a good idea to talk a little more about some of the other properties of functions that I've been glossing over. To start with, let's take this opportunity to type the name of the function at the command line without the parentheses:

```
quadruple
```

```
## function (x)
## {
##   y <- x * 4
##   return(y)
## }
```

As you can see, when you type the name of a function at the command line, R prints out the underlying source code that we used to define the function in the first place. In the case of the `quadruple()` function, this is quite helpful to us – we can read this code and actually see what the function does. For other functions, this is less helpful, as we saw back in Section 3.5 when we tried typing `citation` rather than `citation()`.

8.4.1 Function arguments revisited

Okay, now that we are starting to get a sense for how functions are constructed, let's have a look at two, slightly more complicated functions that I've created. The source code for these functions is contained within the `functionexample2.R` and `functionexample3.R` scripts. Let's start by looking at the first one:

```
## --- functionexample2.R
pow <- function( x, y = 1 ) {
  out <- x^y # raise x to the power y
  return( out )
}
```

and if we type `source("functionexample2.R")` to load the `pow()` function into our workspace, then we can make use of it. As you can see from looking at the code for this function, it has two arguments `x` and `y`, and all it does is raise `x` to the power of `y`. For instance, this command

```
pow(x=3, y=2)
```

```
## [1] 9
```

calculates the value of 3^2 . The interesting thing about this function isn't what it does, since R already has perfectly good mechanisms for calculating powers. Rather, notice that when I defined the function, I specified `y=1` when listing the arguments? That's the default value for `y`. So if we enter a command without specifying a value for `y`, then the function assumes that we want `y=1`:

```
pow( x=3 )
```

```
## [1] 3
```

However, since I didn't specify any default value for `x` when I defined the `pow()` function, we always need to input a value for `x`. If we don't R will spit out an error message.

So now you know how to specify default values for an argument. The other thing I should point out while I'm on this topic is the use of the `...` argument. The `...` argument is a special construct in R which is only used within functions. It is used as a way of matching against multiple user inputs: in other words, `...` is used as a mechanism to allow the user to enter as many inputs as they like. I won't talk at all about the low-level details of how this works at all, but I will show you a simple example of a function that makes use of it. To that end, consider the following script:

```
## --- functionexample3.R
doubleMax <- function( ... ) {
  max.val <- max( ... )  # find the largest value in ...
  out <- 2 * max.val     # double it
  return( out )
}
```

When we type `source("functionexample3.R")`, R creates the `doubleMax()` function. You can type in as many inputs as you like. The `doubleMax()` function identifies the largest value in the inputs, by passing all the user inputs to the `max()` function, and then doubles it. For example:

```
doubleMax( 1,2,5 )
```

```
## [1] 10
```

8.4.2 There's more to functions than this

There's a lot of other details to functions that I've hidden in my description in this chapter. Experienced programmers will wonder exactly how the "scoping rules" work in R,⁵ or want to know how to use a function to create variables in other environments⁶, or if function objects can be assigned as elements of a list⁷ and probably hundreds of other things besides. However, I don't want to have this discussion get too cluttered with details, so I think it's best – at least for the purposes of the current book – to stop here.

8.5 Implicit loops

There's one last topic I want to discuss in this chapter. In addition to providing the explicit looping structures via `while` and `for`, R also provides a collection of functions for *implicit loops*. What I mean by this is that these are functions that carry out operations very similar to those that you'd normally use a loop for. However, instead of typing out the whole loop, the whole thing is done with a single command. The main reason why this can be handy is that – due to the way that R is written – these implicit looping functions are usually about to do the same calculations much faster than the corresponding explicit loops. In most applications that beginners might want to undertake, this probably isn't very important, since most beginners tend to start out working with fairly small data sets and don't usually need to undertake extremely time consuming number crunching. However, because you often see these functions referred to in other contexts, it may be useful to very briefly discuss a few of them.

The first and simplest of these functions is `sapply()`. The two most important arguments to this function are `X`, which specifies a vector containing the data, and `FUN`, which specifies the name of a function that should be applied to each element of the data vector. The following example illustrates the basics of how it works:

⁵Lexical scope.

⁶The `assign()` function.

⁷Yes.

```
words <- c("along", "the", "loom", "of", "the", "land")
sapply( X = words, FUN = nchar )
```

```
## along   the   loom    of    the   land
##      5      3      4      2      3      4
```

Notice how similar this is to the second example of a `for` loop in Section 8.2.2. The `sapply()` function has implicitly looped over the elements of `words`, and for each such element applied the `nchar()` function to calculate the number of letters in the corresponding word.

The second of these functions is `tapply()`, which has three key arguments. As before `X` specifies the data, and `FUN` specifies a function. However, there is also an `INDEX` argument which specifies a grouping variable.⁸ What the `tapply()` function does is loop over all of the different values that appear in the `INDEX` variable. Each such value defines a group: the `tapply()` function constructs the subset of `X` that corresponds to that group, and then applies the function `FUN` to that subset of the data. This probably sounds a little abstract, so let's consider a specific example, using the `nightgarden.Rdata` file that we used in Chapter 7.

```
gender <- c( "male","male","female","female","male" )
age <- c( 10,12,9,11,13 )
tapply( X = age, INDEX = gender, FUN = mean )
```

```
##   female     male
## 10.00000 11.66667
```

In this extract, what we're doing is using `gender` to define two different groups of people, and using their `ages` as the data. We then calculate the `mean()` of the ages, separately for the males and the females. A closely related function is `by()`. It actually does the same thing as `tapply()`, but the output is formatted a bit differently. This time around the three arguments are called `data`, `INDICES` and `FUN`, but they're pretty much the same thing. An example of how to use the `by()` function is shown in the following extract:

```
by( data = age, INDICES = gender, FUN = mean )

## gender: female
## [1] 10
## -----
## gender: male
## [1] 11.66667
```

The `tapply()` and `by()` functions are quite handy things to know about, and are pretty widely used. However, although I do make passing reference to the `tapply()` later on, I don't make much use of them in this book.

Before moving on, I should mention that there are several other functions that work along similar lines, and have suspiciously similar names: `lapply`, `mapply`, `apply`, `vapply`, `rapply` and `eapply`. However, none of these come up anywhere else in this book, so all I wanted to do here is draw your attention to the fact that they exist.

⁸Or a list of such variables.

8.6 Summary

In this chapter I talked about several key programming concepts, things that you should know about if you want to start converting your simple scripts into full fledged programs:

- Writing and using scripts (Section 8.1).
- Using loops (Section 8.2) and implicit loops (Section 8.5).
- Making conditional statements (Section 8.3)
- Writing your own functions (Section 8.4)

As always, there are *lots* of things I'm ignoring in this chapter. It takes a lot of work to become a proper programmer, just as it takes a lot of work to be a proper psychologist or a proper statistician, and this book is certainly not going to provide you with all the tools you need to make that step. However, you'd be amazed at how much you can achieve using only the tools that I've covered up to this point. Loops, conditionals and functions are very powerful things, especially when combined with the various tools discussed in Chapters 3, 4 and 7. Believe it or not, you're off to a pretty good start just by having made it to this point. If you want to keep going, there are (as always!) several other books you might want to look at. One that I've read and enjoyed is "A first course in statistical programming with R" Braun and Murdoch (2007), but quite a few people have suggested to me that "The art of programming with R" Matloff and Matloff (2011) is worth the effort too.

Chapter 9

Introduction to probability

[God] has afforded us only the twilight ... of Probability.

– John Locke

Up to this point in the book, we've discussed some of the key ideas in experimental design, and we've talked a little about how you can summarise a data set. To a lot of people, this is all there is to statistics: it's about calculating averages, collecting all the numbers, drawing pictures, and putting them all in a report somewhere. Kind of like stamp collecting, but with numbers. However, statistics covers much more than that. In fact, descriptive statistics is one of the smallest parts of statistics, and one of the least powerful. The bigger and more useful part of statistics is that it provides that let you make inferences about data.

Once you start thinking about statistics in these terms – that statistics is there to help us draw inferences from data – you start seeing examples of it everywhere. For instance, here's a tiny extract from a newspaper article in the Sydney Morning Herald (30 Oct 2010):

“I have a tough job,” the Premier said in response to a poll which found her government is now the most unpopular Labor administration in polling history, with a primary vote of just 23 per cent.

This kind of remark is entirely unremarkable in the papers or in everyday life, but let's have a think about what it entails. A polling company has conducted a survey, usually a pretty big one because they can afford it. I'm too lazy to track down the original survey, so let's just imagine that they called 1000 NSW voters at random, and 230 (23%) of those claimed that they intended to vote for the ALP. For the 2010 Federal election, the Australian Electoral Commission reported 4,610,795 enrolled voters in NSW; so the opinions of the remaining 4,609,795 voters (about 99.98% of voters) remain unknown to us. Even assuming that no-one lied to the polling company the only thing we can say with 100% confidence is that the true ALP primary vote is somewhere between $230/4610795$ (about 0.005%) and $4610025/4610795$ (about 99.83%). So, on what basis is it legitimate for the polling company, the newspaper, and the readership to conclude that the ALP primary vote is only about 23%?

The answer to the question is pretty obvious: if I call 1000 people at random, and 230 of them say they intend to vote for the ALP, then it seems very unlikely that these are the *only* 230 people out of the entire voting public who actually intend to do so. In other words, we assume that the data collected by the polling company is pretty representative of the population at large. But how representative? Would we be surprised to discover that the true ALP primary vote is actually 24%? 29%? 37%? At this point everyday intuition starts to break down a bit. No-one would be surprised by 24%, and everybody would be surprised by 37%, but it's a bit hard to say whether 29% is plausible. We need some more powerful tools than just looking at the numbers and guessing.

Inferential statistics provides the tools that we need to answer these sorts of questions, and since these kinds of questions lie at the heart of the scientific enterprise, they take up the lions share of every introductory course on statistics and research methods. However, the theory of statistical inference is built on top of **probability theory**. And it is to probability theory that we must now turn. This discussion of probability theory is basically background: there's not a lot of statistics per se in this chapter, and you don't need to understand this material in as much depth as the other chapters in this part of the book. Nevertheless, because probability theory does underpin so much of statistics, it's worth covering some of the basics.

9.1 How are probability and statistics different?

Before we start talking about probability theory, it's helpful to spend a moment thinking about the relationship between probability and statistics. The two disciplines are closely related but they're not identical. Probability theory is "the doctrine of chances". It's a branch of mathematics that tells you how often different kinds of events will happen. For example, all of these questions are things you can answer using probability theory:

- What are the chances of a fair coin coming up heads 10 times in a row?
- If I roll two six sided dice, how likely is it that I'll roll two sixes?
- How likely is it that five cards drawn from a perfectly shuffled deck will all be hearts?
- What are the chances that I'll win the lottery?

Notice that all of these questions have something in common. In each case the "truth of the world" is known, and my question relates to the "what kind of events" will happen. In the first question I *know* that the coin is fair, so there's a 50% chance that any individual coin flip will come up heads. In the second question, I *know* that the chance of rolling a 6 on a single die is 1 in 6. In the third question I *know* that the deck is shuffled properly. And in the fourth question, I *know* that the lottery follows specific rules. You get the idea. The critical point is that probabilistic questions start with a known **model** of the world, and we use that model to do some calculations. The underlying model can be quite simple. For instance, in the coin flipping example, we can write down the model like this:

$$P(\text{heads}) = 0.5$$

which you can read as "the probability of heads is 0.5". As we'll see later, in the same way that percentages are numbers that range from 0% to 100%, probabilities are just numbers that range from 0 to 1. When using this probability model to answer the first question, I don't actually know exactly what's going to happen. Maybe I'll get 10 heads, like the question says. But maybe I'll get three heads. That's the key thing: in probability theory, the *model* is known, but the *data* are not.

So that's probability. What about statistics? Statistical questions work the other way around. In statistics, we *do not* know the truth about the world. All we have is the data, and it is from the data that we want to *learn* the truth about the world. Statistical questions tend to look more like these:

- If my friend flips a coin 10 times and gets 10 heads, are they playing a trick on me?
- If five cards off the top of the deck are all hearts, how likely is it that the deck was shuffled? - If the lottery commissioner's spouse wins the lottery, how likely is it that the lottery was rigged?

This time around, the only thing we have are data. What I *know* is that I saw my friend flip the coin 10 times and it came up heads every time. And what I want to *infer* is whether or not I should conclude that what I just saw was actually a fair coin being flipped 10 times in a row, or whether I should suspect that my friend is playing a trick on me. The data I have look like this:

H H H H H H H H H H

and what I'm trying to do is work out which "model of the world" I should put my trust in. If the coin is fair, then the model I should adopt is one that says that the probability of heads is 0.5; that is, $P(\text{heads}) = 0.5$. If the coin is not fair, then I should conclude that the probability of heads is *not* 0.5, which we would write as $P(\text{heads}) \neq 0.5$. In other words, the statistical inference problem is to figure out which of these probability models is right. Clearly, the statistical question isn't the same as the probability question, but they're deeply connected to one another. Because of this, a good introduction to statistical theory will start with a discussion of what probability is and how it works.

9.2 What does probability mean?

Let's start with the first of these questions. What is "probability"? It might seem surprising to you, but while statisticians and mathematicians (mostly) agree on what the *rules* of probability are, there's much less of a consensus on what the word really *means*. It seems weird because we're all very comfortable using words like "chance", "likely", "possible" and "probable", and it doesn't seem like it should be a very difficult question to answer. If you had to explain "probability" to a five year old, you could do a pretty good job. But if you've ever had that experience in real life, you might walk away from the conversation feeling like you didn't quite get it right, and that (like many everyday concepts) it turns out that you don't *really* know what it's all about.

So I'll have a go at it. Let's suppose I want to bet on a soccer game between two teams of robots, *Arduino Arsenal* and *C Milan*. After thinking about it, I decide that there is an 80% probability that *Arduino Arsenal* winning. What do I mean by that? Here are three possibilities...

- They're robot teams, so I can make them play over and over again, and if I did that, *Arduino Arsenal* would win 8 out of every 10 games on average.
- For any given game, I would only agree that betting on this game is only "fair" if a \$1 bet on *C Milan* gives a \$5 payoff (i.e. I get my \$1 back plus a \$4 reward for being correct), as would a \$4 bet on *Arduino Arsenal* (i.e., my \$4 bet plus a \$1 reward).
- My subjective "belief" or "confidence" in an *Arduino Arsenal* victory is four times as strong as my belief in a *C Milan* victory.

Each of these seems sensible. However they're not identical, and not every statistician would endorse all of them. The reason is that there are different statistical ideologies (yes, really!) and depending on which one you subscribe to, you might say that some of those statements are meaningless or irrelevant. In this section, I give a brief introduction the two main approaches that exist in the literature. These are by no means the only approaches, but they're the two big ones.

9.2.1 The frequentist view

The first of the two major approaches to probability, and the more dominant one in statistics, is referred to as the **frequentist view**, and it defines probability as a **long-run frequency**. Suppose we were to try flipping a fair coin, over and over again. By definition, this is a coin that has $P(H) = 0.5$. What might we observe? One possibility is that the first 20 flips might look like this:

T,H,H,H,H,T,T,H,H,H,T,H,H,T,T,T,T,H

In this case 11 of these 20 coin flips (55%) came up heads. Now suppose that I'd been keeping a running tally of the number of heads (which I'll call N_H) that I've seen, across the first N flips, and calculate the proportion of heads N_H/N every time. Here's what I'd get (I did literally flip coins to produce this!):

number.of.flips	number.of.heads	proportion
1	0	0.00
2	1	0.50
3	2	0.67
4	3	0.75
5	4	0.80
6	4	0.67
7	4	0.57
8	5	0.63
9	6	0.67
10	7	0.70
11	8	0.73
12	8	0.67
13	9	0.69
14	10	0.71
15	10	0.67
16	10	0.63
17	10	0.59
18	10	0.56
19	10	0.53
20	11	0.55

Notice that at the start of the sequence, the *proportion* of heads fluctuates wildly, starting at .00 and rising as high as .80. Later on, one gets the impression that it dampens out a bit, with more and more of the values actually being pretty close to the “right” answer of .50. This is the frequentist definition of probability in a nutshell: flip a fair coin over and over again, and as N grows large (approaches infinity, denoted $N \rightarrow \infty$), the proportion of heads will converge to 50%. There are some subtle technicalities that the mathematicians care about, but qualitatively speaking, that’s how the frequentists define probability. Unfortunately, I don’t have an infinite number of coins, or the infinite patience required to flip a coin an infinite number of times. However, I do have a computer, and computers excel at mindless repetitive tasks. So I asked my computer to simulate flipping a coin 1000 times, and then drew a picture of what happens to the proportion N_H/N as N increases. Actually, I did it four times, just to make sure it wasn’t a fluke. The results are shown in Figure 9.1. As you can see, the *proportion of observed heads* eventually stops fluctuating, and settles down; when it does, the number at which it finally settles is the true probability of heads.

The frequentist definition of probability has some desirable characteristics. Firstly, it is objective: the probability of an event is *necessarily* grounded in the world. The only way that probability statements can make sense is if they refer to (a sequence of) events that occur in the physical universe.¹ Secondly, it is unambiguous: any two people watching the same sequence of events unfold, trying to calculate the probability of an event, must inevitably come up with the same answer. However, it also has undesirable characteristics. Firstly, infinite sequences don’t exist in the physical world. Suppose you picked up a coin from your pocket and started to flip it. Every time it lands, it impacts on the ground. Each impact wears the coin down a bit; eventually, the coin will be destroyed. So, one might ask whether it really makes sense to pretend that an “infinite” sequence of coin flips is even a meaningful concept, or an objective one. We can’t say that an “infinite sequence” of events is a real thing in the physical universe, because the physical universe doesn’t allow infinite anything. More seriously, the frequentist definition has a narrow scope. There are lots of things out there that human beings are happy to assign probability to in everyday language, but cannot (even in theory) be mapped onto a hypothetical sequence of events. For instance, if a meteorologist comes on TV and says, “the probability of rain in Adelaide on 2 November 2048 is 60%” we humans are happy to accept this. But it’s not clear how to define this in frequentist terms. There’s only one city of Adelaide, and only 2 November 2048. There’s no infinite sequence of events here, just a once-off

¹This doesn’t mean that frequentists can’t make hypothetical statements, of course; it’s just that if you want to make a statement about probability, then it must be possible to redescribe that statement in terms of a sequence of potentially observable events, and the relative frequencies of different outcomes that appear within that sequence.

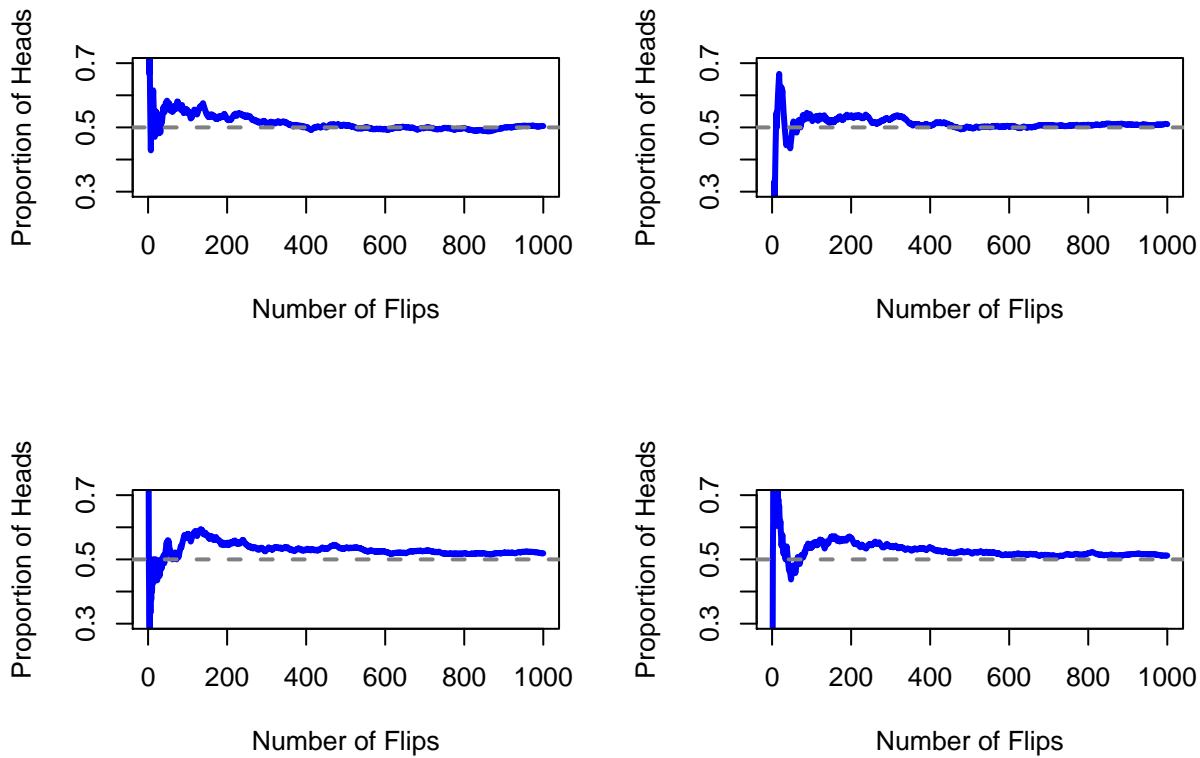


Figure 9.1: An illustration of how frequentist probability works. If you flip a fair coin over and over again, the proportion of heads that you've seen eventually settles down, and converges to the true probability of 0.5. Each panel shows four different simulated experiments: in each case, we pretend we flipped a coin 1000 times, and kept track of the proportion of flips that were heads as we went along. Although none of these sequences actually ended up with an exact value of .5, if we'd extended the experiment for an infinite number of coin flips they would have.

thing. Frequentist probability genuinely *forbids* us from making probability statements about a single event. From the frequentist perspective, it will either rain tomorrow or it will not; there is no “probability” that attaches to a single non-repeatable event. Now, it should be said that there are some very clever tricks that frequentists can use to get around this. One possibility is that what the meteorologist means is something like this: “There is a category of days for which I predict a 60% chance of rain; if we look only across those days for which I make this prediction, then on 60% of those days it will actually rain”. It’s very weird and counterintuitive to think of it this way, but you do see frequentists do this sometimes. And it *will* come up later in this book (see Section 10.5).

9.2.2 The Bayesian view

The *Bayesian view* of probability is often called the subjectivist view, and it is a minority view among statisticians, but one that has been steadily gaining traction for the last several decades. There are many flavours of Bayesianism, making hard to say exactly what “the” Bayesian view is. The most common way of thinking about subjective probability is to define the probability of an event as the *degree of belief* that an intelligent and rational agent assigns to that truth of that event. From that perspective, probabilities don’t exist in the world, but rather in the thoughts and assumptions of people and other intelligent beings. However, in order for this approach to work, we need some way of operationalising “degree of belief”. One way that you can do this is to formalise it in terms of “rational gambling”, though there are many other ways. Suppose that I believe that there’s a 60% probability of rain tomorrow. If someone offers me a bet: if it rains tomorrow, then I win \$5, but if it doesn’t rain then I lose \$5. Clearly, from my perspective, this is a pretty good bet. On the other hand, if I think that the probability of rain is only 40%, then it’s a bad bet to take. Thus, we can operationalise the notion of a “subjective probability” in terms of what bets I’m willing to accept.

What are the advantages and disadvantages to the Bayesian approach? The main advantage is that it allows you to assign probabilities to any event you want to. You don’t need to be limited to those events that are repeatable. The main disadvantage (to many people) is that we can’t be purely objective – specifying a probability requires us to specify an entity that has the relevant degree of belief. This entity might be a human, an alien, a robot, or even a statistician, but there has to be an intelligent agent out there that believes in things. To many people this is uncomfortable: it seems to make probability arbitrary. While the Bayesian approach does require that the agent in question be rational (i.e., obey the rules of probability), it does allow everyone to have their own beliefs; I can believe the coin is fair and you don’t have to, even though we’re both rational. The frequentist view doesn’t allow any two observers to attribute different probabilities to the same event: when that happens, then at least one of them must be wrong. The Bayesian view does not prevent this from occurring. Two observers with different background knowledge can legitimately hold different beliefs about the same event. In short, where the frequentist view is sometimes considered to be too narrow (forbids lots of things that we want to assign probabilities to), the Bayesian view is sometimes thought to be too broad (allows too many differences between observers).

9.2.3 What’s the difference? And who is right?

Now that you’ve seen each of these two views independently, it’s useful to make sure you can compare the two. Go back to the hypothetical robot soccer game at the start of the section. What do you think a frequentist and a Bayesian would say about these three statements? Which statement would a frequentist say is the correct definition of probability? Which one would a Bayesian do? Would some of these statements be meaningless to a frequentist or a Bayesian? If you’ve understood the two perspectives, you should have some sense of how to answer those questions.

Okay, assuming you understand the different, you might be wondering which of them is *right*? Honestly, I don’t know that there is a right answer. As far as I can tell there’s nothing mathematically incorrect about the way frequentists think about sequences of events, and there’s nothing mathematically incorrect about the way that Bayesians define the beliefs of a rational agent. In fact, when you dig down into the details,

Bayesians and frequentists actually agree about a lot of things. Many frequentist methods lead to decisions that Bayesians agree a rational agent would make. Many Bayesian methods have very good frequentist properties.

For the most part, I'm a pragmatist so I'll use any statistical method that I trust. As it turns out, that makes me prefer Bayesian methods, for reasons I'll explain towards the end of the book, but I'm not fundamentally opposed to frequentist methods. Not everyone is quite so relaxed. For instance, consider Sir Ronald Fisher, one of the towering figures of 20th century statistics and a vehement opponent to all things Bayesian, whose paper on the mathematical foundations of statistics referred to Bayesian probability as "an impenetrable jungle [that] arrests progress towards precision of statistical concepts" Fisher (1922). Or the psychologist Paul Meehl, who suggests that relying on frequentist methods could turn you into "a potent but sterile intellectual rake who leaves in his merry path a long train of ravished maidens but no viable scientific offspring" Meehl (1967). The history of statistics, as you might gather, is not devoid of entertainment.

In any case, while I personally prefer the Bayesian view, the majority of statistical analyses are based on the frequentist approach. My reasoning is pragmatic: the goal of this book is to cover roughly the same territory as a typical undergraduate stats class in psychology, and if you want to understand the statistical tools used by most psychologists, you'll need a good grasp of frequentist methods. I promise you that this isn't wasted effort. Even if you end up wanting to switch to the Bayesian perspective, you really should read through at least one book on the "orthodox" frequentist view. And since R is the most widely used statistical language for Bayesians, you might as well read a book that uses R. Besides, I won't completely ignore the Bayesian perspective. Every now and then I'll add some commentary from a Bayesian point of view, and I'll revisit the topic in more depth in Chapter ??.

9.3 Basic probability theory

Ideological arguments between Bayesians and frequentists notwithstanding, it turns out that people mostly agree on the rules that probabilities should obey. There are lots of different ways of arriving at these rules. The most commonly used approach is based on the work of Andrey Kolmogorov, one of the great Soviet mathematicians of the 20th century. I won't go into a lot of detail, but I'll try to give you a bit of a sense of how it works. And in order to do so, I'm going to have to talk about my pants.

9.3.1 Introducing probability distributions

One of the disturbing truths about my life is that I only own 5 pairs of pants: three pairs of jeans, the bottom half of a suit, and a pair of tracksuit pants. Even sadder, I've given them names: I call them X_1 , X_2 , X_3 , X_4 and X_5 . I really do: that's why they call me Mister Imaginative. Now, on any given day, I pick out exactly one of pair of pants to wear. Not even I'm so stupid as to try to wear two pairs of pants, and thanks to years of training I never go outside without wearing pants anymore. If I were to describe this situation using the language of probability theory, I would refer to each pair of pants (i.e., each X) as an **elementary event**. The key characteristic of elementary events is that every time we make an observation (e.g., every time I put on a pair of pants), then the outcome will be one and only one of these events. Like I said, these days I always wear exactly one pair of pants, so my pants satisfy this constraint. Similarly, the set of all possible events is called a **sample space**. Granted, some people would call it a "wardrobe", but that's because they're refusing to think about my pants in probabilistic terms. Sad.

Okay, now that we have a sample space (a wardrobe), which is built from lots of possible elementary events (pants), what we want to do is assign a **probability** of one of these elementary events. For an event X , the probability of that event $P(X)$ is a number that lies between 0 and 1. The bigger the value of $P(X)$, the more likely the event is to occur. So, for example, if $P(X) = 0$, it means the event X is impossible (i.e., I never wear those pants). On the other hand, if $P(X) = 1$ it means that event X is certain to occur (i.e., I always wear those pants). For probability values in the middle, it means that I sometimes wear those pants. For instance, if $P(X) = 0.5$ it means that I wear those pants half of the time.

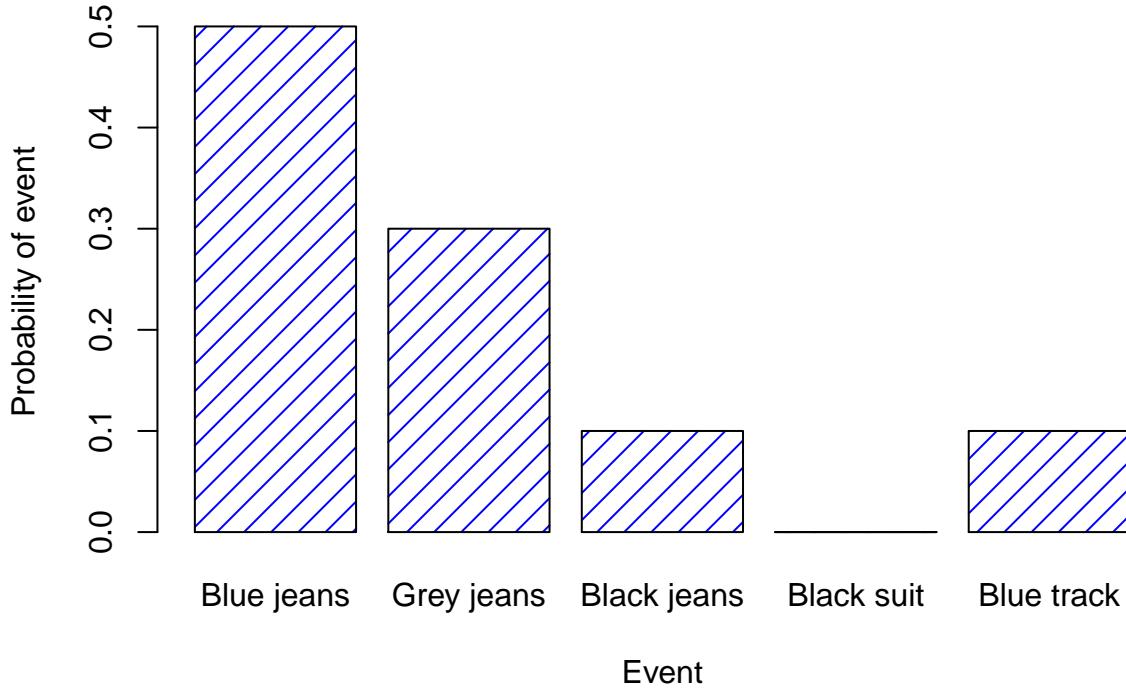


Figure 9.2: A visual depiction of the “pants” probability distribution. There are five “elementary events”, corresponding to the five pairs of pants that I own. Each event has some probability of occurring: this probability is a number between 0 to 1. The sum of these probabilities is 1.

At this point, we’re almost done. The last thing we need to recognise is that “something always happens”. Every time I put on pants, I really do end up wearing pants (crazy, right?). What this somewhat trite statement means, in probabilistic terms, is that the probabilities of the elementary events need to add up to 1. This is known as the **law of total probability**, not that any of us really care. More importantly, if these requirements are satisfied, then what we have is a **probability distribution**. For example, this is an example of a probability distribution

Which.pants	Blue.jeans	Grey.jeans	Black.jeans	Black.suit	Blue.tracksuit
Label	\$X_1\$	\$X_2\$	\$X_3\$	\$X_4\$	\$X_5\$
Probability	\$P(X_1) = .5\$	\$P(X_2) = .3\$	\$P(X_3) = .1\$	\$P(X_4) = 0\$	\$P(X_5) = .1\$

Each of the events has a probability that lies between 0 and 1, and if we add up the probability of all events, they sum to 1. Awesome. We can even draw a nice bar graph (see Section 6.7) to visualise this distribution, as shown in Figure ???. And at this point, we’ve all achieved something. You’ve learned what a probability distribution is, and I’ve finally managed to find a way to create a graph that focuses entirely on my pants. Everyone wins!

The only other thing that I need to point out is that probability theory allows you to talk about **non elementary events** as well as elementary ones. The easiest way to illustrate the concept is with an example. In the pants example, it’s perfectly legitimate to refer to the probability that I wear jeans. In this scenario, the “Dan wears jeans” event said to have happened as long as the elementary event that actually did occur is one of the appropriate ones; in this case “blue jeans”, “black jeans” or “grey jeans”. In mathematical terms, we defined the “jeans” event E to correspond to the set of elementary events (X_1, X_2, X_3) . If any of these

Table 9.1: Some basic rules that probabilities must satisfy. You don't really need to know these rules in order to understand the analyses that we'll talk about later in the book, but they are important if you want to understand probability theory a bit more deeply.

English	Notation	NANA	Formula
Not A	$P(\neg A)$	=	$1 - P(A)$
A or B	$P(A \cup B)$	=	$P(A) + P(B) - P(A \cap B)$
A and B	$P(A \cap B)$	=	$P(A B) P(B)$

elementary events occurs, then E is also said to have occurred. Having decided to write down the definition of the E this way, it's pretty straightforward to state what the probability $P(E)$ is: we just add everything up. In this particular case

$$P(E) = P(X_1) + P(X_2) + P(X_3)$$

and, since the probabilities of blue, grey and black jeans respectively are .5, .3 and .1, the probability that I wear jeans is equal to .9.

At this point you might be thinking that this is all terribly obvious and simple and you'd be right. All we've really done is wrap some basic mathematics around a few common sense intuitions. However, from these simple beginnings it's possible to construct some extremely powerful mathematical tools. I'm definitely not going to go into the details in this book, but what I will do is list – in Table 9.1 – some of the other rules that probabilities satisfy. These rules can be derived from the simple assumptions that I've outlined above, but since we don't actually use these rules for anything in this book, I won't do so here.

9.4 The binomial distribution

As you might imagine, probability distributions vary enormously, and there's an enormous range of distributions out there. However, they aren't all equally important. In fact, the vast majority of the content in this book relies on one of five distributions: the binomial distribution, the normal distribution, the t distribution, the χ^2 ("chi-square") distribution and the F distribution. Given this, what I'll do over the next few sections is provide a brief introduction to all five of these, paying special attention to the binomial and the normal. I'll start with the binomial distribution, since it's the simplest of the five.

9.4.1 Introducing the binomial

The theory of probability originated in the attempt to describe how games of chance work, so it seems fitting that our discussion of the **binomial distribution** should involve a discussion of rolling dice and flipping coins. Let's imagine a simple "experiment": in my hot little hand I'm holding 20 identical six-sided dice. On one face of each die there's a picture of a skull; the other five faces are all blank. If I proceed to roll all 20 dice, what's the probability that I'll get exactly 4 skulls? Assuming that the dice are fair, we know that the chance of any one die coming up skulls is 1 in 6; to say this another way, the skull probability for a single die is approximately .167. This is enough information to answer our question, so let's have a look at how it's done.

As usual, we'll want to introduce some names and some notation. We'll let N denote the number of dice rolls in our experiment; which is often referred to as the **size parameter** of our binomial distribution. We'll also use θ to refer to the probability that a single die comes up skulls, a quantity that is usually called the **success probability** of the binomial.² Finally, we'll use X to refer to the results of our experiment, namely the number of skulls I get when I roll the dice. Since the actual value of X is due to chance, we

²Note that the term "success" is pretty arbitrary, and doesn't actually imply that the outcome is something to be desired. If θ referred to the probability that any one passenger gets injured in a bus crash, I'd still call it the success probability, but that doesn't mean I want people to get hurt in bus crashes!

refer to it as a *random variable*. In any case, now that we have all this terminology and notation, we can use it to state the problem a little more precisely. The quantity that we want to calculate is the probability that $X = 4$ given that we know that $\theta = .167$ and $N = 20$. The general “form” of the thing I’m interested in calculating could be written as

$$P(X | \theta, N)$$

and we’re interested in the special case where $X = 4$, $\theta = .167$ and $N = 20$. There’s only one more piece of notation I want to refer to before moving on to discuss the solution to the problem. If I want to say that X is generated randomly from a binomial distribution with parameters θ and N , the notation I would use is as follows:

$$X \sim \text{Binomial}(\theta, N)$$

Yeah, yeah. I know what you’re thinking: notation, notation, notation. Really, who cares? Very few readers of this book are here for the notation, so I should probably move on and talk about how to use the binomial distribution. I’ve included the formula for the binomial distribution in Table 9.2, since some readers may want to play with it themselves, but since most people probably don’t care that much and because we don’t need the formula in this book, I won’t talk about it in any detail. Instead, I just want to show you what the binomial distribution looks like. To that end, Figure 9.3 plots the binomial probabilities for all possible values of X for our dice rolling experiment, from $X = 0$ (no skulls) all the way up to $X = 20$ (all skulls). Note that this is basically a bar chart, and is no different to the “pants probability” plot I drew in Figure 9.2. On the horizontal axis we have all the possible events, and on the vertical axis we can read off the probability of each of those events. So, the probability of rolling 4 skulls out of 20 times is about 0.20 (the actual answer is 0.2022036, as we’ll see in a moment). In other words, you’d expect that to happen about 20% of the times you repeated this experiment.

9.4.2 Working with the binomial distribution in R

Although some people find it handy to know the formulas in Table 9.2, most people just want to know how to use the distributions without worrying too much about the maths. To that end, R has a function called `dbinom()` that calculates binomial probabilities for us. The main arguments to the function are

- `x`. This is a number, or vector of numbers, specifying the outcomes whose probability you’re trying to calculate.
- `size`. This is a number telling R the size of the experiment.
- `prob`. This is the success probability for any one trial in the experiment.

So, in order to calculate the probability of getting $x = 4$ skulls, from an experiment of `size = 20` trials, in which the probability of getting a skull on any one trial is `prob = 1/6` ... well, the command I would use is simply this:

```
dbinom( x = 4, size = 20, prob = 1/6 )
```

```
## [1] 0.2022036
```

To give you a feel for how the binomial distribution changes when we alter the values of θ and N , let’s suppose that instead of rolling dice, I’m actually flipping coins. This time around, my experiment involves flipping a fair coin repeatedly, and the outcome that I’m interested in is the number of heads that I observe. In this scenario, the success probability is now $\theta = 1/2$. Suppose I were to flip the coin $N = 20$ times. In this example, I’ve changed the success probability, but kept the size of the experiment the same. What does this do to our binomial distribution? Well, as Figure 9.4 shows, the main effect of this is to shift the whole distribution, as you’d expect. Okay, what if we flipped a coin $N = 100$ times? Well, in that case, we get Figure 9.5. The distribution stays roughly in the middle, but there’s a bit more variability in the possible outcomes.

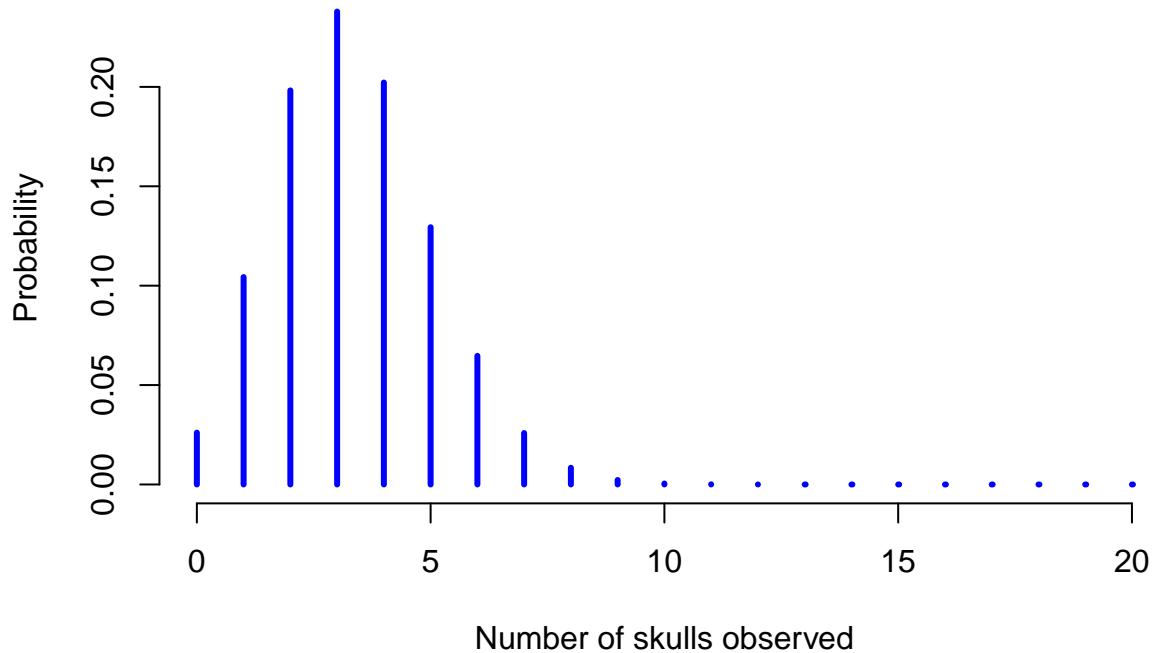


Figure 9.3: The binomial distribution with size parameter of $N = 20$ and an underlying success probability of $\theta = 1/6$. Each vertical bar depicts the probability of one specific outcome (i.e., one possible value of X). Because this is a probability distribution, each of the probabilities must be a number between 0 and 1, and the heights of the bars must sum to 1 as well.

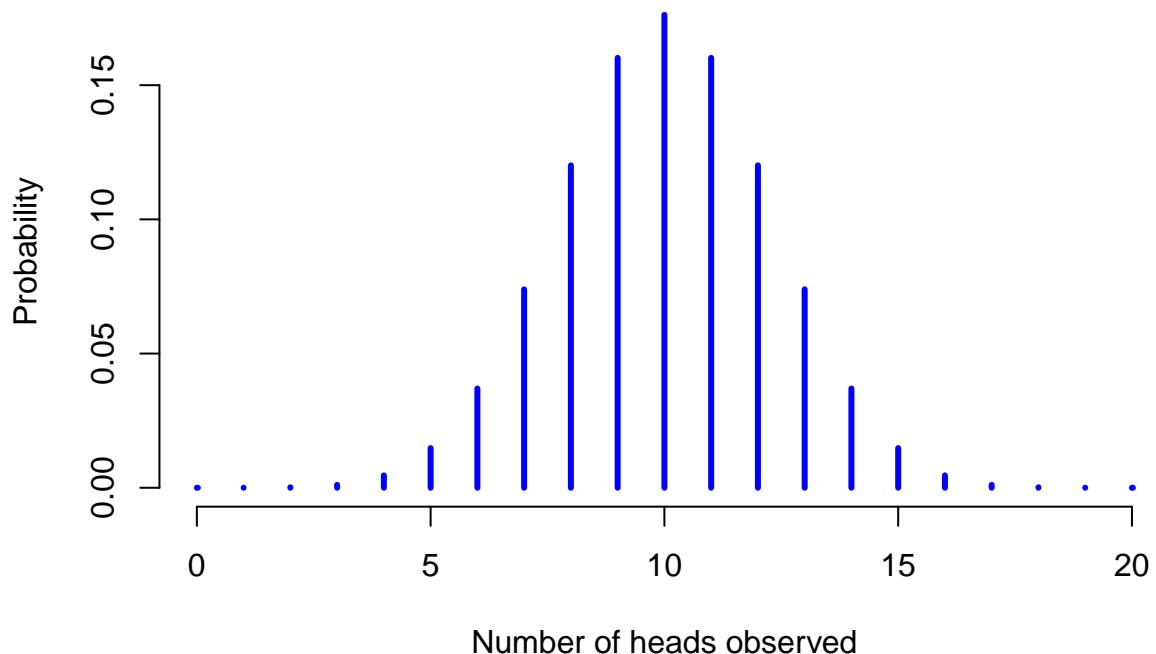


Figure 9.4: Two binomial distributions, involving a scenario in which I'm flipping a fair coin, so the underlying success probability is $\theta = 1/2$. Here we assume I'm flipping the coin $N = 20$ times.

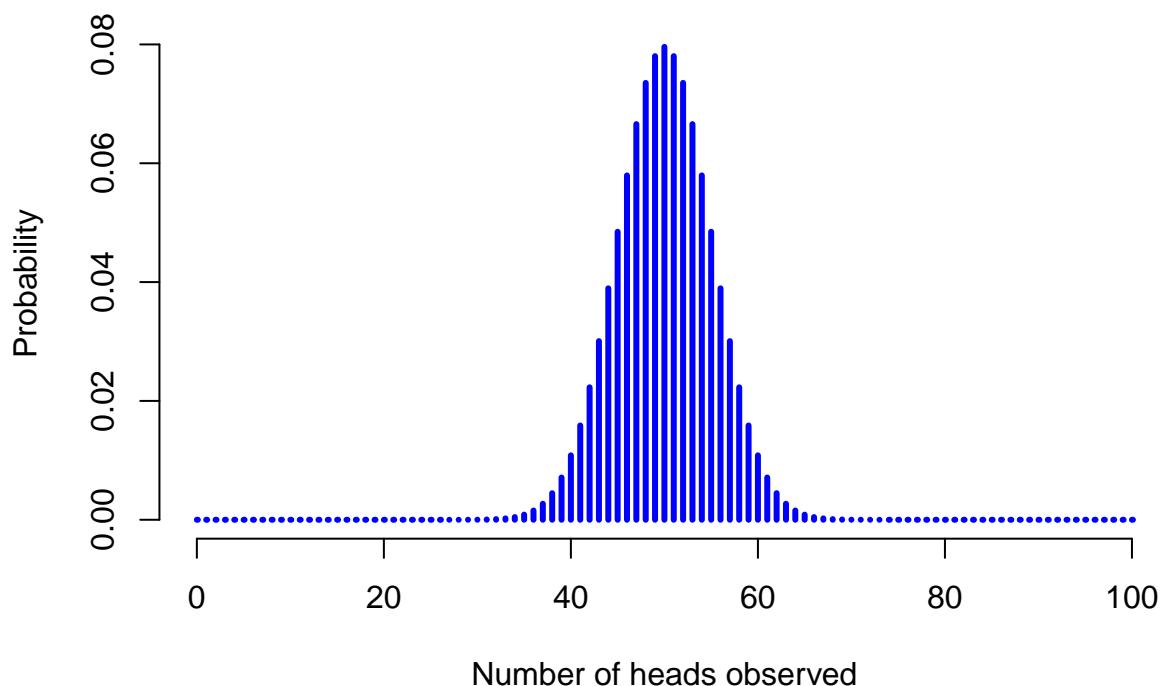


Figure 9.5: Two binomial distributions, involving a scenario in which I'm flipping a fair coin, so the underlying success probability is $\theta = 1/2$. Here we assume that the coin is flipped $N = 100$ times.

Table 9.2: Formulas for the binomial and normal distributions. We don't really use these formulas for anything in this book, but they're pretty important for more advanced work, so I thought it might be best to put them here in a table, where they can't get in the way of the text. In the equation for the binomial, $X!$ is the factorial function (i.e., multiply all whole numbers from 1 to X), and for the normal distribution "exp" refers to the exponential function, which we discussed in the Chapter on Data Handling. If these equations don't make a lot of sense to you, don't worry too much about them.

Binomial	Normal
$\$P(X \theta, N) = \frac{N!}{X!(N-X)!} \theta^X (1-\theta)^{N-X}$	$\$p(X \mu, \sigma) = \frac{1}{\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$

Table 9.3: The naming system for R probability distribution functions. Every probability distribution implemented in R is actually associated with four separate functions, and there is a pretty standardised way for naming these functions.

What.it.does	Prefix	Normal.distribution	Binomial.distribution
probability (density) of	d	dnorm()	dbinom()
cumulative probability of	p	dnorm()	pbinom()
generate random number from	r	rnorm()	rbinom()
q qnorm() qbinom()	q	qnorm()	qbinom()

```
knitr::kable(data.frame(stringsAsFactors=FALSE, Binomial = c("$P(X | \theta, N) = \frac{N!}{X!(N-X)!} \theta^X (1-\theta)^{N-X}$", Normal = c("$p(X | \mu, \sigma) = \frac{1}{\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$"))
```

At this point, I should probably explain the name of the `dbinom()` function. Obviously, the "binom" part comes from the fact that we're working with the binomial distribution, but the "d" prefix is probably a bit of a mystery. In this section I'll give a partial explanation: specifically, I'll explain why there is a prefix. As for why it's a "d" specifically, you'll have to wait until the next section. What's going on here is that R actually provides *four* functions in relation to the binomial distribution. These four functions are `dbinom()`, `pbinom()`, `rbinom()` and `qbinom()`, and each one calculates a different quantity of interest. Not only that, R does the same thing for *every* probability distribution that it implements. No matter what distribution you're talking about, there's a `d` function, a `p` function, a `q` function and a `r` function. This is illustrated in Table 9.3, using the binomial distribution and the normal distribution as examples.

Let's have a look at what all four functions do. Firstly, all four versions of the function require you to specify the `size` and `prob` arguments: no matter what you're trying to get R to calculate, it needs to know what the parameters are. However, they differ in terms of what the other argument is, and what the output is. So let's look at them one at a time.

- The `d` form we've already seen: you specify a particular outcome `x`, and the output is the probability of obtaining exactly that outcome. (the "d" is short for **density**, but ignore that for now).
- The `p` form calculates the **cumulative probability**. You specify a particular quantile `q`, and it tells you the probability of obtaining an outcome *smaller than or equal to* `q`.
- The `q` form calculates the **quantiles** of the distribution. You specify a probability value `p`, and gives you the corresponding percentile. That is, the value of the variable for which there's a probability `p` of obtaining an outcome lower than that value.
- The `r` form is a **random number generator**: specifically, it generates `n` random outcomes from the distribution.

This is a little abstract, so let's look at some concrete examples. Again, we've already covered `dbinom()` so let's focus on the other three versions. We'll start with `pbinom()`, and we'll go back to the skull-dice example. Again, I'm rolling 20 dice, and each die has a 1 in 6 chance of coming up skulls. Suppose, however, that I want to know the probability of rolling 4 *or fewer* skulls. If I wanted to, I could use the `dbinom()`

function to calculate the exact probability of rolling 0 skulls, 1 skull, 2 skulls, 3 skulls and 4 skulls and then add these up, but there's a faster way. Instead, I can calculate this using the `pbinom()` function. Here's the command:

```
pbinom( q = 4, size = 20, prob = 1/6)
```

```
## [1] 0.7687492
```

In other words, there is a 76.9% chance that I will roll 4 or fewer skulls. Or, to put it another way, R is telling us that a value of 4 is actually the 76.9th percentile of this binomial distribution.

Next, let's consider the `qbinom()` function. Let's say I want to calculate the 75th percentile of the binomial distribution. If we're sticking with our skulls example, I would use the following command to do this:

```
qbinom( p = 0.75, size = 20, prob = 1/6)
```

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

Hm. There's something odd going on here. Let's think this through. What the `qbinom()` function appears to be telling us is that the 75th percentile of the binomial distribution is 4, even though we saw from the `pbinom()` function that 4 is *actually* the 76.9th percentile. And it's definitely the `pbinom()` function that is correct. I promise. The weirdness here comes from the fact that our binomial distribution doesn't really *have* a 75th percentile. Not really. Why not? Well, there's a 56.7% chance of rolling 3 or fewer skulls (you can type `pbinom(3, 20, 1/6)` to confirm this if you want), and a 76.9% chance of rolling 4 or fewer skulls. So there's a sense in which the 75th percentile should lie "in between" 3 and 4 skulls. But that makes no sense at all! You can't roll 20 dice and get 3.9 of them come up skulls. This issue can be handled in different ways: you could report an in between value (or *interpolated* value, to use the technical name) like 3.9, you could round down (to 3) or you could round up (to 4). The `qbinom()` function rounds upwards: if you ask for a percentile that doesn't actually exist (like the 75th in this example), R finds the smallest value for which the the percentile rank is *at least* what you asked for. In this case, since the "true" 75th percentile (whatever that would mean) lies somewhere between 3 and 4 skulls, R rounds up and gives you an answer of 4. This subtlety is tedious, I admit, but thankfully it's only an issue for discrete distributions like the binomial (see Section 2.2.5 for a discussion of continuous versus discrete). The other distributions that I'll talk about (normal, *t*, χ^2 and *F*) are all continuous, and so R can always return an exact quantile whenever you ask for it.

Finally, we have the random number generator. To use the `rbinom()` function, you specify how many times R should "simulate" the experiment using the `n` argument, and it will generate random outcomes from the binomial distribution. So, for instance, suppose I were to repeat my die rolling experiment 100 times. I could get R to simulate the results of these experiments by using the following command:

```
rbinom( n = 100, size = 20, prob = 1/6 )
```

```
## [1] 3 2 9 2 4 4 3 7 1 0 1 5 3 5 4 3 3 2 3 1 4 3 2 3 2 0 4 2 4 4 6 1 3 4 7
## [36] 5 4 4 3 4 2 3 1 3 3 4 6 6 2 5 9 1 5 2 3 4 1 3 4 3 4 4 4 4 2 1 3 2 6 3
## [71] 2 4 6 4 4 2 4 1 5 4 2 4 8 3 3 2 3 5 5 3 1 2 3 4 6 2 2 2 1 2
```

As you can see, these numbers are pretty much what you'd expect given the distribution shown in Figure 9.3. Most of the time I roll somewhere between 1 to 5 skulls. There are a lot of subtleties associated with random number generation using a computer,³ but for the purposes of this book we don't need to worry too much about them.

³Since computers are deterministic machines, they can't actually produce truly random behaviour. Instead, what they do is take advantage of various mathematical functions that share a lot of similarities with true randomness. What this means is that any random numbers generated on a computer are *pseudorandom*, and the quality of those numbers depends on the specific method used. By default R uses the "Mersenne twister" method. In any case, you can find out more by typing `?Random`, but as usual the R help files are fairly dense.

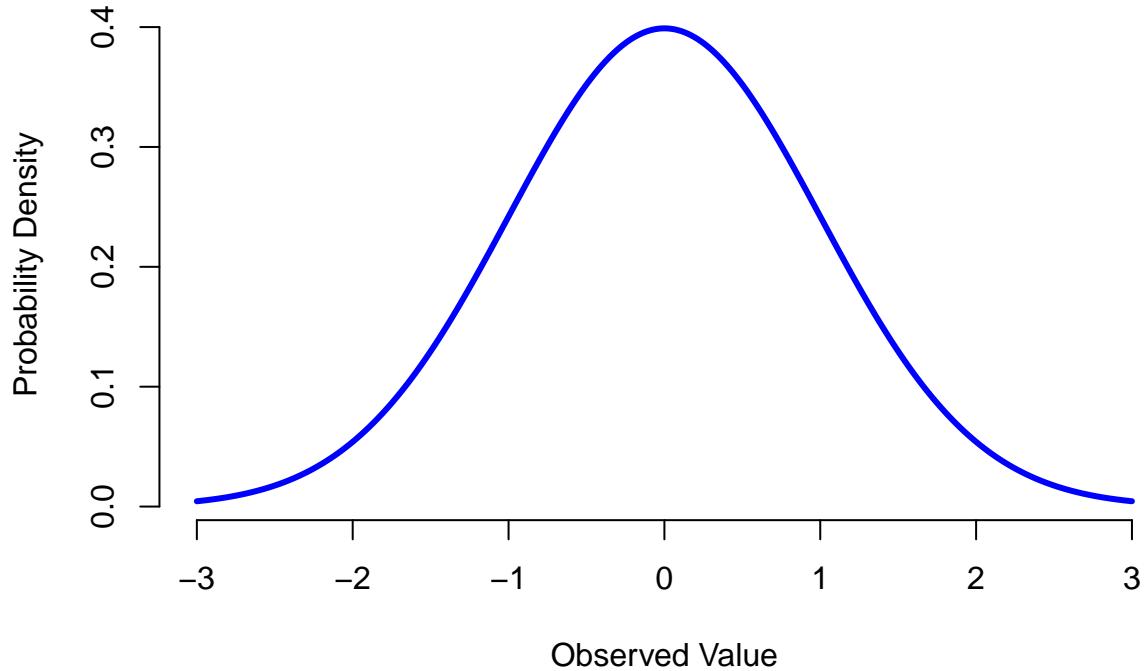


Figure 9.6: {The normal distribution with mean $mu = 0$ and standard deviation $sigma = 1$. The x -axis corresponds to the value of some variable, and the y -axis tells us something about how likely we are to observe that value. However, notice that the y -axis is labelled “Probability Density” and not “Probability”. There is a subtle and somewhat frustrating characteristic of continuous distributions that makes the y axis behave a bit oddly: the height of the curve here isn’t actually the probability of observing a particular x value. On the other hand, it *is* true that the heights of the curve tells you which x values are more likely (the higher ones!).}

9.5 The normal distribution

While the binomial distribution is conceptually the simplest distribution to understand, it’s not the most important one. That particular honour goes to the ***normal distribution***, which is also referred to as “the bell curve” or a “Gaussian distribution”. A normal distribution is described using two parameters, the mean of the distribution μ and the standard deviation of the distribution σ . The notation that we sometimes use to say that a variable X is normally distributed is as follows:

$$X \sim \text{Normal}(\mu, \sigma)$$

Of course, that’s just notation. It doesn’t tell us anything interesting about the normal distribution itself. As was the case with the binomial distribution, I have included the formula for the normal distribution in this book, because I think it’s important enough that everyone who learns statistics should at least look at it, but since this is an introductory text I don’t want to focus on it, so I’ve tucked it away in Table 9.2. Similarly, the R functions for the normal distribution are `dnorm()`, `pnorm()`, `qnorm()` and `rnorm()`. However, they behave in pretty much exactly the same way as the corresponding functions for the binomial distribution, so there’s not a lot that you need to know. The only thing that I should point out is that the

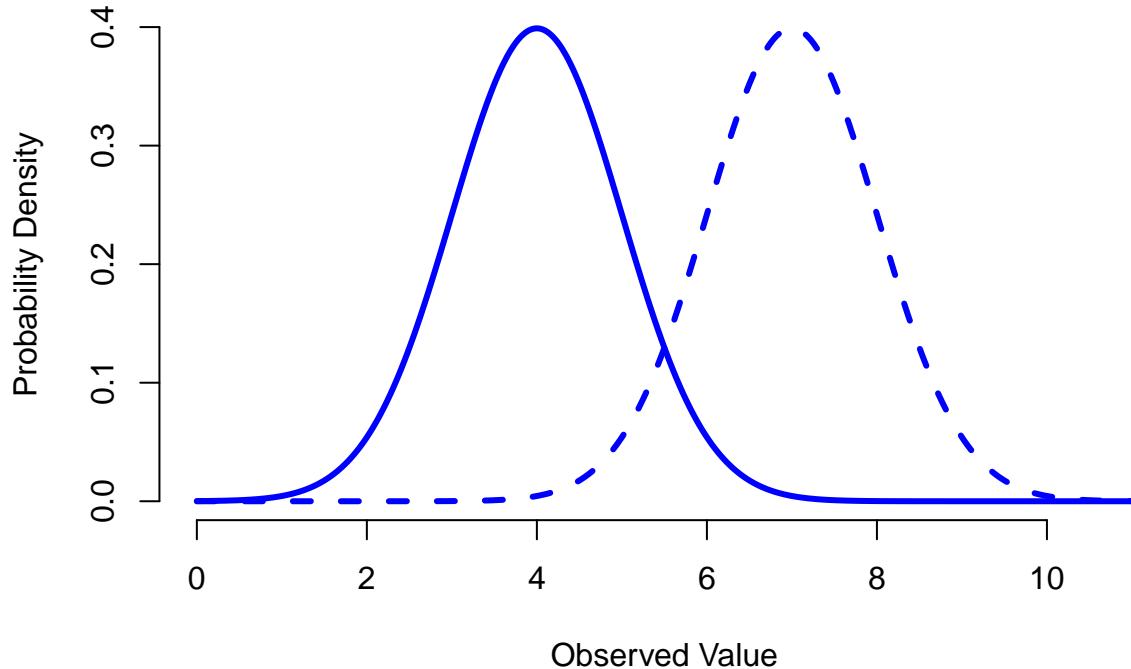


Figure 9.7: An illustration of what happens when you change the mean of a normal distribution. The solid line depicts a normal distribution with a mean of $\mu = 4$. The dashed line shows a normal distribution with a mean of $\mu = 7$. In both cases, the standard deviation is $\sigma = 1$. Not surprisingly, the two distributions have the same shape, but the dashed line is shifted to the right.

argument names for the parameters are `mean` and `sd`. In pretty much every other respect, there's nothing else to add.

Instead of focusing on the maths, let's try to get a sense for what it means for a variable to be normally distributed. To that end, have a look at Figure 9.6, which plots a normal distribution with mean $\mu = 0$ and standard deviation $\sigma = 1$. You can see where the name “bell curve” comes from: it looks a bit like a bell. Notice that, unlike the plots that I drew to illustrate the binomial distribution, the picture of the normal distribution in Figure 9.6 shows a smooth curve instead of “histogram-like” bars. This isn't an arbitrary choice: the normal distribution is continuous, whereas the binomial is discrete. For instance, in the die rolling example from the last section, it was possible to get 3 skulls or 4 skulls, but impossible to get 3.9 skulls. The figures that I drew in the previous section reflected this fact: in Figure 9.3, for instance, there's a bar located at $X = 3$ and another one at $X = 4$, but there's nothing in between. Continuous quantities don't have this constraint. For instance, suppose we're talking about the weather. The temperature on a pleasant Spring day could be 23 degrees, 24 degrees, 23.9 degrees, or anything in between since temperature is a continuous variable, and so a normal distribution might be quite appropriate for describing Spring temperatures.⁴

With this in mind, let's see if we can't get an intuition for how the normal distribution works. Firstly, let's have a look at what happens when we play around with the parameters of the distribution. To that end,

⁴In practice, the normal distribution is so handy that people tend to use it even when the variable isn't actually continuous. As long as there are enough categories (e.g., Likert scale responses to a questionnaire), it's pretty standard practice to use the normal distribution as an approximation. This works out much better in practice than you'd think.

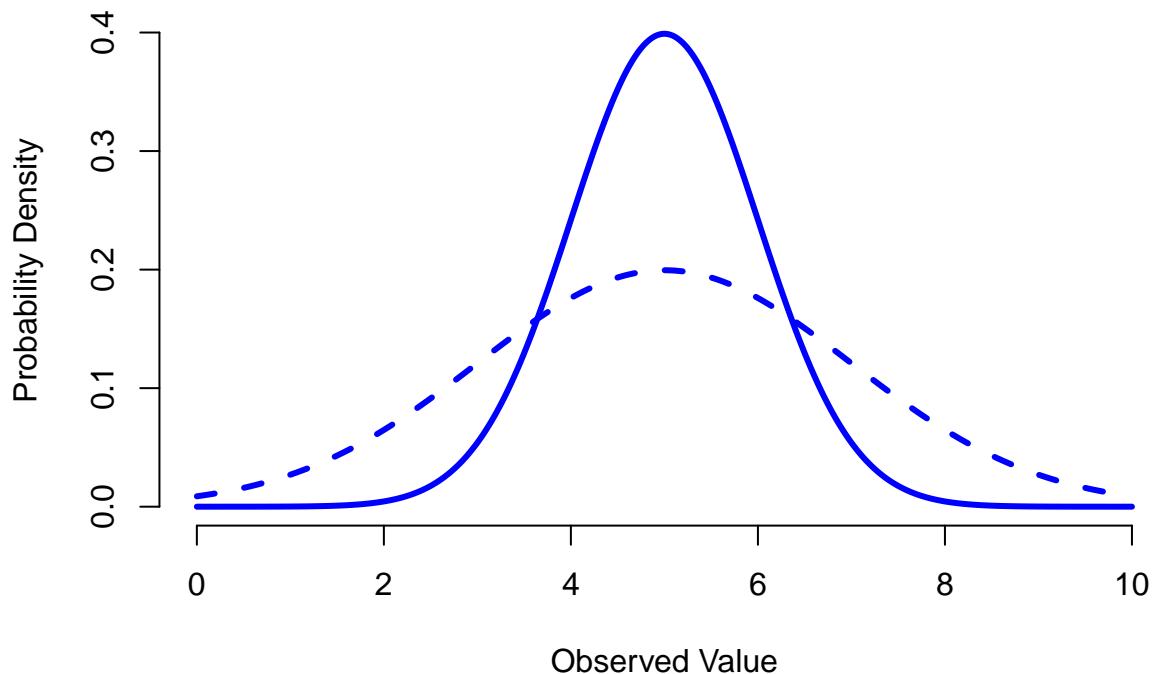


Figure 9.8: An illustration of what happens when you change the the standard deviation of a normal distribution. Both distributions plotted in this figure have a mean of $\mu = 5$, but they have different standard deviations. The solid line plots a distribution with standard deviation $\sigma = 1$, and the dashed line shows a distribution with standard deviation $\sigma = 2$. As a consequence, both distributions are “centred” on the same spot, but the dashed line is wider than the solid one.

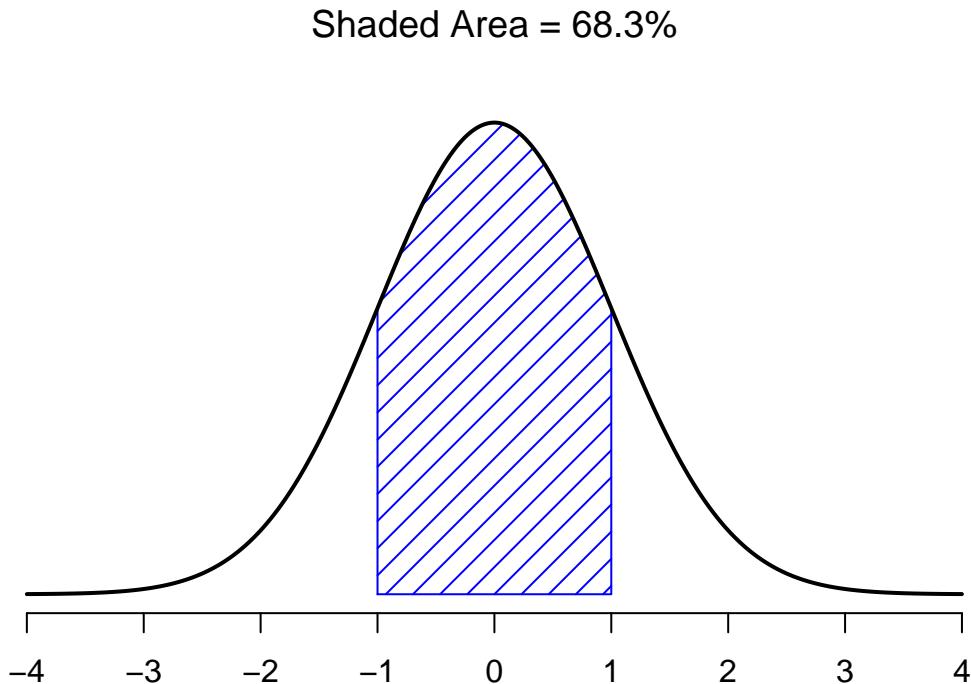


Figure 9.9: The area under the curve tells you the probability that an observation falls within a particular range. The solid lines plot normal distributions with mean $\mu = 0$ and standard deviation $\sigma = 1$. The shaded areas illustrate “areas under the curve” for two important cases. Here we can see that there is a 68.3% chance that an observation will fall within one standard deviation of the mean

Figure 9.7 plots normal distributions that have different means, but have the same standard deviation. As you might expect, all of these distributions have the same “width”. The only difference between them is that they’ve been shifted to the left or to the right. In every other respect they’re identical. In contrast, if we increase the standard deviation while keeping the mean constant, the peak of the distribution stays in the same place, but the distribution gets wider, as you can see in Figure 9.8. Notice, though, that when we widen the distribution, the height of the peak shrinks. This has to happen: in the same way that the heights of the bars that we used to draw a discrete binomial distribution have to *sum* to 1, the total *area under the curve* for the normal distribution must equal 1. Before moving on, I want to point out one important characteristic of the normal distribution. Irrespective of what the actual mean and standard deviation are, 68.3% of the area falls within 1 standard deviation of the mean. Similarly, 95.4% of the distribution falls within 2 standard deviations of the mean, and 99.7% of the distribution is within 3 standard deviations. This idea is illustrated in Figure ??.

9.5.1 Probability density

There’s something I’ve been trying to hide throughout my discussion of the normal distribution, something that some introductory textbooks omit completely. They might be right to do so: this “thing” that I’m hiding is weird and counterintuitive even by the admittedly distorted standards that apply in statistics. Fortunately, it’s not something that you need to understand at a deep level in order to do basic statistics:

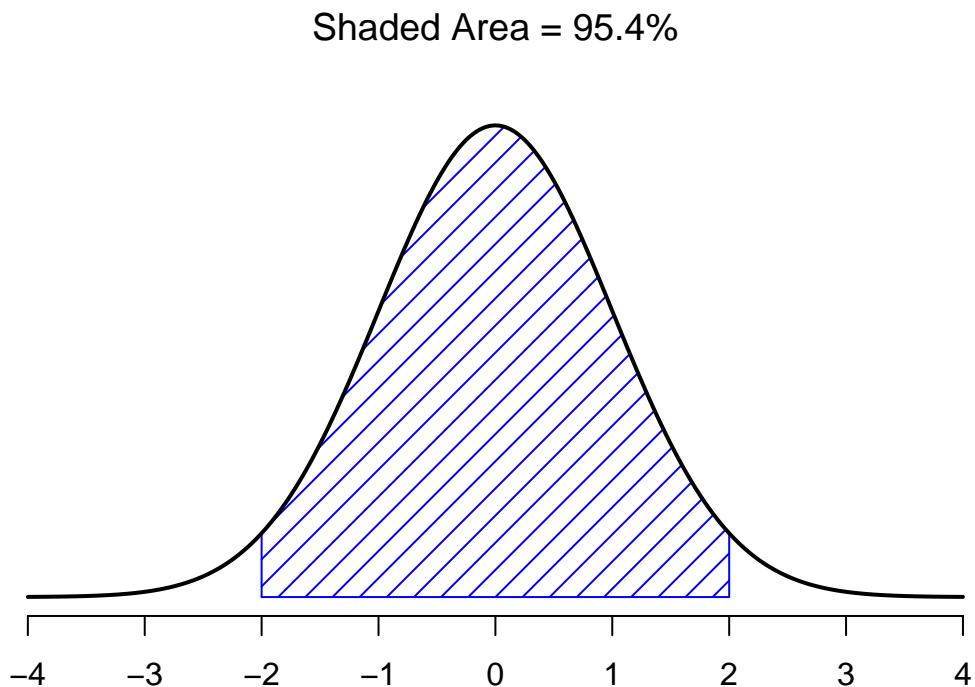


Figure 9.10: The area under the curve tells you the probability that an observation falls within a particular range. The solid lines plot normal distributions with mean $\mu = 0$ and standard deviation $\sigma = 1$. The shaded areas illustrate “areas under the curve” for two important cases. Here we see that there is a 95.4% chance that an observation will fall within two standard deviations of the mean.

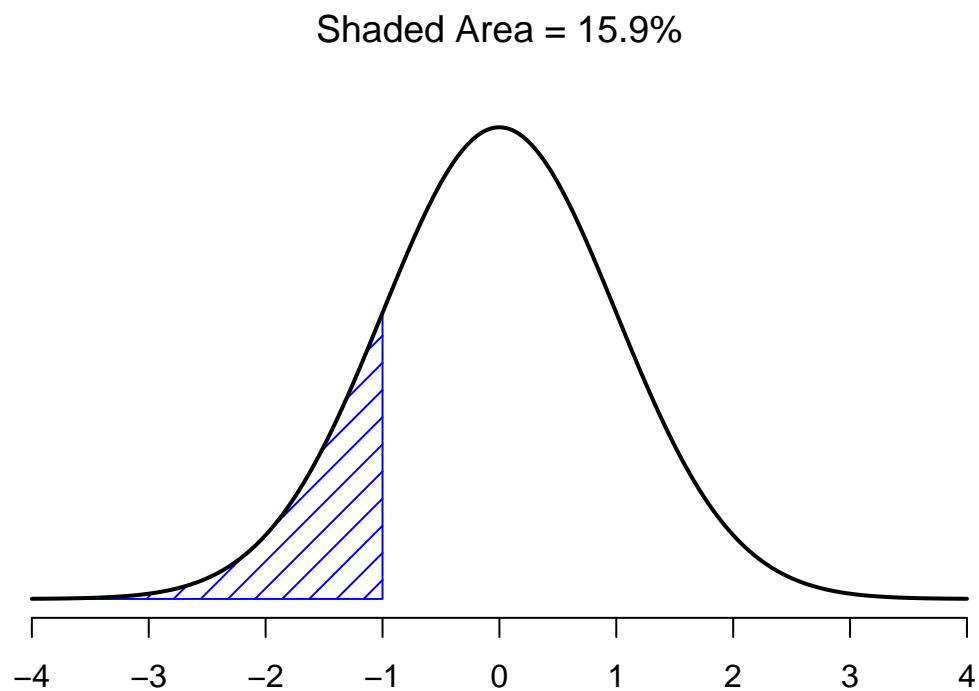


Figure 9.11: Two more examples of the “area under the curve idea”. There is a 15.9% chance that an observation is one standard deviation below the mean or smaller

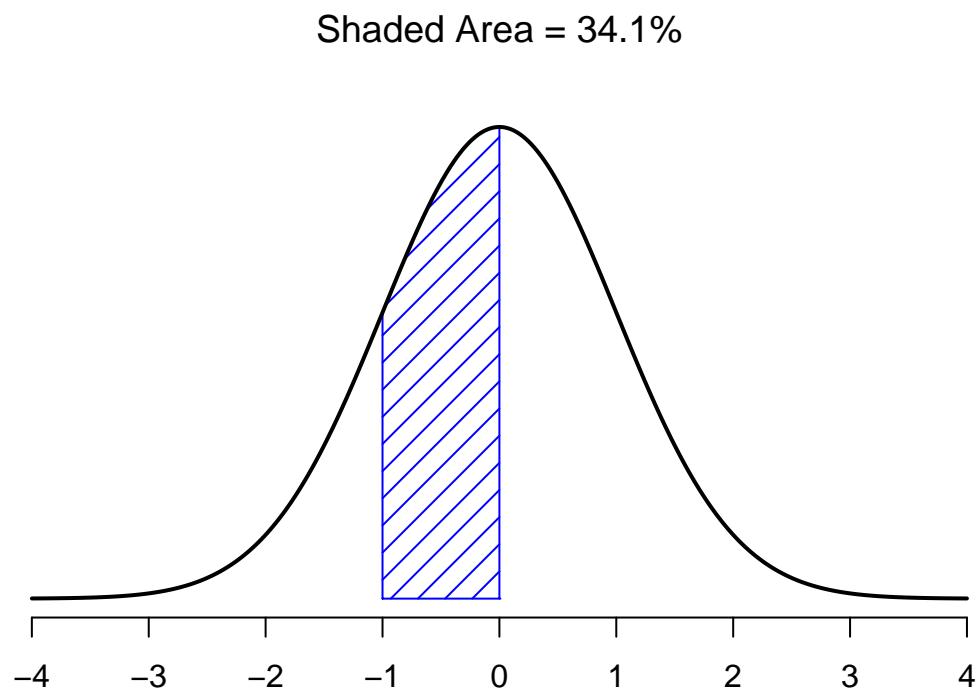


Figure 9.12: There is a 34.1% chance that the observation is greater than one standard deviation below the mean but still below the mean. Notice that if you add these two numbers together you get $15.9 + 34.1 = 50$. For normally distributed data, there is a 50% chance that an observation falls below the mean. And of course that also implies that there is a 50% chance that it falls above the mean.

rather, it's something that starts to become important later on when you move beyond the basics. So, if it doesn't make complete sense, don't worry: try to make sure that you follow the gist of it.

Throughout my discussion of the normal distribution, there's been one or two things that don't quite make sense. Perhaps you noticed that the y -axis in these figures is labelled "Probability Density" rather than density. Maybe you noticed that I used $p(X)$ instead of $P(X)$ when giving the formula for the normal distribution. Maybe you're wondering why R uses the "d" prefix for functions like `dnorm()`. And maybe, just maybe, you've been playing around with the `dnorm()` function, and you accidentally typed in a command like this:

```
dnorm( x = 1, mean = 1, sd = 0.1 )  
  
## [1] 3.989423
```

And if you've done the last part, you're probably very confused. I've asked R to calculate the probability that $x = 1$, for a normally distributed variable with `mean = 1` and standard deviation `sd = 0.1`; and it tells me that the probability is 3.99. But, as we discussed earlier, probabilities *can't* be larger than 1. So either I've made a mistake, or that's not a probability.

As it turns out, the second answer is correct. What we've calculated here isn't actually a probability: it's something else. To understand what that something is, you have to spend a little time thinking about what it really *means* to say that X is a continuous variable. Let's say we're talking about the temperature outside. The thermometer tells me it's 23 degrees, but I know that's not really true. It's not *exactly* 23 degrees. Maybe it's 23.1 degrees, I think to myself. But I know that that's not really true either, because it might actually be 23.09 degrees. But, I know that... well, you get the idea. The tricky thing with genuinely continuous quantities is that you never really know exactly what they are.

Now think about what this implies when we talk about probabilities. Suppose that tomorrow's maximum temperature is sampled from a normal distribution with mean 23 and standard deviation 1. What's the probability that the temperature will be *exactly* 23 degrees? The answer is "zero", or possibly, "a number so close to zero that it might as well be zero". Why is this? It's like trying to throw a dart at an infinitely small dart board: no matter how good your aim, you'll never hit it. In real life you'll never get a value of exactly 23. It'll always be something like 23.1 or 22.99998 or something. In other words, it's completely meaningless to talk about the probability that the temperature is exactly 23 degrees. However, in everyday language, if I told you that it was 23 degrees outside and it turned out to be 22.9998 degrees, you probably wouldn't call me a liar. Because in everyday language, "23 degrees" usually means something like "somewhere between 22.5 and 23.5 degrees". And while it doesn't feel very meaningful to ask about the probability that the temperature is exactly 23 degrees, it does seem sensible to ask about the probability that the temperature lies between 22.5 and 23.5, or between 20 and 30, or any other range of temperatures.

The point of this discussion is to make clear that, when we're talking about continuous distributions, it's not meaningful to talk about the probability of a specific value. However, what we *can* talk about is the probability that the value lies within a particular range of values. To find out the probability associated with a particular range, what you need to do is calculate the "area under the curve". We've seen this concept already: in Figures 9.9 and ??fig:sdnorm1b), the shaded areas shown depict genuine probabilities (e.g., in Figure 9.9 it shows the probability of observing a value that falls within 1 standard deviation of the mean).

Okay, so that explains part of the story. I've explained a little bit about how continuous probability distributions should be interpreted (i.e., area under the curve is the key thing), but I haven't actually explained what the `dnorm()` function actually calculates. Equivalently, what does the formula for $p(x)$ that I described earlier actually mean? Obviously, $p(x)$ doesn't describe a probability, but what is it? The name for this quantity $p(x)$ is a **probability density**, and in terms of the plots we've been drawing, it corresponds to the *height* of the curve. The densities themselves aren't meaningful in and of themselves: but they're "rigged" to ensure that the *area* under the curve is always interpretable as genuine probabilities. To be honest, that's about as much as you really need to know for now.⁵

⁵For those readers who know a little calculus, I'll give a slightly more precise explanation. In the same way that probabilities

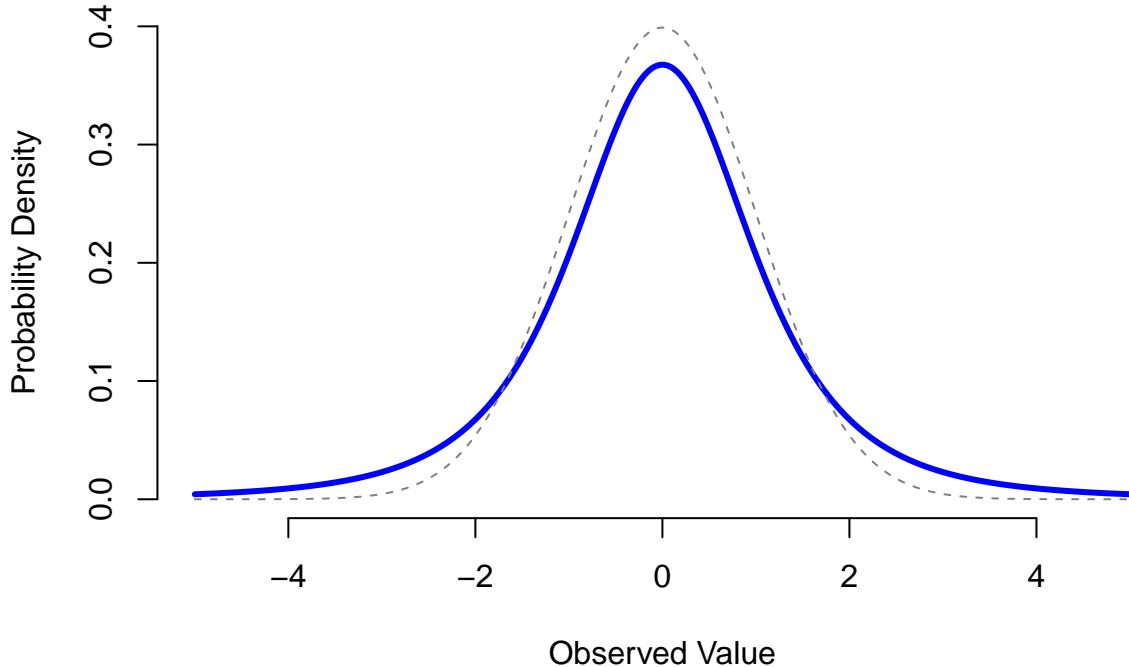


Figure 9.13: A t distribution with 3 degrees of freedom (solid line). It looks similar to a normal distribution, but it's not quite the same. For comparison purposes, I've plotted a standard normal distribution as the dashed line. Note that the “tails” of the t distribution are “heavier” (i.e., extend further outwards) than the tails of the normal distribution? That's the important difference between the two.

9.6 Other useful distributions

The normal distribution is the distribution that statistics makes most use of (for reasons to be discussed shortly), and the binomial distribution is a very useful one for lots of purposes. But the world of statistics is filled with probability distributions, some of which we'll run into in passing. In particular, the three that will appear in this book are the t distribution, the χ^2 distribution and the F distribution. I won't give formulas for any of these, or talk about them in too much detail, but I will show you some pictures.

- The t **distribution** is a continuous distribution that looks very similar to a normal distribution, but has heavier tails: see Figure 9.13. This distribution tends to arise in situations where you think that the data actually follow a normal distribution, but you don't know the mean or standard deviation. As you might expect, the relevant R functions are `dt()`, `pt()`, `qt()` and `rt()`, and we'll run into this distribution again in Chapter ??.
- The χ^2 **distribution** is another distribution that turns up in lots of different places. The situation in which we'll see it is when doing categorical data analysis (Chapter ??), but it's one of those things

are non-negative numbers that must sum to 1, probability densities are non-negative numbers that must integrate to 1 (where the integral is taken across all possible values of X). To calculate the probability that X falls between a and b we calculate the definite integral of the density function over the corresponding range, $\int_a^b p(x) dx$. If you don't remember or never learned calculus, don't worry about this. It's not needed for this book.

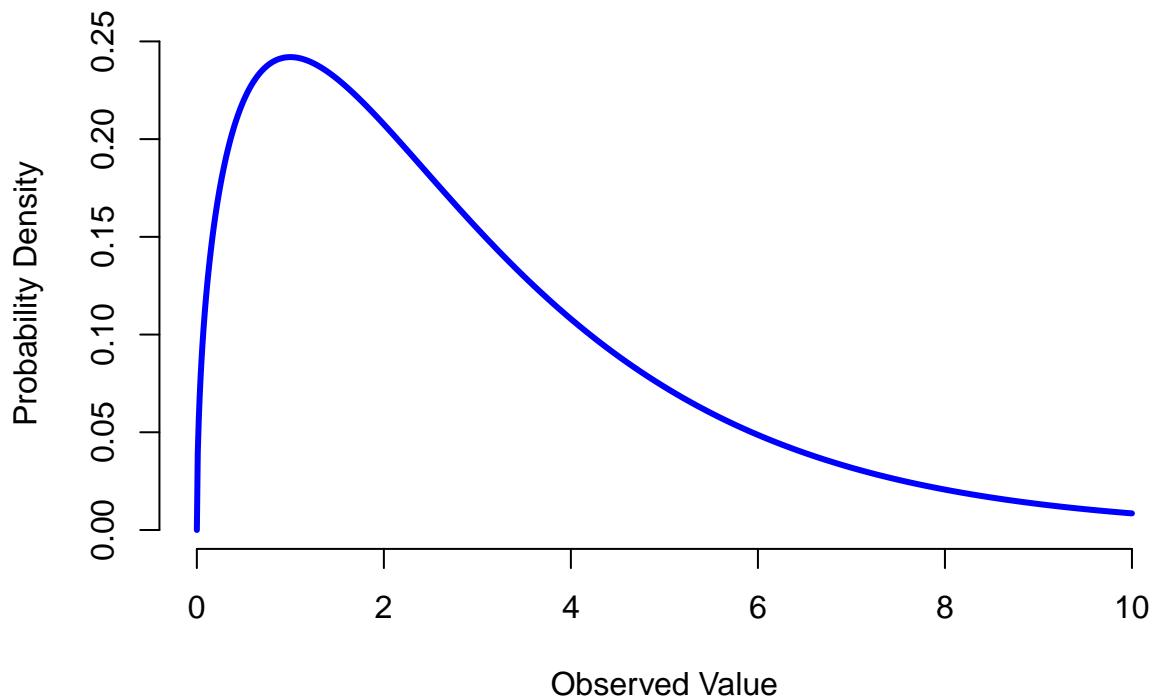


Figure 9.14: A χ^2 distribution with 3 degrees of freedom. Notice that the observed values must always be greater than zero, and that the distribution is pretty skewed. These are the key features of a chi-square distribution.

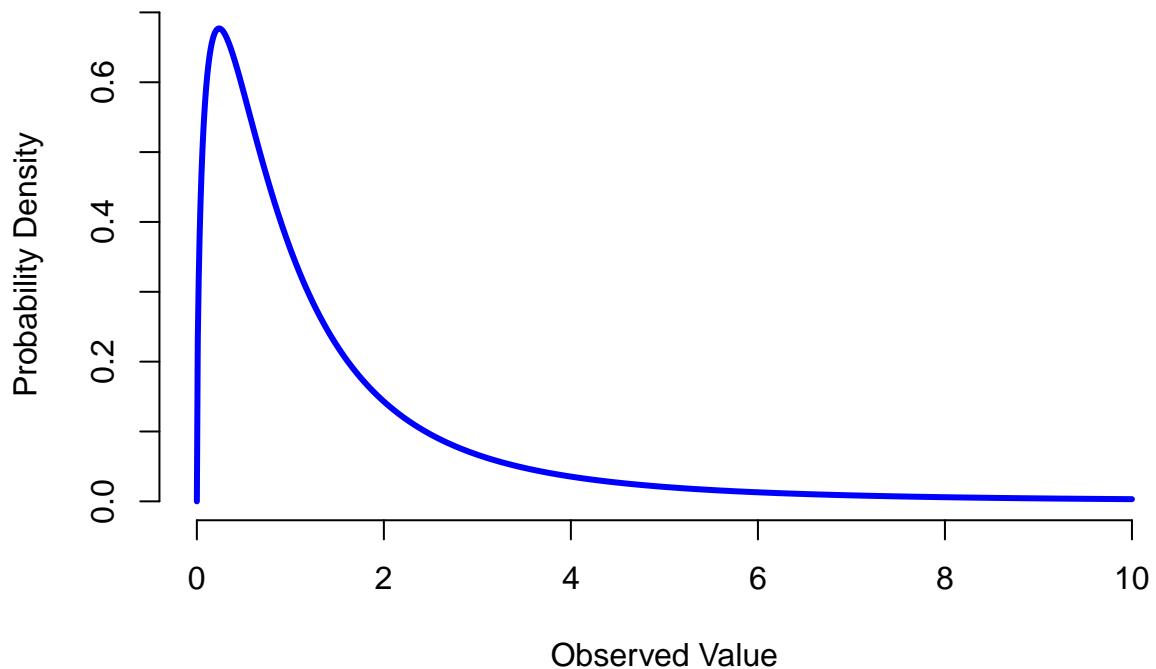


Figure 9.15: An F distribution with 3 and 5 degrees of freedom. Qualitatively speaking, it looks pretty similar to a chi-square distribution, but they're not quite the same in general.

that actually pops up all over the place. When you dig into the maths (and who doesn't love doing that?), it turns out that the main reason why the χ^2 distribution turns up all over the place is that, if you have a bunch of variables that are normally distributed, square their values and then add them up (a procedure referred to as taking a "sum of squares"), this sum has a χ^2 distribution. You'd be amazed how often this fact turns out to be useful. Anyway, here's what a χ^2 distribution looks like: Figure 9.14. Once again, the R commands for this one are pretty predictable: `dchisq()`, `pchisq()`, `qchisq()`, `rchisq()`.

- The *F distribution* looks a bit like a χ^2 distribution, and it arises whenever you need to compare two χ^2 distributions to one another. Admittedly, this doesn't exactly sound like something that any sane person would want to do, but it turns out to be very important in real world data analysis. Remember when I said that χ^2 turns out to be the key distribution when we're taking a "sum of squares"? Well, what that means is if you want to compare two different "sums of squares", you're probably talking about something that has an *F* distribution. Of course, as yet I still haven't given you an example of anything that involves a sum of squares, but I will... in Chapter ???. And that's where we'll run into the *F* distribution. Oh, and here's a picture: Figure 9.15. And of course we can get R to do things with *F* distributions just by using the commands `df()`, `pf()`, `qf()` and `rf()`.

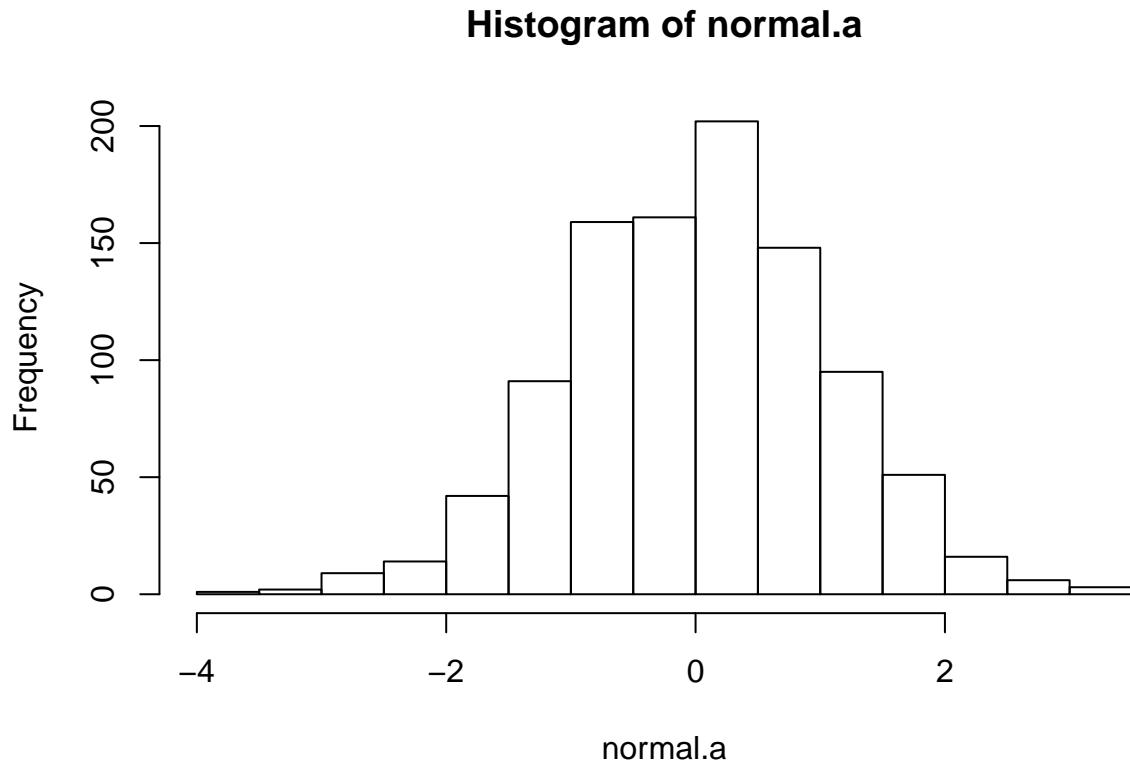
Because these distributions are all tightly related to the normal distribution and to each other, and because they are will turn out to be the important distributions when doing inferential statistics later in this book, I think it's useful to do a little demonstration using R, just to "convince ourselves" that these distributions really are related to each other in the way that they're supposed to be. First, we'll use the `rnorm()` function to generate 1000 normally-distributed observations:

```
normal.a <- rnorm( n=1000, mean=0, sd=1 )
print(head(normal.a))

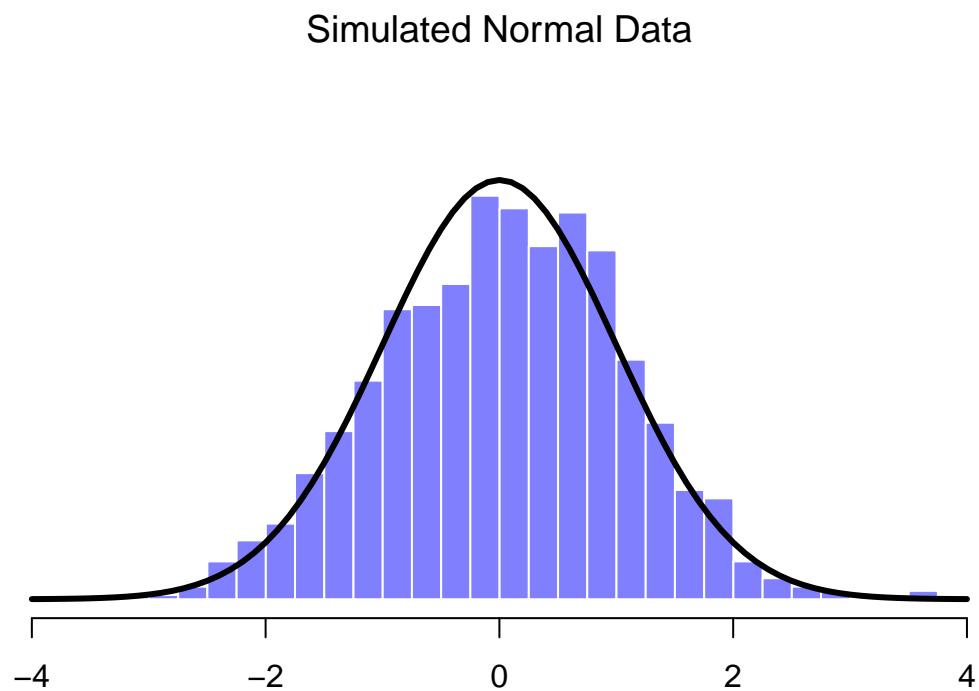
## [1]  0.002520116 -1.759249354 -0.055968257  0.879791922  1.166488549
## [6]  0.789723465
```

So the `normal.a` variable contains 1000 numbers that are normally distributed, and have mean 0 and standard deviation 1, and the actual print out of these numbers goes on for rather a long time. Note that, because the default parameters of the `rnorm()` function are `mean=0` and `sd=1`, I could have shortened the command to `rnorm(n=1000)`. In any case, what we can do is use the `hist()` function to draw a histogram of the data, like so:

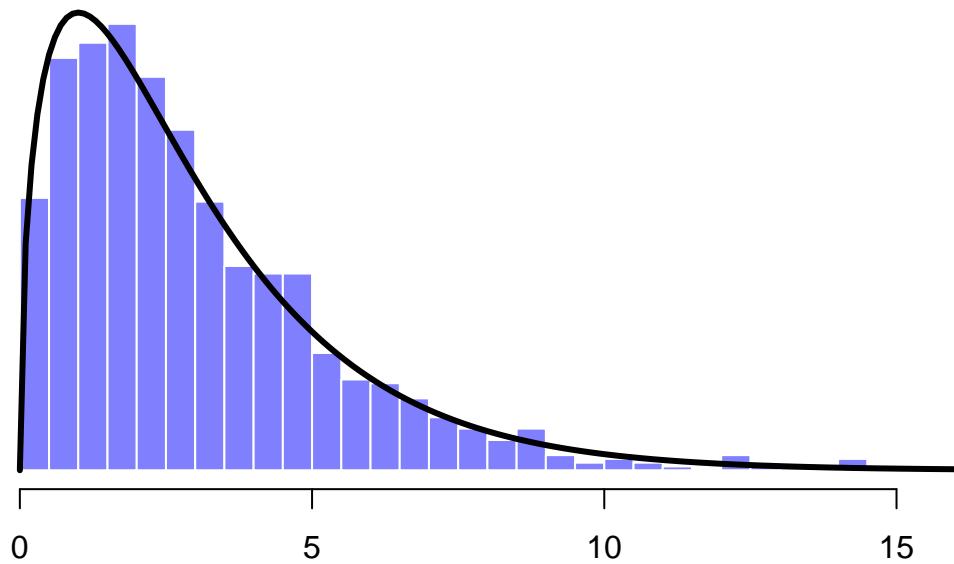
```
hist( normal.a )
```

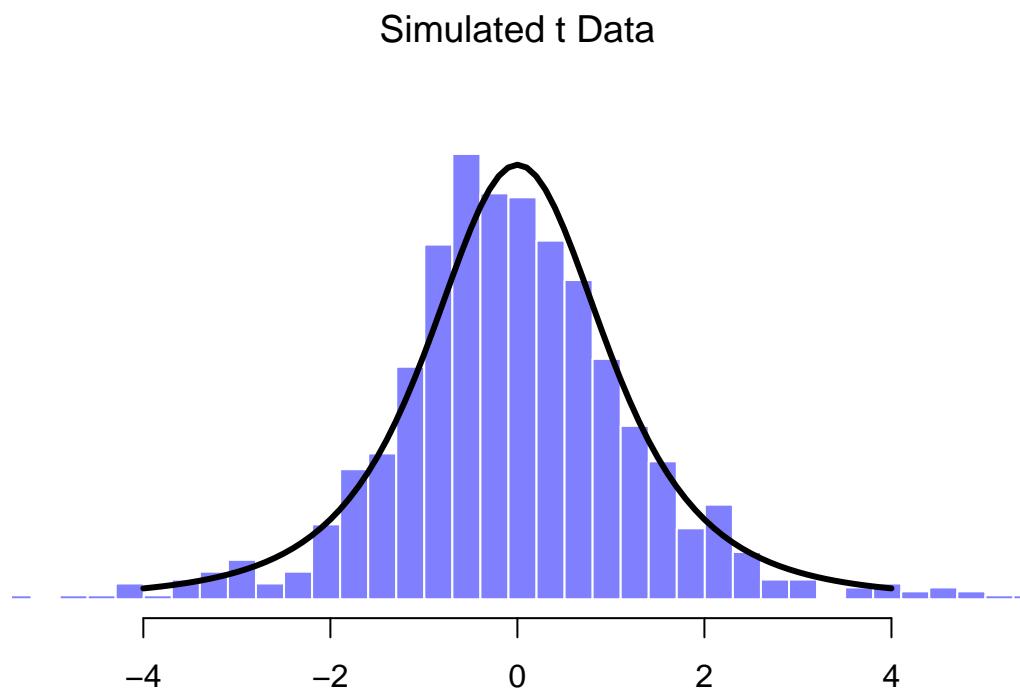


If you do this, you should see something similar to Figure ???. Your plot won't look quite as pretty as the one in the figure, of course, because I've played around with all the formatting (see Chapter 6), and I've also plotted the true distribution of the data as a solid black line (i.e., a normal distribution with mean 0 and standard deviation 1) so that you can compare the data that we just generated to the true distribution.

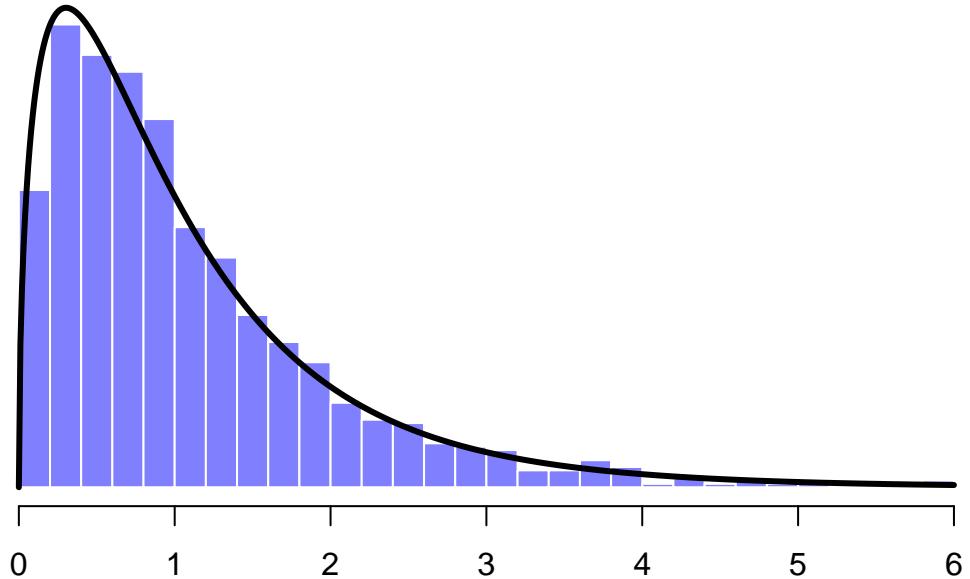


Simulated Chi–Square Data





Simulated F Data



In the previous example all I did was generate lots of normally distributed observations using `rnorm()` and then compared those to the true probability distribution in the figure (using `dnorm()` to generate the black line in the figure, but I didn't show the commands for that). Now let's try something trickier. We'll try to generate some observations that follow a chi-square distribution with 3 degrees of freedom, but instead of using `rchisq()`, we'll start with variables that are normally distributed, and see if we can exploit the known relationships between normal and chi-square distributions to do the work. As I mentioned earlier, a chi-square distribution with k degrees of freedom is what you get when you take k normally-distributed variables (with mean 0 and standard deviation 1), square them, and add them up. Since we want a chi-square distribution with 3 degrees of freedom, we'll need to supplement our `normal.a` data with two more sets of normally-distributed observations, imaginatively named `normal.b` and `normal.c`:

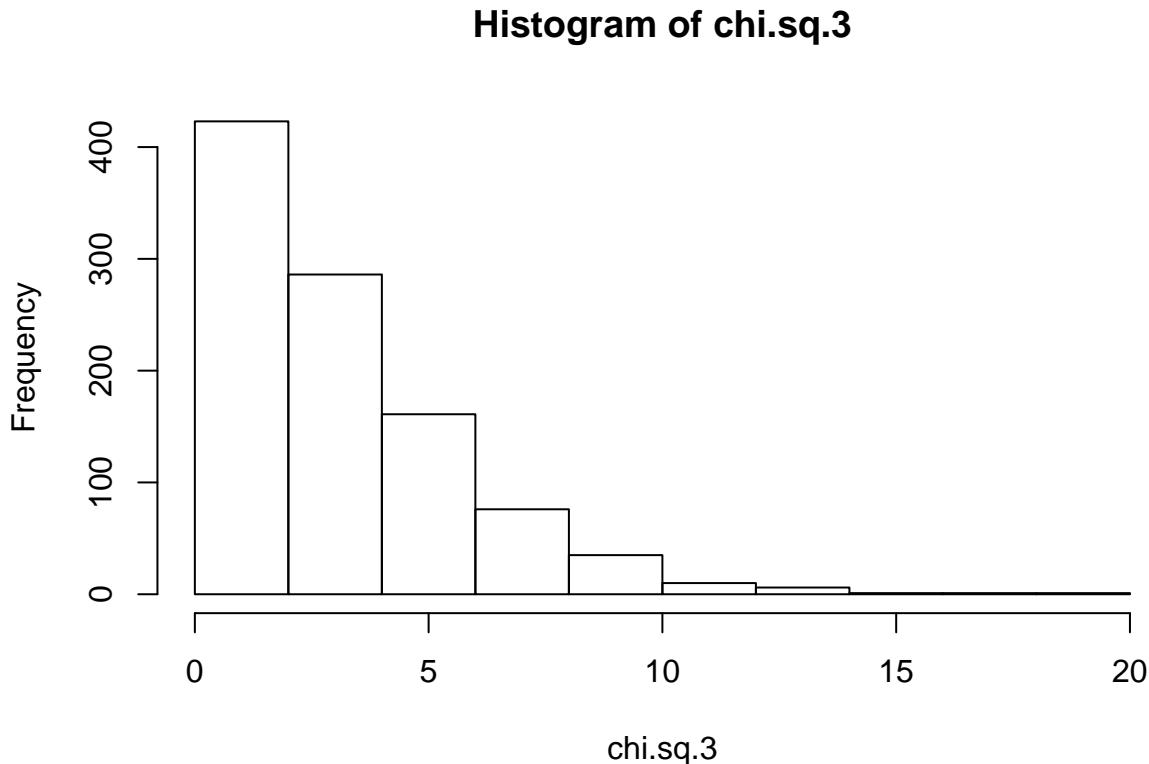
```
normal.b <- rnorm( n=1000 ) # another set of normally distributed data
normal.c <- rnorm( n=1000 ) # and another!
```

Now that we've done that, the theory says we should square these and add them together, like this

```
chi.sq.3 <- (normal.a)^2 + (normal.b)^2 + (normal.c)^2
```

and the resulting `chi.sq.3` variable should contain 1000 observations that follow a chi-square distribution with 3 degrees of freedom. You can use the `hist()` function to have a look at these observations yourself, using a command like this,

```
hist( chi.sq.3 )
```



and you should obtain a result that looks pretty similar to the chi-square plot in Figure ???. Once again, the plot that I've drawn is a little fancier: in addition to the histogram of `chi.sq.3`, I've also plotted a chi-square distribution with 3 degrees of freedom. It's pretty clear that – even though I used `rnorm()` to do all the work rather than `rchisq()` – the observations stored in the `chi.sq.3` variable really do follow a chi-square distribution. Admittedly, this probably doesn't seem all that interesting right now, but later on when we start encountering the chi-square distribution in Chapter ???, it will be useful to understand the fact that these distributions are related to one another.

We can extend this demonstration to the *t* distribution and the *F* distribution. Earlier, I implied that the *t* distribution is related to the normal distribution when the standard deviation is unknown. That's certainly true, and that's the what we'll see later on in Chapter ???, but there's a somewhat more precise relationship between the normal, chi-square and *t* distributions. Suppose we “scale” our chi-square data by dividing it by the degrees of freedom, like so

```
scaled.chi.sq.3 <- chi.sq.3 / 3
```

We then take a set of normally distributed variables and divide them by (the square root of) our scaled chi-square variable which had $df = 3$, and the result is a *t* distribution with 3 degrees of freedom:

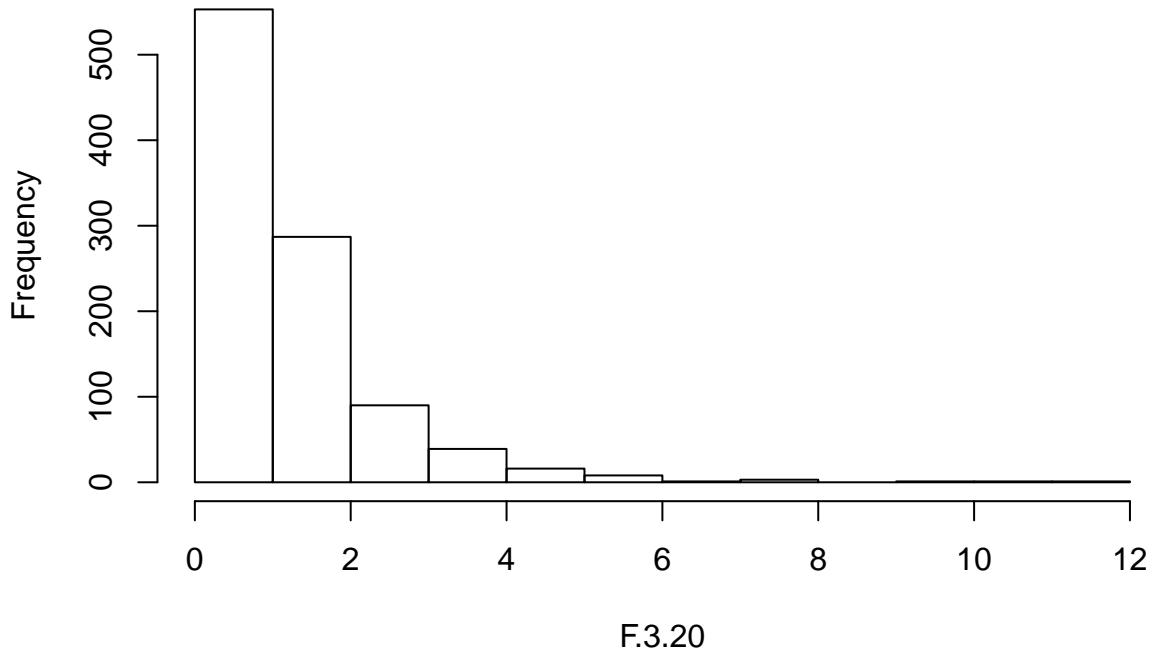
```
normal.d <- rnorm( n=1000 ) # yet another set of normally distributed data
t.3 <- normal.d / sqrt( scaled.chi.sq.3 ) # divide by square root of scaled chi-square to get t
```

If we plot the histogram of `t.3`, we end up with something that looks very similar to the *t* distribution in Figure ???. Similarly, we can obtain an *F* distribution by taking the ratio between two scaled chi-square distributions. Suppose, for instance, we wanted to generate data from an *F* distribution with 3 and 20 degrees of freedom. We could do this using `df()`, but we could also do the same thing by generating two

chi-square variables, one with 3 degrees of freedom, and the other with 20 degrees of freedom. As the example with `chi.sq.3` illustrates, we can actually do this using `rnorm()` if we really want to, but this time I'll take a short cut:

```
chi.sq.20 <- rchisq( 1000, 20) # generate chi square data with df = 20...
scaled.chi.sq.20 <- chi.sq.20 / 20 # scale the chi square variable...
F.3.20 <- scaled.chi.sq.3 / scaled.chi.sq.20 # take the ratio of the two chi squares...
hist( F.3.20 ) # ... and draw a picture
```

Histogram of F.3.20



The resulting `F.3.20` variable does in fact store variables that follow an F distribution with 3 and 20 degrees of freedom. This is illustrated in Figure ??, which plots the histogram of the observations stored in `F.3.20` against the true F distribution with $df_1 = 3$ and $df_2 = 20$. Again, they match.

Okay, time to wrap this section up. We've seen three new distributions: χ^2 , t and F . They're all continuous distributions, and they're all closely related to the normal distribution. I've talked a little bit about the precise nature of this relationship, and shown you some R commands that illustrate this relationship. The key thing for our purposes, however, is not that you have a deep understanding of all these different distributions, nor that you remember the precise relationships between them. The main thing is that you grasp the basic idea that these distributions are all deeply related to one another, and to the normal distribution. Later on in this book, we're going to run into data that are normally distributed, or at least assumed to be normally distributed. What I want you to understand right now is that, if you make the assumption that your data are normally distributed, you shouldn't be surprised to see χ^2 , t and F distributions popping up all over the place when you start trying to do your data analysis.

9.7 Summary

In this chapter we've talked about probability. We've talked what probability means, and why statisticians can't agree on what it means. We talked about the rules that probabilities have to obey. And we introduced the idea of a probability distribution, and spent a good chunk of the chapter talking about some of the more important probability distributions that statisticians work with. The section by section breakdown looks like this:

- Probability theory versus statistics (Section 9.1)
- Frequentist versus Bayesian views of probability (Section 9.2)
- Basics of probability theory (Section 9.3)
- Binomial distribution (Section 9.4), normal distribution (Section 9.5), and others (Section 9.6)

As you'd expect, my coverage is by no means exhaustive. Probability theory is a large branch of mathematics in its own right, entirely separate from its application to statistics and data analysis. As such, there are thousands of books written on the subject and universities generally offer multiple classes devoted entirely to probability theory. Even the "simpler" task of documenting standard probability distributions is a big topic. I've described five standard probability distributions in this chapter, but sitting on my bookshelf I have a 45-chapter book called "Statistical Distributions" Evans et al. (2011) that lists a *lot* more than that. Fortunately for you, very little of this is necessary. You're unlikely to need to know dozens of statistical distributions when you go out and do real world data analysis, and you definitely won't need them for this book, but it never hurts to know that there's other possibilities out there.

Picking up on that last point, there's a sense in which this whole chapter is something of a digression. Many undergraduate psychology classes on statistics skim over this content very quickly (I know mine did), and even the more advanced classes will often "forget" to revisit the basic foundations of the field. Most academic psychologists would not know the difference between probability and density, and until recently very few would have been aware of the difference between Bayesian and frequentist probability. However, I think it's important to understand these things before moving onto the applications. For example, there are a lot of rules about what you're "allowed" to say when doing statistical inference, and many of these can seem arbitrary and weird. However, they start to make sense if you understand that there is this Bayesian/frequentist distinction. Similarly, in Chapter ?? we're going to talk about something called the *t*-test, and if you really want to have a grasp of the mechanics of the *t*-test it really helps to have a sense of what a *t*-distribution actually looks like. You get the idea, I hope.

Chapter 10

Estimating unknown quantities from a sample

At the start of the last chapter I highlighted the critical distinction between *descriptive statistics* and *inferential statistics*. As discussed in Chapter 5, the role of descriptive statistics is to concisely summarise what we *do* know. In contrast, the purpose of inferential statistics is to “learn what we do not know from what we do”. Now that we have a foundation in probability theory, we are in a good position to think about the problem of statistical inference. What kinds of things would we like to learn about? And how do we learn them? These are the questions that lie at the heart of inferential statistics, and they are traditionally divided into two “big ideas”: estimation and hypothesis testing. The goal in this chapter is to introduce the first of these big ideas, estimation theory, but I’m going to witter on about sampling theory first because estimation theory doesn’t make sense until you understand sampling. As a consequence, this chapter divides naturally into two parts Sections 10.1 through 10.3 are focused on sampling theory, and Sections 10.4 and 10.5 make use of sampling theory to discuss how statisticians think about estimation.

10.1 Samples, populations and sampling

In the prelude to Part I discussed the riddle of induction, and highlighted the fact that *all* learning requires you to make assumptions. Accepting that this is true, our first task to come up with some fairly general assumptions about data that make sense. This is where **sampling theory** comes in. If probability theory is the foundations upon which all statistical theory builds, sampling theory is the frame around which you can build the rest of the house. Sampling theory plays a huge role in specifying the assumptions upon which your statistical inferences rely. And in order to talk about “making inferences” the way statisticians think about it, we need to be a bit more explicit about what it is that we’re drawing inferences *from* (the sample) and what it is that we’re drawing inferences *about* (the population).

In almost every situation of interest, what we have available to us as researchers is a **sample** of data. We might have run experiment with some number of participants; a polling company might have phoned some number of people to ask questions about voting intentions; etc. Regardless: the data set available to us is finite, and incomplete. We can’t possibly get every person in the world to do our experiment; a polling company doesn’t have the time or the money to ring up every voter in the country etc. In our earlier discussion of descriptive statistics (Chapter 5, this sample was the only thing we were interested in. Our only goal was to find ways of describing, summarising and graphing that sample. This is about to change.

10.1.1 Defining a population

A sample is a concrete thing. You can open up a data file, and there's the data from your sample. A **population**, on the other hand, is a more abstract idea. It refers to the set of all possible people, or all possible observations, that you want to draw conclusions about, and is generally *much* bigger than the sample. In an ideal world, the researcher would begin the study with a clear idea of what the population of interest is, since the process of designing a study and testing hypotheses about the data that it produces does depend on the population about which you want to make statements. However, that doesn't always happen in practice: usually the researcher has a fairly vague idea of what the population is and designs the study as best he/she can on that basis.

Sometimes it's easy to state the population of interest. For instance, in the “polling company” example that opened the chapter, the population consisted of all voters enrolled at the time of the study – millions of people. The sample was a set of 1000 people who all belong to that population. In most situations the situation is much less simple. In a typical psychological experiment, determining the population of interest is a bit more complicated. Suppose I run an experiment using 100 undergraduate students as my participants. My goal, as a cognitive scientist, is to try to learn something about how the mind works. So, which of the following would count as “the population”:

- All of the undergraduate psychology students at the University of Adelaide?
- Undergraduate psychology students in general, anywhere in the world?
- Australians currently living?
- Australians of similar ages to my sample?
- Anyone currently alive?
- Any human being, past, present or future?
- Any biological organism with a sufficient degree of intelligence operating in a terrestrial environment?
- Any intelligent being?

Each of these defines a real group of mind-possessing entities, all of which might be of interest to me as a cognitive scientist, and it's not at all clear which one ought to be the true population of interest. As another example, consider the Wellesley-Croker game that we discussed in the prelude. The sample here is a specific sequence of 12 wins and 0 losses for Wellesley. What is the population?

- All outcomes until Wellesley and Croker arrived at their destination?
- All outcomes if Wellesley and Croker had played the game for the rest of their lives?
- All outcomes if Wellseley and Croker lived forever and played the game until the world ran out of hills?
- All outcomes if we created an infinite set of parallel universes and the Wellesely/Croker pair made guesses about the same 12 hills in each universe?

Again, it's not obvious what the population is.

10.1.2 Simple random samples

Irrespective of how I define the population, the critical point is that the sample is a subset of the population, and our goal is to use our knowledge of the sample to draw inferences about the properties of the population. The relationship between the two depends on the *procedure* by which the sample was selected. This procedure is referred to as a **sampling method**, and it is important to understand why it matters.

To keep things simple, let's imagine that we have a bag containing 10 chips. Each chip has a unique letter printed on it, so we can distinguish between the 10 chips. The chips come in two colours, black and white. This set of chips is the population of interest, and it is depicted graphically on the left of Figure 10.1. As you can see from looking at the picture, there are 4 black chips and 6 white chips, but of course in real life we wouldn't know that unless we looked in the bag. Now imagine you run the following “experiment”:

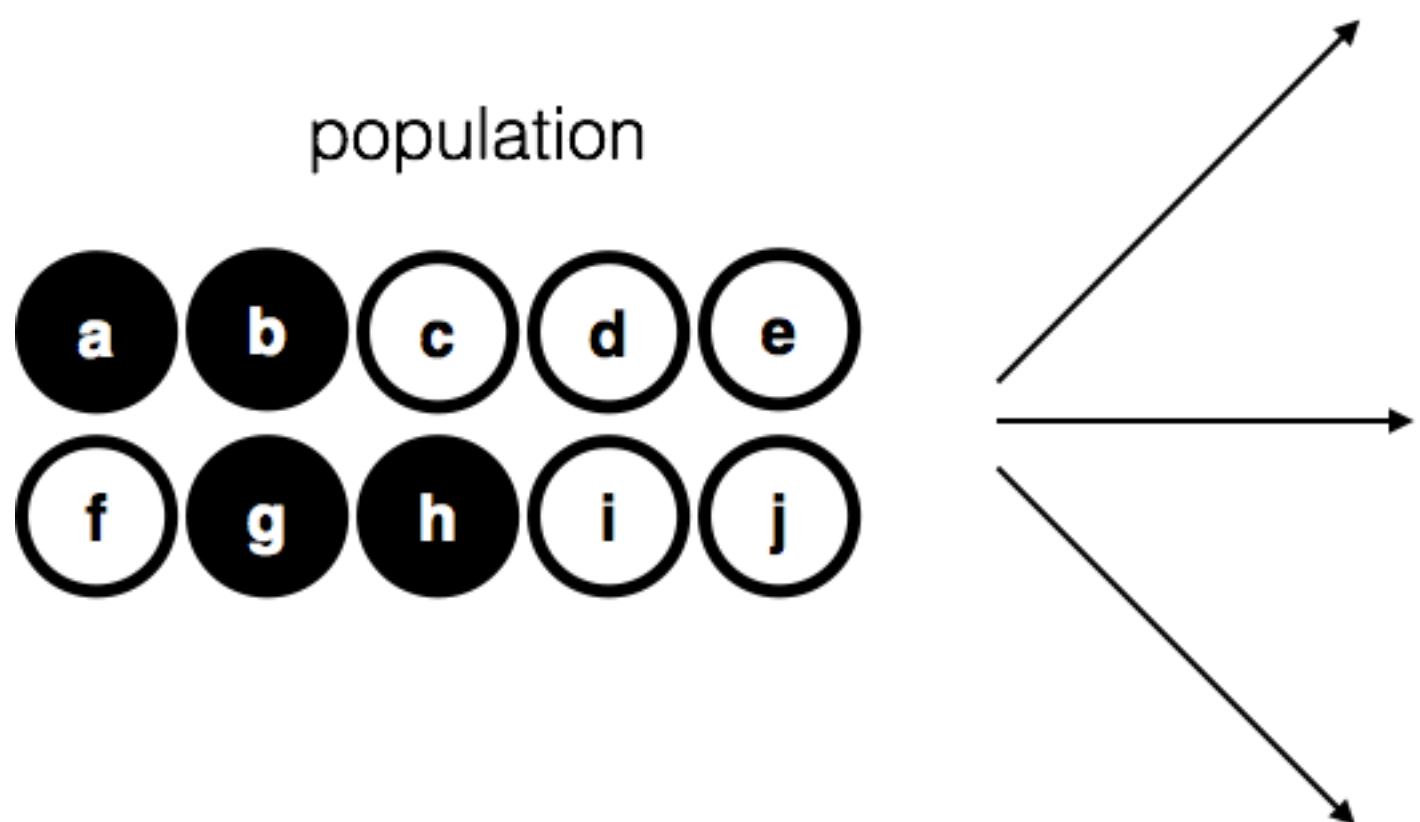


Figure 10.1: Simple random sampling without replacement from a finite population

you shake up the bag, close your eyes, and pull out 4 chips without putting any of them back into the bag. First out comes the a chip (black), then the c chip (white), then j (white) and then finally b (black). If you wanted, you could then put all the chips back in the bag and repeat the experiment, as depicted on the right hand side of Figure 10.1. Each time you get different results, but the procedure is identical in each case. The fact that the same procedure can lead to different results each time, we refer to it as a *random* process.¹ However, because we shook the bag before pulling any chips out, it seems reasonable to think that every chip has the same chance of being selected. A procedure in which every member of the population has the same chance of being selected is called a ***simple random sample***. The fact that we did *not* put the chips back in the bag after pulling them out means that you can't observe the same thing twice, and in such cases the observations are said to have been sampled ***without replacement***.

To help make sure you understand the importance of the sampling procedure, consider an alternative way in which the experiment could have been run. Suppose that my 5-year old son had opened the bag, and decided to pull out four black chips without putting any of them back in the bag. This *biased* sampling scheme is depicted in Figure 10.2. Now consider the evidentiary value of seeing 4 black chips and 0 white chips. Clearly, it depends on the sampling scheme, does it not? If you know that the sampling scheme is biased to select only black chips, then a sample that consists of only black chips doesn't tell you very much about the population! For this reason, statisticians really like it when a data set can be considered a simple random sample, because it makes the data analysis *much* easier.

A third procedure is worth mentioning. This time around we close our eyes, shake the bag, and pull out a chip. This time, however, we record the observation and then put the chip back in the bag. Again we close our eyes, shake the bag, and pull out a chip. We then repeat this procedure until we have 4 chips. Data sets generated in this way are still simple random samples, but because we put the chips back in the bag immediately after drawing them it is referred to as a sample ***with replacement***. The difference between this situation and the first one is that it is possible to observe the same population member multiple times, as illustrated in Figure 10.3.

In my experience, most psychology experiments tend to be sampling without replacement, because the same person is not allowed to participate in the experiment twice. However, most statistical theory is based on the assumption that the data arise from a simple random sample *with* replacement. In real life, this very rarely matters. If the population of interest is large (e.g., has more than 10 entities!) the difference between sampling with- and without- replacement is too small to be concerned with. The difference between simple random samples and biased samples, on the other hand, is not such an easy thing to dismiss.

10.1.3 Most samples are not simple random samples

As you can see from looking at the list of possible populations that I showed above, it is almost impossible to obtain a simple random sample from most populations of interest. When I run experiments, I'd consider it a minor miracle if my participants turned out to be a random sampling of the undergraduate psychology students at Adelaide university, even though this is by far the narrowest population that I might want to generalise to. A thorough discussion of other types of sampling schemes is beyond the scope of this book, but to give you a sense of what's out there I'll list a few of the more important ones:

- ***Stratified sampling***. Suppose your population is (or can be) divided into several different subpopulations, or *strata*. Perhaps you're running a study at several different sites, for example. Instead of trying to sample randomly from the population as a whole, you instead try to collect a separate random sample from each of the strata. Stratified sampling is sometimes easier to do than simple random sampling, especially when the population is already divided into the distinct strata. It can also be more efficient than simple random sampling, especially when some of the subpopulations are rare. For

¹The proper mathematical definition of randomness is extraordinarily technical, and way beyond the scope of this book. We'll be non-technical here and say that a process has an element of randomness to it whenever it is possible to repeat the process and get different answers each time.

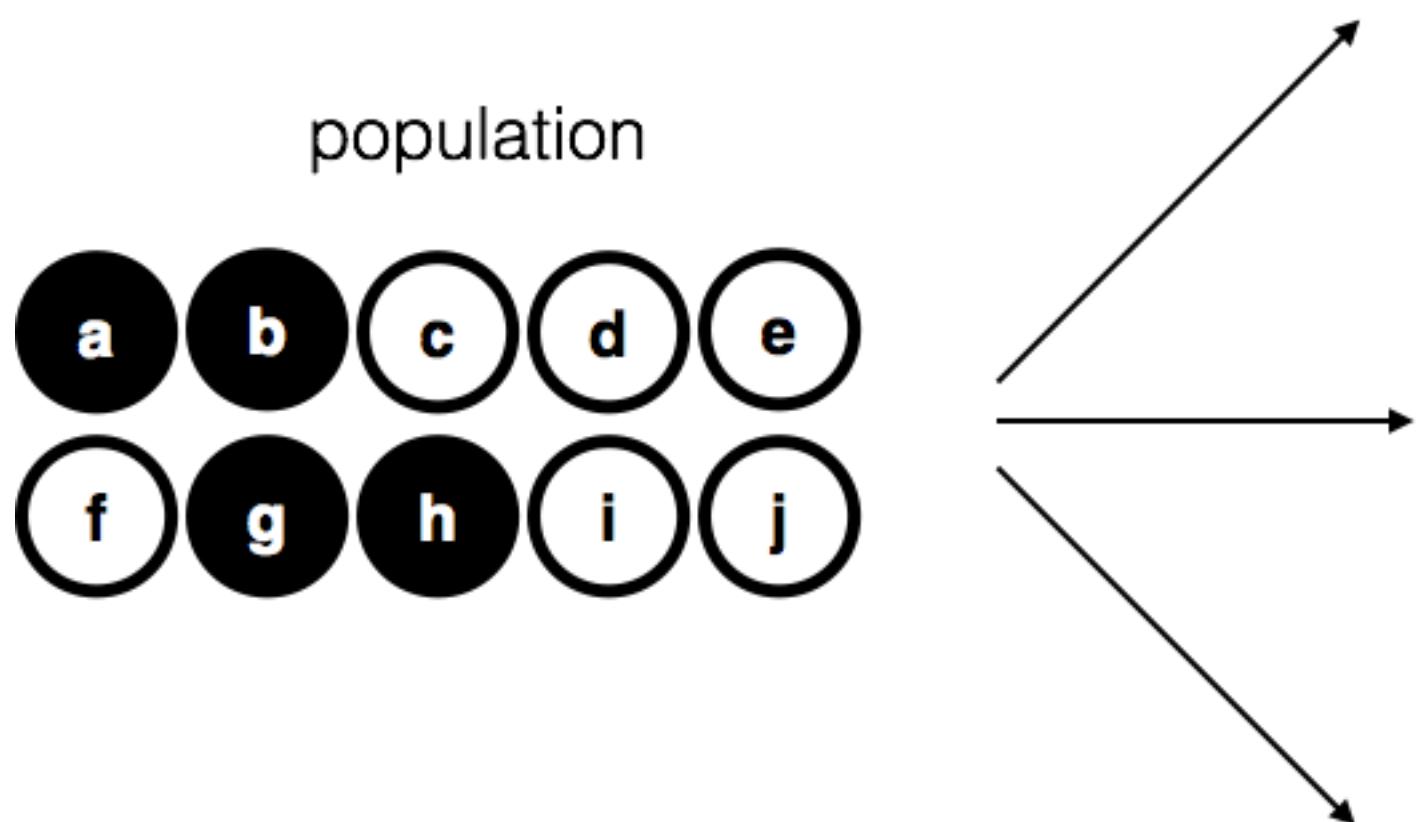


Figure 10.2: Biased sampling without replacement from a finite population

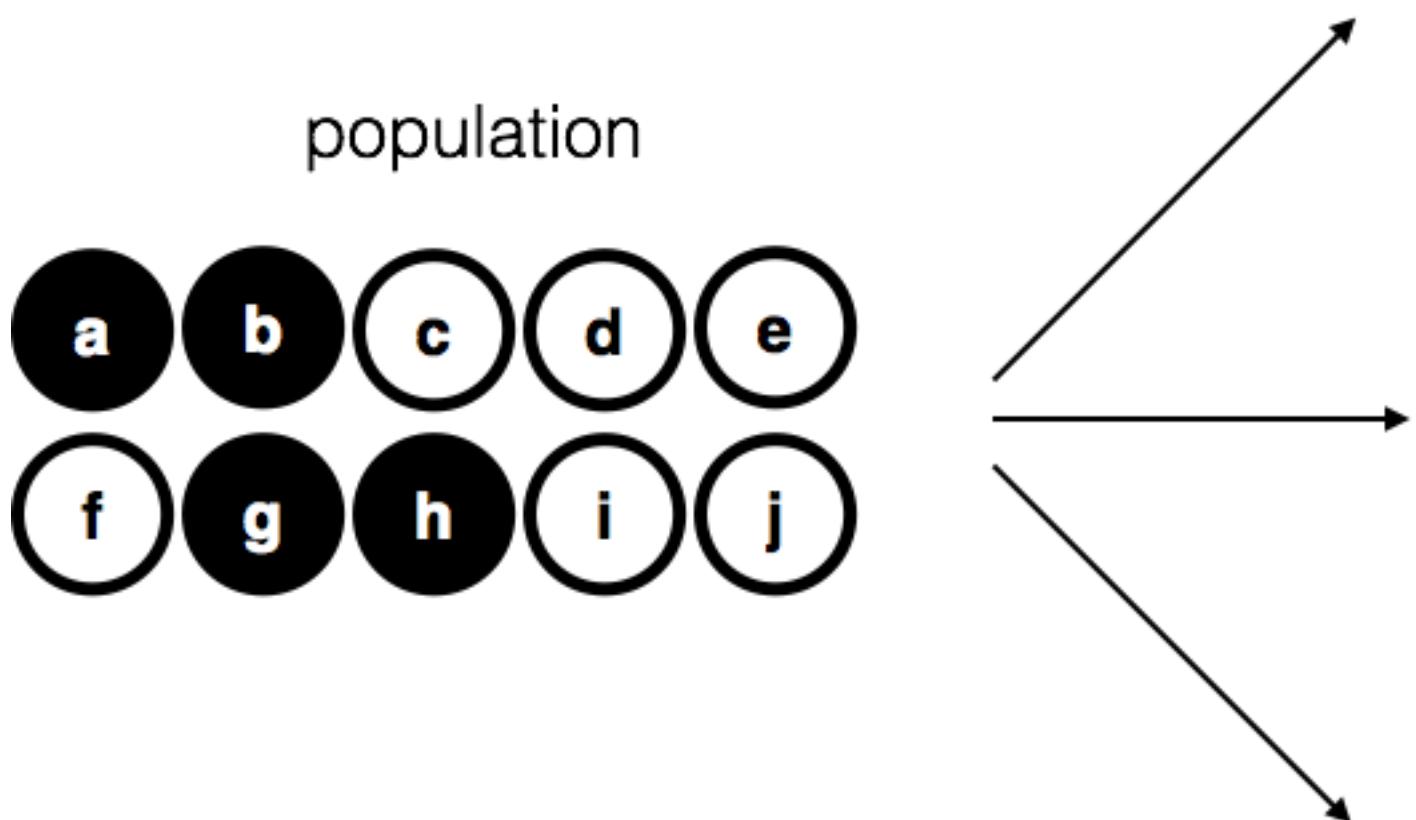


Figure 10.3: Simple random sampling *with* replacement from a finite population

instance, when studying schizophrenia it would be much better to divide the population into two² strata (schizophrenic and not-schizophrenic), and then sample an equal number of people from each group. If you selected people randomly, you would get so few schizophrenic people in the sample that your study would be useless. This specific kind of stratified sampling is referred to as *oversampling* because it makes a deliberate attempt to over-represent rare groups.

- *Snowball sampling* is a technique that is especially useful when sampling from a “hidden” or hard to access population, and is especially common in social sciences. For instance, suppose the researchers want to conduct an opinion poll among transgender people. The research team might only have contact details for a few trans folks, so the survey starts by asking them to participate (stage 1). At the end of the survey, the participants are asked to provide contact details for other people who might want to participate. In stage 2, those new contacts are surveyed. The process continues until the researchers have sufficient data. The big advantage to snowball sampling is that it gets you data in situations that might otherwise be impossible to get any. On the statistical side, the main disadvantage is that the sample is highly non-random, and non-random in ways that are difficult to address. On the real life side, the disadvantage is that the procedure can be unethical if not handled well, because hidden populations are often hidden for a reason. I chose transgender people as an example here to highlight this: if you weren’t careful you might end up outing people who don’t want to be outed (very, very bad form), and even if you don’t make that mistake it can still be intrusive to use people’s social networks to study them. It’s certainly very hard to get people’s informed consent *before* contacting them, yet in many cases the simple act of contacting them and saying “hey we want to study you” can be hurtful. Social networks are complex things, and just because you can use them to get data doesn’t always mean you should.
- *Convenience sampling* is more or less what it sounds like. The samples are chosen in a way that is convenient to the researcher, and not selected at random from the population of interest. Snowball sampling is one type of convenience sampling, but there are many others. A common example in psychology are studies that rely on undergraduate psychology students. These samples are generally non-random in two respects: firstly, reliance on undergraduate psychology students automatically means that your data are restricted to a single subpopulation. Secondly, the students usually get to pick which studies they participate in, so the sample is a self selected subset of psychology students not a randomly selected subset. In real life, most studies are convenience samples of one form or another. This is sometimes a severe limitation, but not always.

10.1.4 How much does it matter if you don’t have a simple random sample?

Okay, so real world data collection tends not to involve nice simple random samples. Does that matter? A little thought should make it clear to you that it *can* matter if your data are not a simple random sample: just think about the difference between Figures 10.1 and 10.2. However, it’s not quite as bad as it sounds. Some types of biased samples are entirely unproblematic. For instance, when using a stratified sampling technique you actually *know* what the bias is because you created it deliberately, often to *increase* the effectiveness of your study, and there are statistical techniques that you can use to adjust for the biases you’ve introduced (not covered in this book!). So in those situations it’s not a problem.

More generally though, it’s important to remember that random sampling is a means to an end, not the end in itself. Let’s assume you’ve relied on a convenience sample, and as such you can assume it’s biased. A bias in your sampling method is only a problem if it causes you to draw the wrong conclusions. When viewed from that perspective, I’d argue that we don’t need the sample to be randomly generated in *every* respect: we only need it to be random with respect to the psychologically-relevant phenomenon of interest. Suppose I’m doing a study looking at working memory capacity. In study 1, I actually have the ability to sample randomly from all human beings currently alive, with one exception: I can only sample people born on a Monday. In study 2, I am able to sample randomly from the Australian population. I want to generalise my results to the population of all living humans. Which study is better? The answer, obviously, is study 1.

²Nothing in life is that simple: there’s not an obvious division of people into binary categories like “schizophrenic” and “not schizophrenic”. But this isn’t a clinical psychology text, so please forgive me a few simplifications here and there.

Why? Because we have no reason to think that being “born on a Monday” has any interesting relationship to working memory capacity. In contrast, I can think of several reasons why “being Australian” might matter. Australia is a wealthy, industrialised country with a very well-developed education system. People growing up in that system will have had life experiences much more similar to the experiences of the people who designed the tests for working memory capacity. This shared experience might easily translate into similar beliefs about how to “take a test”, a shared assumption about how psychological experimentation works, and so on. These things might actually matter. For instance, “test taking” style might have taught the Australian participants how to direct their attention exclusively on fairly abstract test materials relative to people that haven’t grown up in a similar environment; leading to a misleading picture of what working memory capacity is.

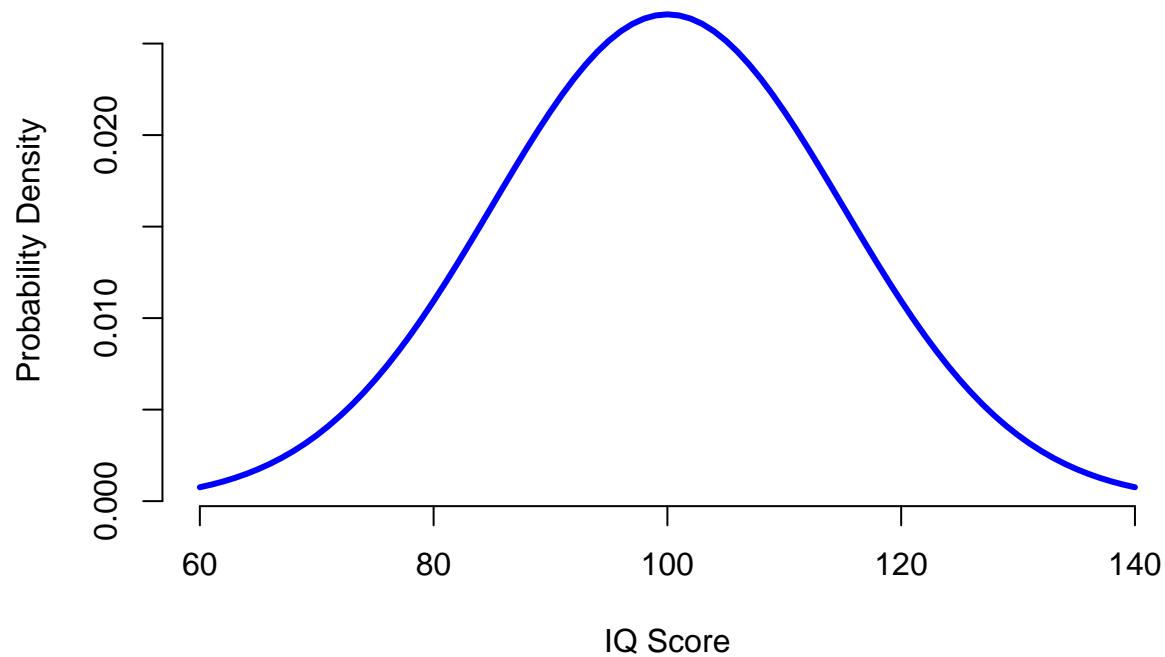
There are two points hidden in this discussion. Firstly, when designing your own studies, it’s important to think about what population you care about, and try hard to sample in a way that is appropriate to that population. In practice, you’re usually forced to put up with a “sample of convenience” (e.g., psychology lecturers sample psychology students because that’s the least expensive way to collect data, and our coffers aren’t exactly overflowing with gold), but if so you should at least spend some time thinking about what the dangers of this practice might be.

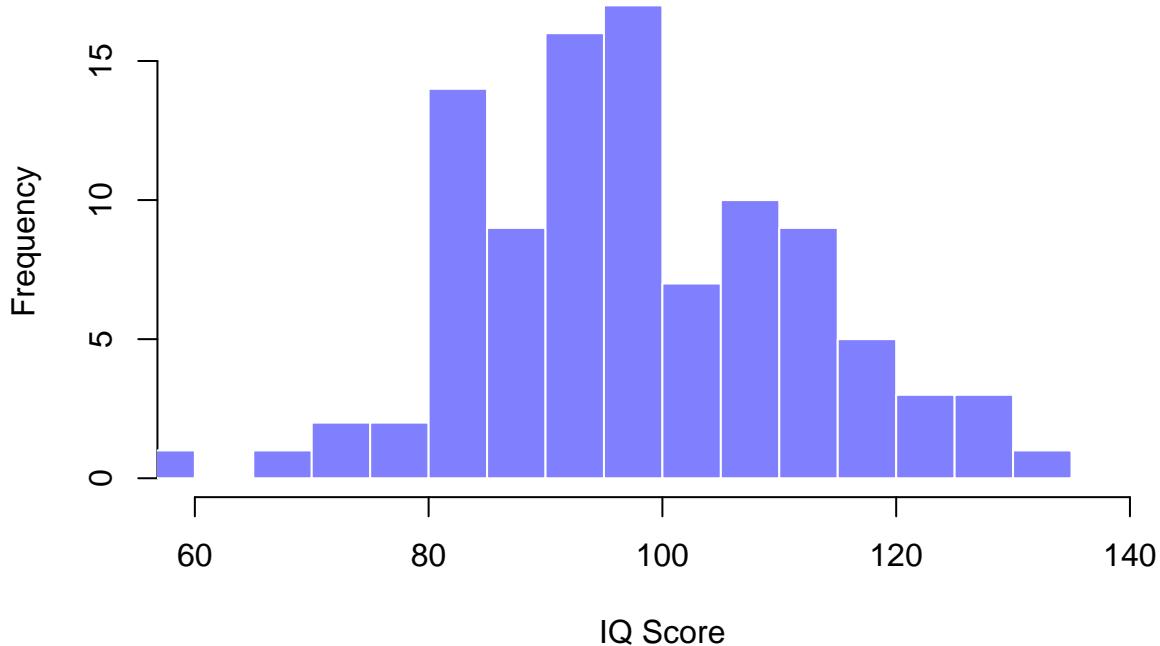
Secondly, if you’re going to criticise someone else’s study because they’ve used a sample of convenience rather than laboriously sampling randomly from the entire human population, at least have the courtesy to offer a specific theory as to *how* this might have distorted the results. Remember, everyone in science is aware of this issue, and does what they can to alleviate it. Merely pointing out that “the study only included people from group BLAH” is entirely unhelpful, and borders on being insulting to the researchers, who are *of course* aware of the issue. They just don’t happen to be in possession of the infinite supply of time and money required to construct the perfect sample. In short, if you want to offer a responsible critique of the sampling process, then be *helpful*. Rehashing the blindingly obvious truisms that I’ve been rambling on about in this section isn’t helpful.

10.1.5 Population parameters and sample statistics

Okay. Setting aside the thorny methodological issues associated with obtaining a random sample and my rather unfortunate tendency to rant about lazy methodological criticism, let’s consider a slightly different issue. Up to this point we have been talking about populations the way a scientist might. To a psychologist, a population might be a group of people. To an ecologist, a population might be a group of bears. In most cases the populations that scientists care about are concrete things that actually exist in the real world. Statisticians, however, are a funny lot. On the one hand, they *are* interested in real world data and real science in the same way that scientists are. On the other hand, they also operate in the realm of pure abstraction in the way that mathematicians do. As a consequence, statistical theory tends to be a bit abstract in how a population is defined. In much the same way that psychological researchers operationalise our abstract theoretical ideas in terms of concrete measurements (Section 2.1, statisticians operationalise the concept of a “population” in terms of mathematical objects that they know how to work with. You’ve already come across these objects in Chapter 9: they’re called probability distributions.

The idea is quite simple. Let’s say we’re talking about IQ scores. To a psychologist, the population of interest is a group of actual humans who have IQ scores. A statistician “simplifies” this by operationally defining the population as the probability distribution depicted in Figure ???. IQ tests are designed so that the average IQ is 100, the standard deviation of IQ scores is 15, and the distribution of IQ scores is normal. These values are referred to as the ***population parameters*** because they are characteristics of the entire population. That is, we say that the population mean μ is 100, and the population standard deviation σ is 15.





```
## [1] "n= 100 mean= 97.7573699832847 sd= 14.2057100089253"
```

```
## [1] "n= 10000 mean= 100.228103572035 sd= 15.1325692887095"
```

Now suppose I run an experiment. I select 100 people at random and administer an IQ test, giving me a simple random sample from the population. My sample would consist of a collection of numbers like this:

106 101 98 80 74 ... 107 72 100

Each of these IQ scores is sampled from a normal distribution with mean 100 and standard deviation 15. So if I plot a histogram of the sample, I get something like the one shown in Figure 10.4b. As you can see, the histogram is *roughly* the right shape, but it's a very crude approximation to the true population distribution shown in Figure 10.4a. When I calculate the mean of my sample, I get a number that is fairly close to the population mean 100 but not identical. In this case, it turns out that the people in my sample have a mean IQ of 98.5, and the standard deviation of their IQ scores is 15.9. These **sample statistics** are properties of my data set, and although they are fairly similar to the true population values, they are not the same. In general, sample statistics are the things you can calculate from your data set, and the population parameters are the things you want to learn about. Later on in this chapter I'll talk about how you can estimate population parameters using your sample statistics (Section 10.4) and how to work out how confident you are in your estimates (Section 10.5) but before we get to that there's a few more ideas in sampling theory that you need to know about.

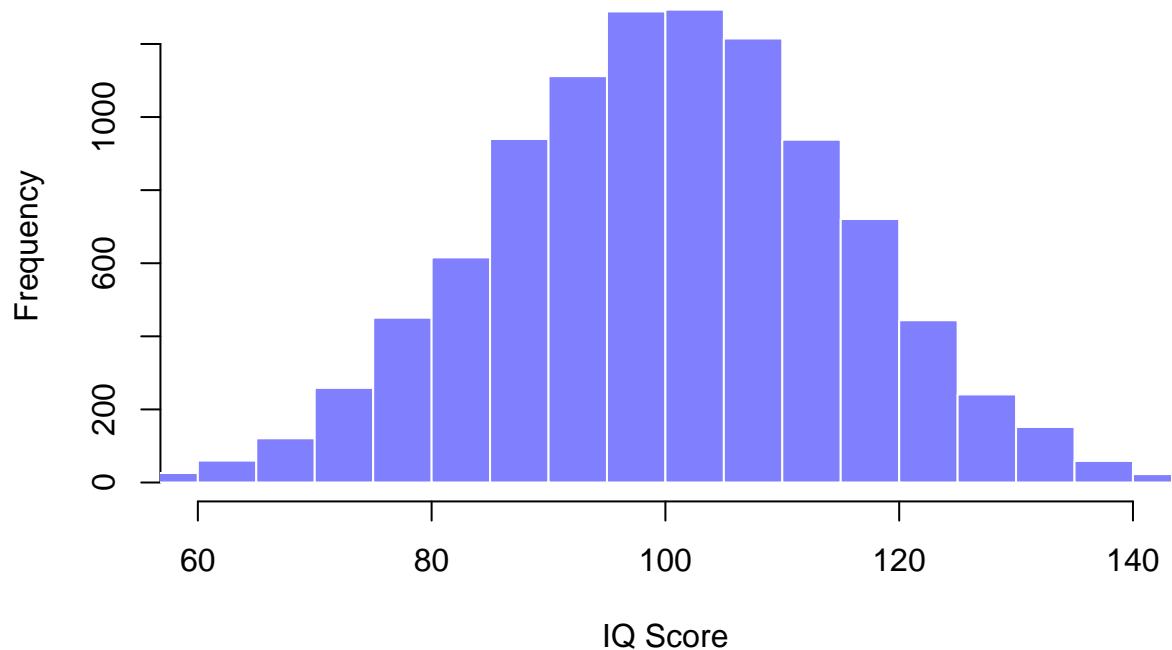


Figure 10.4: The population distribution of IQ scores (panel a) and two samples drawn randomly from it. In panel b we have a sample of 100 observations, and panel c we have a sample of 10,000 observations.

10.2 The law of large numbers

In the previous section I showed you the results of one fictitious IQ experiment with a sample size of $N = 100$. The results were somewhat encouraging: the true population mean is 100, and the sample mean of 98.5 is a pretty reasonable approximation to it. In many scientific studies that level of precision is perfectly acceptable, but in other situations you need to be a lot more precise. If we want our sample statistics to be much closer to the population parameters, what can we do about it?

The obvious answer is to collect more data. Suppose that we ran a much larger experiment, this time measuring the IQs of 10,000 people. We can simulate the results of this experiment using R. In Section 9.5 I introduced the `rnorm()` function, which generates random numbers sampled from a normal distribution. For an experiment with a sample size of `n = 10000`, and a population with `mean = 100` and `sd = 15`, R produces our fake IQ data using these commands:

```
IQ <- rnorm(n = 10000, mean = 100, sd = 15) # generate IQ scores
IQ <- round(IQ) # IQs are whole numbers!
print(head(IQ))
```

```
## [1] 107 114 117  76 103  95
```

I can compute the mean IQ using the command `mean(IQ)` and the standard deviation using the command `sd(IQ)`, and I can draw a histogram using `hist()`. The histogram of this much larger sample is shown in Figure 10.4c. Even a moment's inspection makes clear that the larger sample is a much better approximation to the true population distribution than the smaller one. This is reflected in the sample statistics: the mean IQ for the larger sample turns out to be 99.9, and the standard deviation is 15.1. These values are now very close to the true population.

I feel a bit silly saying this, but the thing I want you to take away from this is that large samples generally give you better information. I feel silly saying it because it's so bloody obvious that it shouldn't need to be said. In fact, it's such an obvious point that when Jacob Bernoulli – one of the founders of probability theory – formalised this idea back in 1713, he was kind of a jerk about it. Here's how he described the fact that we all share this intuition:

For even the most stupid of men, by some instinct of nature, by himself and without any instruction (which is a remarkable thing), is convinced that the more observations have been made, the less danger there is of wandering from one's goal Stigler (1986)

Okay, so the passage comes across as a bit condescending (not to mention sexist), but his main point is correct: it really does feel obvious that more data will give you better answers. The question is, why is this so? Not surprisingly, this intuition that we all share turns out to be correct, and statisticians refer to it as the **law of large numbers**. The law of large numbers is a mathematical law that applies to many different sample statistics, but the simplest way to think about it is as a law about averages. The sample mean is the most obvious example of a statistic that relies on averaging (because that's what the mean is... an average), so let's look at that. When applied to the sample mean, what the law of large numbers states is that as the sample gets larger, the sample mean tends to get closer to the true population mean. Or, to say it a little bit more precisely, as the sample size “approaches” infinity (written as $N \rightarrow \infty$) the sample mean approaches the population mean ($\bar{X} \rightarrow \mu$).³

I don't intend to subject you to a proof that the law of large numbers is true, but it's one of the most important tools for statistical theory. The law of large numbers is the thing we can use to justify our belief

³Technically, the law of large numbers pertains to any sample statistic that can be described as an average of independent quantities. That's certainly true for the sample mean. However, it's also possible to write many other sample statistics as averages of one form or another. The variance of a sample, for instance, can be rewritten as a kind of average and so is subject to the law of large numbers. The minimum value of a sample, however, cannot be written as an average of anything and is therefore not governed by the law of large numbers.

that collecting more and more data will eventually lead us to the truth. For any particular data set, the sample statistics that we calculate from it will be wrong, but the law of large numbers tells us that if we keep collecting more data those sample statistics will tend to get closer and closer to the true population parameters.

10.3 Sampling distributions and the central limit theorem

The law of large numbers is a very powerful tool, but it's not going to be good enough to answer all our questions. Among other things, all it gives us is a "long run guarantee". In the long run, if we were somehow able to collect an infinite amount of data, then the law of large numbers guarantees that our sample statistics will be correct. But as John Maynard Keynes famously argued in economics, a long run guarantee is of little use in real life:

[The] long run is a misleading guide to current affairs. In the long run we are all dead. Economists set themselves too easy, too useless a task, if in tempestuous seasons they can only tell us, that when the storm is long past, the ocean is flat again. Keynes (1923)

As in economics, so too in psychology and statistics. It is not enough to know that we will *eventually* arrive at the right answer when calculating the sample mean. Knowing that an infinitely large data set will tell me the exact value of the population mean is cold comfort when my *actual* data set has a sample size of $N = 100$. In real life, then, we must know something about the behaviour of the sample mean when it is calculated from a more modest data set!

10.3.1 Sampling distribution of the mean

With this in mind, let's abandon the idea that our studies will have sample sizes of 10000, and consider a very modest experiment indeed. This time around we'll sample $N = 5$ people and measure their IQ scores. As before, I can simulate this experiment in R using the `rnorm()` function:

```
> IQ.1 <- round( rnorm(n=5, mean=100, sd=15 ) )
> IQ.1
[1] 90 82 94 99 110
```

The mean IQ in this sample turns out to be exactly 95. Not surprisingly, this is much less accurate than the previous experiment. Now imagine that I decided to *replicate* the experiment. That is, I repeat the procedure as closely as possible: I randomly sample 5 new people and measure their IQ. Again, R allows me to simulate the results of this procedure:

```
> IQ.2 <- round( rnorm(n=5, mean=100, sd=15 ) )
> IQ.2
[1] 78 88 111 111 117
```

This time around, the mean IQ in my sample is 101. If I repeat the experiment 10 times I obtain the results shown in Table ??, and as you can see the sample mean varies from one replication to the next.

NANA	Person.1	Person.2	Person.3	Person.4	Person.5	Sample.Mean	caption
Replication 1	90	82	94	99	110	95.0	Ten replications of the IQ experim
Replication 2	78	88	111	111	117	101.0	Ten replications of the IQ experim
Replication 3	111	122	91	98	86	101.6	Ten replications of the IQ experim
Replication 4	98	96	119	99	107	103.8	Ten replications of the IQ experim
Replication 5	105	113	103	103	98	104.4	Ten replications of the IQ experim
Replication 6	81	89	93	85	114	92.4	Ten replications of the IQ experim
Replication 7	100	93	108	98	133	106.4	Ten replications of the IQ experim
Replication 8	107	100	105	117	85	102.8	Ten replications of the IQ experim
Replication 9	86	119	108	73	116	100.4	Ten replications of the IQ experim
Replication 10	95	126	112	120	76	105.8	Ten replications of the IQ experim

Now suppose that I decided to keep going in this fashion, replicating this “five IQ scores” experiment over and over again. Every time I replicate the experiment I write down the sample mean. Over time, I’d be amassing a new data set, in which every experiment generates a single data point. The first 10 observations from my data set are the sample means listed in Table ??, so my data set starts out like this:

95.0 101.0 101.6 103.8 104.4 ...

What if I continued like this for 10,000 replications, and then drew a histogram? Using the magical powers of R that’s exactly what I did, and you can see the results in Figure 10.5. As this picture illustrates, the average of 5 IQ scores is usually between 90 and 110. But more importantly, what it highlights is that if we replicate an experiment over and over again, what we end up with is a *distribution* of sample means! This distribution has a special name in statistics: it’s called the ***sampling distribution of the mean***.

Sampling distributions are another important theoretical idea in statistics, and they’re crucial for understanding the behaviour of small samples. For instance, when I ran the very first “five IQ scores” experiment, the sample mean turned out to be 95. What the sampling distribution in Figure 10.5 tells us, though, is that the “five IQ scores” experiment is not very accurate. If I repeat the experiment, the sampling distribution tells me that I can expect to see a sample mean anywhere between 80 and 120.

10.3.2 Sampling distributions exist for any sample statistic!

One thing to keep in mind when thinking about sampling distributions is that *any* sample statistic you might care to calculate has a sampling distribution. For example, suppose that each time I replicated the “five IQ scores” experiment I wrote down the largest IQ score in the experiment. This would give me a data set that started out like this:

110 117 122 119 113 ...

Doing this over and over again would give me a very different sampling distribution, namely the ***sampling distribution of the maximum***. The sampling distribution of the maximum of 5 IQ scores is shown in Figure 10.6. Not surprisingly, if you pick 5 people at random and then find the person with the highest IQ score, they’re going to have an above average IQ. Most of the time you’ll end up with someone whose IQ is measured in the 100 to 140 range.

10.3.3 The central limit theorem

An illustration of the how sampling distribution of the mean depends on sample size. In each panel, I generated 10,000 samples of IQ data, and calculated the mean IQ observed within each of these data sets. The histograms in these plots show the distribution of these means (i.e., the sampling distribution of the

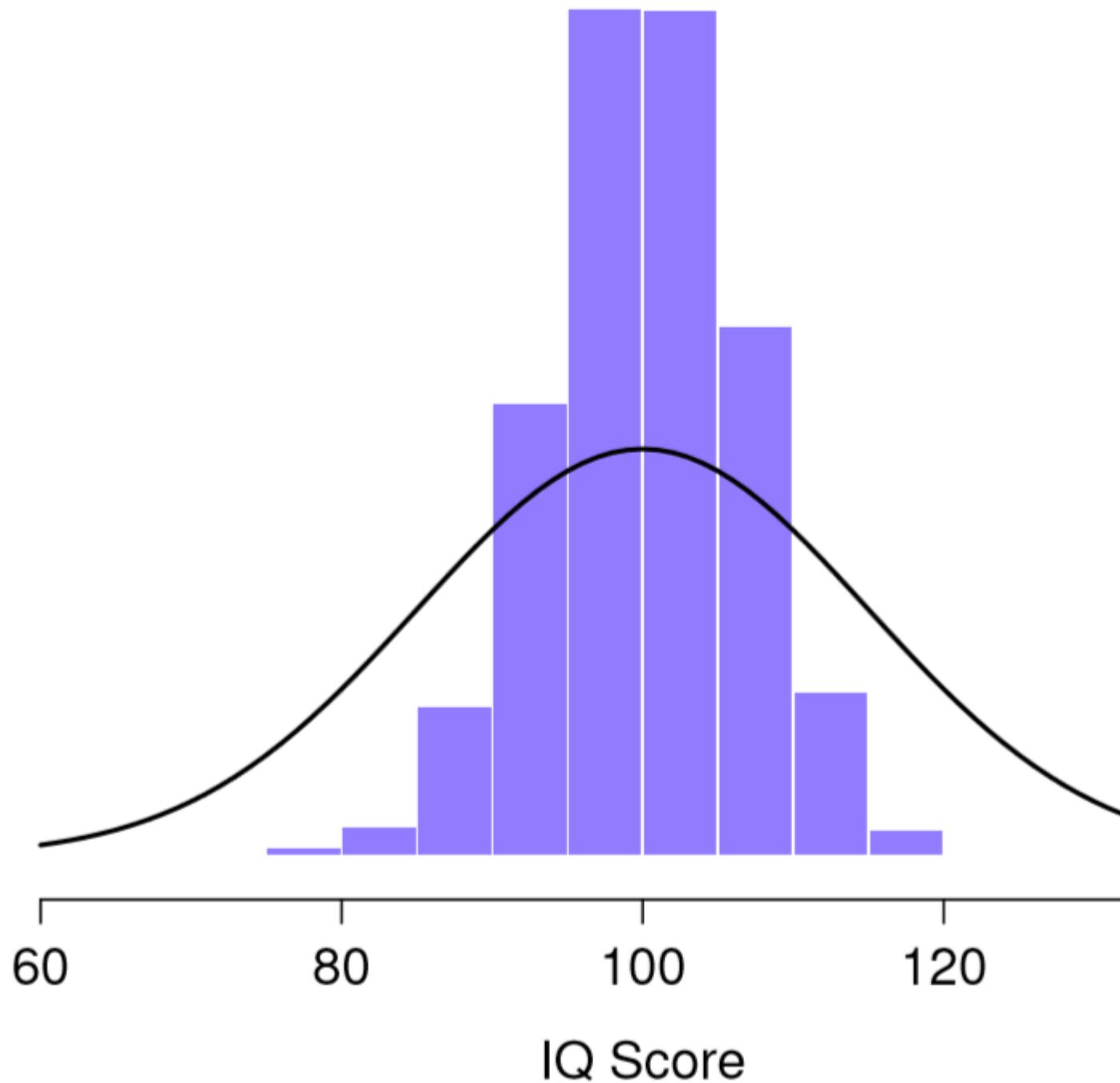


Figure 10.5: The sampling distribution of the mean for the "five IQ scores experiment". If you sample 5 people at random and calculate their *average* IQ, you'll almost certainly get a number between 80 and 120, even though there are quite a lot of individuals who have IQs above 120 or below 80. For comparison, the black line plots the population distribution of IQ scores.

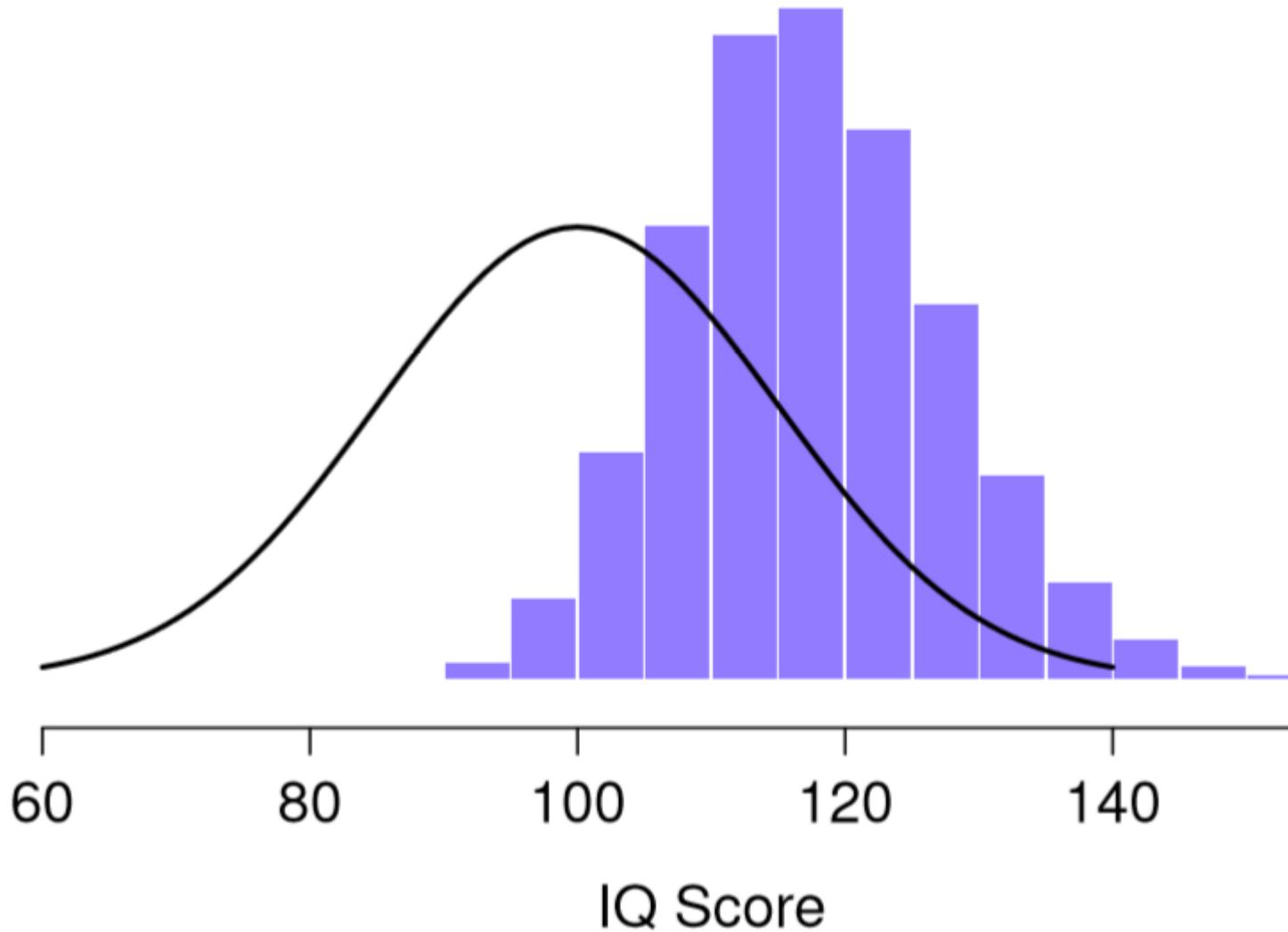


Figure 10.6: The sampling distribution of the *maximum* for the "five IQ scores experiment". If you sample 5 people at random and select the one with the highest IQ score, you'll probably see someone with an IQ between 100 and 140.

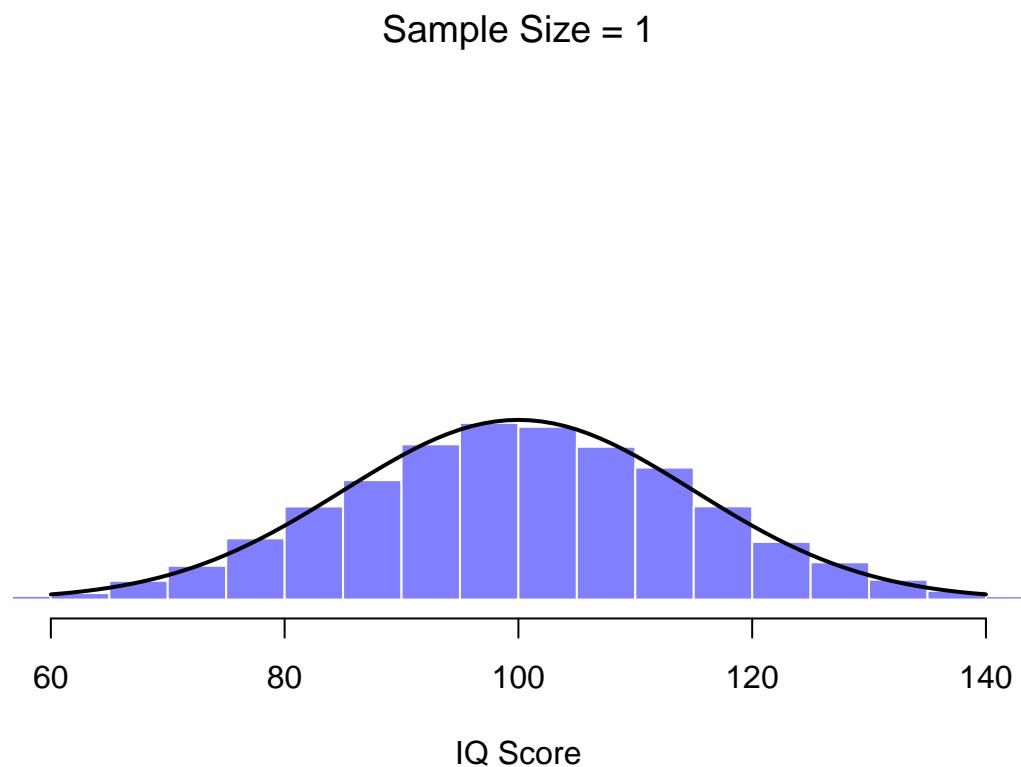


Figure 10.7: Each data set contained only a single observation, so the mean of each sample is just one person's IQ score. As a consequence, the sampling distribution of the mean is of course identical to the population distribution of IQ scores.

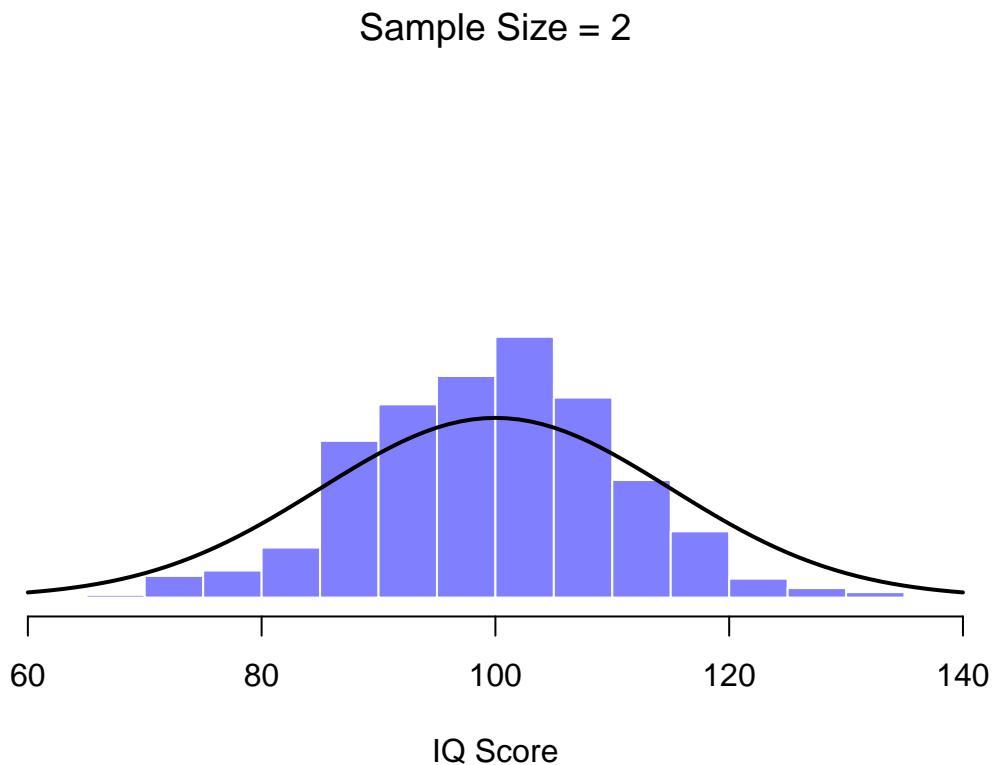


Figure 10.8: When we raise the sample size to 2, the mean of any one sample tends to be closer to the population mean than a one person's IQ score, and so the histogram (i.e., the sampling distribution) is a bit narrower than the population distribution.

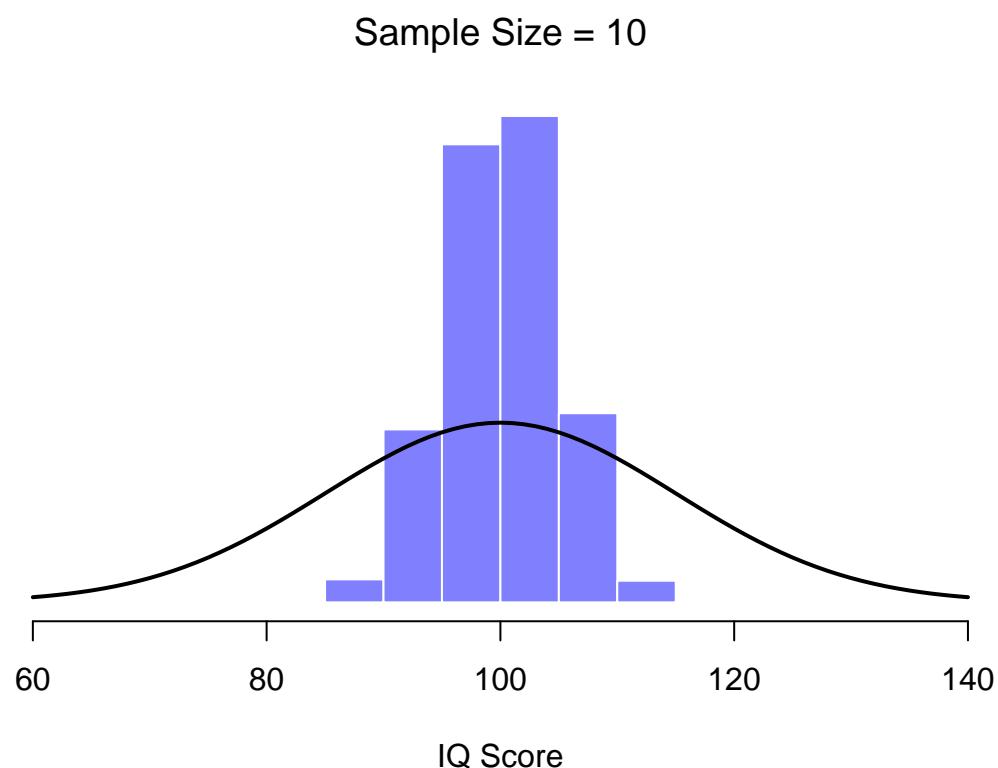


Figure 10.9: By the time we raise the sample size to 10, we can see that the distribution of sample means tend to be fairly tightly clustered around the true population mean.

mean). Each individual IQ score was drawn from a normal distribution with mean 100 and standard deviation 15, which is shown as the solid black line).

At this point I hope you have a pretty good sense of what sampling distributions are, and in particular what the sampling distribution of the mean is. In this section I want to talk about how the sampling distribution of the mean changes as a function of sample size. Intuitively, you already know part of the answer: if you only have a few observations, the sample mean is likely to be quite inaccurate: if you replicate a small experiment and recalculate the mean you'll get a very different answer. In other words, the sampling distribution is quite wide. If you replicate a large experiment and recalculate the sample mean you'll probably get the same answer you got last time, so the sampling distribution will be very narrow. You can see this visually in Figures 10.7, 10.8 and 10.9: the bigger the sample size, the narrower the sampling distribution gets. We can quantify this effect by calculating the standard deviation of the sampling distribution, which is referred to as the **standard error**. The standard error of a statistic is often denoted SE, and since we're usually interested in the standard error of the sample *mean*, we often use the acronym SEM. As you can see just by looking at the picture, as the sample size N increases, the SEM decreases.

Okay, so that's one part of the story. However, there's something I've been glossing over so far. All my examples up to this point have been based on the "IQ scores" experiments, and because IQ scores are roughly normally distributed, I've assumed that the population distribution is normal. What if it isn't normal? What happens to the sampling distribution of the mean? The remarkable thing is this: no matter what shape your population distribution is, as N increases the sampling distribution of the mean starts to look more like a normal distribution. To give you a sense of this, I ran some simulations using R. To do this, I started with the "ramped" distribution shown in the histogram in Figure ???. As you can see by comparing the triangular shaped histogram to the bell curve plotted by the black line, the population distribution doesn't look very much like a normal distribution at all. Next, I used R to simulate the results of a large number of experiments. In each experiment I took $N = 2$ samples from this distribution, and then calculated the sample mean. Figure ?? plots the histogram of these sample means (i.e., the sampling distribution of the mean for $N = 2$). This time, the histogram produces a \cap -shaped distribution: it's still not normal, but it's a lot closer to the black line than the population distribution in Figure ???. When I increase the sample size to $N = 4$, the sampling distribution of the mean is very close to normal (Figure ??, and by the time we reach a sample size of $N = 8$ it's almost perfectly normal. In other words, as long as your sample size isn't tiny, the sampling distribution of the mean will be approximately normal no matter what your population distribution looks like!

```
# needed for printing
width <- 6
height <- 6

# parameters of the beta
a <- 2
b <- 1

# mean and standard deviation of the beta
s <- sqrt( a*b / (a+b)^2 / (a+b+1) )
m <- a / (a+b)

# define function to draw a plot
plotOne <- function(n,N=50000) {

  # generate N random sample means of size n
  X <- matrix(rbeta(n*N,a,b),n,N)
  X <- colMeans(X)

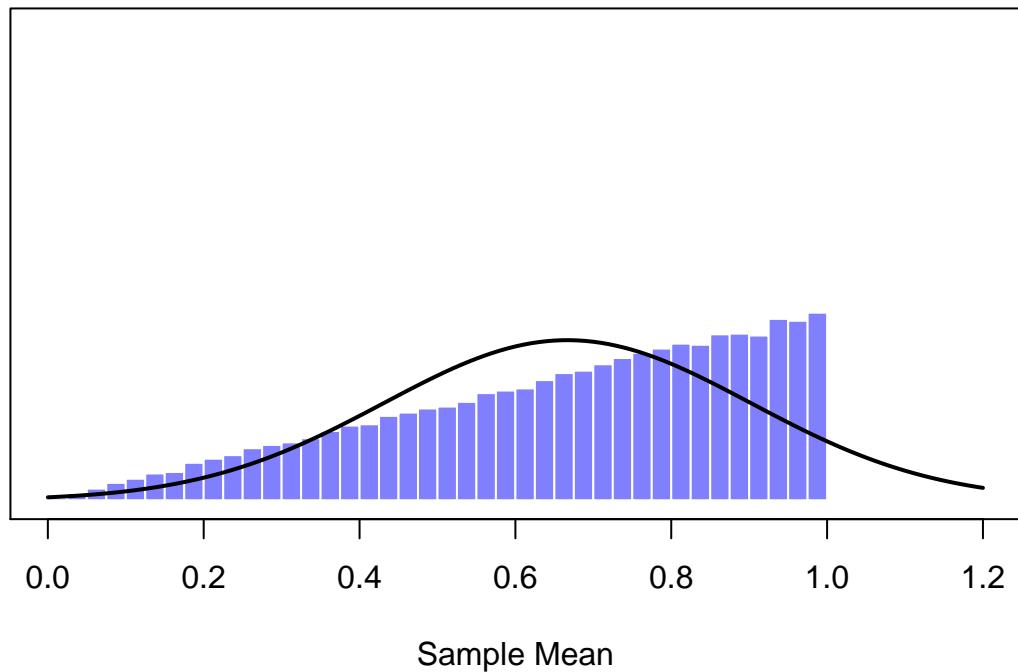
  # plot the data
  hist( X, breaks=seq(0,1,.025), border="white", freq=FALSE,
```

```
col=ifelse(colour,emphColLight,emphGrey),
xlab="Sample Mean", ylab="", xlim=c(0,1.2),
main=paste("Sample Size =",n), axes=FALSE,
font.main=1, ylim=c(0,5)
)
box()
axis(1)
#axis(2)

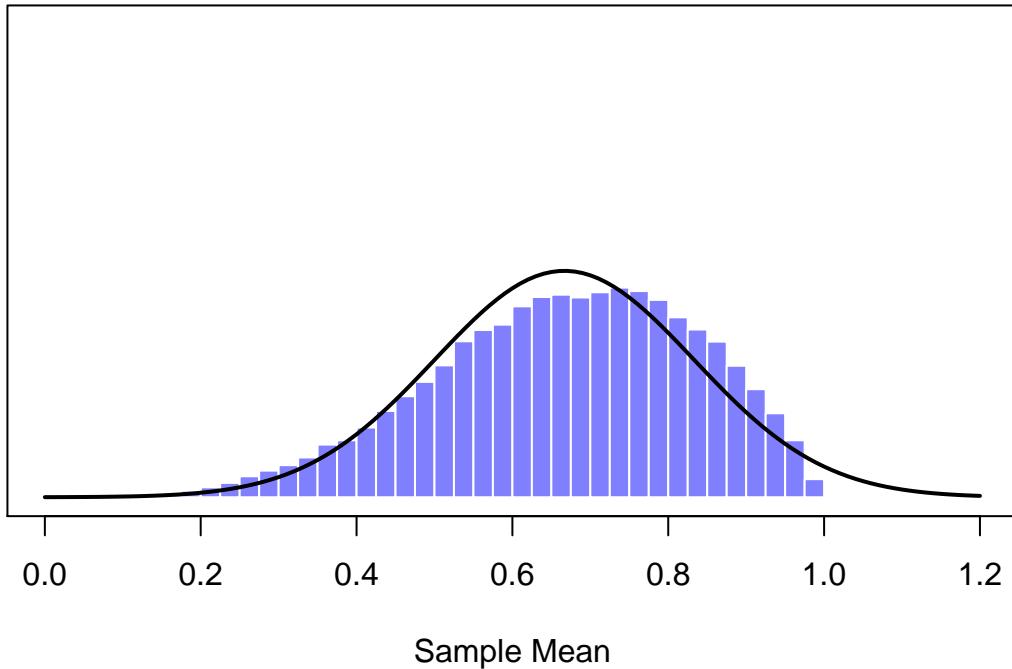
# plot the theoretical distribution
lines( x <- seq(0,1.2,.01), dnorm(x,m,s/sqrt(n)),
      lwd=2, col="black", type="l"
)
}

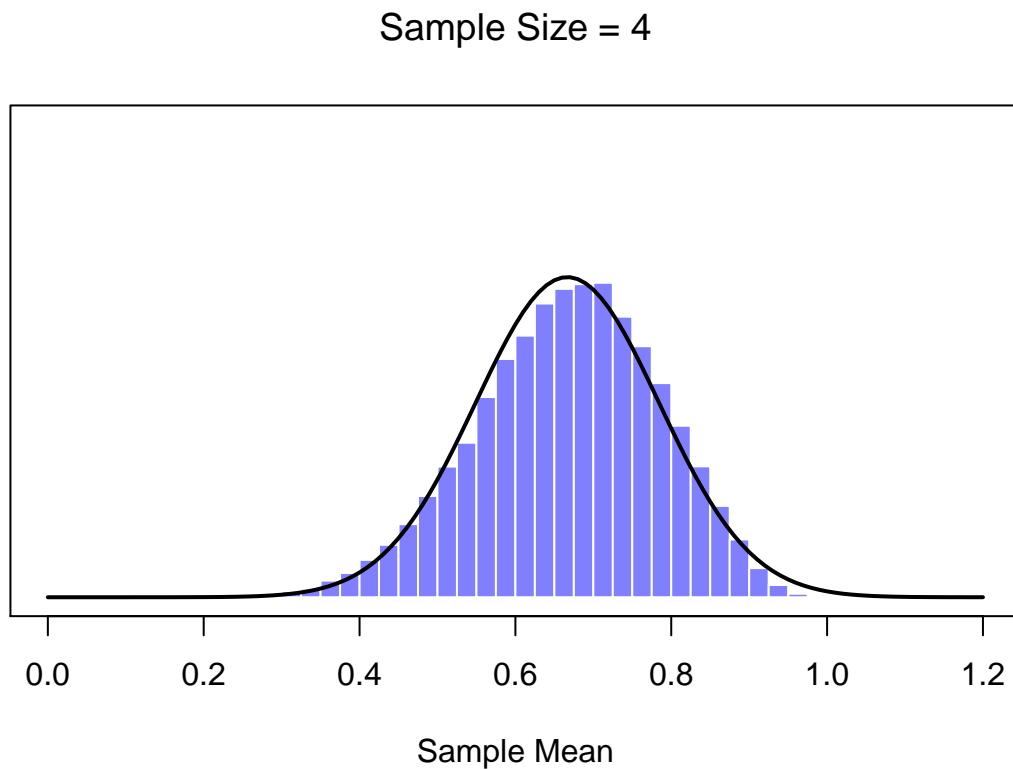
for( i in c(1,2,4,8)) {
  plotOne(i)}
```

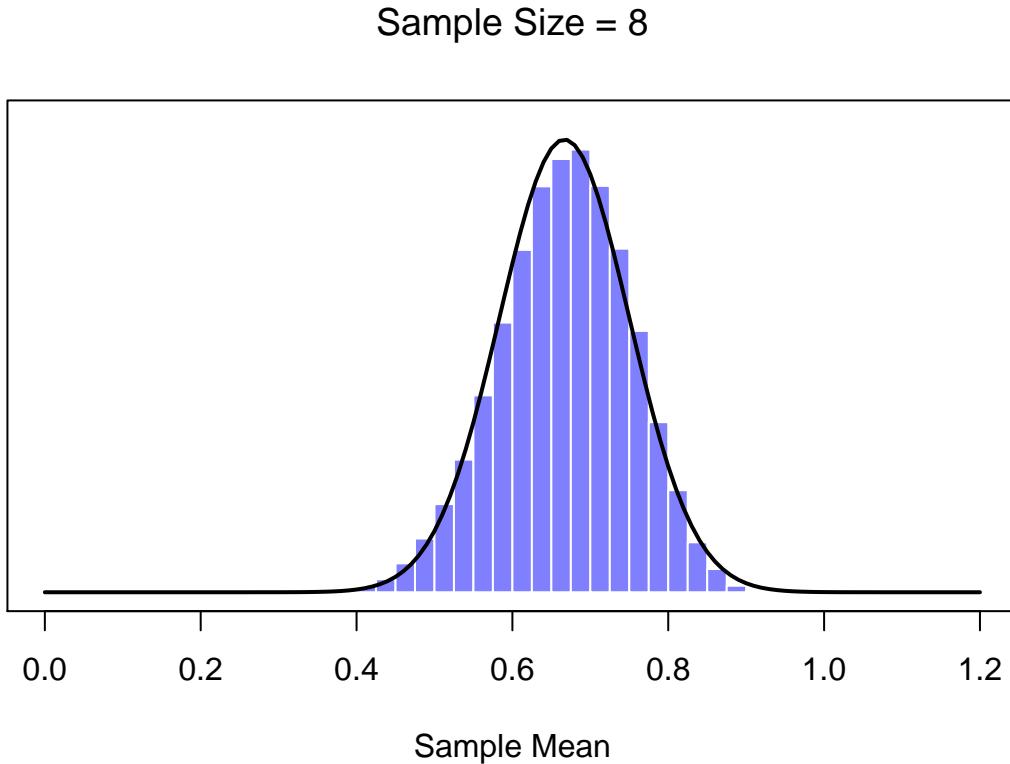
Sample Size = 1



Sample Size = 2







On the basis of these figures, it seems like we have evidence for all of the following claims about the sampling distribution of the mean:

- The mean of the sampling distribution is the same as the mean of the population
- The standard deviation of the sampling distribution (i.e., the standard error) gets smaller as the sample size increases
- The shape of the sampling distribution becomes normal as the sample size increases

As it happens, not only are all of these statements true, there is a very famous theorem in statistics that proves all three of them, known as the ***central limit theorem***. Among other things, the central limit theorem tells us that if the population distribution has mean μ and standard deviation σ , then the sampling distribution of the mean also has mean μ , and the standard error of the mean is

$$\text{SEM} = \frac{\sigma}{\sqrt{N}}$$

Because we divide the population standard deviation σ by the square root of the sample size N , the SEM gets smaller as the sample size increases. It also tells us that the shape of the sampling distribution becomes normal.⁴

This result is useful for all sorts of things. It tells us why large experiments are more reliable than small ones, and because it gives us an explicit formula for the standard error it tells us *how much* more reliable a

⁴As usual, I'm being a bit sloppy here. The central limit theorem is a bit more general than this section implies. Like most introductory stats texts, I've discussed one situation where the central limit theorem holds: when you're taking an average across lots of independent events drawn from the same distribution. However, the central limit theorem is much broader than this. There's a whole class of things called "*U*-statistics" for instance, all of which satisfy the central limit theorem and therefore become normally distributed for large sample sizes. The mean is one such statistic, but it's not the only one.

large experiment is. It tells us why the normal distribution is, well, *normal*. In real experiments, many of the things that we want to measure are actually averages of lots of different quantities (e.g., arguably, “general” intelligence as measured by IQ is an average of a large number of “specific” skills and abilities), and when that happens, the averaged quantity should follow a normal distribution. Because of this mathematical law, the normal distribution pops up over and over again in real data.

10.4 Estimating population parameters

In all the IQ examples in the previous sections, we actually knew the population parameters ahead of time. As every undergraduate gets taught in their very first lecture on the measurement of intelligence, IQ scores are *defined* to have mean 100 and standard deviation 15. However, this is a bit of a lie. How do we know that IQ scores have a true population mean of 100? Well, we know this because the people who designed the tests have administered them to very large samples, and have then “rigged” the scoring rules so that their sample has mean 100. That’s not a bad thing of course: it’s an important part of designing a psychological measurement. However, it’s important to keep in mind that this theoretical mean of 100 only attaches to the population that the test designers used to design the tests. Good test designers will actually go to some lengths to provide “test norms” that can apply to lots of different populations (e.g., different age groups, nationalities etc).

This is very handy, but of course almost every research project of interest involves looking at a different population of people to those used in the test norms. For instance, suppose you wanted to measure the effect of low level lead poisoning on cognitive functioning in Port Pirie, a South Australian industrial town with a lead smelter. Perhaps you decide that you want to compare IQ scores among people in Port Pirie to a comparable sample in Whyalla, a South Australian industrial town with a steel refinery.⁵ Regardless of which town you’re thinking about, it doesn’t make a lot of sense simply to *assume* that the true population mean IQ is 100. No-one has, to my knowledge, produced sensible norming data that can automatically be applied to South Australian industrial towns. We’re going to have to *estimate* the population parameters from a sample of data. So how do we do this?

10.4.1 Estimating the population mean

Suppose we go to Port Pirie and 100 of the locals are kind enough to sit through an IQ test. The average IQ score among these people turns out to be $\bar{X} = 98.5$. So what is the true mean IQ for the entire population of Port Pirie? Obviously, we don’t know the answer to that question. It could be 97.2, but it could also be 103.5. Our sampling isn’t exhaustive so we cannot give a definitive answer. Nevertheless if I was forced at gunpoint to give a “best guess” I’d have to say 98.5. That’s the essence of statistical estimation: giving a best guess.

In this example, estimating the unknown poulation parameter is straightforward. I calculate the sample mean, and I use that as my *estimate of the population mean*. It’s pretty simple, and in the next section I’ll explain the statistical justification for this intuitive answer. However, for the moment what I want to do is make sure you recognise that the sample statistic and the estimate of the population parameter are

⁵Please note that if you were *actually* interested in this question, you would need to be a *lot* more careful than I’m being here. You *can’t* just compare IQ scores in Whyalla to Port Pirie and assume that any differences are due to lead poisoning. Even if it were true that the only differences between the two towns corresponded to the different refineries (and it isn’t, not by a long shot), you need to account for the fact that people already *believe* that lead pollution causes cognitive deficits: if you recall back to Chapter 2, this means that there are different demand effects for the Port Pirie sample than for the Whyalla sample. In other words, you might end up with an illusory group difference in your data, caused by the fact that people *think* that there is a real difference. I find it pretty implausible to think that the locals wouldn’t be well aware of what you were trying to do if a bunch of researchers turned up in Port Pirie with lab coats and IQ tests, and even less plausible to think that a lot of people would be pretty resentful of you for doing it. Those people won’t be as co-operative in the tests. Other people in Port Pirie might be *more* motivated to do well because they don’t want their home town to look bad. The motivational effects that would apply in Whyalla are likely to be weaker, because people don’t have any concept of “iron ore poisoning” in the same way that they have a concept for “lead poisoning”. Psychology is *hard*.

conceptually different things. A sample statistic is a description of your data, whereas the estimate is a guess about the population. With that in mind, statisticians often use different notation to refer to them. For instance, if true population mean is denoted μ , then we would use $\hat{\mu}$ to refer to our estimate of the population mean. In contrast, the sample mean is denoted \bar{X} or sometimes m . However, in simple random samples, the estimate of the population mean is identical to the sample mean: if I observe a sample mean of $\bar{X} = 98.5$, then my estimate of the population mean is also $\hat{\mu} = 98.5$. To help keep the notation clear, here's a handy table:

```
knitr::kable(data.frame(stringsAsFactors=FALSE,
  Symbol = c("$\\bar{X}$", "$\\mu$", "$\\hat{\\mu}$"),
  What.is.it = c("Sample mean", "True population mean",
    "Estimate of the population mean"),
  Do.we.know.what.it.is = c("Yes calculated from the raw data",
    "Almost never known for sure",
    "Yes identical to the sample mean")))
```

Symbol	What.is.it	Do.we.know.what.it.is
\bar{X}	Sample mean	Yes calculated from the raw data
μ	True population mean	Almost never known for sure
$\hat{\mu}$	Estimate of the population mean	Yes identical to the sample mean

10.4.2 Estimating the population standard deviation

So far, estimation seems pretty simple, and you might be wondering why I forced you to read through all that stuff about sampling theory. In the case of the mean, our estimate of the population parameter (i.e. $\hat{\mu}$) turned out to be identical to the corresponding sample statistic (i.e. \bar{X}). However, that's not always true. To see this, let's have a think about how to construct an *estimate of the population standard deviation*, which we'll denote $\hat{\sigma}$. What shall we use as our estimate in this case? Your first thought might be that we could do the same thing we did when estimating the mean, and just use the sample statistic as our estimate. That's almost the right thing to do, but not quite.

Here's why. Suppose I have a sample that contains a single observation. For this example, it helps to consider a sample where you have no intuitions at all about what the true population values might be, so let's use something completely fictitious. Suppose the observation in question measures the *cromulence* of my shoes. It turns out that my shoes have a cromulence of 20. So here's my sample:

20

This is a perfectly legitimate sample, even if it does have a sample size of $N = 1$. It has a sample mean of 20, and because every observation in this sample is equal to the sample mean (obviously!) it has a sample standard deviation of 0. As a description of the *sample* this seems quite right: the sample contains a single observation and therefore there is no variation observed within the sample. A sample standard deviation of $s = 0$ is the right answer here. But as an estimate of the *population* standard deviation, it feels completely insane, right? Admittedly, you and I don't know anything at all about what "cromulence" is, but we know something about data: the only reason that we don't see any variability in the *sample* is that the sample is too small to display any variation! So, if you have a sample size of $N = 1$, it *feels* like the right answer is just to say "no idea at all".

Notice that you *don't* have the same intuition when it comes to the sample mean and the population mean. If forced to make a best guess about the population mean, it doesn't feel completely insane to guess that the population mean is 20. Sure, you probably wouldn't feel very confident in that guess, because you have only the one observation to work with, but it's still the best guess you can make.

Let's extend this example a little. Suppose I now make a second observation. My data set now has $N = 2$ observations of the cromulence of shoes, and the complete sample now looks like this:

20, 22

This time around, our sample is *just* large enough for us to be able to observe some variability: two observations is the bare minimum number needed for any variability to be observed! For our new data set, the sample mean is $\bar{X} = 21$, and the sample standard deviation is $s = 1$. What intuitions do we have about the population? Again, as far as the population mean goes, the best guess we can possibly make is the sample mean: if forced to guess, we'd probably guess that the population mean cromulence is 21. What about the standard deviation? This is a little more complicated. The sample standard deviation is only based on two observations, and if you're at all like me you probably have the intuition that, with only two observations, we haven't given the population "enough of a chance" to reveal its true variability to us. It's not just that we suspect that the estimate is *wrong*: after all, with only two observations we expect it to be wrong to some degree. The worry is that the error is *systematic*. Specifically, we suspect that the sample standard deviation is likely to be smaller than the population standard deviation.

This intuition feels right, but it would be nice to demonstrate this somehow. There are in fact mathematical proofs that confirm this intuition, but unless you have the right mathematical background they don't help very much. Instead, what I'll do is use R to simulate the results of some experiments. With that in mind, let's return to our IQ studies. Suppose the true population mean IQ is 100 and the standard deviation is 15. I can use the `rnorm()` function to generate the the results of an experiment in which I measure $N = 2$ IQ scores, and calculate the sample standard deviation. If I do this over and over again, and plot a histogram of these sample standard deviations, what I have is the *sampling distribution of the standard deviation*. I've plotted this distribution in Figure 10.10. Even though the true population standard deviation is 15, the average of the *sample* standard deviations is only 8.5. Notice that this is a very different result to what we found in Figure 10.8 when we plotted the sampling distribution of the mean. If you look at that sampling distribution, what you see is that the population mean is 100, and the average of the sample means is also 100.

Now let's extend the simulation. Instead of restricting ourselves to the situation where we have a sample size of $N = 2$, let's repeat the exercise for sample sizes from 1 to 10. If we plot the average sample mean and average sample standard deviation as a function of sample size, you get the results shown in Figure 10.11. On the left hand side (panel a), I've plotted the average sample mean and on the right hand side (panel b), I've plotted the average standard deviation. The two plots are quite different: *on average*, the average sample mean is equal to the population mean. It is an ***unbiased estimator***, which is essentially the reason why your best estimate for the population mean is the sample mean.⁶ The plot on the right is quite different: on average, the sample standard deviation s is *smaller* than the population standard deviation σ . It is a ***biased estimator***. In other words, if we want to make a "best guess" $\hat{\sigma}$ about the value of the population standard deviation σ , we should make sure our guess is a little bit larger than the sample standard deviation s .

The fix to this systematic bias turns out to be very simple. Here's how it works. Before tackling the standard deviation, let's look at the variance. If you recall from Section 5.2, the sample variance is defined to be the average of the squared deviations from the sample mean. That is:

$$s^2 = \frac{1}{N} \sum_{i=1}^N (X_i - \bar{X})^2$$

The sample variance s^2 is a biased estimator of the population variance σ^2 . But as it turns out, we only need to make a tiny tweak to transform this into an unbiased estimator. All we have to do is divide by $N - 1$ rather than by N . If we do that, we obtain the following formula:

$$\hat{\sigma}^2 = \frac{1}{N - 1} \sum_{i=1}^N (X_i - \bar{X})^2$$

⁶I should note that I'm hiding something here. Unbiasedness is a desirable characteristic for an estimator, but there are other things that matter besides bias. However, it's beyond the scope of this book to discuss this in any detail. I just want to draw your attention to the fact that there's some hidden complexity here.

Population Standard Deviation

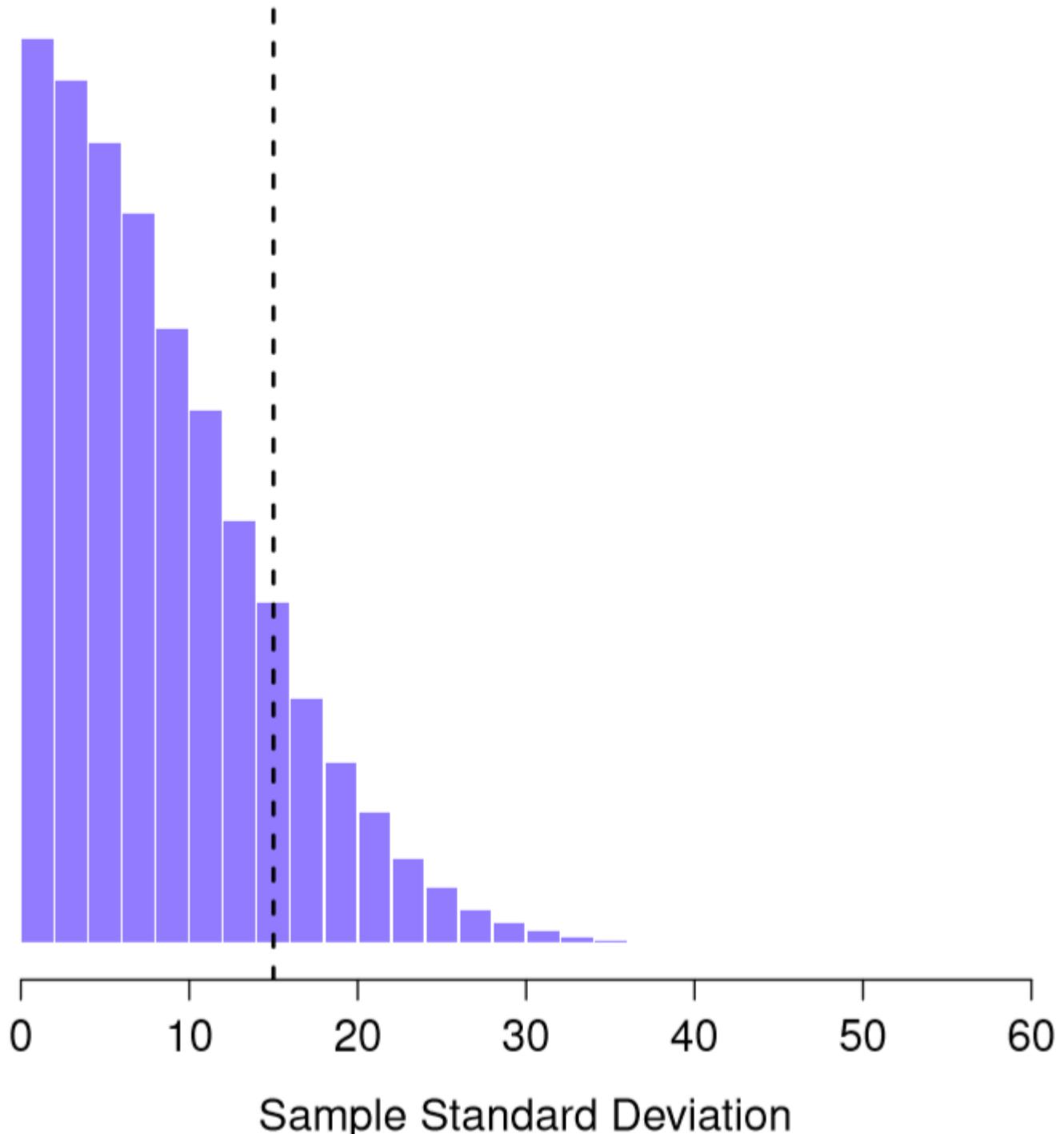


Figure 10.10: The sampling distribution of the sample standard deviation for a "two IQ scores" experiment. The true population standard deviation is 15 (dashed line), but as you can see from the histogram, the vast majority of experiments will produce a much smaller sample standard deviation than this. On average, this experiment would produce a sample standard deviation of only 8.5, well below the true value! In other words, the sample standard deviation is a **biased** estimate of the population standard deviation.

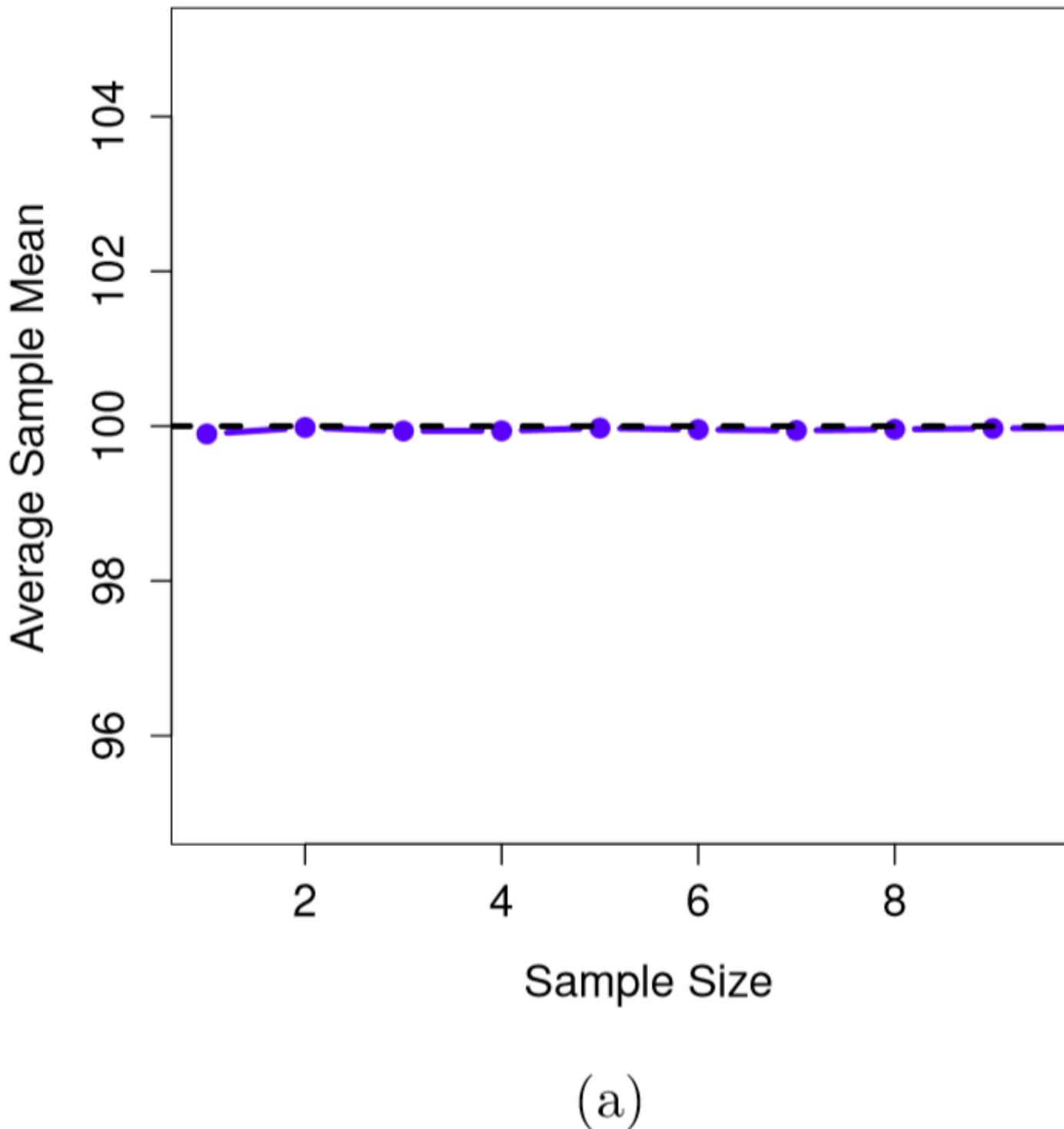


Figure 10.11: An illustration of the fact that the sample mean is an unbiased estimator of the population mean (panel a), but the sample standard deviation is a biased estimator of the population standard deviation (panel b). To generate the figure, I generated 10,000 simulated data sets with 1 observation each, 10,000 more with 2 observations, and so on up to a sample size of 10. Each data set consisted of fake IQ data: that is, the data were normally distributed with a true population mean of 100 and standard deviation 15. *On average*, the sample means turn out to be 100, regardless of sample size (panel a). However, the sample standard deviations turn out to be systematically too small (panel b), especially for small sample sizes.

This is an unbiased estimator of the population variance σ^2 . Moreover, this finally answers the question we raised in Section 5.2. Why did R give us slightly different answers when we used the `var()` function? Because the `var()` function calculates $\hat{\sigma}^2$ not s^2 , that's why. A similar story applies for the standard deviation. If we divide by $N - 1$ rather than N , our estimate of the population standard deviation becomes:

$$\hat{\sigma} = \sqrt{\frac{1}{N-1} \sum_{i=1}^N (X_i - \bar{X})^2}$$

and when we use R's built in standard deviation function `sd()`, what it's doing is calculating $\hat{\sigma}$, not s .⁷

One final point: in practice, a lot of people tend to refer to $\hat{\sigma}$ (i.e., the formula where we divide by $N - 1$) as the *sample* standard deviation. Technically, this is incorrect: the *sample* standard deviation should be equal to s (i.e., the formula where we divide by N). These aren't the same thing, either conceptually or numerically. One is a property of the sample, the other is an estimated characteristic of the population. However, in almost every real life application, what we actually care about is the estimate of the population parameter, and so people always report $\hat{\sigma}$ rather than s . This is the right number to report, of course, it's that people tend to get a little bit imprecise about terminology when they write it up, because "sample standard deviation" is shorter than "estimated population standard deviation". It's no big deal, and in practice I do the same thing everyone else does. Nevertheless, I think it's important to keep the two *concepts* separate: it's never a good idea to confuse "known properties of your sample" with "guesses about the population from which it came". The moment you start thinking that s and $\hat{\sigma}$ are the same thing, you start doing exactly that.

To finish this section off, here's another couple of tables to help keep things clear:

```
knitr::kable(data.frame(stringsAsFactors=FALSE,
  Symbol = c("$s$",
             "$\\sigma$",
             "$\\hat{\\sigma}$",
             "$s^2$",
             "$\\sigma^2$",
             "$\\hat{\\sigma}^2$"),
  What.is.it = c("Sample standard deviation",
                "Population standard deviation",
                "Estimate of the population standard deviation",
                "Sample variance",
                "Population variance",
                "Estimate of the population variance"),
  Do.we.know.what.it.is = c("Yes - calculated from the raw data",
                           "Almost never known for sure",
                           "Yes - but not the same as the sample standard deviation",
                           "Yes - calculated from the raw data",
                           "Almost never known for sure",
                           "Yes - but not the same as the sample variance"))
))
```

Symbol	What.is.it	Do.we.know.what.it.is
\$s\$	Sample standard deviation	Yes - calculated from the raw data
\$\sigma\$	Population standard deviation	Almost never known for sure
\$\hat{\sigma}\$	Estimate of the population standard deviation	Yes - but not the same as the sample standard deviation
\$s^2\$	Sample variance	Yes - calculated from the raw data
\$\sigma^2\$	Population variance	Almost never known for sure
\$\hat{\sigma}^2\$	Estimate of the population variance	Yes - but not the same as the sample variance

⁷Okay, I'm hiding something else here. In a bizarre and counterintuitive twist, since $\hat{\sigma}^2$ is an unbiased estimator of σ^2 , you'd assume that taking the square root would be fine, and $\hat{\sigma}$ would be an unbiased estimator of σ . Right? Weirdly, it's not. There's actually a subtle, tiny bias in $\hat{\sigma}$. This is just bizarre: $\hat{\sigma}^2$ is an unbiased estimator of the population variance σ^2 , but when you take the square root, it turns out that $\hat{\sigma}$ is a biased estimator of the population standard deviation σ . Weird, weird, weird, right? So, why is $\hat{\sigma}$ biased? The technical answer is "because non-linear transformations (e.g., the square root) don't commute with expectation", but that just sounds like gibberish to everyone who hasn't taken a course in mathematical statistics. Fortunately, it doesn't matter for practical purposes. The bias is small, and in real life everyone uses $\hat{\sigma}$ and it works just fine. Sometimes mathematics is just annoying.

10.5 Estimating a confidence interval

Statistics means never having to say you're certain – Unknown origin⁸ but I've never found the original source.

Up to this point in this chapter, I've outlined the basics of sampling theory which statisticians rely on to make guesses about population parameters on the basis of a sample of data. As this discussion illustrates, one of the reasons we need all this sampling theory is that every data set leaves us with a some of uncertainty, so our estimates are never going to be perfectly accurate. The thing that has been missing from this discussion is an attempt to *quantify* the amount of uncertainty that attaches to our estimate. It's not enough to be able guess that, say, the mean IQ of undergraduate psychology students is 115 (yes, I just made that number up). We also want to be able to say something that expresses the degree of certainty that we have in our guess. For example, it would be nice to be able to say that there is a 95% chance that the true mean lies between 109 and 121. The name for this is a **confidence interval** for the mean.

Armed with an understanding of sampling distributions, constructing a confidence interval for the mean is actually pretty easy. Here's how it works. Suppose the true population mean is μ and the standard deviation is σ . I've just finished running my study that has N participants, and the mean IQ among those participants is \bar{X} . We know from our discussion of the central limit theorem (Section 10.3.3) that the sampling distribution of the mean is approximately normal. We also know from our discussion of the normal distribution Section 9.5 that there is a 95% chance that a normally-distributed quantity will fall within two standard deviations of the true mean. To be more precise, we can use the `qnorm()` function to compute the 2.5th and 97.5th percentiles of the normal distribution

```
qnorm( p = c(.025, .975) )
```

```
## [1] -1.959964 1.959964
```

Okay, so I lied earlier on. The more correct answer is that 95% chance that a normally-distributed quantity will fall within 1.96 standard deviations of the true mean. Next, recall that the standard deviation of the sampling distribution is referred to as the standard error, and the standard error of the mean is written as SEM. When we put all these pieces together, we learn that there is a 95% probability that the sample mean \bar{X} that we have actually observed lies within 1.96 standard errors of the population mean. Mathematically, we write this as:

$$\mu - (1.96 \times \text{SEM}) \leq \bar{X} \leq \mu + (1.96 \times \text{SEM})$$

where the SEM is equal to σ/\sqrt{N} , and we can be 95% confident that this is true. However, that's not answering the question that we're actually interested in. The equation above tells us what we should expect about the sample mean, given that we know what the population parameters are. What we *want* is to have this work the other way around: we want to know what we should believe about the population parameters, given that we have observed a particular sample. However, it's not too difficult to do this. Using a little high school algebra, a sneaky way to rewrite our equation is like this:

$$\bar{X} - (1.96 \times \text{SEM}) \leq \mu \leq \bar{X} + (1.96 \times \text{SEM})$$

What this is telling is is that the range of values has a 95% probability of containing the population mean μ . We refer to this range as a **95% confidence interval**, denoted CI_{95} . In short, as long as N is sufficiently large – large enough for us to believe that the sampling distribution of the mean is normal – then we can write this as our formula for the 95% confidence interval:

$$\text{CI}_{95} = \bar{X} \pm \left(1.96 \times \frac{\sigma}{\sqrt{N}} \right)$$

⁸This quote appears on a great many t-shirts and websites, and even gets a mention in a few academic papers (e.g., \url{http://www.amstat.org/publications/jse/v10n3/friedman.html})

Of course, there's nothing special about the number 1.96: it just happens to be the multiplier you need to use if you want a 95% confidence interval. If I'd wanted a 70% confidence interval, I could have used the `qnorm()` function to calculate the 15th and 85th quantiles:

```
qnorm( p = c(.15, .85) )
## [1] -1.036433 1.036433
```

and so the formula for CI_{70} would be the same as the formula for CI_{95} except that we'd use 1.04 as our magic number rather than 1.96.

10.5.1 A slight mistake in the formula

As usual, I lied. The formula that I've given above for the 95% confidence interval is approximately correct, but I glossed over an important detail in the discussion. Notice my formula requires you to use the standard error of the mean, SEM, which in turn requires you to use the true population standard deviation σ . Yet, in Section @ref(pointestimates I stressed the fact that we don't actually *know* the true population parameters. Because we don't know the true value of σ , we have to use an estimate of the population standard deviation $\hat{\sigma}$ instead. This is pretty straightforward to do, but this has the consequence that we need to use the quantiles of the t -distribution rather than the normal distribution to calculate our magic number; and the answer depends on the sample size. When N is very large, we get pretty much the same value using `qt()` that we would if we used `qnorm()`...

```
N <- 10000  # suppose our sample size is 10,000
qt( p = .975, df = N-1)  # calculate the 97.5th quantile of the t-dist
## [1] 1.960201
```

But when N is small, we get a much bigger number when we use the t distribution:

```
N <- 10  # suppose our sample size is 10
qt( p = .975, df = N-1)  # calculate the 97.5th quantile of the t-dist
## [1] 2.262157
```

There's nothing too mysterious about what's happening here. Bigger values mean that the confidence interval is wider, indicating that we're more uncertain about what the true value of μ actually is. When we use the t distribution instead of the normal distribution, we get bigger numbers, indicating that we have more uncertainty. And why do we have that extra uncertainty? Well, because our estimate of the population standard deviation $\hat{\sigma}$ might be wrong! If it's wrong, it implies that we're a bit less sure about what our sampling distribution of the mean actually looks like... and this uncertainty ends up getting reflected in a wider confidence interval.

10.5.2 Interpreting a confidence interval

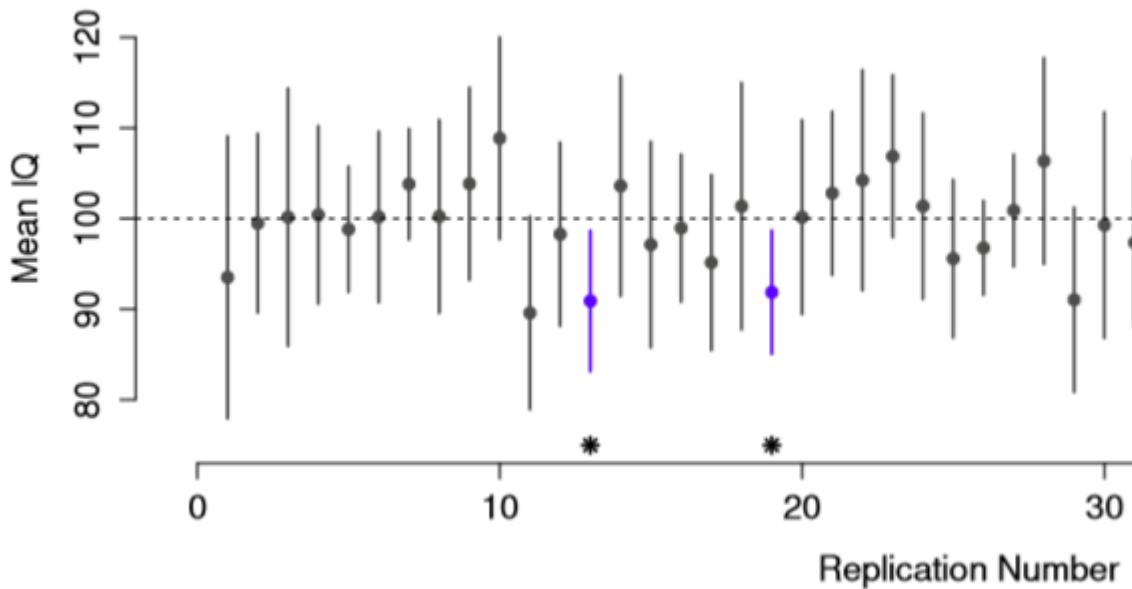
The hardest thing about confidence intervals is understanding what they *mean*. Whenever people first encounter confidence intervals, the first instinct is almost always to say that “there is a 95% probability that the true mean lies inside the confidence interval”. It's simple, and it seems to capture the common sense idea of what it means to say that I am “95% confident”. Unfortunately, it's not quite right. The intuitive definition relies very heavily on your own personal *beliefs* about the value of the population mean. I say

that I am 95% confident because those are my beliefs. In everyday life that's perfectly okay, but if you remember back to Section 9.2, you'll notice that talking about personal belief and confidence is a Bayesian idea. Personally (speaking as a Bayesian) I have no problem with the idea that the phrase "95% probability" is allowed to refer to a personal belief. However, confidence intervals are *not* Bayesian tools. Like everything else in this chapter, confidence intervals are *frequentist* tools, and if you are going to use frequentist methods then it's not appropriate to attach a Bayesian interpretation to them. If you use frequentist methods, you must adopt frequentist interpretations!

Okay, so if that's not the right answer, what is? Remember what we said about frequentist probability: the only way we are allowed to make "probability statements" is to talk about a sequence of events, and to count up the frequencies of different kinds of events. From that perspective, the interpretation of a 95% confidence interval must have something to do with replication. Specifically: if we replicated the experiment over and over again and computed a 95% confidence interval for each replication, then 95% of those *intervals* would contain the true mean. More generally, 95% of all confidence intervals constructed using this procedure should contain the true population mean. This idea is illustrated in Figure 10.5.2, which shows 50 confidence intervals constructed for a "measure 10 IQ scores" experiment (top panel) and another 50 confidence intervals for a "measure 25 IQ scores" experiment (bottom panel). A bit fortuitously, across the 100 replications that I simulated, it turned out that exactly 95 of them contained the true mean.

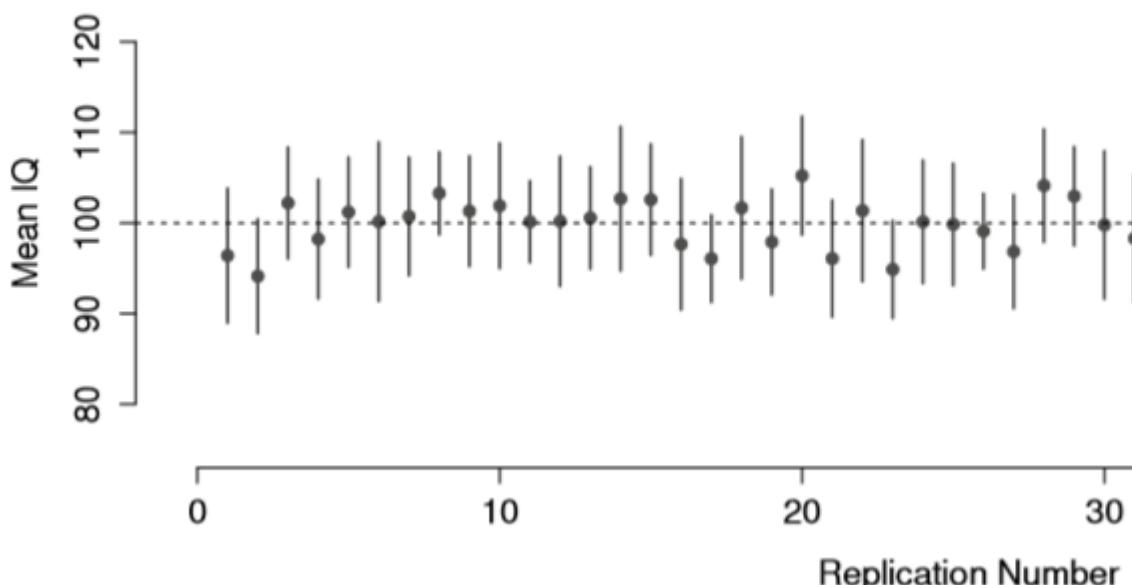
Sample Size = 10

(a)



Sample Size = 25

(b)



\begin{figure}

\caption{95\% confidence intervals. The top (panel a) shows 50 simulated replications of an experiment in which we measure the IQs of 10 people. The dot marks the location of the sample mean, and the line shows the 95\% confidence interval. In total 47 of the 50 confidence intervals do contain the true mean (i.e., 100), but the three intervals marked with asterisks do not. The lower graph (panel b) shows a similar simulation,

but this time we simulate replications of an experiment that measures the IQs of 25 people.} \end{figure}

The critical difference here is that the Bayesian claim makes a probability statement about the population mean (i.e., it refers to our uncertainty about the population mean), which is not allowed under the frequentist interpretation of probability because you can't "replicate" a population! In the frequentist claim, the population mean is fixed and no probabilistic claims can be made about it. Confidence intervals, however, are repeatable so we can replicate experiments. Therefore a frequentist is allowed to talk about the probability that the *confidence interval* (a random variable) contains the true mean; but is not allowed to talk about the probability that the *true population mean* (not a repeatable event) falls within the confidence interval.

I know that this seems a little pedantic, but it does matter. It matters because the difference in interpretation leads to a difference in the mathematics. There is a Bayesian alternative to confidence intervals, known as *credible intervals*. In most situations credible intervals are quite similar to confidence intervals, but in other cases they are drastically different. As promised, though, I'll talk more about the Bayesian perspective in Chapter ??.

10.5.3 Calculating confidence intervals in R

As far as I can tell, the core packages in R don't include a simple function for calculating confidence intervals for the mean. They *do* include a lot of complicated, extremely powerful functions that can be used to calculate confidence intervals associated with lots of different things, such as the `confint()` function that we'll use in Chapter ???. But I figure that when you're first learning statistics, it might be useful to start with something simpler. As a consequence, the `lsr` package includes a function called `ciMean()` which you can use to calculate your confidence intervals. There are two arguments that you might want to specify:⁹

- `x`. This should be a numeric vector containing the data.
- `conf`. This should be a number, specifying the confidence level. By default, `conf = .95`, since 95% confidence intervals are the de facto standard in psychology.

So, for example, if I load the `afl24.Rdata` file, calculate the confidence interval associated with the mean attendance:

```
> ciMean( x = afl$attendance )
  2.5%    97.5%
31597.32 32593.12
```

Hopefully that's fairly clear.

10.5.4 Plotting confidence intervals in R

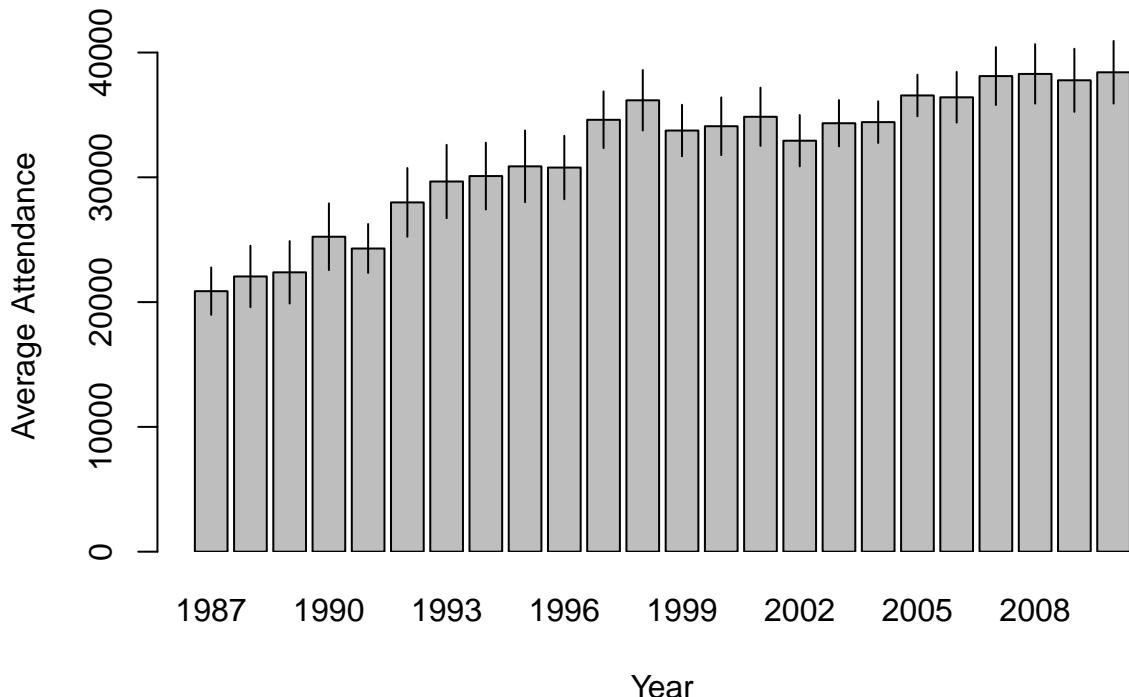
There's several different ways you can draw graphs that show confidence intervals as error bars. I'll show three versions here, but this certainly doesn't exhaust the possibilities. In doing so, what I'm assuming is that you want to draw is a plot showing the means and confidence intervals for one variable, broken down by different levels of a second variable. For instance, in our `afl` data that we discussed earlier, we might be interested in plotting the average `attendance` by `year`. I'll do this using two different functions, `bargraph.CI()` and `lineplot.CI()` (both of which are in the `sciplot` package). Assuming that you've installed these packages on your system (see Section 4.2 if you've forgotten how to do this), you'll need to load them. You'll also need to load the `lsr` package, because we'll make use of the `ciMean()` function to actually calculate the confidence intervals

⁹As of the current writing, these are the only arguments to the function. However, I am planning to add a bit more functionality to `ciMean()`. However, regardless of what those future changes might look like, the `x` and `conf` arguments will remain the same, and the commands used in this book will still work.

```
load( "./rbook-master/data/afl24.Rdata" ) # contains the "afl" data frame
library( sciplot )      # bargraph.CI() and lineplot.CI() functions
library( lsr )          # ciMean() function
```

Here's how to plot the means and confidence intervals drawn using `bargraph.CI()`.

```
bargraph.CI( x.factor = year,           # grouping variable
              response = attendance,    # outcome variable
              data = afl,               # data frame with the variables
              ci.fun= ciMean,          # name of the function to calculate CIs
              xlab = "Year",            # x-axis label
              ylab = "Average Attendance" # y-axis label
)
```



We can use the same arguments when calling the `lineplot.CI()` function:

```
lineplot.CI( x.factor = year,           # grouping variable
              response = attendance,    # outcome variable
              data = afl,               # data frame with the variables
              ci.fun= ciMean,          # name of the function to calculate CIs
              xlab = "Year",            # x-axis label
              ylab = "Average Attendance" # y-axis label
)
```

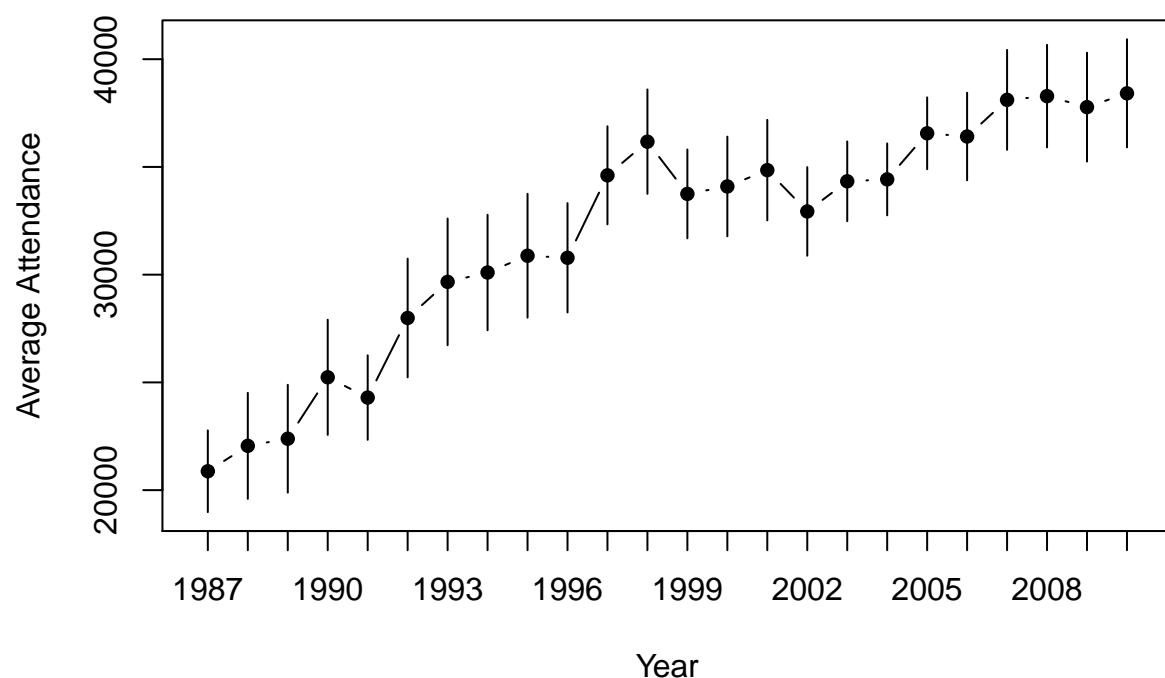


Figure 10.12: Means and 95% confidence intervals for AFL attendance, plotted separately for each year from 1987 to 2010. This graph was drawn using the `lineplot.CI()` function.

10.6 Summary

In this chapter I've covered two main topics. The first half of the chapter talks about sampling theory, and the second half talks about how we can use sampling theory to construct estimates of the population parameters. The section breakdown looks like this:

- Basic ideas about samples, sampling and populations (Section 10.1)
- Statistical theory of sampling: the law of large numbers (Section 10.2), sampling distributions and the central limit theorem (Section 10.3).
- Estimating means and standard deviations (Section 10.4)
- Estimating a confidence interval (Section 10.5)

As always, there's a lot of topics related to sampling and estimation that aren't covered in this chapter, but for an introductory psychology class this is fairly comprehensive I think. For most applied researchers you won't need much more theory than this. One big question that I haven't touched on in this chapter is what you do when you don't have a simple random sample. There is a lot of statistical theory you can draw on to handle this situation, but it's well beyond the scope of this book.

Chapter 11

Hypothesis testing

The process of induction is the process of assuming the simplest law that can be made to harmonize with our experience. This process, however, has no logical foundation but only a psychological one. It is clear that there are no grounds for believing that the simplest course of events will really happen. It is an hypothesis that the sun will rise tomorrow: and this means that we do not know whether it will rise.

– Ludwig Wittgenstein¹

In the last chapter, I discussed the ideas behind estimation, which is one of the two “big ideas” in inferential statistics. It’s now time to turn out attention to the other big idea, which is *hypothesis testing*. In its most abstract form, hypothesis testing really a very simple idea: the researcher has some theory about the world, and wants to determine whether or not the data actually support that theory. However, the details are messy, and most people find the theory of hypothesis testing to be the most frustrating part of statistics. The structure of the chapter is as follows. Firstly, I’ll describe how hypothesis testing works, in a fair amount of detail, using a simple running example to show you how a hypothesis test is “built”. I’ll try to avoid being too dogmatic while doing so, and focus instead on the underlying logic of the testing procedure.² Afterwards, I’ll spend a bit of time talking about the various dogmas, rules and heresies that surround the theory of hypothesis testing.

11.1 A menagerie of hypotheses

Eventually we all succumb to madness. For me, that day will arrive once I’m finally promoted to full professor. Safely ensconced in my ivory tower, happily protected by tenure, I will finally be able to take leave of my senses (so to speak), and indulge in that most thoroughly unproductive line of psychological research: the search for extrasensory perception (ESP).³

Let’s suppose that this glorious day has come. My first study is a simple one, in which I seek to test whether clairvoyance exists. Each participant sits down at a table, and is shown a card by an experimenter. The

¹The quote comes from Wittgenstein’s (1922) text, *Tractatus Logico-Philosophicus*.

²A technical note. The description below differs subtly from the standard description given in a lot of introductory texts. The orthodox theory of null hypothesis testing emerged from the work of Sir Ronald Fisher and Jerzy Neyman in the early 20th century; but Fisher and Neyman actually had very different views about how it should work. The standard treatment of hypothesis testing that most texts use is a hybrid of the two approaches. The treatment here is a little more Neyman-style than the orthodox view, especially as regards the meaning of the p value.

³My apologies to anyone who actually believes in this stuff, but on my reading of the literature on ESP, it’s just not reasonable to think this is real. To be fair, though, some of the studies are rigorously designed; so it’s actually an interesting area for thinking about psychological research design. And of course it’s a free country, so you can spend your own time and effort proving me wrong if you like, but I wouldn’t think that’s a terribly practical use of your intellect.

card is black on one side and white on the other. The experimenter takes the card away, and places it on a table in an adjacent room. The card is placed black side up or white side up completely at random, with the randomisation occurring only after the experimenter has left the room with the participant. A second experimenter comes in and asks the participant which side of the card is now facing upwards. It's purely a one-shot experiment. Each person sees only one card, and gives only one answer; and at no stage is the participant actually in contact with someone who knows the right answer. My data set, therefore, is very simple. I have asked the question of N people, and some number X of these people have given the correct response. To make things concrete, let's suppose that I have tested $N = 100$ people, and $X = 62$ of these got the answer right... a surprisingly large number, sure, but is it large enough for me to feel safe in claiming I've found evidence for ESP? This is the situation where hypothesis testing comes in useful. However, before we talk about how to *test* hypotheses, we need to be clear about what we mean by hypotheses.

11.1.1 Research hypotheses versus statistical hypotheses

The first distinction that you need to keep clear in your mind is between research hypotheses and statistical hypotheses. In my ESP study, my overall scientific goal is to demonstrate that clairvoyance exists. In this situation, I have a clear research goal: I am hoping to discover evidence for ESP. In other situations I might actually be a lot more neutral than that, so I might say that my research goal is to determine whether or not clairvoyance exists. Regardless of how I want to portray myself, the basic point that I'm trying to convey here is that a research hypothesis involves making a substantive, testable scientific claim... if you are a psychologist, then your research hypotheses are fundamentally *about* psychological constructs. Any of the following would count as ***research hypotheses***:

- *Listening to music reduces your ability to pay attention to other things.* This is a claim about the causal relationship between two psychologically meaningful concepts (listening to music and paying attention to things), so it's a perfectly reasonable research hypothesis.
- *Intelligence is related to personality.* Like the last one, this is a relational claim about two psychological constructs (intelligence and personality), but the claim is weaker: correlational not causal.
- *Intelligence is* speed of information processing.* *This hypothesis has a quite different character: it's not actually a relational claim at all. It's an ontological claim about the fundamental character of intelligence (and I'm pretty sure it's wrong). It's worth expanding on this one actually: It's usually easier to think about how to construct experiments to test research hypotheses of the form "does X affect Y?" than it is to address claims like "what is X?" And in practice, what usually happens is that you find ways of testing relational claims that follow from your ontological ones. For instance, if I believe that intelligence is* speed of information processing in the brain, my experiments will often involve looking for relationships between measures of intelligence and measures of speed.* As a consequence, most everyday research questions do tend to be relational in nature, but they're almost always motivated by deeper ontological questions about the state of nature.

Notice that in practice, my research hypotheses could overlap a lot. My ultimate goal in the ESP experiment might be to test an ontological claim like "ESP exists", but I might operationally restrict myself to a narrower hypothesis like "Some people can 'see' objects in a clairvoyant fashion". That said, there are some things that really don't count as proper research hypotheses in any meaningful sense:

- *Love is a battlefield.* This is too vague to be testable. While it's okay for a research hypothesis to have a degree of vagueness to it, it has to be possible to operationalise your theoretical ideas. Maybe I'm just not creative enough to see it, but I can't see how this can be converted into any concrete research design. If that's true, then this isn't a scientific research hypothesis, it's a pop song. That doesn't mean it's not interesting – a lot of deep questions that humans have fall into this category. Maybe one day science will be able to construct testable theories of love, or to test to see if God exists, and so on; but right now we can't, and I wouldn't bet on ever seeing a satisfying scientific approach to either.

- *The first rule of tautology club is the first rule of tautology club.* This is not a substantive claim of any kind. It's true by definition. No conceivable state of nature could possibly be inconsistent with this claim. As such, we say that this is an unfalsifiable hypothesis, and as such it is outside the domain of science. Whatever else you do in science, your claims must have the possibility of being wrong.
- *More people in my experiment will say “yes” than “no”.* This one fails as a research hypothesis because it's a claim about the data set, not about the psychology (unless of course your actual research question is whether people have some kind of “yes” bias!). As we'll see shortly, this hypothesis is starting to sound more like a statistical hypothesis than a research hypothesis.

As you can see, research hypotheses can be somewhat messy at times; and ultimately they are *scientific* claims. **Statistical hypotheses** are neither of these two things. Statistical hypotheses must be mathematically precise, and they must correspond to specific claims about the characteristics of the data generating mechanism (i.e., the “population”). Even so, the intent is that statistical hypotheses bear a clear relationship to the substantive research hypotheses that you care about! For instance, in my ESP study my research hypothesis is that some people are able to see through walls or whatever. What I want to do is to “map” this onto a statement about how the data were generated. So let's think about what that statement would be. The quantity that I'm interested in within the experiment is $P(\text{"correct"})$, the true-but-unknown probability with which the participants in my experiment answer the question correctly. Let's use the Greek letter θ (theta) to refer to this probability. Here are four different statistical hypotheses:

- If ESP doesn't exist and if my experiment is well designed, then my participants are just guessing. So I should expect them to get it right half of the time and so my statistical hypothesis is that the true probability of choosing correctly is $\theta = 0.5$.
- Alternatively, suppose ESP does exist and participants can see the card. If that's true, people will perform better than chance. The statistical hypothesis would be that $\theta > 0.5$.
- A third possibility is that ESP does exist, but the colours are all reversed and people don't realise it (okay, that's wacky, but you never know...). If that's how it works then you'd expect people's performance to be *below* chance. This would correspond to a statistical hypothesis that $\theta < 0.5$.
- Finally, suppose ESP exists, but I have no idea whether people are seeing the right colour or the wrong one. In that case, the only claim I could make about the data would be that the probability of making the correct answer is *not* equal to 50. This corresponds to the statistical hypothesis that $\theta \neq 0.5$.

All of these are legitimate examples of a statistical hypothesis because they are statements about a population parameter and are meaningfully related to my experiment.

What this discussion makes clear, I hope, is that when attempting to construct a statistical hypothesis test the researcher actually has two quite distinct hypotheses to consider. First, he or she has a research hypothesis (a claim about psychology), and this corresponds to a statistical hypothesis (a claim about the data generating population). In my ESP example, these might be

Dan.s.research.hypothesis	Dan.s.statistical.hypothesis
ESP.exists	$\$\\theta \\neq 0.5\$$

And the key thing to recognise is this: *a statistical hypothesis test is a test of the statistical hypothesis, not the research hypothesis.* If your study is badly designed, then the link between your research hypothesis and your statistical hypothesis is broken. To give a silly example, suppose that my ESP study was conducted in a situation where the participant can actually see the card reflected in a window; if that happens, I would be able to find very strong evidence that $\theta \neq 0.5$, but this would tell us nothing about whether “ESP exists”.

11.1.2 Null hypotheses and alternative hypotheses

So far, so good. I have a research hypothesis that corresponds to what I want to believe about the world, and I can map it onto a statistical hypothesis that corresponds to what I want to believe about how the data were generated. It's at this point that things get somewhat counterintuitive for a lot of people. Because

what I'm about to do is invent a new statistical hypothesis (the “null” hypothesis, H_0) that corresponds to the exact opposite of what I want to believe, and then focus exclusively on that, almost to the neglect of the thing I'm actually interested in (which is now called the “alternative” hypothesis, H_1). In our ESP example, the null hypothesis is that $\theta = 0.5$, since that's what we'd expect if ESP *didn't* exist. My hope, of course, is that ESP is totally real, and so the *alternative* to this null hypothesis is $\theta \neq 0.5$. In essence, what we're doing here is dividing up the possible values of θ into two groups: those values that I really hope aren't true (the null), and those values that I'd be happy with if they turn out to be right (the alternative). Having done so, the important thing to recognise is that the goal of a hypothesis test is *not* to show that the alternative hypothesis is (probably) true; the goal is to show that the null hypothesis is (probably) false. Most people find this pretty weird.

The best way to think about it, in my experience, is to imagine that a hypothesis test is a criminal trial⁴... *the trial of the null hypothesis*. The null hypothesis is the defendant, the researcher is the prosecutor, and the statistical test itself is the judge. Just like a criminal trial, there is a presumption of innocence: the null hypothesis is *deemed* to be true unless you, the researcher, can prove beyond a reasonable doubt that it is false. You are free to design your experiment however you like (within reason, obviously!), and your goal when doing so is to maximise the chance that the data will yield a conviction... for the crime of being false. The catch is that the statistical test sets the rules of the trial, and those rules are designed to protect the null hypothesis – specifically to ensure that if the null hypothesis is actually true, the chances of a false conviction are guaranteed to be low. This is pretty important: after all, the null hypothesis doesn't get a lawyer. And given that the researcher is trying desperately to prove it to be false, *someone* has to protect it.

11.2 Two types of errors

Before going into details about how a statistical test is constructed, it's useful to understand the philosophy behind it. I hinted at it when pointing out the similarity between a null hypothesis test and a criminal trial, but I should now be explicit. Ideally, we would like to construct our test so that we never make any errors. Unfortunately, since the world is messy, this is never possible. Sometimes you're just really unlucky: for instance, suppose you flip a coin 10 times in a row and it comes up heads all 10 times. That feels like very strong evidence that the coin is biased (and it is!), but of course there's a 1 in 1024 chance that this would happen even if the coin was totally fair. In other words, in real life we *always* have to accept that there's a chance that we did the wrong thing. As a consequence, the goal behind statistical hypothesis testing is not to *eliminate* errors, but to *minimise* them.

At this point, we need to be a bit more precise about what we mean by “errors”. Firstly, let's state the obvious: it is either the case that the null hypothesis is true, or it is false; and our test will either reject the null hypothesis or retain it.⁵ So, as the table below illustrates, after we run the test and make our choice, one of four things might have happened:

	retain \$H_0\$	reject \$H_0\$
\$H_0\$ is true	correct decision	error (type I)
\$H_0\$ is false	error (type II)	correct decision

⁴This analogy only works if you're from an adversarial legal system like UK/US/Australia. As I understand these things, the French inquisitorial system is quite different.

⁵An aside regarding the language you use to talk about hypothesis testing. Firstly, one thing you really want to avoid is the word “prove”: a statistical test really doesn't *prove* that a hypothesis is true or false. Proof implies certainty, and as the saying goes, statistics means never having to say you're certain. On that point almost everyone would agree. However, beyond that there's a fair amount of confusion. Some people argue that you're only allowed to make statements like “rejected the null”, “failed to reject the null”, or possibly “retained the null”. According to this line of thinking, you can't say things like “accept the alternative” or “accept the null”. Personally I think this is too strong: in my opinion, this conflates null hypothesis testing with Karl Popper's falsificationist view of the scientific process. While there are similarities between falsificationism and null hypothesis testing, they aren't equivalent. However, while I personally think it's fine to talk about accepting a hypothesis (on the proviso that “acceptance” doesn't actually mean that it's necessarily true, especially in the case of the null hypothesis), many people will disagree. And more to the point, you should be aware that this particular weirdness exists, so that you're not caught unawares by it when writing up your own results.

As a consequence there are actually *two* different types of error here. If we reject a null hypothesis that is actually true, then we have made a ***type I error***. On the other hand, if we retain the null hypothesis when it is in fact false, then we have made a ***type II error***.

Remember how I said that statistical testing was kind of like a criminal trial? Well, I meant it. A criminal trial requires that you establish “beyond a reasonable doubt” that the defendant did it. All of the evidentiary rules are (in theory, at least) designed to ensure that there’s (almost) no chance of wrongfully convicting an innocent defendant. The trial is designed to protect the rights of a defendant: as the English jurist William Blackstone famously said, it is “better that ten guilty persons escape than that one innocent suffer.” In other words, a criminal trial doesn’t treat the two types of error in the same way... punishing the innocent is deemed to be much worse than letting the guilty go free. A statistical test is pretty much the same: the single most important design principle of the test is to *control* the probability of a type I error, to keep it below some fixed probability. This probability, which is denoted α , is called the ***significance level*** of the test (or sometimes, the *size* of the test). And I’ll say it again, because it is so central to the whole set-up... a hypothesis test is said to have significance level α if the type I error rate is no larger than α .

So, what about the type II error rate? Well, we’d also like to keep those under control too, and we denote this probability by β . However, it’s much more common to refer to the ***power*** of the test, which is the probability with which we reject a null hypothesis when it really is false, which is $1 - \beta$. To help keep this straight, here’s the same table again, but with the relevant numbers added:

	retain H_0	reject H_0
H_0 is true	$1 - \alpha$ (probability of correct retention)	α (type I error rate)
H_0 is false	β (type II error rate)	$1 - \beta$ (power of the test)

A “powerful” hypothesis test is one that has a small value of β , while still keeping α fixed at some (small) desired level. By convention, scientists make use of three different α levels: .05, .01 and .001. Notice the asymmetry here... the tests are designed to *ensure* that the α level is kept small, but there’s no corresponding guarantee regarding β . We’d certainly *like* the type II error rate to be small, and we try to design tests that keep it small, but this is very much secondary to the overwhelming need to control the type I error rate. As Blackstone might have said if he were a statistician, it is “better to retain 10 false null hypotheses than to reject a single true one”. To be honest, I don’t know that I agree with this philosophy – there are situations where I think it makes sense, and situations where I think it doesn’t – but that’s neither here nor there. It’s how the tests are built.

11.3 Test statistics and sampling distributions

At this point we need to start talking specifics about how a hypothesis test is constructed. To that end, let’s return to the ESP example. Let’s ignore the actual data that we obtained, for the moment, and think about the structure of the experiment. Regardless of what the actual numbers are, the *form* of the data is that X out of N people correctly identified the colour of the hidden card. Moreover, let’s suppose for the moment that the null hypothesis really is true: ESP doesn’t exist, and the true probability that anyone picks the correct colour is exactly $\theta = 0.5$. What would we *expect* the data to look like? Well, obviously, we’d expect the proportion of people who make the correct response to be pretty close to 50%. Or, to phrase this in more mathematical terms, we’d say that X/N is approximately 0.5. Of course, we wouldn’t expect this fraction to be *exactly* 0.5: if, for example we tested $N = 100$ people, and $X = 53$ of them got the question right, we’d probably be forced to concede that the data are quite consistent with the null hypothesis. On the other hand, if $X = 99$ of our participants got the question right, then we’d feel pretty confident that the null hypothesis is wrong. Similarly, if only $X = 3$ people got the answer right, we’d be similarly confident that the null was wrong. Let’s be a little more technical about this: we have a quantity X that we can calculate by looking at our data; after looking at the value of X , we make a decision about whether to believe that the null hypothesis is correct, or to reject the null hypothesis in favour of the alternative. The name for this thing that we calculate to guide our choices is a ***test statistic***.

Having chosen a test statistic, the next step is to state precisely which values of the test statistic would

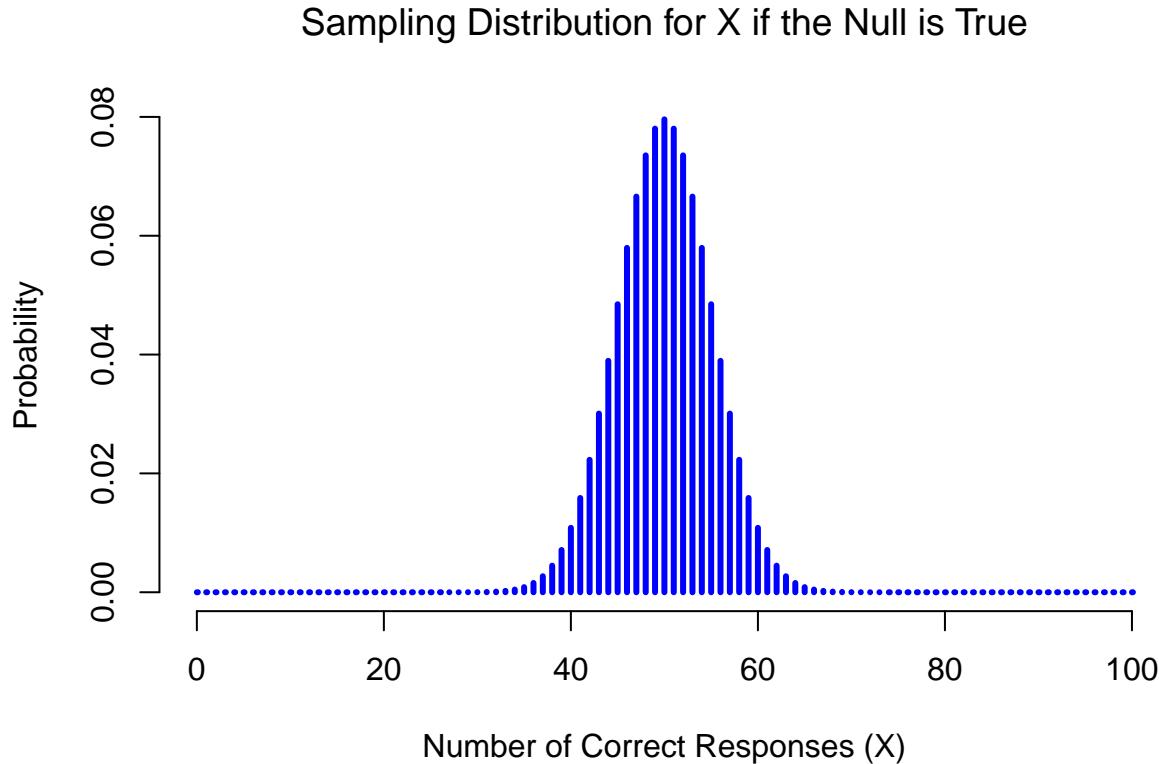


Figure 11.1: The sampling distribution for our test statistic X when the null hypothesis is true. For our ESP scenario, this is a binomial distribution. Not surprisingly, since the null hypothesis says that the probability of a correct response is $\theta = .5$, the sampling distribution says that the most likely value is 50 (out of 100) correct responses. Most of the probability mass lies between 40 and 60.

cause is to reject the null hypothesis, and which values would cause us to keep it. In order to do so, we need to determine what the *sampling distribution of the test statistic* would be if the null hypothesis were actually true (we talked about sampling distributions earlier in Section 10.3.1). Why do we need this? Because this distribution tells us exactly what values of X our null hypothesis would lead us to expect. And therefore, we can use this distribution as a tool for assessing how closely the null hypothesis agrees with our data.

How do we actually determine the sampling distribution of the test statistic? For a lot of hypothesis tests this step is actually quite complicated, and later on in the book you'll see me being slightly evasive about it for some of the tests (some of them I don't even understand myself). However, sometimes it's very easy. And, fortunately for us, our ESP example provides us with one of the easiest cases. Our population parameter θ is just the overall probability that people respond correctly when asked the question, and our test statistic X is the *count* of the number of people who did so, out of a sample size of N . We've seen a distribution like this before, in Section 9.4: that's exactly what the binomial distribution describes! So, to use the notation and terminology that I introduced in that section, we would say that the null hypothesis predicts that X is binomially distributed, which is written

$$X \sim \text{Binomial}(\theta, N)$$

Since the null hypothesis states that $\theta = 0.5$ and our experiment has $N = 100$ people, we have the sampling distribution we need. This sampling distribution is plotted in Figure 11.1. No surprises really: the null

hypothesis says that $X = 50$ is the most likely outcome, and it says that we're almost certain to see somewhere between 40 and 60 correct responses.

11.4 Making decisions

Okay, we're very close to being finished. We've constructed a test statistic (X), and we chose this test statistic in such a way that we're pretty confident that if X is close to $N/2$ then we should retain the null, and if not we should reject it. The question that remains is this: exactly which values of the test statistic should we associate with the null hypothesis, and which exactly values go with the alternative hypothesis? In my ESP study, for example, I've observed a value of $X = 62$. What decision should I make? Should I choose to believe the null hypothesis, or the alternative hypothesis?

11.4.1 Critical regions and critical values

To answer this question, we need to introduce the concept of a ***critical region*** for the test statistic X . The critical region of the test corresponds to those values of X that would lead us to reject null hypothesis (which is why the critical region is also sometimes called the rejection region). How do we find this critical region? Well, let's consider what we know:

- X should be very big or very small in order to reject the null hypothesis.
- If the null hypothesis is true, the sampling distribution of X is $\text{Binomial}(0.5, N)$.
- If $\alpha = .05$, the critical region must cover 5% of this sampling distribution.

It's important to make sure you understand this last point: the critical region corresponds to those values of X for which we would reject the null hypothesis, and the sampling distribution in question describes the probability that we would obtain a particular value of X if the null hypothesis were actually true. Now, let's suppose that we chose a critical region that covers 20% of the sampling distribution, and suppose that the null hypothesis is actually true. What would be the probability of incorrectly rejecting the null? The answer is of course 20%. And therefore, we would have built a test that had an α level of 0.2. If we want $\alpha = .05$, the critical region is only *allowed* to cover 5% of the sampling distribution of our test statistic.

As it turns out, those three things uniquely solve the problem: our critical region consists of the most *extreme values*, known as the ***tails*** of the distribution. This is illustrated in Figure 11.2. As it turns out, if we want $\alpha = .05$, then our critical regions correspond to $X \leq 40$ and $X \geq 60$.⁶ That is, if the number of people saying "true" is between 41 and 59, then we should retain the null hypothesis. If the number is between 0 to 40 or between 60 to 100, then we should reject the null hypothesis. The numbers 40 and 60 are often referred to as the ***critical values***, since they define the edges of the critical region.

At this point, our hypothesis test is essentially complete: (1) we choose an α level (e.g., $\alpha = .05$), (2) come up with some test statistic (e.g., X) that does a good job (in some meaningful sense) of comparing H_0 to H_1 , (3) figure out the sampling distribution of the test statistic on the assumption that the null hypothesis is true (in this case, binomial) and then (4) calculate the critical region that produces an appropriate α level (0-40 and 60-100). All that we have to do now is calculate the value of the test statistic for the real data (e.g., $X = 62$) and then compare it to the critical values to make our decision. Since 62 is greater than the critical value of 60, we would reject the null hypothesis. Or, to phrase it slightly differently, we say that the test has produced a ***significant*** result.

⁶Strictly speaking, the test I just constructed has $\alpha = .057$, which is a bit too generous. However, if I'd chosen 39 and 61 to be the boundaries for the critical region, then the critical region only covers 3.5% of the distribution. I figured that it makes more sense to use 40 and 60 as my critical values, and be willing to tolerate a 5.7% type I error rate, since that's as close as I can get to a value of $\alpha = .05$.

Critical Regions for a Two-Sided Test

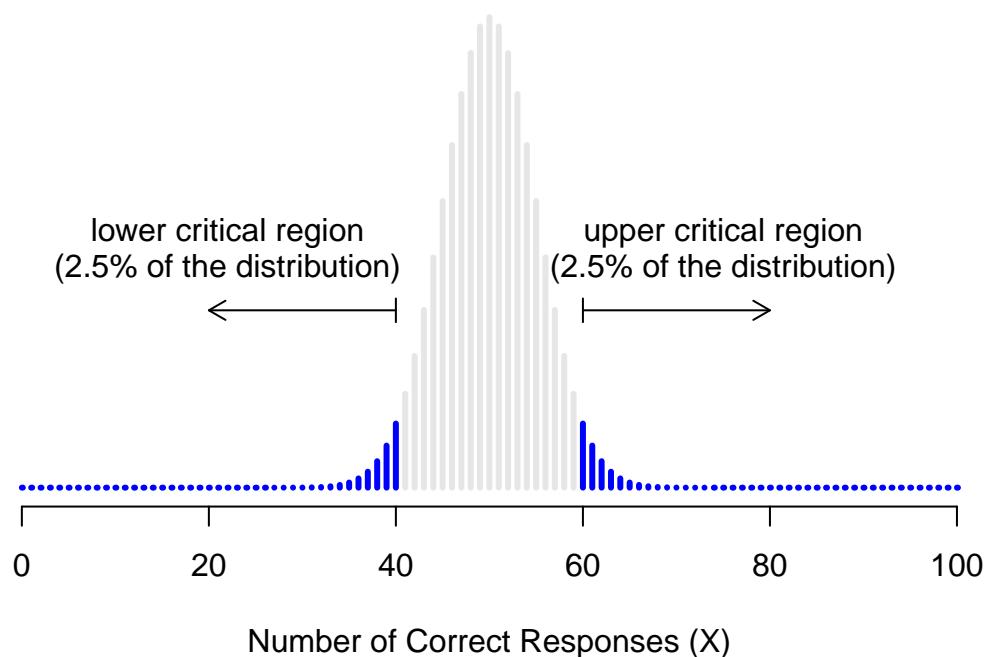


Figure 11.2: The critical region associated with the hypothesis test for the ESP study, for a hypothesis test with a significance level of $\alpha = .05$. The plot itself shows the sampling distribution of X under the null hypothesis: the grey bars correspond to those values of X for which we would retain the null hypothesis. The black bars show the critical region: those values of X for which we would reject the null. Because the alternative hypothesis is two sided (i.e., allows both $\theta < .5$ and $\theta > .5$), the critical region covers both tails of the distribution. To ensure an α level of $.05$, we need to ensure that each of the two regions encompasses 2.5% of the sampling distribution.

11.4.2 A note on statistical “significance”

Like other occult techniques of divination, the statistical method has a private jargon deliberately contrived to obscure its methods from non-practitioners.

– Attributed to G. O. Ashley⁷

A very brief digression is in order at this point, regarding the word “significant”. The concept of statistical significance is actually a very simple one, but has a very unfortunate name. If the data allow us to reject the null hypothesis, we say that “the result is *statistically significant*”, which is often shortened to “the result is significant”. This terminology is rather old, and dates back to a time when “significant” just meant something like “indicated”, rather than its modern meaning, which is much closer to “important”. As a result, a lot of modern readers get very confused when they start learning statistics, because they think that a “significant result” must be an important one. It doesn’t mean that at all. All that “statistically significant” means is that the data allowed us to reject a null hypothesis. Whether or not the result is actually important in the real world is a very different question, and depends on all sorts of other things.

11.4.3 The difference between one sided and two sided tests

There’s one more thing I want to point out about the hypothesis test that I’ve just constructed. If we take a moment to think about the statistical hypotheses I’ve been using,

$$\begin{aligned} H_0 : \theta &= .5 \\ H_1 : \theta &\neq .5 \end{aligned}$$

we notice that the alternative hypothesis covers *both* the possibility that $\theta < .5$ and the possibility that $\theta > .5$. This makes sense if I really think that ESP could produce better-than-chance performance *or* worse-than-chance performance (and there are some people who think that). In statistical language, this is an example of a **two-sided test**. It’s called this because the alternative hypothesis covers the area on both “sides” of the null hypothesis, and as a consequence the critical region of the test covers both tails of the sampling distribution (2.5% on either side if $\alpha = .05$), as illustrated earlier in Figure 11.2.

However, that’s not the only possibility. It might be the case, for example, that I’m only willing to believe in ESP if it produces better than chance performance. If so, then my alternative hypothesis would only covers the possibility that $\theta > .5$, and as a consequence the null hypothesis now becomes $\theta \leq .5$:

$$\begin{aligned} H_0 : \theta &\leq .5 \\ H_1 : \theta &> .5 \end{aligned}$$

When this happens, we have what’s called a **one-sided test**, and when this happens the critical region only covers one tail of the sampling distribution. This is illustrated in Figure 11.3.

11.5 The p value of a test

In one sense, our hypothesis test is complete; we’ve constructed a test statistic, figured out its sampling distribution if the null hypothesis is true, and then constructed the critical region for the test. Nevertheless, I’ve actually omitted the most important number of all: **the p value**. It is to this topic that we now turn. There are two somewhat different ways of interpreting a p value, one proposed by Sir Ronald Fisher and the other by Jerzy Neyman. Both versions are legitimate, though they reflect very different ways of thinking about hypothesis tests. Most introductory textbooks tend to give Fisher’s version only, but I think that’s a bit of a shame. To my mind, Neyman’s version is cleaner, and actually better reflects the logic of the null hypothesis test. You might disagree though, so I’ve included both. I’ll start with Neyman’s version...

⁷The internet seems fairly convinced that Ashley said this, though I can’t for the life of me find anyone willing to give a source for the claim.

Critical Region for a One-Sided Test

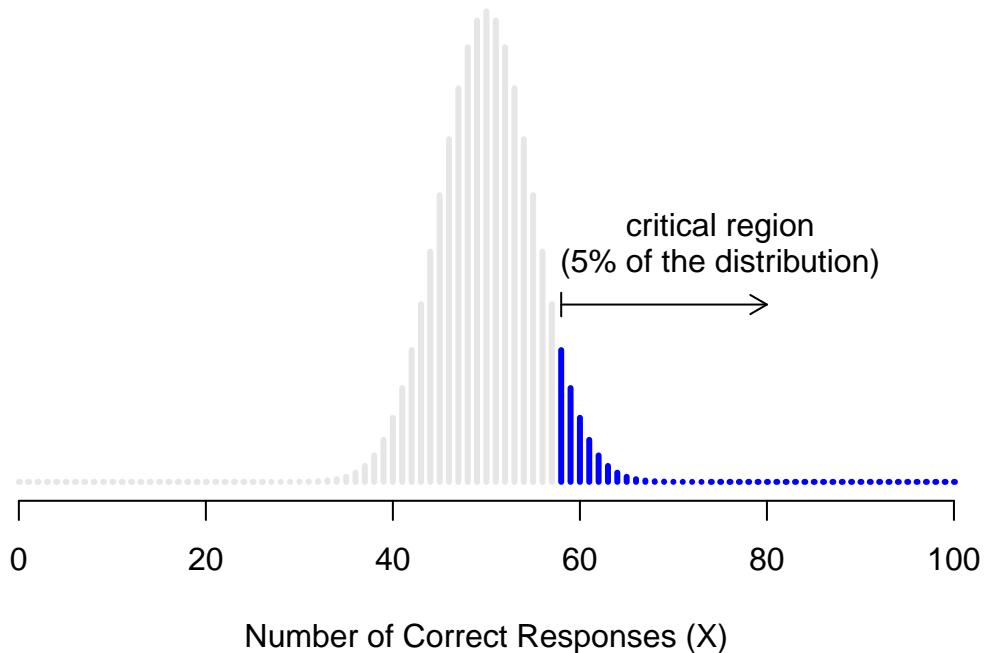


Figure 11.3: The critical region for a one sided test. In this case, the alternative hypothesis is that $\theta > .05$, so we would only reject the null hypothesis for large values of X . As a consequence, the critical region only covers the upper tail of the sampling distribution; specifically the upper 5% of the distribution. Contrast this to the two-sided version earlier)

11.5.1 A softer view of decision making

One problem with the hypothesis testing procedure that I've described is that it makes no distinction at all between a result that's "barely significant" and those that are "highly significant". For instance, in my ESP study the data I obtained only just fell inside the critical region - so I did get a significant effect, but was a pretty near thing. In contrast, suppose that I'd run a study in which $X = 97$ out of my $N = 100$ participants got the answer right. This would obviously be significant too, but by a much larger margin; there's really no ambiguity about this at all. The procedure that I described makes no distinction between the two. If I adopt the standard convention of allowing $\alpha = .05$ as my acceptable Type I error rate, then both of these are significant results.

This is where the p value comes in handy. To understand how it works, let's suppose that we ran lots of hypothesis tests on the same data set: but with a different value of α in each case. When we do that for my original ESP data, what we'd get is something like this

Value of α	Reject the null?
0.05	Yes
0.04	Yes
0.03	Yes
0.02	No
0.01	No

When we test ESP data ($X = 62$ successes out of $N = 100$ observations) using α levels of .03 and above, we'd always find ourselves rejecting the null hypothesis. For α levels of .02 and below, we always end up retaining the null hypothesis. Therefore, somewhere between .02 and .03 there must be a smallest value of α that would allow us to reject the null hypothesis for this data. This is the p value; as it turns out the ESP data has $p = .021$. In short:

p is defined to be the smallest Type I error rate (α) that you have to be willing to tolerate if you want to reject the null hypothesis.

If it turns out that p describes an error rate that you find intolerable, then you must retain the null. If you're comfortable with an error rate equal to p , then it's okay to reject the null hypothesis in favour of your preferred alternative.

In effect, p is a summary of all the possible hypothesis tests that you could have run, taken across all possible α values. And as a consequence it has the effect of “softening” our decision process. For those tests in which $p \leq \alpha$ you would have rejected the null hypothesis, whereas for those tests in which $p > \alpha$ you would have retained the null. In my ESP study I obtained $X = 62$, and as a consequence I’ve ended up with $p = .021$. So the error rate I have to tolerate is 2.1%. In contrast, suppose my experiment had yielded $X = 97$. What happens to my p value now? This time it’s shrunk to $p = 1.36 \times 10^{-25}$, which is a tiny, tiny⁸ Type I error rate. For this second case I would be able to reject the null hypothesis with a lot more confidence, because I only have to be “willing” to tolerate a type I error rate of about 1 in 10 trillion trillion in order to justify my decision to reject.

11.5.2 The probability of extreme data

The second definition of the p -value comes from Sir Ronald Fisher, and it's actually this one that you tend to see in most introductory statistics textbooks. Notice how, when I constructed the critical region, it corresponded to the *tails* (i.e., extreme values) of the sampling distribution? That's not a coincidence: almost all "good" tests have this characteristic (good in the sense of minimising our type II error rate, β). The reason for that is that a good critical region almost always corresponds to those values of the test statistic that are least likely to be observed if the null hypothesis is true. If this rule is true, then we can

define the p -value as the probability that we would have observed a test statistic that is at least as extreme as the one we actually did get. In other words, if the data are extremely implausible according to the null hypothesis, then the null hypothesis is probably wrong.

11.5.3 A common mistake

Okay, so you can see that there are two rather different but legitimate ways to interpret the p value, one based on Neyman's approach to hypothesis testing and the other based on Fisher's. Unfortunately, there is a third explanation that people sometimes give, especially when they're first learning statistics, and it is *absolutely and completely wrong*. This mistaken approach is to refer to the p value as "the probability that the null hypothesis is true". It's an intuitively appealing way to think, but it's wrong in two key respects: (1) null hypothesis testing is a frequentist tool, and the frequentist approach to probability does *not* allow you to assign probabilities to the null hypothesis... according to this view of probability, the null hypothesis is either true or it is not; it cannot have a "5% chance" of being true. (2) even within the Bayesian approach, which does let you assign probabilities to hypotheses, the p value would not correspond to the probability that the null is true; this interpretation is entirely inconsistent with the mathematics of how the p value is calculated. Put bluntly, despite the intuitive appeal of thinking this way, there is *no* justification for interpreting a p value this way. Never do it.

11.6 Reporting the results of a hypothesis test

When writing up the results of a hypothesis test, there's usually several pieces of information that you need to report, but it varies a fair bit from test to test. Throughout the rest of the book I'll spend a little time talking about how to report the results of different tests (see Section ?? for a particularly detailed example), so that you can get a feel for how it's usually done. However, regardless of what test you're doing, the one thing that you always have to do is say something about the p value, and whether or not the outcome was significant.

The fact that you have to do this is unsurprising; it's the whole point of doing the test. What might be surprising is the fact that there is some contention over exactly how you're supposed to do it. Leaving aside those people who completely disagree with the entire framework underpinning null hypothesis testing, there's a certain amount of tension that exists regarding whether or not to report the exact p value that you obtained, or if you should state only that $p < \alpha$ for a significance level that you chose in advance (e.g., $p < .05$).

11.6.1 The issue

To see why this is an issue, the key thing to recognise is that p values are *terribly* convenient. In practice, the fact that we can compute a p value means that we don't actually have to specify any α level at all in order to run the test. Instead, what you can do is calculate your p value and interpret it directly: if you get $p = .062$, then it means that you'd have to be willing to tolerate a Type I error rate of 6.2% to justify rejecting the null. If you personally find 6.2% intolerable, then you retain the null. Therefore, the argument goes, why don't we just report the actual p value and let the reader make up their own minds about what an acceptable Type I error rate is? This approach has the big advantage of "softening" the decision making process – in fact, if you accept the Neyman definition of the p value, that's the whole point of the p value. We no longer have a fixed significance level of $\alpha = .05$ as a bright line separating "accept" from "reject" decisions; and this removes the rather pathological problem of being forced to treat $p = .051$ in a fundamentally different way to $p = .049$.

This flexibility is both the advantage and the disadvantage to the p value. The reason why a lot of people don't like the idea of reporting an exact p value is that it gives the researcher a bit *too much* freedom. In particular, it lets you change your mind about what error tolerance you're willing to put up with *after* you

Table 11.1: A commonly adopted convention for reporting p values: in many places it is conventional to report one of four different things (e.g., $p < .05$) as shown below. I've included the "significance stars" notation (i.e., a * indicates $p < .05$) because you sometimes see this notation produced by statistical software. It's also worth noting that some people will write *n.s.* (not significant) rather than $p > .05$.

Usual notation	Signif. stars	Signif. stars
$p > .05$	NA	The test wasn't significant
$p < .05$	*	The test was significant at $\alpha = .05$ but not at $\alpha = .01$ or $\alpha = .001$
$p < .01$	**	The test was significant at $\alpha = .05$ and $\alpha = .01$ but not at $\alpha = .001$
$p < .001$	***	The test was significant at all levels

look at the data. For instance, consider my ESP experiment. Suppose I ran my test, and ended up with a p value of .09. Should I accept or reject? Now, to be honest, I haven't yet bothered to think about what level of Type I error I'm "really" willing to accept. I don't have an opinion on that topic. But I *do* have an opinion about whether or not ESP exists, and I *definitely* have an opinion about whether my research should be published in a reputable scientific journal. And amazingly, now that I've looked at the data I'm starting to think that a 9% error rate isn't so bad, especially when compared to how annoying it would be to have to admit to the world that my experiment has failed. So, to avoid looking like I just made it up after the fact, I now say that my α is .1: a 10% type I error rate isn't too bad, and at that level my test is significant! I win.

In other words, the worry here is that I might have the best of intentions, and be the most honest of people, but the temptation to just "shade" things a little bit here and there is really, really strong. As anyone who has ever run an experiment can attest, it's a long and difficult process, and you often get *very* attached to your hypotheses. It's hard to let go and admit the experiment didn't find what you wanted it to find. And that's the danger here. If we use the "raw" p -value, people will start interpreting the data in terms of what they *want* to believe, not what the data are actually saying... and if we allow that, well, why are we bothering to do science at all? Why not let everyone believe whatever they like about anything, regardless of what the facts are? Okay, that's a bit extreme, but that's where the worry comes from. According to this view, you really *must* specify your α value in advance, and then only report whether the test was significant or not. It's the only way to keep ourselves honest.

11.6.2 Two proposed solutions

In practice, it's pretty rare for a researcher to specify a single α level ahead of time. Instead, the convention is that scientists rely on three standard significance levels: .05, .01 and .001. When reporting your results, you indicate which (if any) of these significance levels allow you to reject the null hypothesis. This is summarised in Table 11.1. This allows us to soften the decision rule a little bit, since $p < .01$ implies that the data meet a stronger evidentiary standard than $p < .05$ would. Nevertheless, since these levels are fixed in advance by convention, it does prevent people choosing their α level after looking at the data.

Nevertheless, quite a lot of people still prefer to report exact p values. To many people, the advantage of allowing the reader to make up their own mind about how to interpret $p = .06$ outweighs any disadvantages. In practice, however, even among those researchers who prefer exact p values it is quite common to just write $p < .001$ instead of reporting an exact value for small p . This is in part because a lot of software doesn't actually print out the p value when it's that small (e.g., SPSS just writes $p = .000$ whenever $p < .001$), and in part because a very small p value can be kind of misleading. The human mind sees a number like .0000000001 and it's hard to suppress the gut feeling that the evidence in favour of the alternative hypothesis is a near certainty. In practice however, this is usually wrong. Life is a big, messy, complicated thing: and every statistical test ever invented relies on simplifications, approximations and assumptions. As a consequence, it's probably not reasonable to walk away from *any* statistical analysis with a feeling of confidence stronger than $p < .001$ implies. In other words, $p < .001$ is really code for "as far as *this test* is concerned, the evidence is overwhelming."

In light of all this, you might be wondering exactly what you should do. There's a fair bit of contradictory advice on the topic, with some people arguing that you should report the exact p value, and other people arguing that you should use the tiered approach illustrated in Table 11.1. As a result, the best advice I can give is to suggest that you look at papers/reports written in your field and see what the convention seems to be. If there doesn't seem to be any consistent pattern, then use whichever method you prefer.

11.7 Running the hypothesis test in practice

At this point some of you might be wondering if this is a “real” hypothesis test, or just a toy example that I made up. It’s real. In the previous discussion I built the test from first principles, thinking that it was the simplest possible problem that you might ever encounter in real life. However, this test already exists: it’s called the *binomial test*, and it’s implemented by an R function called `binom.test()`. To test the null hypothesis that the response probability is one-half $p = .5$,⁹ using data in which $x = 62$ of $n = 100$ people made the correct response, here’s how to do it in R:

```
binom.test( x=62, n=100, p=.5 )

##
##  Exact binomial test
##
## data: 62 and 100
## number of successes = 62, number of trials = 100, p-value =
## 0.02098
## alternative hypothesis: true probability of success is not equal to 0.5
## 95 percent confidence interval:
##  0.5174607 0.7152325
## sample estimates:
## probability of success
##                         0.62
```

Right now, this output looks pretty unfamiliar to you, but you can see that it’s telling you more or less the right things. Specifically, the p -value of 0.02 is less than the usual choice of $\alpha = .05$, so you can reject the null. We’ll talk a lot more about how to read this sort of output as we go along; and after a while you’ll hopefully find it quite easy to read and understand. For now, however, I just wanted to make the point that R contains a whole lot of functions corresponding to different kinds of hypothesis test. And while I’ll usually spend quite a lot of time explaining the logic behind how the tests are built, every time I discuss a hypothesis test the discussion will end with me showing you a fairly simple R command that you can use to run the test in practice.

11.8 Effect size, sample size and power

In previous sections I’ve emphasised the fact that the major design principle behind statistical hypothesis testing is that we try to control our Type I error rate. When we fix $\alpha = .05$ we are attempting to ensure that only 5% of true null hypotheses are incorrectly rejected. However, this doesn’t mean that we don’t care about Type II errors. In fact, from the researcher’s perspective, the error of failing to reject the null when it is actually false is an extremely annoying one. With that in mind, a secondary goal of hypothesis testing is to try to minimise β , the Type II error rate, although we don’t usually *talk* in terms of minimising Type II errors. Instead, we talk about maximising the *power* of the test. Since power is defined as $1 - \beta$, this is the same thing.

⁹Note that the p here has nothing to do with a p value. The p argument in the `binom.test()` function corresponds to the probability of making a correct response, according to the null hypothesis. In other words, it’s the θ value.

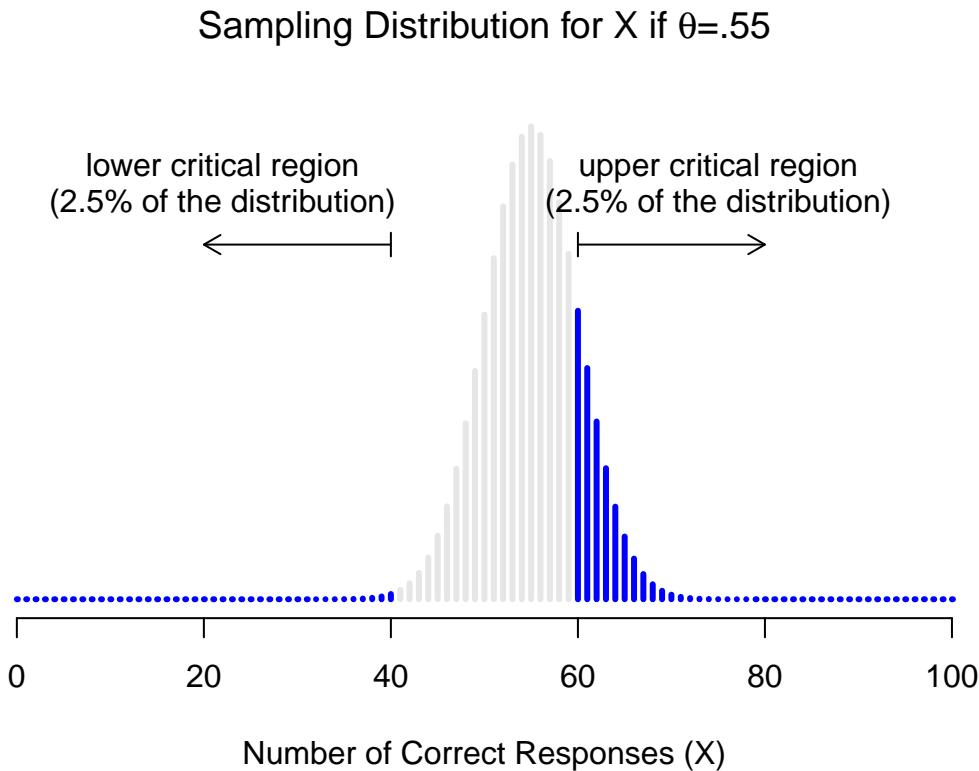


Figure 11.4: Sampling distribution under the *alternative* hypothesis, for a population parameter value of $\theta = 0.55$. A reasonable proportion of the distribution lies in the rejection region.

11.8.1 The power function

Let's take a moment to think about what a Type II error actually is. A Type II error occurs when the alternative hypothesis is true, but we are nevertheless unable to reject the null hypothesis. Ideally, we'd be able to calculate a single number β that tells us the Type II error rate, in the same way that we can set $\alpha = .05$ for the Type I error rate. Unfortunately, this is a lot trickier to do. To see this, notice that in my ESP study the alternative hypothesis actually corresponds to lots of possible values of θ . In fact, the alternative hypothesis corresponds to every value of θ *except* 0.5. Let's suppose that the true probability of someone choosing the correct response is 55% (i.e., $\theta = .55$). If so, then the *true* sampling distribution for X is not the same one that the null hypothesis predicts: the most likely value for X is now 55 out of 100. Not only that, the whole sampling distribution has now shifted, as shown in Figure 11.4. The critical regions, of course, do not change: by definition, the critical regions are based on what the null hypothesis predicts. What we're seeing in this figure is the fact that when the null hypothesis is wrong, a much larger proportion of the sampling distribution falls in the critical region. And of course that's what should happen: the probability of rejecting the null hypothesis is larger when the null hypothesis is actually false! However $\theta = .55$ is not the only possibility consistent with the alternative hypothesis. Let's instead suppose that the true value of θ is actually 0.7. What happens to the sampling distribution when this occurs? The answer, shown in Figure 11.5, is that almost the entirety of the sampling distribution has now moved into the critical region. Therefore, if $\theta = 0.7$ the probability of us correctly rejecting the null hypothesis (i.e., the power of the test) is much larger than if $\theta = 0.55$. In short, while $\theta = .55$ and $\theta = .70$ are both part of the alternative hypothesis, the Type II error rate is different.

What all this means is that the power of a test (i.e., $1 - \beta$) depends on the true value of θ . To illustrate this,

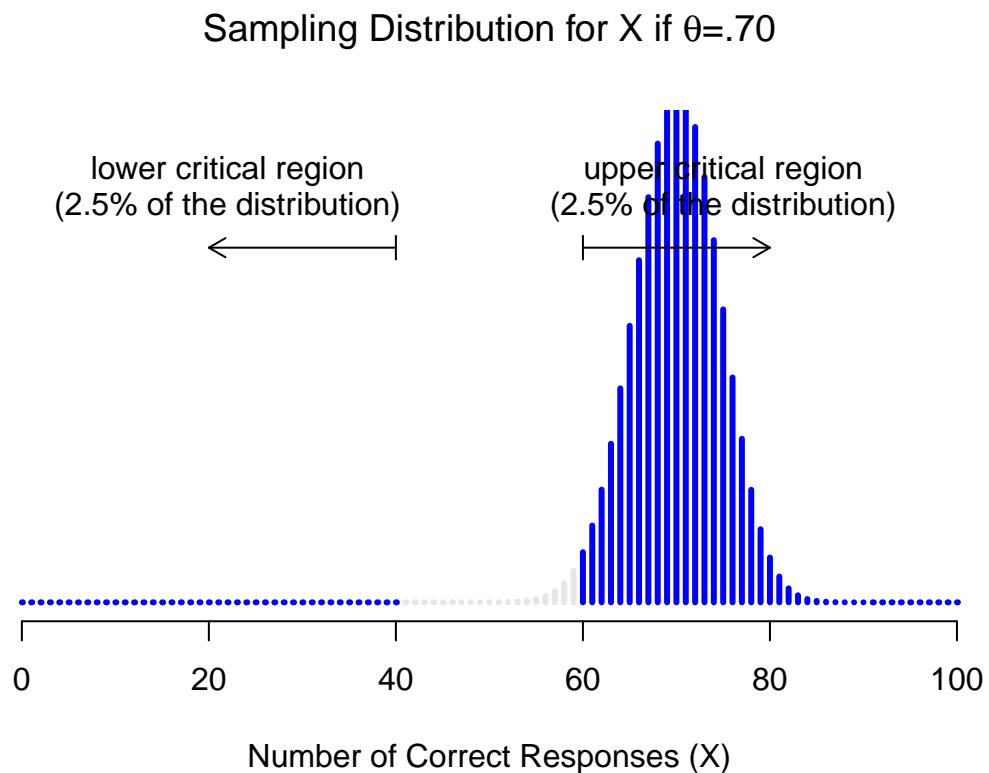


Figure 11.5: Sampling distribution under the *alternative* hypothesis, for a population parameter value of $\theta = 0.70$. Almost all of the distribution lies in the rejection region.

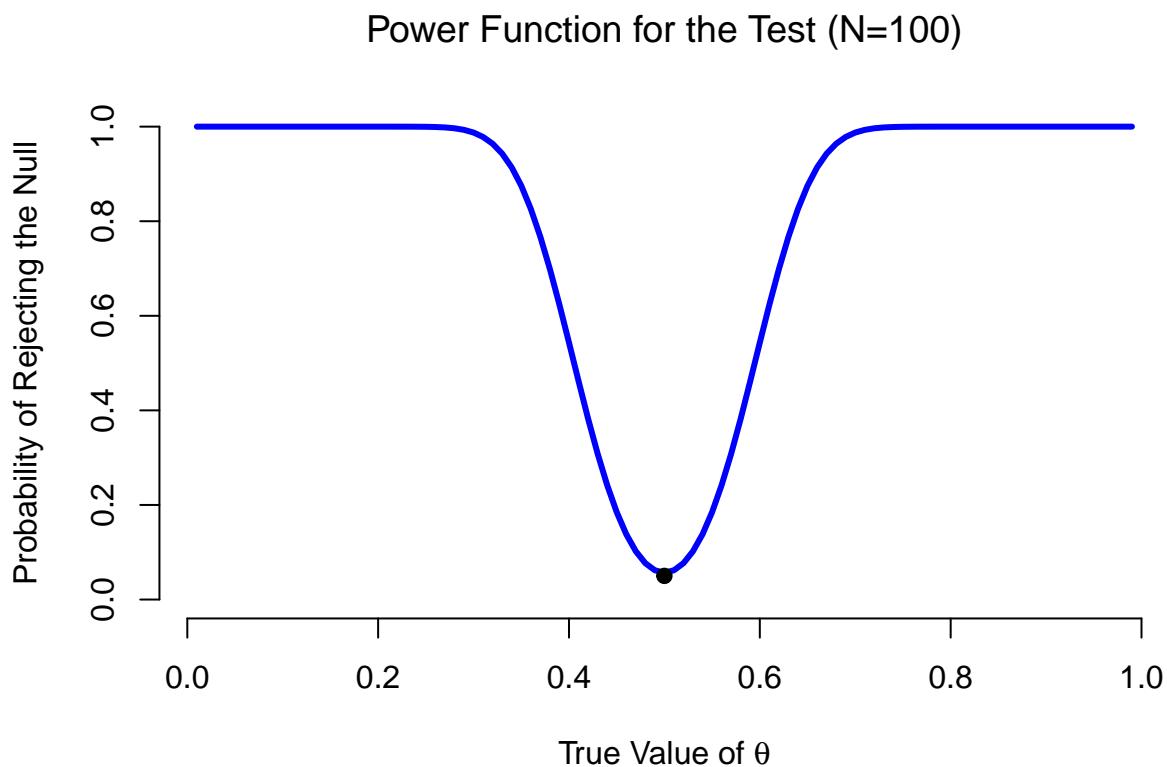


Figure 11.6: The probability that we will reject the null hypothesis, plotted as a function of the true value of θ . Obviously, the test is more powerful (greater chance of correct rejection) if the true value of θ is very different from the value that the null hypothesis specifies (i.e., $\theta = .5$). Notice that when θ actually is equal to .5 (plotted as a black dot), the null hypothesis is in fact true: rejecting the null hypothesis in this instance would be a Type I error.

I've calculated the expected probability of rejecting the null hypothesis for all values of θ , and plotted it in Figure 11.6. This plot describes what is usually called the ***power function*** of the test. It's a nice summary of how good the test is, because it actually tells you the power ($1 - \beta$) for all possible values of θ . As you can see, when the true value of θ is very close to 0.5, the power of the test drops very sharply, but when it is further away, the power is large.

11.8.2 Effect size

Since all models are wrong the scientist must be alert to what is importantly wrong. It is inappropriate to be concerned with mice when there are tigers abroad

– George Box 1976

The plot shown in Figure 11.6 captures a fairly basic point about hypothesis testing. If the true state of the world is very different from what the null hypothesis predicts, then your power will be very high; but if the true state of the world is similar to the null (but not identical) then the power of the test is going to be very low. Therefore, it's useful to be able to have some way of quantifying how "similar" the true state of the world is to the null hypothesis. A statistic that does this is called a measure of ***effect size*** (e.g. Cohen, 1988; Ellis, 2010). Effect size is defined slightly differently in different contexts,¹⁰ (and so this section just talks in general terms) but the qualitative idea that it tries to capture is always the same: how big is the difference between the *true* population parameters, and the parameter values that are assumed by the null hypothesis? In our ESP example, if we let $\theta_0 = 0.5$ denote the value assumed by the null hypothesis, and let θ denote the true value, then a simple measure of effect size could be something like the difference between the true value and null (i.e., $\theta - \theta_0$), or possibly just the magnitude of this difference, $\text{abs}(\theta - \theta_0)$.

	big effect size	small effect size
significant result	difference is real, and of practical importance	difference is real, but might not be interesting
non-significant result	no effect observed	no effect observed

Why calculate effect size? Let's assume that you've run your experiment, collected the data, and gotten a significant effect when you ran your hypothesis test. Isn't it enough just to say that you've gotten a significant effect? Surely that's the *point* of hypothesis testing? Well, sort of. Yes, the point of doing a hypothesis test is to try to demonstrate that the null hypothesis is wrong, but that's hardly the only thing we're interested in. If the null hypothesis claimed that $\theta = .5$, and we show that it's wrong, we've only really told half of the story. Rejecting the null hypothesis implies that we believe that $\theta \neq .5$, but there's a big difference between $\theta = .51$ and $\theta = .8$. If we find that $\theta = .8$, then not only have we found that the null hypothesis is wrong, it appears to be *very* wrong. On the other hand, suppose we've successfully rejected the null hypothesis, but it looks like the true value of θ is only $.51$ (this would only be possible with a large study). Sure, the null hypothesis is wrong, but it's not at all clear that we actually *care*, because the effect size is so small. In the context of my ESP study we might still care, since any demonstration of real psychic powers would actually be pretty cool¹¹, but in other contexts a 1% difference isn't very interesting, even if it is a real difference. For instance, suppose we're looking at differences in high school exam scores between males and females, and it turns out that the female scores are 1% higher on average than the males. If I've got data from thousands of students, then this difference will almost certainly be *statistically significant*, but regardless of how small the p value is it's just not very interesting. You'd hardly want to go around proclaiming a crisis in boys education on the basis of such a tiny difference would you? It's for this reason that it is becoming more standard (slowly, but surely) to report some kind of standard measure of effect size along with the results of the hypothesis test. The hypothesis test itself tells you whether you should believe that the effect you have observed is real (i.e., not just due to chance); the effect size tells you whether or not you should care.

¹⁰There's an R package called `compute.es` that can be used for calculating a very broad range of effect size measures; but for the purposes of the current book we won't need it: all of the effect size measures that I'll talk about here have functions in the `lsr` package

¹¹Although in practice a very small effect size is worrying, because even very minor methodological flaws might be responsible for the effect; and in practice no experiment is perfect, so there are always methodological issues to worry about.

11.8.3 Increasing the power of your study

Not surprisingly, scientists are fairly obsessed with maximising the power of their experiments. We want our experiments to work, and so we want to maximise the chance of rejecting the null hypothesis if it is false (and of course we usually want to believe that it is false!) As we've seen, one factor that influences power is the effect size. So the first thing you can do to increase your power is to increase the effect size. In practice, what this means is that you want to design your study in such a way that the effect size gets magnified. For instance, in my ESP study I might believe that psychic powers work best in a quiet, darkened room; with fewer distractions to cloud the mind. Therefore I would try to conduct my experiments in just such an environment: if I can strengthen people's ESP abilities somehow, then the true value of θ will go up¹² and therefore my effect size will be larger. In short, clever experimental design is one way to boost power; because it can alter the effect size.

Unfortunately, it's often the case that even with the best of experimental designs you may have only a small effect. Perhaps, for example, ESP really does exist, but even under the best of conditions it's very very weak. Under those circumstances, your best bet for increasing power is to increase the sample size. In general, the more observations that you have available, the more likely it is that you can discriminate between two hypotheses. If I ran my ESP experiment with 10 participants, and 7 of them correctly guessed the colour of the hidden card, you wouldn't be terribly impressed. But if I ran it with 10,000 participants and 7,000 of them got the answer right, you would be much more likely to think I had discovered something. In other words, power increases with the sample size. This is illustrated in Figure 11.7, which shows the power of the test for a true parameter of $\theta = 0.7$, for all sample sizes N from 1 to 100, where I'm assuming that the null hypothesis predicts that $\theta_0 = 0.5$.

```
## [1] 0.00000000 0.00000000 0.00000000 0.00000000 0.00000000 0.11837800
## [7] 0.08257300 0.05771362 0.19643626 0.14945203 0.11303734 0.25302172
## [13] 0.20255096 0.16086106 0.29695959 0.24588947 0.38879291 0.33269435
## [19] 0.28223844 0.41641377 0.36272868 0.31341925 0.43996501 0.38859619
## [25] 0.51186665 0.46049782 0.41129777 0.52752694 0.47870819 0.58881596
## [31] 0.54162450 0.49507894 0.59933871 0.55446069 0.65155826 0.60907715
## [37] 0.69828554 0.65867614 0.61815357 0.70325017 0.66542910 0.74296156
## [43] 0.70807163 0.77808343 0.74621569 0.71275488 0.78009449 0.74946571
## [49] 0.81000236 0.78219322 0.83626633 0.81119597 0.78435605 0.83676444
## [55] 0.81250680 0.85920268 0.83741123 0.87881491 0.85934395 0.83818214
## [61] 0.87858194 0.85962510 0.89539581 0.87849413 0.91004390 0.89503851
## [67] 0.92276845 0.90949768 0.89480727 0.92209753 0.90907263 0.93304809
## [73] 0.92153987 0.94254237 0.93240638 0.92108426 0.94185449 0.93185881
## [79] 0.95005094 0.94125189 0.95714694 0.94942195 0.96327866 0.95651332
## [85] 0.94886329 0.96265653 0.95594208 0.96796884 0.96208909 0.97255504
## [91] 0.96741721 0.97650832 0.97202770 0.97991117 0.97601093 0.97153910
## [97] 0.97944717 0.97554675 0.98240749 0.97901142
```

Because power is important, whenever you're contemplating running an experiment it would be pretty useful to know how much power you're likely to have. It's never possible to know for sure, since you can't possibly know what your effect size is. However, it's often (well, sometimes) possible to guess how big it should be. If so, you can guess what sample size you need! This idea is called **power analysis**, and if it's feasible to do it, then it's very helpful, since it can tell you something about whether you have enough time or money to be able to run the experiment successfully. It's increasingly common to see people arguing that power analysis should be a required part of experimental design, so it's worth knowing about. I don't discuss power analysis in this book, however. This is partly for a boring reason and partly for a substantive one. The boring reason is that I haven't had time to write about power analysis yet. The substantive one is that I'm still a little

¹²Notice that the true population parameter θ doesn't necessarily correspond to an immutable fact of nature. In this context θ is just the true probability that people would correctly guess the colour of the card in the other room. As such the population parameter can be influenced by all sorts of things. Of course, this is all on the assumption that ESP actually exists!

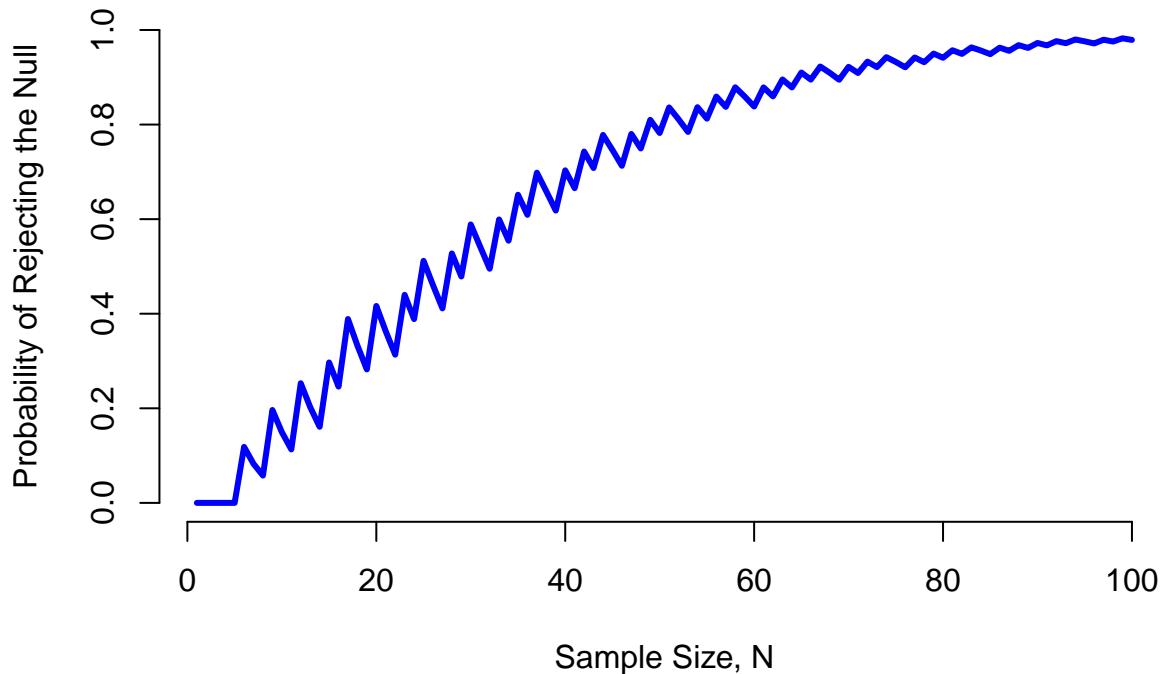


Figure 11.7: The power of our test, plotted as a function of the sample size N . In this case, the true value of θ is 0.7, but the null hypothesis is that $\theta = 0.5$. Overall, larger N means greater power. (The small zig-zags in this function occur because of some odd interactions between θ , α and the fact that the binomial distribution is discrete; it doesn't matter for any serious purpose)

suspicious of power analysis. Speaking as a researcher, I have very rarely found myself in a position to be able to do one – it’s either the case that (a) my experiment is a bit non-standard and I don’t know how to define effect size properly, (b) I literally have so little idea about what the effect size will be that I wouldn’t know how to interpret the answers. Not only that, after extensive conversations with someone who does stats consulting for a living (my wife, as it happens), I can’t help but notice that in practice the *only* time anyone ever asks her for a power analysis is when she’s helping someone write a grant application. In other words, the only time any scientist ever seems to want a power analysis in real life is when they’re being forced to do it by bureaucratic process. It’s not part of anyone’s day to day work. In short, I’ve always been of the view that while power is an important concept, power *analysis* is not as useful as people make it sound, except in the rare cases where (a) someone has figured out how to calculate power for your actual experimental design and (b) you have a pretty good idea what the effect size is likely to be. Maybe other people have had better experiences than me, but I’ve personally never been in a situation where both (a) and (b) were true. Maybe I’ll be convinced otherwise in the future, and probably a future version of this book would include a more detailed discussion of power analysis, but for now this is about as much as I’m comfortable saying about the topic.

11.9 Some issues to consider

What I’ve described to you in this chapter is the orthodox framework for null hypothesis significance testing (NHST). Understanding how NHST works is an absolute necessity, since it has been the dominant approach to inferential statistics ever since it came to prominence in the early 20th century. It’s what the vast majority of working scientists rely on for their data analysis, so even if you hate it you need to know it. However, the approach is not without problems. There are a number of quirks in the framework, historical oddities in how it came to be, theoretical disputes over whether or not the framework is right, and a lot of practical traps for the unwary. I’m not going to go into a lot of detail on this topic, but I think it’s worth briefly discussing a few of these issues.

11.9.1 Neyman versus Fisher

The first thing you should be aware of is that orthodox NHST is actually a mash-up of two rather different approaches to hypothesis testing, one proposed by Sir Ronald Fisher and the other proposed by Jerzy Neyman (for a historical summary see Lehmann, 2011). The history is messy because Fisher and Neyman were real people whose opinions changed over time, and at no point did either of them offer “the definitive statement” of how we should interpret their work many decades later. That said, here’s a quick summary of what I take these two approaches to be.

First, let’s talk about Fisher’s approach. As far as I can tell, Fisher assumed that you only had the one hypothesis (the null), and what you want to do is find out if the null hypothesis is inconsistent with the data. From his perspective, what you should do is check to see if the data are “sufficiently unlikely” according to the null. In fact, if you remember back to our earlier discussion, that’s how Fisher defines the *p*-value. According to Fisher, if the null hypothesis provided a very poor account of the data, you could safely reject it. But, since you don’t have any other hypotheses to compare it to, there’s no way of “accepting the alternative” because you don’t necessarily have an explicitly stated alternative. That’s more or less all that there was to it.

In contrast, Neyman thought that the point of hypothesis testing was as a guide to action, and his approach was somewhat more formal than Fisher’s. His view was that there are multiple things that you could *do* (accept the null or accept the alternative) and the point of the test was to tell you which one the data support. From this perspective, it is critical to specify your alternative hypothesis properly. If you don’t know what the alternative hypothesis is, then you don’t know how powerful the test is, or even which action makes sense. His framework genuinely requires a competition between different hypotheses. For Neyman, the *p* value didn’t directly measure the probability of the data (or data more extreme) under the null, it was

more of an abstract description about which “possible tests” were telling you to accept the null, and which “possible tests” were telling you to accept the alternative.

As you can see, what we have today is an odd mishmash of the two. We talk about having both a null hypothesis and an alternative (Neyman), but usually¹³ define the p value in terms of extreme data (Fisher), but we still have α values (Neyman). Some of the statistical tests have explicitly specified alternatives (Neyman) but others are quite vague about it (Fisher). And, according to some people at least, we’re not allowed to talk about accepting the alternative (Fisher). It’s a mess: but I hope this at least explains why it’s a mess.

11.9.2 Bayesians versus frequentists

Earlier on in this chapter I was quite emphatic about the fact that you *cannot* interpret the p value as the probability that the null hypothesis is true. NHST is fundamentally a frequentist tool (see Chapter 9) and as such it does not allow you to assign probabilities to hypotheses: the null hypothesis is either true or it is not. The Bayesian approach to statistics interprets probability as a degree of belief, so it’s totally okay to say that there is a 10% chance that the null hypothesis is true: that’s just a reflection of the degree of confidence that you have in this hypothesis. You aren’t allowed to do this within the frequentist approach. Remember, if you’re a frequentist, a probability can only be defined in terms of what happens after a large number of independent replications (i.e., a long run frequency). If this is your interpretation of probability, talking about the “probability” that the null hypothesis is true is complete gibberish: a null hypothesis is either true or it is false. There’s no way you can talk about a long run frequency for this statement. To talk about “the probability of the null hypothesis” is as meaningless as “the colour of freedom”. It doesn’t have one!

Most importantly, this *isn’t* a purely ideological matter. If you decide that you are a Bayesian and that you’re okay with making probability statements about hypotheses, you have to follow the Bayesian rules for calculating those probabilities. I’ll talk more about this in Chapter ??, but for now what I want to point out to you is the p value is a *terrible* approximation to the probability that H_0 is true. If what you want to know is the probability of the null, then the p value is not what you’re looking for!

11.9.3 Traps

As you can see, the theory behind hypothesis testing is a mess, and even now there are arguments in statistics about how it “should” work. However, disagreements among statisticians are not our real concern here. Our real concern is practical data analysis. And while the “orthodox” approach to null hypothesis significance testing has many drawbacks, even an unrepentant Bayesian like myself would agree that they can be useful if used responsibly. Most of the time they give sensible answers, and you can use them to learn interesting things. Setting aside the various ideologies and historical confusions that we’ve discussed, the fact remains that the biggest danger in all of statistics is *thoughtlessness*. I don’t mean stupidity, here: I literally mean thoughtlessness. The rush to interpret a result without spending time thinking through what each test actually says about the data, and checking whether that’s consistent with how you’ve interpreted it. That’s where the biggest trap lies.

To give an example of this, consider the following example (see Gelman and Stern, 2006). Suppose I’m running my ESP study, and I’ve decided to analyse the data separately for the male participants and the female participants. Of the male participants, 33 out of 50 guessed the colour of the card correctly. This is a significant effect ($p = .03$). Of the female participants, 29 out of 50 guessed correctly. This is not a significant effect ($p = .32$). Upon observing this, it is extremely tempting for people to start wondering why there is a difference between males and females in terms of their psychic abilities. However, this is wrong. If you think about it, we haven’t *actually* run a test that explicitly compares males to females. All we have done is compare males to chance (binomial test was significant) and compared females to chance (binomial test was

¹³Although this book describes both Neyman’s and Fisher’s definition of the p value, most don’t. Most introductory textbooks will only give you the Fisher version.

non significant). If we want to argue that there is a real difference between the males and the females, we should probably run a test of the null hypothesis that there is no difference! We can do that using a different hypothesis test,¹⁴ but when we do that it turns out that we have no evidence that males and females are significantly different ($p = .54$). Now do you think that there's anything fundamentally different between the two groups? Of course not. What's happened here is that the data from both groups (male and female) are pretty borderline: by pure chance, one of them happened to end up on the magic side of the $p = .05$ line, and the other one didn't. That doesn't actually imply that males and females are different. This mistake is so common that you should always be wary of it: the difference between significant and not-significant is *not* evidence of a real difference – if you want to say that there's a difference between two groups, then you have to test for that difference!

The example above is just that: an example. I've singled it out because it's such a common one, but the bigger picture is that data analysis can be tricky to get right. Think about *what* it is you want to test, *why* you want to test it, and whether or not the answers that your test gives could possibly make any sense in the real world.

11.10 Summary

Null hypothesis testing is one of the most ubiquitous elements to statistical theory. The vast majority of scientific papers report the results of some hypothesis test or another. As a consequence it is almost impossible to get by in science without having at least a cursory understanding of what a p -value means, making this one of the most important chapters in the book. As usual, I'll end the chapter with a quick recap of the key ideas that we've talked about:

- Research hypotheses and statistical hypotheses. Null and alternative hypotheses. (Section 11.1).
- Type 1 and Type 2 errors (Section 11.2)
- Test statistics and sampling distributions (Section 11.3)
- Hypothesis testing as a decision making process (Section 11.4)
- p -values as “soft” decisions (Section 11.5)
- Writing up the results of a hypothesis test (Section 11.6)
- Effect size and power (Section 11.8)
- A few issues to consider regarding hypothesis testing (Section 11.9)

Later in the book, in Chapter ??, I'll revisit the theory of null hypothesis tests from a Bayesian perspective, and introduce a number of new tools that you can use if you aren't particularly fond of the orthodox approach. But for now, though, we're done with the abstract statistical theory, and we can start discussing specific data analysis tools.

¹⁴In this case, the Pearson chi-square test of independence (Chapter ??; `chisq.test()` in R) is what we use; see also the `prop.test()` function.

Bibliography

- Adair, G. (1984). The hawthorne effect: A reconsideration of the methodological artifact. *Journal of Applied Psychology*, 69:334–345.
- Bickel, P. J., Hammel, E. A., and O’Connell, J. W. (1975). Sex bias in graduate admissions: Data from Berkeley. *Science*, 187:398–404.
- Braun, J. and Murdoch, D. J. (2007). *A first course in statistical programming with R*. Cambridge University Press Cambridge.
- Campbell, D. T. and Stanley, J. C. (1963). *Experimental and Quasi-Experimental Designs for Research*. Houghton Mifflin, Boston, MA.
- Cohen, J. (1988). *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum, 2nd edition.
- Ellis, P. D. (2010). *The Essential Guide to Effect Sizes: Statistical Power, Meta-Analysis, and the Interpretation of Research Results*. Cambridge University Press, Cambridge, UK.
- Ellman, M. (2002). Soviet repression statistics: some comments. *Europe-Asia Studies*, 54(7):1151–1172.
- Evans, J. S. B. T., Barston, J. L., and Pollard, P. (1983). On the conflict between logic and belief in syllogistic reasoning. *Memory and Cognition*, 11:295–306.
- Evans, M., Hastings, N., and Peacock, B. (2011). *Statistical Distributions* (3rd ed). Wiley.
- Fisher, R. A. (1922). On the mathematical foundation of theoretical statistics. *Philosophical Transactions of the Royal Society A*, 222:309–368.
- Fox, J. and Weisberg, S. (2011). *An R Companion to Applied Regression*. Sage, Los Angeles, 2nd edition.
- Gelman, A. and Stern, H. (2006). The difference between “significant” and “not significant” is not itself statistically significant. *The American Statistician*, 60:328–331.
- Hothsall, D. (2004). *History of Psychology*. McGraw-Hill.
- Ioannidis, J. P. A. (2005). Why most published research findings are false. *PLoS Med*, 2(8):697–701.
- Kahneman, D. and Tversky, A. (1973). On the psychology of prediction. *Psychological Review*, 80:237–251.
- Keynes, J. M. (1923). *A Tract on Monetary Reform*. Macmillan and Company, London.
- Kühberger, A., Fritz, A., and Scherndl, T. (2014). Publication bias in psychology: A diagnosis based on the correlation between effect size and sample size. *Public Library of Science One*, 9:1–8.
- Lehmann, E. L. (2011). *Fisher, Neyman, and the Creation of Classical Statistics*. Springer.
- Matloff, N. and Matloff, N. S. (2011). *The art of R programming: A tour of statistical software design*. No Starch Press.
- Meehl, P. H. (1967). Theory testing in psychology and physics: A methodological paradox. *Philosophy of Science*, 34:103–115.

- Pfungst, O. (1911). *Clever Hans (The horse of Mr. von Osten): A contribution to experimental animal and human psychology.* Henry Holt.
- R Core Team (2013). *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria.
- Rosenthal, R. (1966). *Experimenter effects in behavioral research.* Appleton.
- Spector, P. (2008). *Data Manipulation with R.* Springer, New York, NY.
- Stevens, S. S. (1946). On the theory of scales of measurement. *Science*, 103:677–680.
- Stigler, S. M. (1986). *The History of Statistics.* Harvard University Press, Cambridge, MA.
- Teator, P. (2011). *R Cookbook.* O'Reilly, Sebastopol, CA.