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
A social science"

– W.H. Auden[1]

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]

[1]The 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

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

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.

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, J. S. B. T. Evans, Barston, and Pollard (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

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

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.

1.2 The cautionary tale of Simpson's paradox

The following is a true story. In 1973, the University of California, Berkeley got into some trouble over its admissions of students into postgraduate courses. Specifically, the thing that caused the problem was that the gender breakdown of their admissions looked like this...

	Number of applicants	Percent admitted
Males	8442	44%
Females	4321	35%

...and they got sued. 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 females, you'd probably think that I was either crazy or sexist.

Oddly, it's actually sort of true ... after Berkeley got sued, people started looking *very* carefully at the admissions data (Bickel, Hammel, & O'Connell, 1975). And remarkably, 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. The table below shows the admission figures for the six largest departments (with the names of the departments removed for privacy reasons):

	Males		Females	
Department	Applicants	Percent admitted	Applicants	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%

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

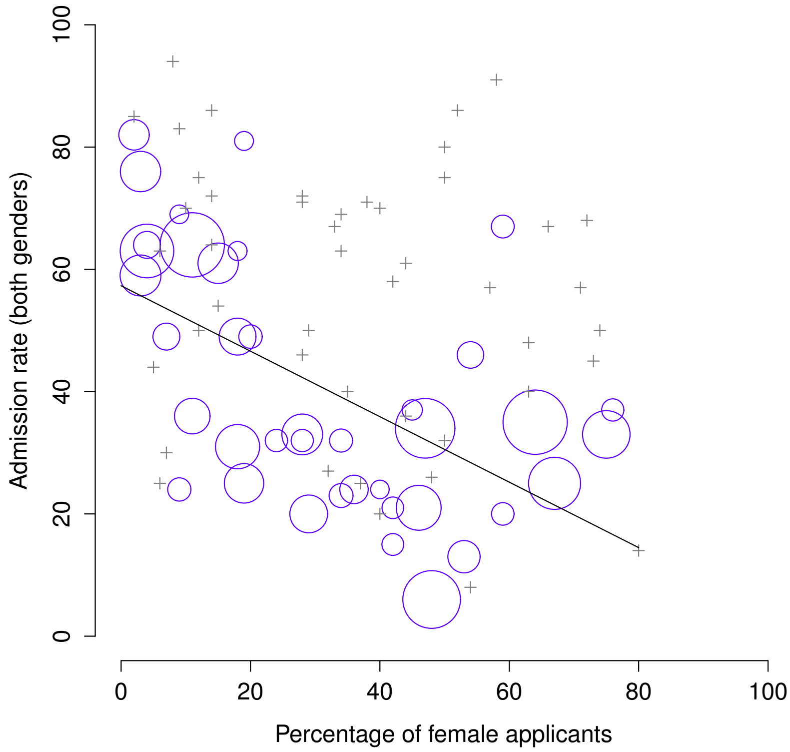


Figure 1.1: The Berkeley 1973 college admissions data. This figure plots the admission rate for the 85 departments that had at least one female applicant, as a function of the percentage of applicants that were female. The plot is a redrawing of Figure 1 from Bickel et al. (1975). Circles plot departments 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.

....

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 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 purely legal perspective, that puts the university in the clear. Postgraduate admissions are determined at the level of the individual department (and there are very 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 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 object 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.[4]

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

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:

  - 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.

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 Fisher[1]

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.*

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 *Klinfelter'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:

- 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 to 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 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
(1) Temperatures are rising, because of human activity
(4) Temperatures are not rising
(2) 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.[2]

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

[2] Actually, 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 it's 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 2.1: 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		✓
ordinal		✓
interval	✓	✓
ratio	✓	✓

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 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 2.1 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 it's 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 a 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?
- **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.

Table 2.2: 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

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).[3] This is summarised in Table 2.2.

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 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

[3] Annoyingly, 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.

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 it's 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" an 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 variable[4] 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

[4]The 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.

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, 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 & 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 prophesy.

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 a 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.[5] 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 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,

[5] Some 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.

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 Finnegan's 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 Finnegan's Wake causes insanity[6]. 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

[6]Clearly, the real effect is that only insane people would even try to read Finnegan's Wake.

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, Fritz, and Scherndl (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* (Section 2.1). 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* (Section 2.2). 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* (Section 2.3). 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* (Section 2.4). 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* (Section 2.5). 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* (Section 2.6). 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.

Part II.

An introduction to R

3. Getting started with R

Robots are nice to work with.
–Roger Zelazny[1]

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

[1]Source: *Dismal Light* (1968).

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.

- 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]

[2] Although 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.

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 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 way[4] 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,

[3] If 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.

[4] Tip 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.

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:

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 current version as I type this is 0.98.501, which means that it's in beta testing (the official release is version 1.0). Even so, I think that the beta version of Rstudio provides a better user experience than anything else I've tried: it really is the best option available in my opinion. 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

[5] This 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.

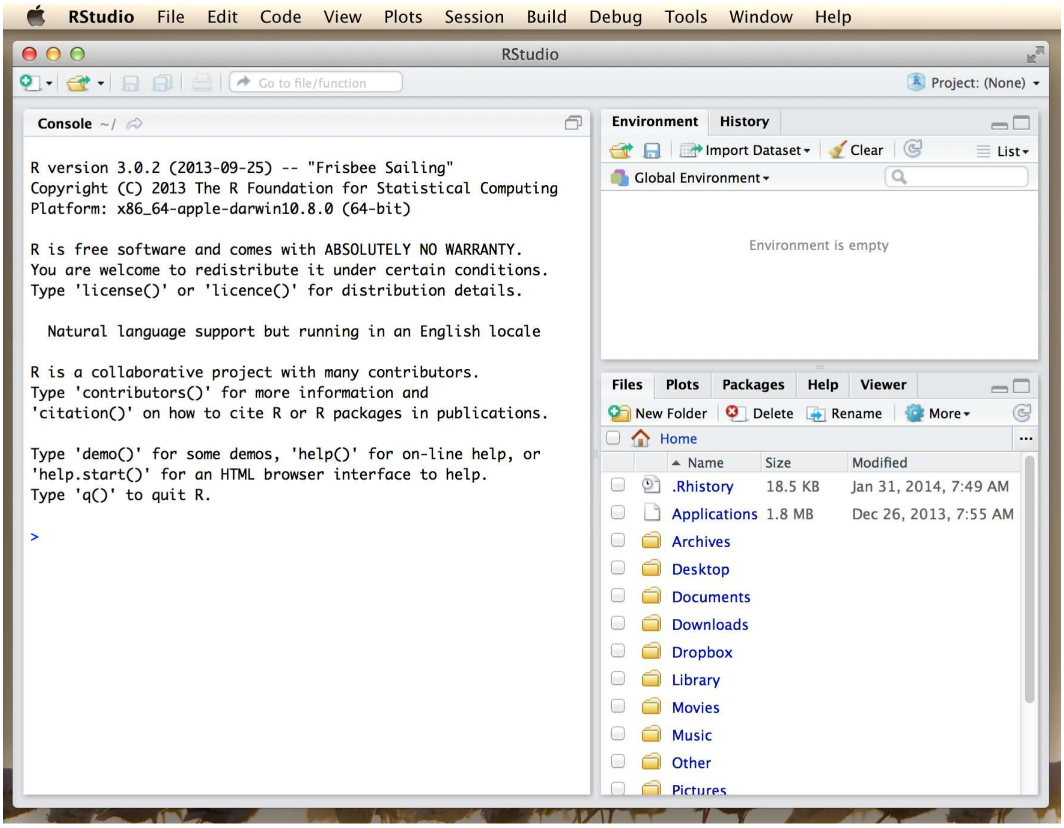


Figure 3.1: An R session in progress running through Rstudio. The picture shows Rstudio running on a Mac, but the Windows interface is almost identical.

....

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.

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.

[6] Seriously. 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.

3.2.1 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.

3.2.2 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 (2013). R: A language and environment
  for statistical computing. R Foundation for
  Statistical Computing, Vienna, Austria. URL
  http://www.R-project.org/.

BLAH BLAH BLAH

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). Obviously, the `BLAH BLAH BLAH` part isn't actually part of what R prints out: when you see that it means that I've chopped out some parts of the output that I don't think are very interesting or relevant. I'll do that a lot. Anyway, getting back to my original point, 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: unexpected symbol in "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.

3.2.3 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:

```
> 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
```

[7] If 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. There are others as well, which I'll discuss in Chapter 7.

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. 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:

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

```
> 57 * 61
[1] 3477
```

So that's handy.

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. 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
```

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.[9] 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.

[9] For 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.[10] 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 enter[11]

```
> 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.[12] 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".[13]

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:

[10] If 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.

[11] As we'll discuss later, by doing this we are implicitly using the `print()` function.

[12] Actually, 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 the `assign()` function, and the `<<-` and `->>` operators. However, we won't be using these at all in this book.

[13] A 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 rules[14] illustrates:

- 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?`.

[14] Actually, 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 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`, `NA_integer_`, `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,[15] 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

[15] For 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`.

on, to do actual statistics), you're going to need to start using **functions**.[16] 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

$$\sqrt{25} = 5$$

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 255 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 255 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 $\text{abs}(x)$. 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:

[16] A 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 `` `+`() ``. Thus `10+20` is equivalent to the function call `` `+`(20, 30) ``. Not surprisingly, no-one ever uses this version. Because that would be stupid.

```
> 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 root[17] 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

[17] A 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.

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:

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

Notice that this is kind of similar in spirit to variable assignment (Section 3.4), 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,[18] 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 )
> round( x = 3.1415, digits = 2 )
> round( digits = 2, x = 3.1415 )
```

but this one does not...

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

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.

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

[18] The 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.

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.[19]

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.

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 3.3. 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 Figure 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.

---

[19] For 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 running R in the terminal then you're already familiar with using tab autocomplete.

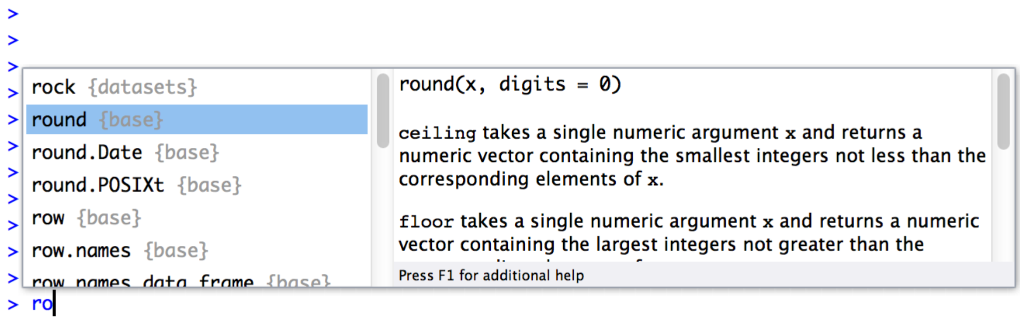


Figure 3.2: Start typing the name of a function or a variable, and hit the "tab" key. Rstudio brings up a little dialog box like this one that lets you select the one you want, and even prints out a little information about it.

....

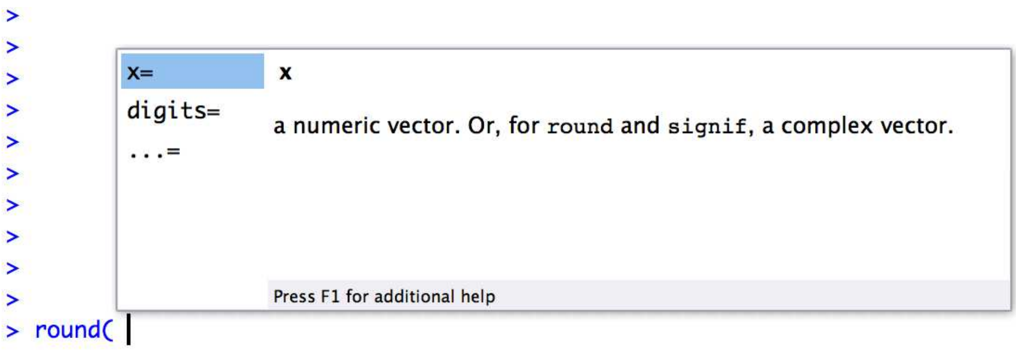


Figure 3.3: If you've typed the name of a function already along with the left parenthesis and then hit the "tab" key, Rstudio brings up a different window to the one shown in Figure 3.2. This one lists all the arguments to the function on the left, and information about each argument on the right.

....

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 escape[20] 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 3.4. 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

[20]Incidentally, that always works: if you've started typing a command and you want to clear it and start again, hit escape.

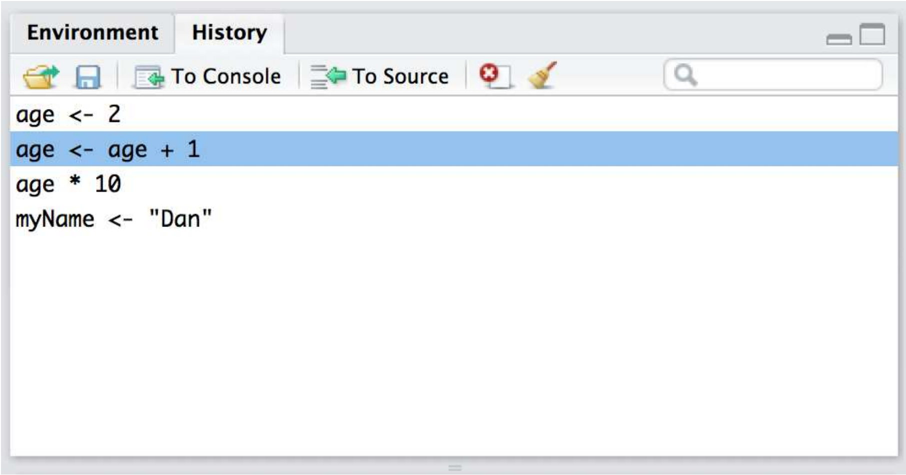


Figure 3.4: The history panel is located in the top right hand side of the Rstudio window. Click on the word "History" and it displays this panel.

....

Console" button).[21]

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:[22]

[21] Another 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.

[22] Notice 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.

```
> sales.by.month <- c(0, 100, 200, 50, 0, 0, 0, 0, 0, 0, 0, 0)
> sales.by.month
 [1]   0 100 200  50   0   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
 [9]   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!

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   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    0    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]  0.00000 25.00000 45.16129 11.66667  0.00000  0.00000  0.00000  0.00000  0.00000
[10]  0.00000  0.00000  0.00000
```

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 `hello` 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`.[23] 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")
> months
 [1] "January"   "February"  "March"     "April"     "May"       "June"
 [7] "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"
```

[23]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.

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 humans[24] but it's definitely

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

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
```

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
```

[25] Note 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`

....

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`:

```
> 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
```

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:

Table 3.3: Some more logical operators.

operation	operator	example input	answer
not	`!`	`!(1==1)`	`FALSE`
or	`|`	`(1==1) | (2==3)`	`TRUE`
and	`&`	`(1==1) & (2==3)`	`FALSE`

```
> (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 3.7) 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

[26] A 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.

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: 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:

```
> 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 3.7. 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 FALSE 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 FALSE FALSE
```

[27] Warning! `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`.

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] FALSE
> "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   0   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   0   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 `TRUE`s and `FALSE`s 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
 [1] "high" "high" "low"  "out"  "out"  "high" "high" "high" "high" "high" "high"
[12] "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]

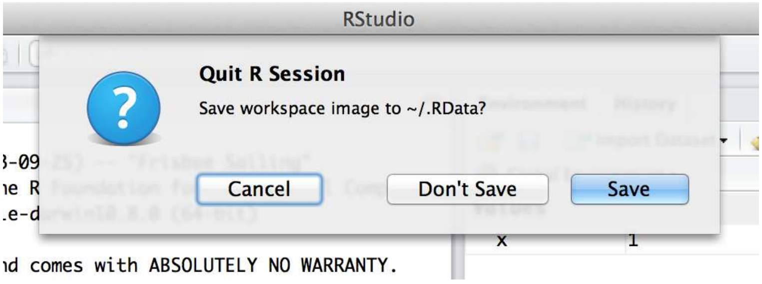


Figure 3.5: The dialog box that shows up when you try to close Rstudio.

3.11 Quitting R

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 addition 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

[28] Well, 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.

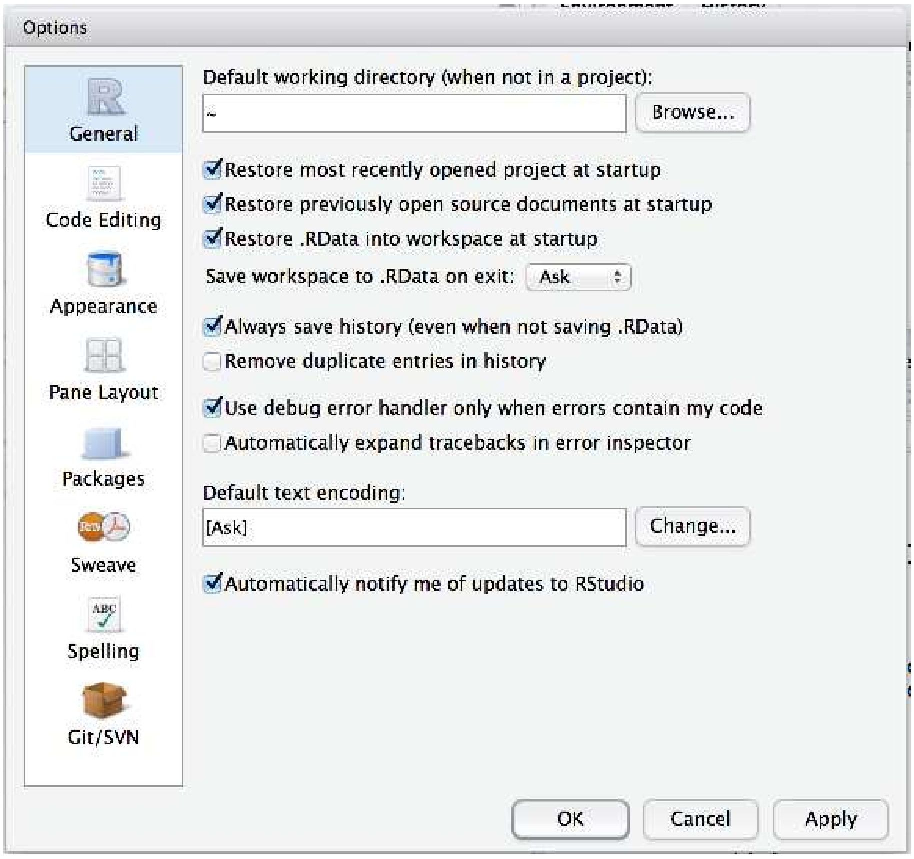


Figure 3.6: The options window in Rstudio. On a Mac, you can open this window by going to the "Rstudio" menu and selecting "Preferences". On a Windows machine you go to the "Tools" menu and select "Global Options".

....

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".

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 (Section 3.1).

- *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 3.2), and in doing so learned how to perform basic calculations using the arithmetic operators `+`, `-`, `*`, `/` and `^`. (Section 3.3)
- *Introduction to functions.* We saw several different functions, three that are used to perform numeric calculations (`sqrt()`, `abs()`, `round()`; Section 3.5), one that applies to text (`nchar()`; Section 3.8.1), and one that works on any variable (`length()`; Section 3.7.5). In doing so, we talked a bit about how argument names work, and learned about default values for arguments. (Section 3.5.1)
- *Introduction to variables.* We learned the basic idea behind variables, and how to assign values to variables using the assignment operator `<-` (Section 3.4). We also learned how to create vectors using the combine function `c()`. (Section 3.7)
- *Data types.* Learned the distinction between numeric, character and logical data; including the basics of how to enter and use each of them. (Sections 3.4 to 3.9)
- *Logical operations.* Learned how to use the logical operators `==`, `!=`, `<`, `>`, `<=`, `=>`, `!`, `&` and `|`. (Section 3.9). 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...

4. Additional R concepts

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:

```
> seeker <- 3.1415
> lover <- 2.7183
> keeper <- seeker * lover
```

[1] Notice 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.

```
> print( keeper )
[1] 8.539539
```

So, from now on, you'll start seeing some # 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.

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

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

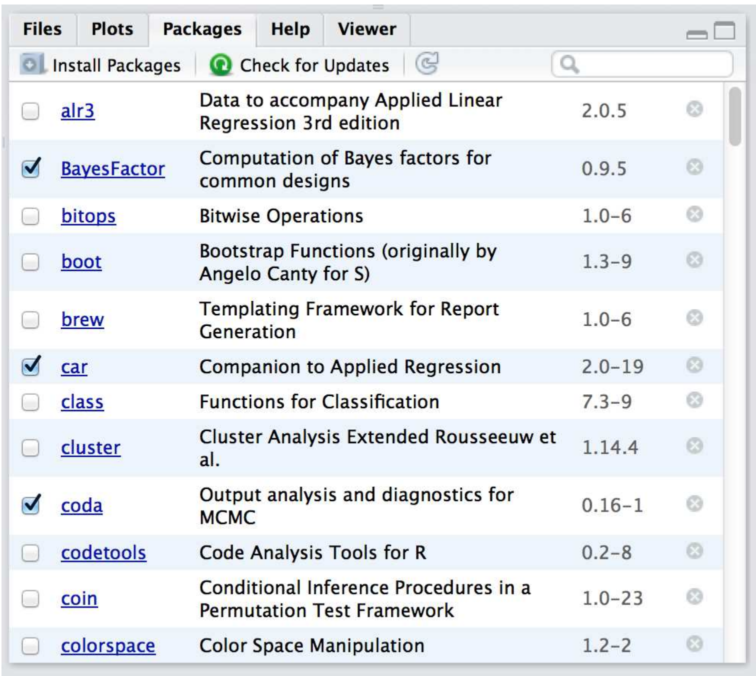


Figure 4.1: The packages panel.

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

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

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:

```
> 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)
```

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)
Attaching package: 'psych'

The following object is masked from 'package:car':

    logit
```

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

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()` command[7]

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 4.2a. 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 4.2b), 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/cl/thhsyrz53g73q0w1kb5z3l_80000gn/T//RtmpmQ9VT3/downloaded_packages
```

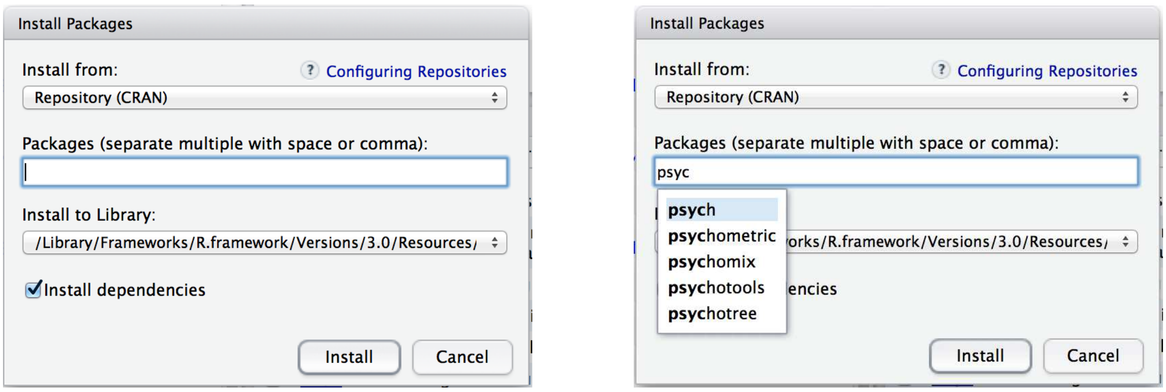
Despite the long and tedious response, all thar really means is "I've installed the psych package". I find it best to humour the talkative little automaton. I don't actually read any of this garbage, I just politely say "thanks" and go back to whatever I was doing.

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.

[6] Tip 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.

[7] It is not very difficult.



(a) (b)

Figure 4.2: The package installation dialog box in Rstudio (panel a). When you start typing, you'll see a dropdown menu suggest a list of possible packages that you might want to install (panel b)

. . . .

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 4.3. In this window, each row refers to a package that needs to be updated. You can to 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.

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.

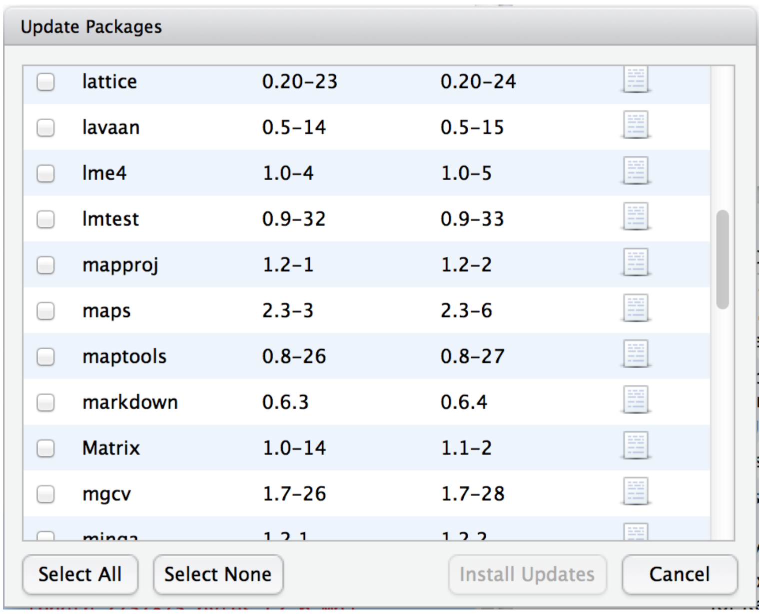


Figure 4.3: The Rstudio dialog box for updating packages.

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

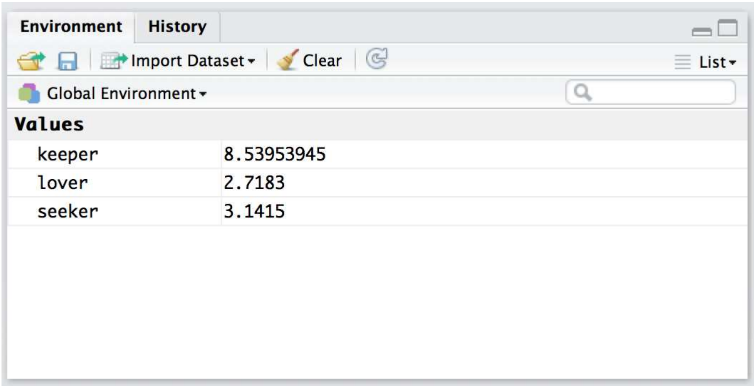


Figure 4.4: The Rstudio "Environment" panel shows you the contents of the workspace. The view shown above is the "list" view. To switch to the grid view, click on the menu item on the top right that currently reads "list". Select "grid" from the dropdown menu, and then it will switch to a view like the one shown in Figure 4.5.

....

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 4.4 and 4.5. If you're using the commmand line, then the `objects()` function may come in handy:

```
> objects()
[1] "keeper" "lover"  "seeker"
```

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 --
   keeper        numeric        1
   lover         numeric        1
   seeker        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

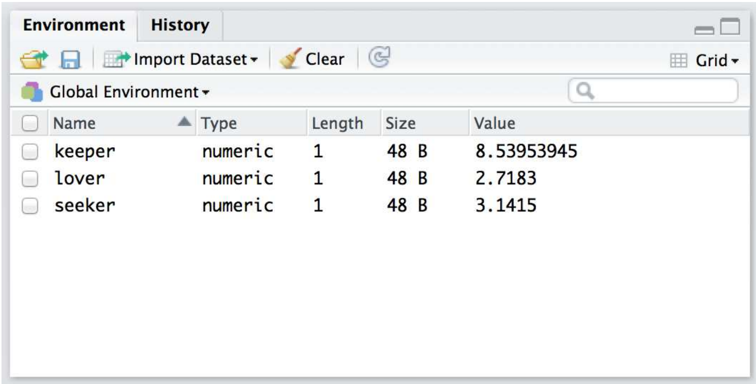


Figure 4.5: The Rstudio "Environment" panel shows you the contents of the workspace. Compare this "grid" view to the "list" view in Figure 4.4

....

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 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 4.5), 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

[8]This 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!

[9]Mind 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.

```
> who()
   -- Name --    -- Class --    -- Size --
   keeper        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 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 as 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 drive[10] 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\dan\Rbook\LSR.pdf
```

and what that *means* is that the book is called `LSR.pdf`, and it's in a folder called `Rbook` 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,

[10] Well, the partition, technically.

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\dan\Rbook`). The table below shows several addresses in relation to my current one:

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

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

```
~\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] "afl24.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" )
```

[11] One 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.

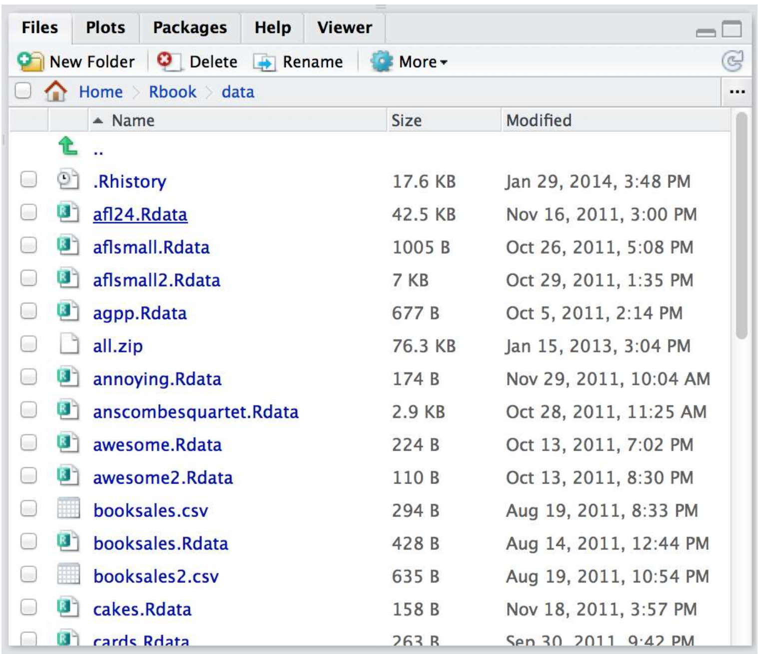


Figure 4.6: The "file panel" is the area shown in the lower right hand corner. It provides a very easy way to browse and navigate your computer using R. See main text for details.

....

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 4.6 – 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 4.6. 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

```
/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 (`.xls` 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.

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

[12] Notably those with `.rda`, `.Rd`, `.Rhistory`, `.rdb` and `.rdx` extensions

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 4.6, 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.

4.5.3 Importing data from CSV files using R

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 4.7, 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 it's capabilities in Section 7.9

[13] In 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.

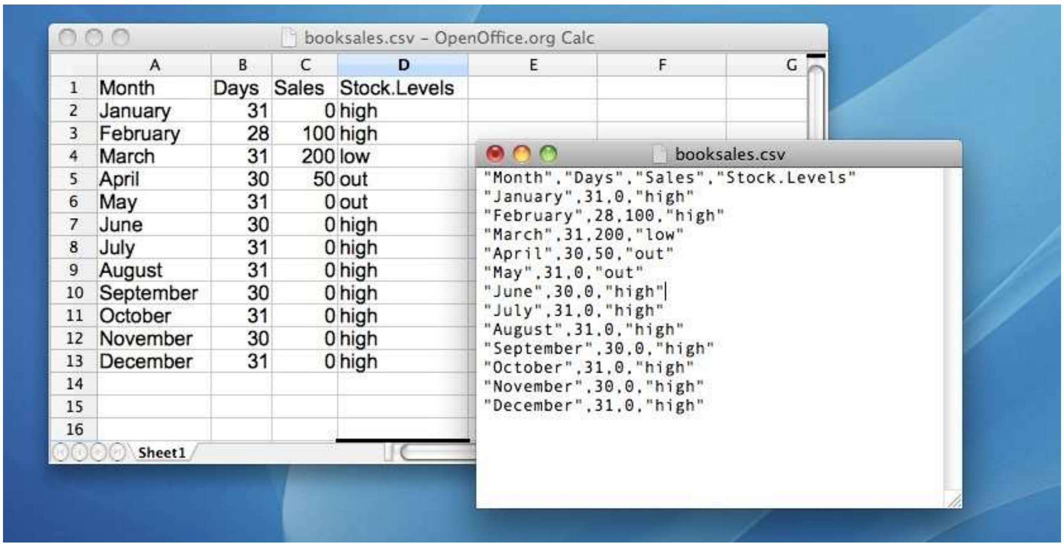


Figure 4.7: The `booksales.csv` data file. On the left, I've opened the file in using a spreadsheet program (OpenOffice), which shows that the file is basically a table. On the right, the same file is open in a standard text editor (the TextEdit program on a Mac), which shows how the file is formatted. The entries in the table are wrapped in quote marks and separated by commas.

....

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 a variable, which I imaginatively called `books`.[14] 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
```

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

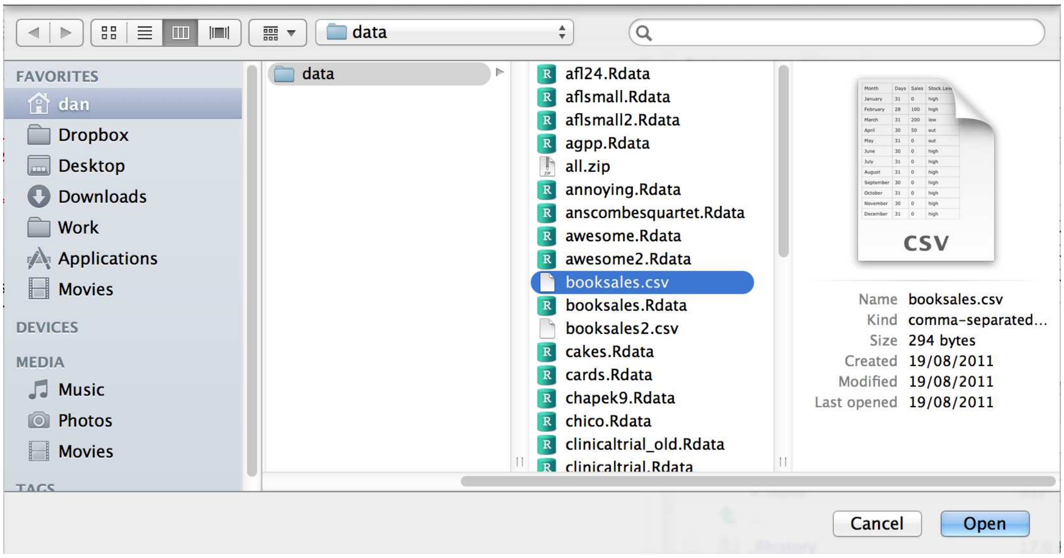


Figure 4.8: A dialog box on a Mac asking you to select the CSV file R should try to import. Mac users will recognise this immediately: it's the usual way in which a Mac asks you to find a file. Windows users won't see this: they'll see the usual explorer window that Windows always gives you when it wants you to select a file.

....

```
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 4.8. 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

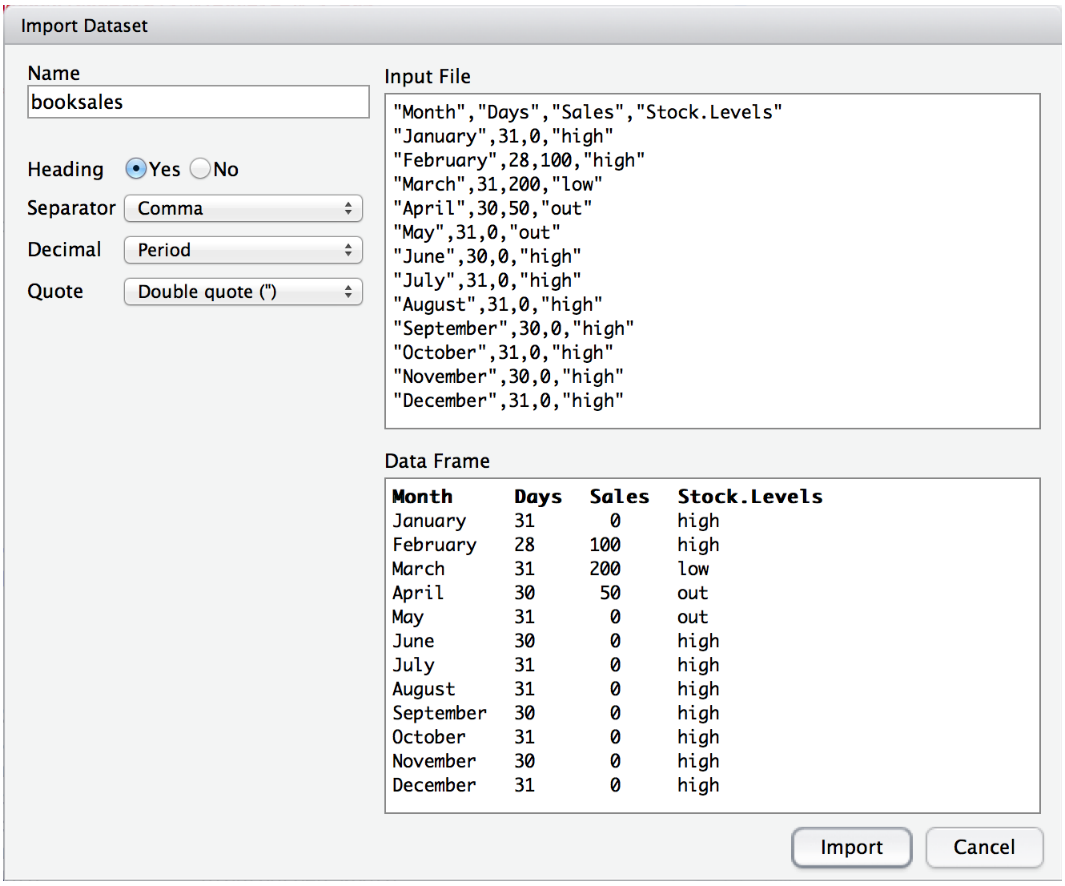


Figure 4.9: The Rstudio window for importing a CSV file into R.

that looks like the one in Figure 4.9.

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.

- 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 R

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 4.4 and 4.5) 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 4.8). 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.

[15] A 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.

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 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
```

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

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