Advanced Statistics Remix

David Schuster

2021-08-23

Contents

About this Book

This is a textbook for my advanced statistics course, first used in Fall 2021.

It is a remix of existing open source educational materials. I am contributing very little text. The primary source of content is Navarro (2018).

Attribution

Where authors are indicated throughout this text, the content has been copied verbatim with no more than minor editorial changes. Note that this differs from an APA style manuscript in which all verbatim text is typically quoted or blockquoted.

Navarro, D. (2018). Learning statistics with R: A tutorial for psychology students and other beginners (version 0.6). Retrieved from https://learningstatisticswithr.com

Crump, M. J. C., Navarro, D., & Suzuki, J. (2019, June 5). Answering Questions with Data (Textbook): Introductory Statistics for Psychology Students. https://doi.org/10.17605/OSF.IO/JZE52

Further, text copied from Navarro (2018) is from the Bookdown translation by Emily Kothe.

License

Chapter 1

Statistics for Research

1.1 Introduction

Text by David Schuster

Video: Applied Statistics

Statistics is a rich and diverse field with endless theories and application. To call this a "statistics course" may be too vague to be useful. Will we study the theory behind the statistics or study how statistics are used? Let's narrow our scope. This course is primarily concerned with statistical methods used by researchers. That is, researchers systematically (deliberately and consistently) gather evidence in order to generate new knowledge. Statistics provide an important tool to help researchers be more systematic in their discovery of new knowledge. Researchers are to statisticians as video game players are to video game designers. Most video game players enjoy playing games but don't necessarily care about how the game is constructed, coded, developed, and sold. Similarly, most researchers don't necessarily care about how mathematical theory supports statistical concepts; instead, they want to use statistics to answer their research questions. Unfortunately, unlike playing a video game, statistical methods provide very little feedback (and usually none at all) about whether or not they are being used correctly. Because of this, researchers do have to understand a bit about how statistical methods work.

You can summarize all of this by saying that you are studying **applied statistics**, the use of statistical methods to address research problems (Cohen). Throughout this course, we will emphasize the statistical knowledge needed to understand and produce research. When theory is introduced (we might think of theoretical statistics as the opposite of applied statistics), it will be included because it helps understanding of concepts we need as researchers.

Doing research is exciting and important because it's our best tool for solving big societal problems, discovering solutions, and separating fact from fiction. Because of this, many researchers are more fascinated by science and their field of study than they are about statistical methods. I have been teaching statistics for over ten years and can confirm that student attitudes about this topic vary widely. If you do not feel like you love statistics in this moment, that is a common feeling. At the same time, you should be aware that many professionals happily and confidently use statistics in their careers and/or daily lives without identifying as mathematicians. This is not to disparage the mathematical perspective, only to say research and mathematics are interesting in different ways. If you do not yet think studying statistics is useful or interesting, perhaps you will challenge your attitudes about math and statistics throughout this course. And if not, perhaps you will provide useful feedback to your instructor!

There are two broad situations where researchers need statistics. Researchers need statistics to:

1. Gather observations in a systematic way (**measurement**)
2. Summarize their observations (**descriptive statistics**)
3. Make conclusions about populations based on the observations (**inferential statistics**)

In the next sections, we will unpack these three functions.

1.2 Measurement

Text by David Schuster

Video: Measurement

The very beginning of statistics, and the most fundamental building block, is data. Data are what we get when we combine numbers and meaning. If I write down any number that comes to mind, I am generating numbers but not data (because there is no meaning). If I wonder about how many students attempt to cross the busy street outside of my office, I am starting to develop a research question (I would say there is meaning involved) but there are no numbers yet, so no data. If I go outside and count the number of pedestrians that cross the street in an hour, I am now gathering data.

Different kinds of data contain different kinds of information. I have already been simplifying my definitions by suggesting that data always involves numbers. If my research question is, "do students walk across the street?" and I go outside for an hour and observe them to do so, then I have gathered data. But, are these data very informative? What is the difference between observing that "some students cross the street per hour" and "40 students cross the street per hour"? They both say the same thing, but the second version provides more specific information. As another example, I could say that it is hot out today, or I could say that it is 99 degrees Fahrenheit (37 degrees Celsius). I could use either of these labels to describe the same day.

The process of gathering data is called **measurement**. It will be useful for us to classify measures according to the kind of data they contain. We will classify measurement in three ways (from Stevens, 1946):

1. According to their *level of measurement*
2. Whether or not they are *continuous or discrete*
3. Whether they represent *qualitative or quantitative* data.

Once you understand these classifications, you should be able to classify a measure in these three ways.

1.2.1 Level of Measurement

A stair diagram is used because higher levels of measurement satisfy all the requirements of the levels below.

```
   Ratio scale/ratio measurement.  Examples: weight, length
  Interval scale/interval measurement.  Example: Fahrenheit temperature
 Ordinal scale/ordinal measurement.  Example: the order in which people finish a race
Nominal scale/Nominal measurement.  Example: which is your favorite fruit?
```

Notice that these levels are stair steps. Each level has all the characteristics of the level below it. Interval scales meet all the requirements of ordinal and nominal scales as well (plus they meet the additional requirement for interval scales).

To determine the level of measurement, ask yourself these questions:

1. Can you rank/order the numbers? (if no, nominal scale. if yes, keep going) example: kinds of fish. can you rank halibut and mullet? (no, nominal scale) example: Olympic medals, can you rank gold, silver, and bronze? (yes, keep going)
2. If you add/subtract the numbers, does the result have meaning? (if no, ordinal scale. if yes, keep going) example: 30 degrees F plus 10 degrees equals 40 degrees (yes, keep going) example: 1st place plus 2 equals 3rd place? (no, this does not make sense, ordinal scale)
3. Does the score have a value of 0 that means ‘none’ or ‘nothing’? (if no, interval scale. if yes, ratio scale) example: counting people; 0 people means no people (yes, ratio scale) example: 0 degrees F means no heat? (no, interval scale)

That last property, having a zero meaning none/nothing/not any is called a **true zero**. Fahrenheit temperature does not have a true zero (it is just another temperature), but Kelvin does (zero degrees is absolute zero and indicates no heat energy).

I find making up values to be a helpful strategy, as I did when I asked Question 2, above. It does not matter what the values are, so you can invent ones to make the questions more concrete.

When students are confused about classifying measures, the most common pattern I see is that they abandon the stair-step-question method. I recommend not trying to skip answering the questions, even as you start to get comfortable with this concept. Start from question 1, and continue up the levels until the answer is no. It takes a few more seconds but is much more reliable. And, remember that each level has all the properties of the levels below it. In other words, ratio scales meet all the requirements of interval scales, ordinal scales, and nominal scales. For this reason, I find trying to match definitions to examples is more confusing than the stair-step method (which was taught to me by my graduate advisor, and I am still using it!).

The second common point of confusion happens when students focus on the data instead of the measurement scale. When we classify measures, we are classifying the measurement scale. The measurement scale includes all possible data that could ever be observed (even if only theoretically). I usually use an exercise question that asks students to classify the level of measurement of the age of a football stadium. Like all measures of duration, the best answer is ratio. Often students are uncomfortable picking that answer, because they do not see anyone observing a football stadium to be 0 years (or days, hours, or minutes) old. How could a football stadium have no age? Even though it is unlikely that a list of football stadium ages would ever observe this, there is an instant where a football stadium has been constructed or opened and is therefore 0 years (days, hours, and minutes) old. All this to say, do not get distracted by what values are the most common or realistic. Instead, when classifying measurement scales, focus on all possible values.

1.2.2 Continuous or Discrete

Separately, you can decide if your variable is continuous or discrete. If you can have an infinite number of fractions of a value, it's continuous. If you cannot, the measure is discrete. example: 5 yards, 5.0005 yards, 5.5 years, and 5.500001 yards are all valid measurements (continuous) example: Olympic medals; the measurement between gold and silver does not exist (discrete)

There may be instances where a grey area exists; at some level, all variables are discrete. For example, you could subdivide a measurement of length down to the molecule. At that point, you cannot have fractional values. Try to avoid over-thinking this issue. If you can reasonably talk about fractional values (half seconds; twenty-five cents are a fraction of a dollar) then the measure is continuous. If you cannot (there is no such thing as half a dog or an eighth of an employee), then the measure is discrete.

1.2.3 Qualitative or Quantitative

Quantitative data is associated with a numerical value. **Qual**itative data is associated with labels that have no numerical value. Nominal and ordinal data are qualitative. Interval and ratio data are quantitative.

1.2.4 Distribution: A collection of our observations

When we make repeated, related observations and collect them together, we have data. When we represent data in numerical or categorical form, we form a **distribution**. When you see distribution, think of a collection of scores.

1.3 Descriptive Statistics: Summarizing our observations

Text by David Schuster

Video: Descriptive Statistics

The problem with distributions is that any collection of more than a couple observations quickly overwhelms our limited working memory and attention. We need a way to summarize distributions. Descriptive statistics does exactly that. A descriptive statistic summarizes a distribution (put another way, it measures a property of a distribution) using a single value.

Descriptive statistics lets us summarize two properties of distributions:

1. The value of the scores (central tendency)
2. How spread out the scores are from each other (variability)

Measures of central tendency are averages. There are multiple ways of expressing an average. Mean, median, and mode are different kinds of averages. That is about all we need for right now. Later, we will go into more detail on how these useful tools work.

Measures of variability put a number on how spread out the data are. Think about your workplace–Are some employees more content than others? Is everyone pretty much in agreement that your workplace is great (or awful)? If most people tend to agree, then we might say your workplace satisfaction has low variability. If there was not so much agreement, we might say your workplace satisfaction has high variability. With measures of variability, we can do even better by quantifying variability. Variability is the concept–how different are scores in the distribution? Measures of variability turn this into a value. Measures of variability include range, sum of squares, variance, and standard deviation.

When there is no variability, we call the value a *constant.* For almost anything you can think to measure about people, there are no constants. We live in a world of complex variability. For me, this is one of the most fascinating and challenging aspects of psychology. You can easily manufacture a bolt to have the same property as another, but psychologists get a front row seat at the amazing diversity of human thought and behavior. Describing and making predictions about variability is also a linkage between statistics and the study of human diversity, which we will consider in more detail later in this course.

1.4 Inferential Statistics: Generalizing from our observations

Video: Inferential Statistics

Inferential statistics is the process of drawing conclusions about a group of interest (called a population) using a limited set of data (called a sample). Fundamentally, inferential statistics uses probability theory and logic that allow you to make conclusions about populations.

We will cover a number of inferential stat techniques in this course. These include the *t*-test, ANOVA, multiple regression, and others. Other terms associated with inferential statistics (we will define and discuss later, for now, just know they are part of inferential statistics) include null hypothesis significance testing (NHST) and Bayesian statistics.

As an example of a population we might want to study, imagine I am interested in studying middle school students' reading comprehension in the United States, and I want to see if it changes over time. To understand this population directly, I would have to measure the reading comprehension of every member. This is impossible. Instead, I take a random sample from the population by mailing surveys to 50 random middle school students with consent of their parents, I can use descriptive statistics to understand my sample data (50 scores) and inferential statistics to generalize the results to the population (millions of scores).

1.4.1 Populations and Samples: Who (or what) the research is about

A population is the entire group of interest. Examples: people, nursing home residents, repeat customers, etc. The population is the group we want to study.

Populations can be any group you want to draw conclusions about. The researcher defines the population, and this frames the entire research project. The findings of a study intending to measure college students may not apply to older adults. The population is the group to which you will generalize your findings.

The descriptive statistics we will cover can be applied to populations. If we can measure everybody and calculate the average, then we have calculated a **population parameter**. A **population distribution**, which is constructed by measuring every member of a population, is called a **census**. Most of the time, our populations of interest are very large, and it is impossible to measure everybody. How could you give a survey to every single college student in the United States? You would first need a list of every college student in the United States. What are some of the problems with conducting a study in this way? You might think of the ethical obstacles, meaning that it is all but guaranteed you would not get every college student to agree to participate. You might also think of the logistical obstacles, such as the time and cost associated with advertising and administering tens of millions of surveys (even digital ones). But even generating a list of the population would be impossible. Imagine these other concerns did not exist and there was such a list of every student in the United States. Would that list be accurate? Put another way, for how long would that list remain accurate? Every day, new students begin college and other students graduate or leave school. A list of all college students in the United States would only be accurate for an instant. In this seemingly-straightforward population example, we see that even the list of members is constantly changing. For all but the smallest populations of people, population-level research is not possible. Even a precise count of such populations are not possible. This hints at a point we will revisit later in the course–sampling and statistics can be useful regardless of the population size. For this reason, sampling is a powerful tool for understanding populations.

A **sample** is a smaller set from the population. The collection of scores from a sample is called a **sample distribution**. When statistics are computed from a sample distribution, they are called sample statistics, or just statistics.

There are many ways to measure a subset of a population; we call the strategy for obtaining a sample the **sampling method**. The best sampling method is **random sampling**. It has a precise definition: A random sample means that every member of the population has an equal chance of being selected. To do this properly, a researcher should generate a list of every member of the population and select from the list at random. To be a truly random sample, every individual selected would have to participate in your study. True random samples meet this definition but there are other, more practical, sampling techniques that approximate random sampling; the closer to a random sample of the population, the more likely the sample will represent the population.

We can think of random sampling as one end of a spectrum with **convenience sampling** at the other. A researcher using a convenience sample asks whoever is available to participate in the study. The resulting sample is biased due to proximity, availability, and convenience. Put another way, convenience sampling is less systematic and more…well, convenient. The further away from a true random sample, the less likely it is that the sample collected will represent the population.

Inferential statistics are a collection of techniques to make conclusions about populations based on sample data. As you have seen, without this tool, we could never measure all the individuals we would wish to study. If you have not studied inferential statistics before, it may seem surprising, perhaps a bit unbelievable, that we could make conclusions from such little data. Inferential statistics is not magic; it does not guarantee perfect conclusions. We will see that researchers make certain assumptions when they use inferential statistics and they generate tentative conclusions. Often, the data suggest an answer rather than provide a definitive answer. Sometimes, the research results in more new questions than answers. These features suggest that science is challenging and takes skill to be done well. One of the goals of this course is to help you be a better researcher and a better evaluator of others' research.

Psychologists and others who study people often take for granted that the **units of analysis** are people. Often, this is the case. Through this lens, populations are groups of people and samples are made up of people. Nothing about statistics requires our observations to be about people, however. We could just as easily measure the number of miles a tire will last before it fails or the loudness of a lion's roar. We can also measure collections of people, such as the performance of a company, the frequency of communication of team members, or the outcomes of students in a school.

1.4.2 Constructs provide the context

The idea or concept represented by our data is called the **construct**. There is an important distinction between constructs and measures. A construct is a "concept, model, or schematic idea" (Shadish, Cook, & Campbell, 2002, p. 506). Constructs are the big ideas that researchers are interested in measuring: depression, patient outcomes, prevalence of cumulative trauma disorders, or even sales. For constructs in the social sciences, there is often disagreement and debate about how to define a construct. To do science, we must be able to quantify our observations (collect data) on the constructs. To go from a construct (the idea) to a measure requires an operational definition. An *operational definition* describes how a construct is measured.

Table 1.2: Admission figures for the six largest departments by gender

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

1.5 The cautionary tale of Simpson's paradox

Text by Navarro (2018)

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

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

…and the were worried about being sued.[1] Given that there were nearly 13,000 applicants, a difference of 9% in admission rates between males and females is just way too big to be a coincidence. Pretty compelling data, right? And if I were to say to you that these data *actually* reflect a weak bias in favour of women (sort of!), you'd probably think that I was either crazy or sexist.

Oddly, it's actually sort of true …when people started looking more carefully at the admissions data (Bickel et al., 1975) they told a rather different story. Specifically, when they looked at it on a department by department basis, it turned out that most of the departments actually had a slightly *higher* success rate for female applicants than for male applicants. Table 1.2 shows the admission figures for the six largest departments (with the names of the departments removed for privacy reasons):

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

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

Here's what's going on. Firstly, notice that the departments are *not* equal to one another in terms of their admission percentages: some departments (e.g., engineering, chemistry) tended to admit a high percentage of the qualified applicants, whereas others (e.g., English) tended to reject most of the candidates, even if they were high quality. So, among the six departments shown above, notice that department A is the most generous, followed by B, C, D, E and F in that order. Next, notice that males and females tended to apply to different departments. If we rank the departments in terms of the total number of male applicants, we get **A**>**B**>D>C>F>E (the "easy" departments are in bold). On the whole, males tended to apply to the departments that had high admission rates. Now compare this to how the female applicants distributed themselves. Ranking the departments in terms of the total number of female applicants produces a quite different ordering C>E>D>F>**A**>**B**. In other words, what these data seem to be suggesting is that the female applicants tended to apply to "harder" departments. And in fact, if we look at all Figure 1.1 we see that this trend is systematic, and quite striking. This effect is known as Simpson's paradox. It's not common, but it does happen in real life, and most people are very surprised by it when they first encounter it, and many people refuse to even believe that it's real. It is very real. And while there are lots of very subtle statistical lessons buried in there, I want to use it to make a much more important point …doing research is hard, and there are *lots* of subtle, counterintuitive traps lying in wait for the unwary. That's reason #2 why scientists love statistics, and why we teach research methods. Because science is hard, and the truth is sometimes cunningly hidden in the nooks and crannies of complicated data.

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

That was the basis for my somewhat glib remarks earlier, but that's not exactly the whole story, is it? After all, if we're interested in this from a more sociological and psychological perspective, we might want to ask *why* there are

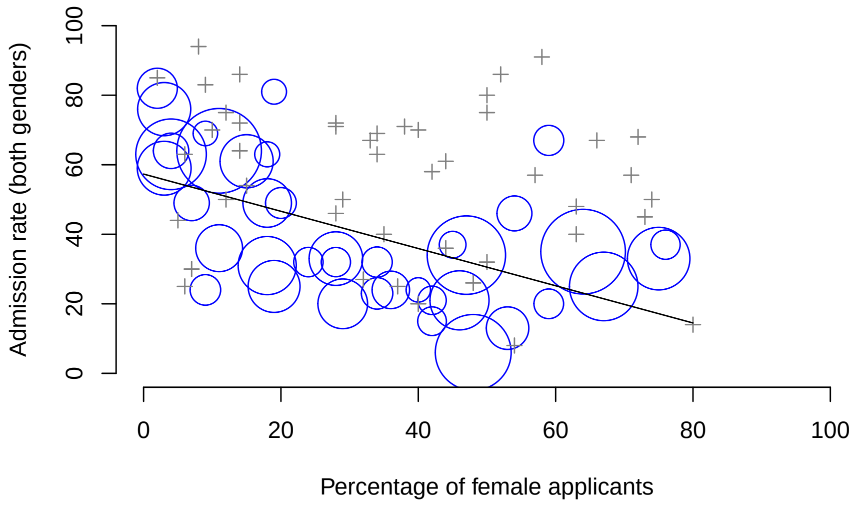


Figure 1.1: The Berkeley 1973 college admissions data. This figure plots the admission rate for the 85 departments that had at least one female applicant, as a function of the percentage of applicants that were female. The plot is a redrawing of Figure 1 from Bickel et al. (1975). Circles plot departments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

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.

1.6 A brief introduction to research design

Text by Navarro (2018)

> *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[2]

Note that this section 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]Presidential Address to the First Indian Statistical Congress, 1938. Source: http://en.wikiquote.org/wiki/Ronald_Fisher

1.6.1 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**.
- My **chromosomal gender** is *male*.
- My **self-identified gender** is *male*.[3]

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.

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

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).

1.6.2 Operationalisation: defining your measurement

Video: Operationalization

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.

1.6.3 The "role" of variables: predictors and outcomes

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

Now, when we doing an analysis, we have different names for X and Y, since they play different roles in the analysis. The classical names for these roles are ***independent variable*** (IV) and ***dependent variable*** (DV). The IV is the variable that you use to do the explaining (i.e., X) and the DV is the variable being explained (i.e., Y). The logic behind these names goes like this: if there really is a relationship between X and Y then we can say that Y depends on X, and if we have designed our study "properly" then X isn't dependent on anything else. However, I personally find those names horrible: they're hard to remember and they're highly misleading, because (a) the IV is never actually "independent of everything else" and (b) if there's no relationship, then the DV doesn't actually depend on the IV. And in fact, because I'm not the only person who thinks that IV and DV are just awful names, there are a number of alternatives that I find more appealing. The terms that I'll use in these notes are ***predictors*** and ***outcomes***. The idea here is that what you're trying to do is use X (the predictors) to make guesses about Y (the outcomes).[4] This is summarised in Table 1.3.

[4] 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.

Table 1.3: 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

1.6.4 Experimental and non-experimental research

Video: Experimental, Quasi-, and Non-Experimental Research Designs

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.

1.6.4.1 Experimental research

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

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

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

1.7 Causality, Research, and Statistics

Text by David Schuster

Video: Causality

1.7.1 Experimental, Quasi-Experimental, and Non-Experimental Studies

In this section, I would like to add a bit more precision to the general concepts explained by Navarro (2018).

Research psychology is a process of identifying constructs and describing how they relate to other constructs. We can classify research designs as experiments, quasi-experiments, and non-experiments.

Experiments are the only kind of research that shows causal relationships (that is, that construct A causes a change in construct B). So an experiment could show if smoking causes lung cancer. To do this, experiments need two things (or they are not experiments)

All experiments have a manipulation. This means that the experimenter changes something within the environment of the experiment (called an independent variable) to see if it causes a change in the outcome (called a dependent variable). For our smoking example, a manipulation would be assigning one group of participants to a lifetime of smoking and another group of participants to a lifetime of no smoking.

Experiments require random assignment. The experimenter decides when to vary the levels of the manipulation (change the manipulation) based on random assignment. Random assignment means that every participant has the same chance as being in one condition as another. For our smoking example, random assignment means each participant has a 50% chance of being in the smoking group.

As may be clear from the smoking example, we cannot always do experiments because of ethical (it would be wrong to assign people to smoke) or practical reasons (you cannot randomly assign people to genders, for example). The solution is a quasi- or non-experimental study.

In summary: experiments are powerful because they uniquely demonstrate causality (causal relationships). However, experiments require a manipulation and random assignment, which are not always possible.

In a quasi-experimental study, there is a manipulation but no random assignment. Whenever participants are assigned to levels of a manipulation non-randomly, the research is quasi-experimental. In a quasi-experimental smoking study, we could ask people if they had smoked before and assign them to smoking or non-smoking groups based on that answer.

In summary: quasi-experiments do not require random assignment, but they do not show casual relationships.

In a non-experimental study, no manipulation is done. If you want to look at the effects of gender on lung cancer, you would simply observe (collect data on) the genders of patients. By only observing, you would not be manipulating gender.

The differences between quasi- and non-experimental studies are sometimes slight (Pedhauzer & Schmelkin, 1991); if the researcher is manipulating an IV, then the work is quasi-experimental.

In summary: non-experimental studies are observational. Like quasi-experimental studies, they do not show causal relationships.

It's worth repeating that only experiments demonstrate causality. Quasi- and non-experiments can show that a relationship exists but do not say whether one variable causes the other. Any non-causal relationship has three possible explanations:

1. A $\rightarrow$ B one variable causes another; in an experiment, this is the only explanation
2. B $\leftarrow$ A the relationship is reversed; the first variable is actually the outcome
3. C $\rightarrow$ A; C $\rightarrow$ B a third variable exists that was not measured in the study; the third variable causes a change in both A and B. There are many 'C' variables, potentially.

In a non-experimental smoking study, you could not say whether smoking causes lung cancer or people who are predisposed to lung cancer are more likely to smoke. A third possibility is that a separate, third variable causes both lung cancer and a desire to smoke.

1.7.2 Demonstrating Causality

In the 19th century, John Stewart Mill said that we could be satisfied that a relationship is causal if the following three things could be demonstrated:

1. The cause preceded the effect
2. The cause was related to the effect
3. We can find no plausible alternative explanation for the effect other than the cause

Experiments aim to identify causal relationships by manipulating something, observing the outcome, seeing a relationship, and using various methods to reduce other explanations.

1.7.3 Statistics and Causality

Statistics are an important tool for establishing causality, but it's important to know that the choice of statistical technique does not affect the level of causal evidence; demonstrating causality is the job of the research design, not the statistics.

A common misconception arises from the term correlational research design, which people use as a label for quasi-experimental and non-experimental research. It is easy to confuse this term with correlation which is a statistical technique.

Recall that statistics has two branches: Descriptive stats provides tools to summarize variability. Inferential stats provides tools for generalizing samples to populations.

To demonstrate causality, we need to satisfy Mill's second requirement. Inferential statistics can help us do that. Two techniques are particularly useful: correlation (and its statistic r) and the t¬-test (and its statistic, t). Next, we will see how these techniques work.

1.7.4 Validity and Reliability

Text by David Schuster

1.7.4.1 Define validity and reliability

Reliability and validity are fundamental to critiquing psychological research and to developing your own high-quality research. There are different types of validity and reliability that are relevant to us, which sometimes confuses people.

Because of this, introductory textbooks often present convoluted definitions of these concepts. Fortunately, the real definitions are simple:

Reliability means consistency. Something is reliable if it is consistent. The more consistency, the more reliability.

Validity means truth. Something is valid if it is true. Truth is either-or; there is no such thing as "more true" or "less true."

In other words, good psychological science requires certain types of consistency and for some of the claims we make to be true. Next, we will look at the specific kinds of reliability and validity that are important for scientists.

1.7.4.2 Types of consistency = Types of Reliability

Reliability

Here are arguably the three most important types of reliability:

Type of Reliability	Situation	Definition	How to assess
Test-retest	You administer a measure to a participant, then wait some period of time, and give them the test again. The participant's true score on the measure has not changed (e.g., IQ, personality).	The extent to which a measure is consistent across different administrations	Look for a correlation between the two administrations
Interrater	A measure involves two or more raters who record subjective observations (e.g., counting the number of times a participant has a tic, counting the number of times a married couple shows affection)	The extent to which two observers are consistent in their ratings	Look for a correlation between the two raters
Internal consistency	You are measuring a construct using several items (e.g., five items all rating your enjoyment of a course)	The extent to which items on a measure are consistent with each other; expected if the items measure the same construct	Cronbach's alpha (.7 is acceptable, .8 is good, and .9 is excellent)

1.7.4.3 Validity is a property of inferences

Video: Validity & Threats

Validity is a specific kind of truth. Validity is the truth of an inference, or a claim. In other words, validity is a property of inferences. An inference (a claim) is valid if it is true.

For example, I could claim that the earth is round. Hopefully, it is a claim that you accept as being true. If you agree, then you could label my claim as valid.

Validity in research is frequently misunderstood, which leads to bizarre and confusing definitions of validity. There is no such thing as “a valid study.” Only claims about the study are valid or not. There is also no such thing as “a valid researcher.” A researcher can make claims. Only the researcher’s claims are valid or not. There is also no such thing as “more valid” or “increasing validity.” Validity is truth of a claim. Either a claim is true, or it is not.

For better or for worse, we usually don’t know with 100% certainty if a claim is true or false (if we did, we wouldn’t need the research). Therefore, research methods get very interesting when we listen to other researcher’s claims and then debate if we agree with them or not. When we do this, we are evaluating the validity of claims made about the study. Next, let’s look at different types of claims (inferences) that are made in research.

1.7.4.4 Types of inferences in a study = Types of validity

Here are some of the most important types of validity.

Type of Validity	Type of Claim	Definition	Example claim
Construct validity	The study operations represent the constructs of interest	The truth of claims that study operations match study constructs	“The Stanford-Binet was used to measure IQ”
Internal validity	The study IV caused a change in the study DV	The truth of claims that the IV causes changes in the DV	“The control group reported lower levels of stress than the experimental group, suggesting that the manipulation raised stress.”

Type of Validity	Type of Claim	Definition	Example claim
External validity	The study results apply to situation X	The truth of claims that the findings will apply as participants/units/variables/settings change.	"Although data were collected from college students, a similar effect would be expected in working adults."
Statistical conclusion validity	The statistical analysis was significant or not significant	The truth of claims about the size and direction of the relationship between the IV and the DV. Or, that the statistical results are correct.	"p < .05, indicating a significant difference"

Finally, you might encounter these other types of validity, but they are less clearly defined and evaluated:

- Content validity: The truth of claims that a measure adequately samples (includes the important elements of) the domain of interest. For example, if IQ includes both verbal and math ability, an IQ test would need to have both verbal and math items.
- Face validity: The truth of claims that a study operation "seems like" the construct. For example, a study about distractions from mobile devices might not support claims of "seeming real" if the phone in the study is a paper mockup.
- Criterion validity: The truth of claims that a measure can predict or correlate with some outcome of interest. A personality test as part of a job application would have criterion validity if it predicted applicants' success in the job.

1.7.4.5 Threats to validity

Threats to validity are specific reasons why an inference about a study is wrong. They can help us anticipate problems in the design of our own research. The best way to address threats to validity is to change the design of our research. Understanding threats to validity also helps you critique research done by others.

Chapter 2

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

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

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" (limited versions of the real thing) very cheaply; they sell full powered "educational versions" at a price that makes me wince; and they sell commercial licences with a staggeringly high price tag. The business model here is to suck you in during your student days, and then leave you dependent on their tools when you go out into the real world. It's hard to blame them for trying, but personally I'm not in favour of shelling out thousands of dollars if I can avoid it. And you can avoid it: if you make use of packages like R that are open source and free, you never get trapped having to pay exorbitant licensing fees.
- 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.

2.1 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]

2.1.1 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

[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.

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 (see Section 2.1.4 for instructions).

2.1.2 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 (see Section 2.1.4 for instructions).

2.1.3 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

[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.

from the command line just by typing `R`. However, if you're feeling envious of Windows and Mac users for their fancy GUIs, you can download RStudio too (see Section 2.1.4 for instructions).

2.1.4 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/

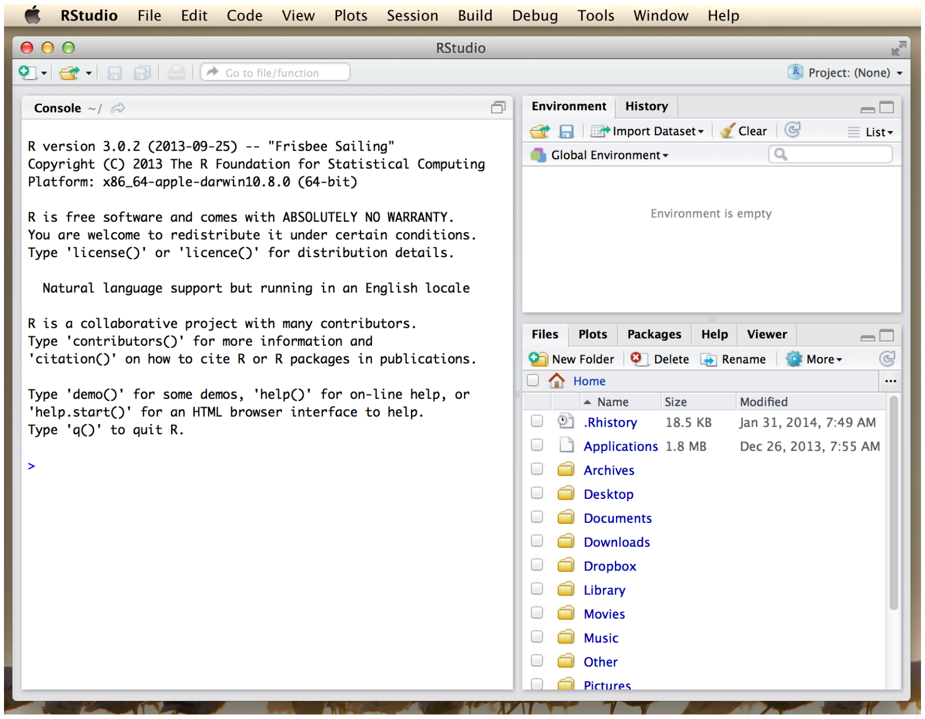
When you visit the RStudio website, you'll probably be struck by how much cleaner and simpler it is than the CRAN website,[5] and how obvious it is what you need to do: click the big green button that says "Download".

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 2.1 shows a screenshot of an R session in progress. In this screenshot, you can see 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.

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.



```
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.

>
```

Name	Size	Modified
.Rhistory	18.5 KB	Jan 31, 2014, 7:49 AM
Applications	1.8 MB	Dec 26, 2013, 7:55 AM
Archives		
Desktop		
Documents		
Downloads		
Dropbox		
Library		
Movies		
Music		
Other		
Pictures		

Figure 2.1: An R session in progress running through RStudio. The picture shows RStudio running on a Mac, but the Windows interface is almost identical.

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

2.1.5 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!

2.2 Typing commands at the R console

Video: RStudio for the Total Beginner

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.

2.2.1 An important digression about formatting

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

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

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

[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.

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

```
10 + 20
```

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

There are two main differences.

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

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

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

```
10 + 20
```

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

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

2.2.2 Be very careful to avoid typos

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

```
10 = 20
```

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

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

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

```
10 - 20
```

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

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

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

2.2.3 R is (a bit) flexible with spacing

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

```
10    + 20
```

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

or this

```
10+20
```

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

and I would get exactly the same answer. However, that doesn't mean that you can insert spaces in any old place. When we looked at the startup documentation in Section 2.1.5 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 (2020). R: A language and environment for statistical
##   computing. R Foundation for Statistical Computing, Vienna, Austria.
##   URL https://www.R-project.org/.
##
## A BibTeX entry for LaTeX users is
##
##   @Manual{,
##     title = {R: A Language and Environment for Statistical Computing},
##     author = {{R Core Team}},
##     organization = {R Foundation for Statistical Computing},
##     address = {Vienna, Austria},
##     year = {2020},
##     url = {https://www.R-project.org/},
##   }
##
## We have invested a lot of time and effort in creating R, please cite it
## when using it for data analysis. See also 'citation("pkgname")' for
## citing R packages.
```

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

```
citation ()
```

```
citation(  )
```

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

```
citat ion()
```

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

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

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

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

```
> 10+
+
```

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

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

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

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

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

```
> citblation(
+
+
```

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

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

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

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

```
citation
## function (package = "base", lib.loc = NULL, auto = NULL)
```

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

```
## {
##     dir <- system.file(package = package, lib.loc = lib.loc)
##     if (dir == "")
##         stop(gettextf("package '%s' not found", package), domain = NA)

BLAH BLAH BLAH
```

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

2.3 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!

2.3.1 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 2.1 lists the operators that correspond

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

Table 2.1: Basic arithmetic operations in R. These five operators are used very frequently throughout the text, so it's important to be familiar with them at the outset.

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

to the basic arithmetic we learned in primary school: addition, subtraction, multiplication and division.

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

```
57 * 61
```

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

So that's handy.

2.3.2 Taking powers

The first four operations listed in Table 2.1 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 **??**.

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.

2.3.3 Doing calculations in the right order

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

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

```
1 + 2 * 4
```

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

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

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

R had decided instead to add `1+2` (to get an interim answer of 3) and then multiplied by 4, it would have come up with an answer of `12`.

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

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

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

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

```
4 / 2 * 3
```

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

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

[10]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.

2.4 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.

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

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

```
sales <- 350
```

When you hit enter, R doesn't print out any output.[11] It just gives you another command prompt. However, behind the scenes R has created a variable called `sales` and given it a value of `350`. You can check that this has happened by asking R to print the variable on screen. And the simplest way to do *that* is to type the name of the variable and hit enter[12].

```
sales
```

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

[11]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.

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

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

Okay, so now we know how to assign variables. Actually, there's a bit more you should know. Firstly, one of the curious features of R is that there are several different ways of making assignments. In addition to the `<-` operator, we can also use `->` and `=`, and it's pretty important to understand the differences between them.[13] Let's start by considering `->`, since that's the easy one (we'll discuss the use of `=` in Section 2.5.1. As you might expect from just looking at the symbol, it's almost identical to `<-`. It's just that the arrow (i.e., the assignment) goes from left to right. So if I wanted to define my `sales` variable using `->`, I would write it like this:

```
350 -> sales
```

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

2.4.2 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:

```
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
```

[13] 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.

[14] 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.

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.

2.4.3 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[15] illustrates:

[15] 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 can only use the upper case alphabetic characters `A-Z` as well as the lower case characters `a-z`. You can also include numeric characters `0-9` in the variable name, as well as the period `.` or underscore `_` character. In other words, you can use `SaL.e_s` as a variable name (though I can't think why you would want to), but you can't use `Sales?`.
- Variable names cannot include spaces: therefore `my sales` is not a valid name, but `my.sales` is.
- Variable names are case sensitive: that is, `Sales` and `sales` are *different* variable names.
- Variable names must start with a letter or a period. You can't use something like `_sales` or `1sales` as a variable name. You can use `.sales` as a variable name if you want, but it's not usually a good idea. By convention, variables starting with a `.` are used for special purposes, so you should avoid doing so.
- Variable names cannot be one of the reserved keywords. These are special names that R needs to keep "safe" from us mere users, so you can't use them as the names of variables. The keywords are: `if`, `else`, `repeat`, `while`, `function`, `for`, `in`, `next`, `break`, `TRUE`, `FALSE`, `NULL`, `Inf`, `NaN`, `NA`, `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,[16] but it's important to be consistent.

2.5 Using functions to do calculations

The symbols `+`, `-`, `*` and so on are examples of operators. As we've seen, you can do quite a lot of calculations just by using these operators. However, in order to do more advanced calculations (and later on, to do actual statistics), you're going to need to start using ***functions***.[17] I'll talk in more detail about functions and how they work in Section **??**, 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:

[16]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`.

[17]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.

```
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:

```
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[18] 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`.

2.5.1 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()`

[18]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.

function supports this, by allowing you to input a second argument to the function that specifies the number of decimal places that you want to round the number to. In other words, I could do this:

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

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

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

```
round( x = 3.1415, digits = 2 )
```

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

Notice that this is kind of similar in spirit to variable assignment (Section 2.4), 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,[19] since that makes it clearer to you what I'm doing. However, one important thing to note is that when specifying the arguments using their names, it doesn't matter what order you type them in. But if you don't use the argument names, then you have to input the arguments in the correct order. In other words, these three commands all produce the same output…

[19]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.

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

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

```
round( x = 3.1415, digits = 2 )
```

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

```
round( digits = 2, x = 3.1415 )
```

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

but this one does not…

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

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

How do you find out what the correct order is? There's a few different ways, but the easiest one is to look at the help documentation for the function (see Section 2.25. 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.

2.6 Letting RStudio help you with your commands

Time for a bit of a digression. At this stage you know how to type in basic commands, including how to use R functions. And it's probably beginning to dawn on you that there are a *lot* of R functions, all of which have their own arguments. You're probably also worried that you're going to have to remember all of them! Thankfully, it's not that bad. In fact, very few data analysts bother to try to remember all the commands. What they really do is use tricks to make their lives easier. The first (and arguably most important one) is to use the internet. If you don't know how a particular R function works, Google it. Second, you can look up the R help documentation. I'll talk more about these two tricks in Section 2.25. But right now I want to call your attention to a couple of simple tricks that RStudio makes available to you.

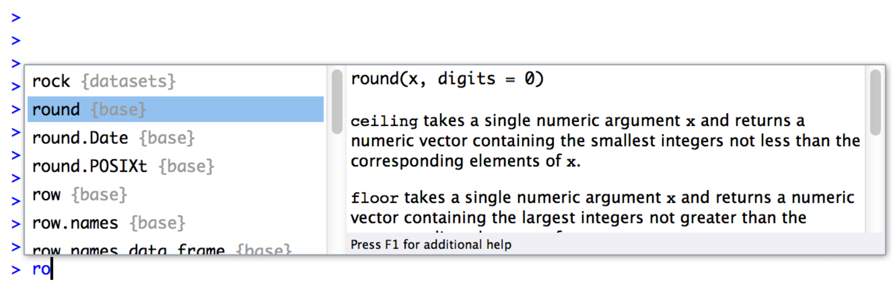
2.6.1 Autocomplete using "tab"

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

Let's stick to our example above and assume that what you want to do is to round a number. This time around, start typing the name of the function that you want, and then hit the "tab" key. RStudio will then display a little window like the one shown in Figure 2.2. 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 2.15. In Figure 2.2 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 2.2. 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

[20]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.



```
rock {datasets}
round {base}
round.Date {base}
round.POSIXt {base}
row {base}
row.names {base}
row.names.data.frame {base}
```

```
round(x, digits = 0)

ceiling takes a single numeric argument x and returns a
numeric vector containing the smallest integers not less than the
corresponding elements of x.

floor takes a single numeric argument x and returns a numeric
vector containing the largest integers not greater than the
Press F1 for additional help
```

```
> ro
```

Figure 2.2: 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.

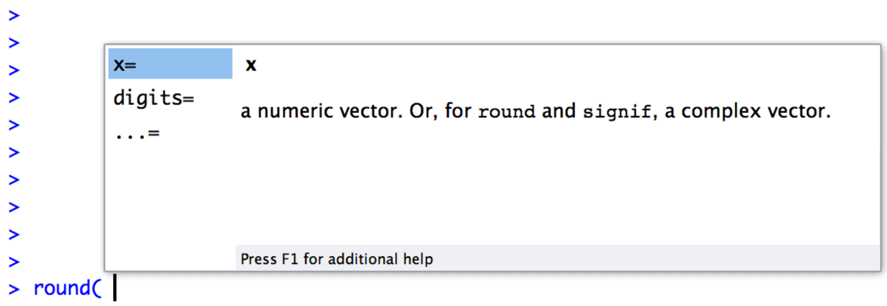
all the information you need. But RStudio goes a bit further, and provides some additional information about the function underneath. Sometimes that additional information is very helpful, sometimes it’s not: RStudio pulls that text from the R help documentation, and my experience is that the helpfulness 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 2.3. 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 2.2. 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.

2.6.2 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[21] Using the up and down keys can be really handy if you’ve typed a long

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



```
>
>
>
>
>
>
>
>
>
>
> round( |
```

x=
digits=
...=
x
a numeric vector. Or, for round and signif, a complex vector.
Press F1 for additional help

Figure 2.3: If you've typed the name of a function already along with the left parenthesis and then hit the "tab" key, RStudio brings up a different window to the one shown above. This one lists all the arguments to the function on the left, and information about each argument on the right.

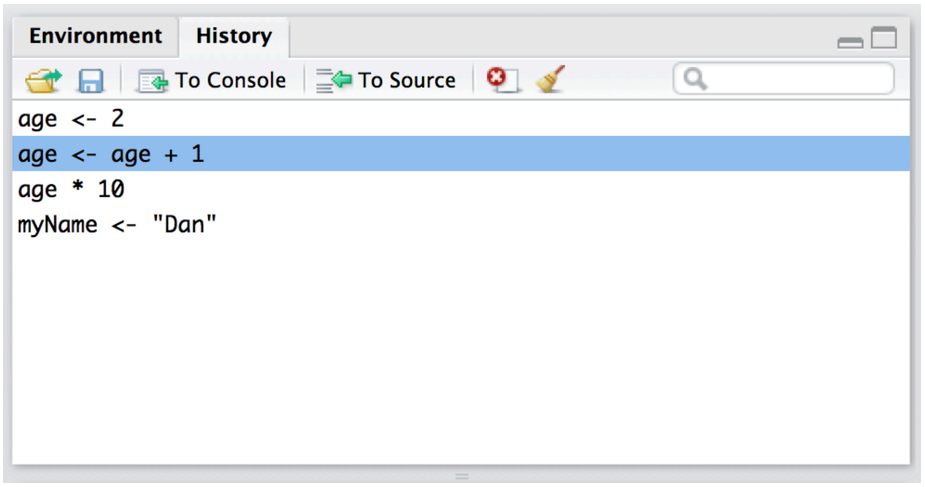
command that had one typo in it. Rather than having to type it all again from scratch, you can use the up key to bring up the command and fix it.

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 2.4. 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 Console" button).[22]

2.7 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 2.24, so let's return to our discussion of variables. When I introduced variables in Section 2.4 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.

[22]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.



```
age <- 2
age <- age + 1
age * 10
myName <- "Dan"
```

Figure 2.4: 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.

2.7.1 Creating a vector

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

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

[23]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.

2.7.2 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
##  [5]   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 three 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 `[5]` and the third with `[9]`. It's pretty clear what's happening here. For the first row, R has printed out the 1st element through to the 4th element, so it starts that row with a `[1]`. For the second row, R has printed out the 5th element of the vector through to the 8th one, and so it begins that row with a `[5]` 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!

2.7.3 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 2.10).

2.7.4 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.

2.7.5 Useful things to know about vectors

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

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

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

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

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

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

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

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

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

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

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

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

2.8 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`.[24] To do so, I could type in a command like this

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

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

[24]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.

```
months[4]
```

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

2.8.1 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 ??, 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.

2.9 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.

2.9.1 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[25] but it's definitely not true of R. R is not infinitely malleable. It has rather firm opinions on the topic of what is and isn't true, at least as regards basic mathematics. If I ask it to calculate `2 + 2`, it always gives the same answer, and it's not bloody 5:

```
2 + 2
```

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

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

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

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

What I've done here is use the ***equality operator***, `==`, to force R to make a "true or false" judgement.[26] Okay, let's see what R thinks of the Party slogan:

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

[26] Note that this is a very different operator to the assignment operator `=` that I talked about in Section 2.4. 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.

```
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.

2.9.2 Logical operations

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

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

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

or this

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

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

Not only that, but as Table 2.2 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`:

Table 2.2: Some logical operators. Technically I should be calling these ”binary relational operators”, but quite frankly I don’t want to. It’s my book so no-one can make me.

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

```
99 < 100
```

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

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

```
100 < 100
```

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

```
100 < 99
```

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

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

```
100 <= 100
```

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

```
100 <= 99
```

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

Table 2.3: 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'

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 2.3.

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

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

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

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

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

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

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

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

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

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

```
! FALSE
```

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

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

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

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

But there are many situations where you really do need to use the `!` operator. We'll see some later on.[27]

2.9.3 Storing and using logical data

Up to this point, I've introduced *numeric data* (in Sections 2.4 and 2.7) and *character data* (in Section 2.8). So you might not be surprised to discover that these `TRUE` and `FALSE` values that R has been producing are actually a third kind of data, called *logical data*. That is, when I asked R if `2 + 2 == 5` and it said `[1] FALSE` in reply, it was actually producing information that we can store in variables. For instance, I could create a variable called `is.the.Party.correct`, which would store R's opinion:

[27]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.

```
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).[28] So this works:

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

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

but this doesn't:

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

```
## Error in eval(expr, envir, enclos): object 'f' not found
```

2.9.4 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 2.7) and vectors of text data (Section 2.8). Again, we can define them directly via the `c()` function, like this:

[28]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`.

```
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 2.7. 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
## [12] FALSE
```

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

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

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

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

2.9.5 Applying logical operation to text

In a moment (Section 2.10) 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 **??**.

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
```

2.10 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.

2.10.1 Extracting multiple elements

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

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

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

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

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

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

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

```
sales.by.month[ c(2,3,4,5,6,7,8) ]
```

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

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

```
2:8
```

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

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

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

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

So that's kind of neat.

2.10.2 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<-c("high", "high", "low", "out", "out", "high",
                "high", "high", "high", "high", "high", "high")

stock.levels
```

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

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

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

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

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

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

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

or this

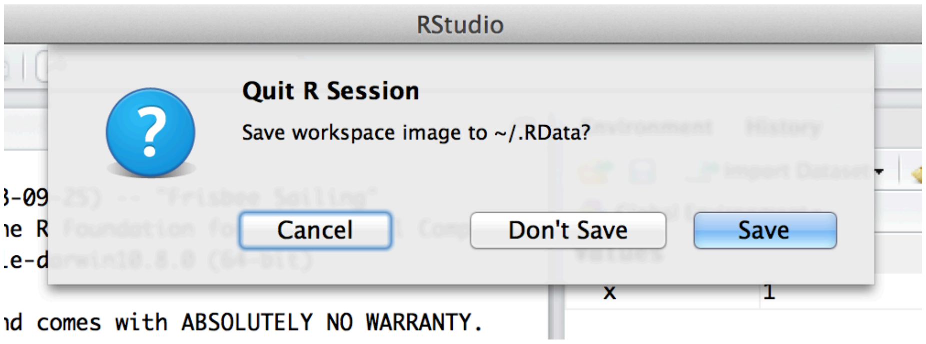
```
months[stock.levels != "high" ]
```

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

Either way, I get the answer I want.

At this point, I hope you can see why logical indexing is such a useful thing. It's a very basic, yet very powerful way to manipulate data. We'll talk a lot more about how to manipulate data in Chapter **??**, 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.[29]

2.11 Quitting R



```
3-09
ne R
.e-d

nd comes with ABSOLUTELY NO WARRANTY.
```

Figure 2.5: The dialog box that shows up when you try to close RStudio.

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 2.18, 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 2.5. If you're using a text based interface you'll see this:

[29]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.

```
q()

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

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

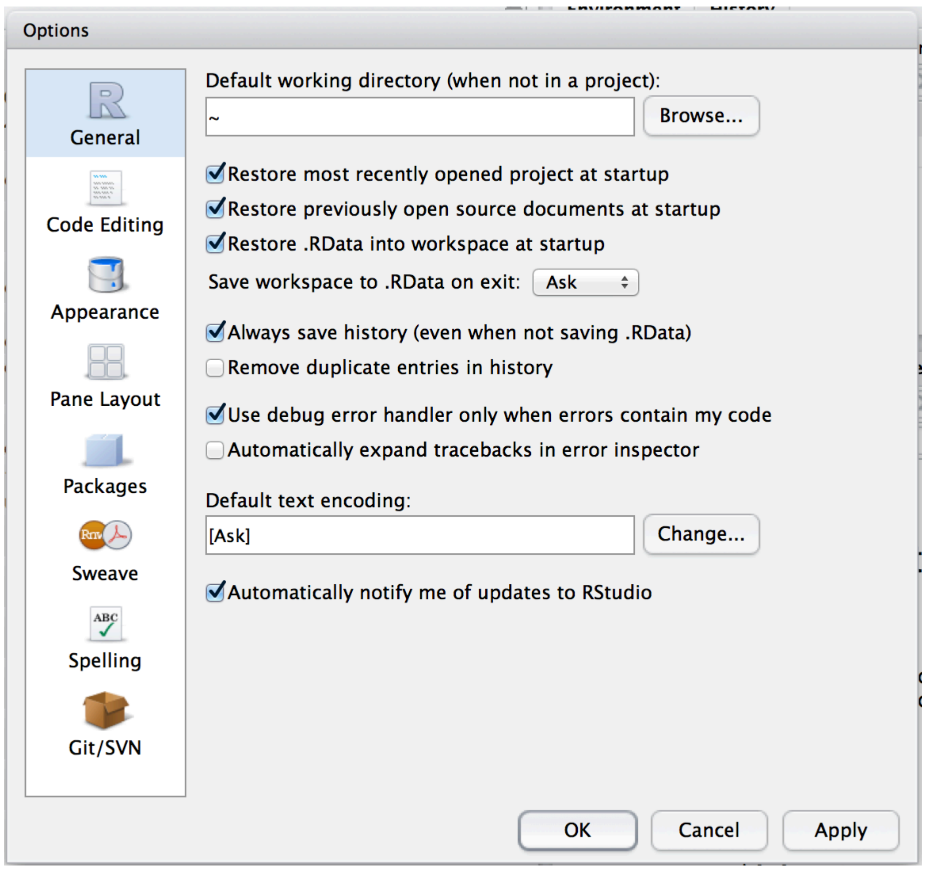
What does this actually *mean*? What’s going on is that R wants to know if you want to save all those variables that you’ve been creating, so that you can use them later. This sounds like a great idea, so it’s really tempting to type `y` or click the “Save” button. To be honest though, I very rarely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

In fact, every time I install R on a new machine one of the first things I do is change the settings so that it never asks me again. You can do this in RStudio really easily: use the menu system to find the RStudio option; the dialog box that comes up will give you an option to tell R never to whine about this again (see Figure 2.6. 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”.

2.12 Summary

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

- Getting started. We downloaded and installed R and RStudio
- Basic commands. We talked a bit about the logic of how R works and in particular how to type commands into the R console (Section @ref(#firstcommand), and in doing so learned how to perform basic calculations using the arithmetic operators `+`, `-`, `*`, `/` and `^`.
- Introduction to functions. We saw several different functions, three that are used to perform numeric calculations (`sqrt()`, `abs()`, `round()`, one that applies to text (`nchar()`; Section 2.8.1), and one that works on any


Options

Default working directory (when not in a project):
~

- [✓] Restore most recently opened project at startup
- [✓] Restore previously open source documents at startup
- [✓] Restore .RData into workspace at startup

Save workspace to .RData on exit: Ask

- [✓] Always save history (even when not saving .RData)
- [ ] Remove duplicate entries in history
- [✓] Use debug error handler only when errors contain my code
- [ ] Automatically expand tracebacks in error inspector

Default text encoding:
[Ask]

- [✓] Automatically notify me of updates to RStudio

Figure 2.6: 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"

variable (`length()`; Section 2.7.5). In doing so, we talked a bit about how argument names work, and learned about default values for arguments. (Section 2.5.1)
- Introduction to variables. We learned the basic idea behind variables, and how to assign values to variables using the assignment operator `<-` (Section 2.4). We also learned how to create vectors using the combine function `c()` (Section 2.7).
- Data types. Learned the distinction between numeric, character and logical data; including the basics of how to enter and use each of them. (Sections 2.4 to 2.9)
- Logical operations. Learned how to use the logical operators `==`, `!=`, `<`, `>`, `<=`, `=>`, `!`, `&` and `|`. And learned how to use logical indexing. (Section 2.10)

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

2.13 Additional R concepts

Form follows function

– Louis Sullivan

So far, 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 section: to build on the introductory content from the last section, to get you to the point that we can start using R for statistics. Broadly speaking, the section comes in two parts. The first half of the section 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 **??** and **??**.

2.14 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 **??**). 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:[30]

```
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
print( keeper )
```

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

You might have already noticed that the code extracts in Chapter 2 included the # character, but from now on, you'll start seeing # characters appearing in the extracts, with some human-readable explanatory remarks next to them. These are still perfectly legitimate commands, since R knows that it should ignore the # character and everything after it. But hopefully they'll help make things a little easier to understand.

2.15 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

[30]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.

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.[31] 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.

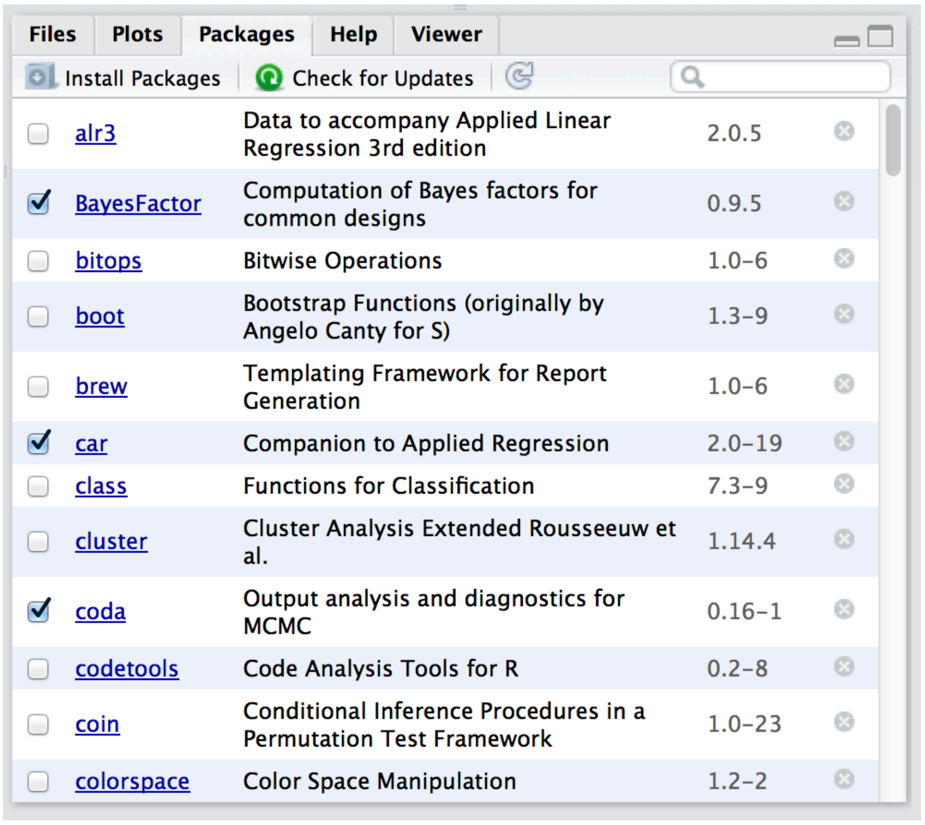
This two step process might seem a little odd at first, but the designers of R had very good reasons to do it this way,[32] and you get the hang of it pretty quickly.

2.15.1 The package panel in RStudio

Right, lets get started. The first thing you need to do is look in the lower right hand panel in RStudio. You'll see a tab labelled "Packages". Click on the tab,

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

[32]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.



☐	alr3	Data to accompany Applied Linear Regression 3rd edition	2.0.5
✓	BayesFactor	Computation of Bayes factors for common designs	0.9.5
☐	bitops	Bitwise Operations	1.0–6
☐	boot	Bootstrap Functions (originally by Angelo Canty for S)	1.3–9
☐	brew	Templating Framework for Report Generation	1.0–6
✓	car	Companion to Applied Regression	2.0–19
☐	class	Functions for Classification	7.3–9
☐	cluster	Cluster Analysis Extended Rousseeuw et al.	1.14.4
✓	coda	Output analysis and diagnostics for MCMC	0.16–1
☐	codetools	Code Analysis Tools for R	0.2–8
☐	coin	Conditional Inference Procedures in a Permutation Test Framework	1.0–23
☐	colorspace	Color Space Manipulation	1.2–2

Figure 2.7: The packages panel.

and you'll see a list of packages that looks something like Figure 2.7. Every row in the panel corresponds to a different package, and every column is a useful piece of information about that package.[33] 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).

2.15.2 Loading a package

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

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

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

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

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

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

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.

2.15.3 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.

2.15.4 A few extra comments

Sections 2.15.2 and 2.15.3 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()`.[34] 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)
```

```
## Loading required package: carData
```

```
library(psych)
```

```
##
## Attaching package: 'psych'
```

```
## The following object is masked from 'package:car':
##
##     logit
```

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.[35]

2.15.5 Downloading new packages

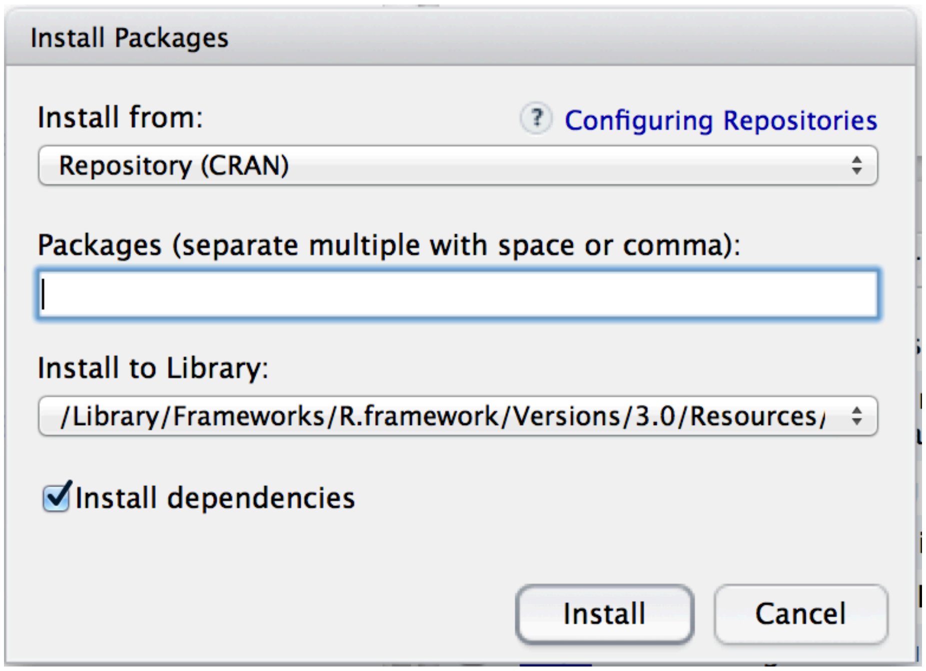
One of the main selling points for R is that there are thousands of packages that have been written for it, and these are all available online. So whereabouts online

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

[35]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.

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[36]

Using the RStudio tools is, again, dead simple. In the top left hand corner of the packages panel (Figure 2.7) 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 2.8.


Install Packages

Install from: Repository (CRAN)

Packages (separate multiple with space or comma):

Install to Library: /Library/Frameworks/R.framework/Versions/3.0/Resources/

✓ Install dependencies

Figure 2.8: The package installation dialog box in RStudio

There are a few different buttons and boxes you can play with. Ignore most of them. Just go to the line that says "Packages" and start typing the name of the package that you want. As you type, you'll see a dropdown menu appear (Figure 2.9), 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

[36]It is not very difficult.

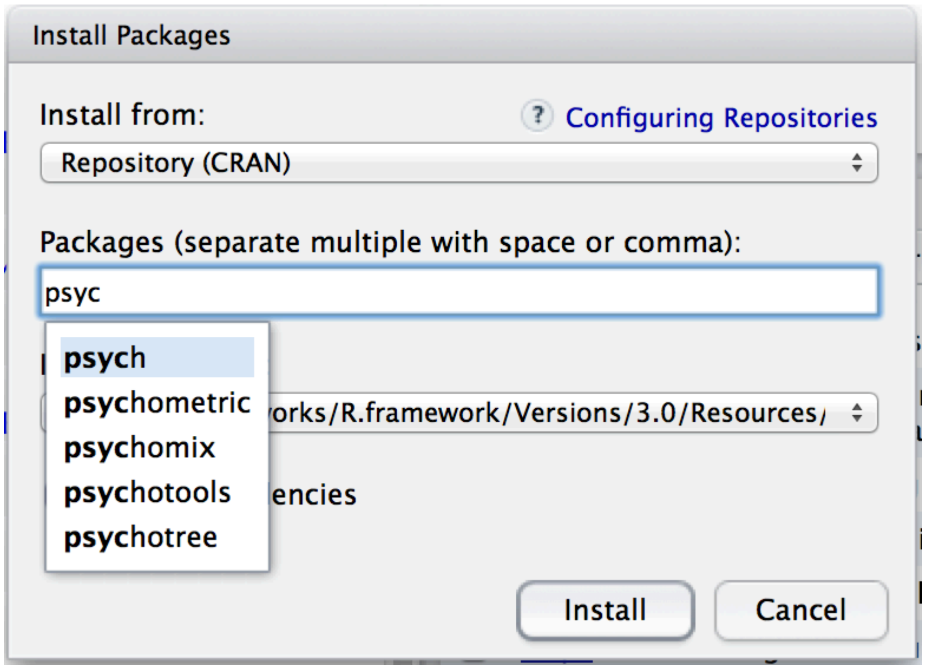


Figure 2.9: When you start typing, you'll see a dropdown menu suggest a list of possible packages that you might want to install

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:

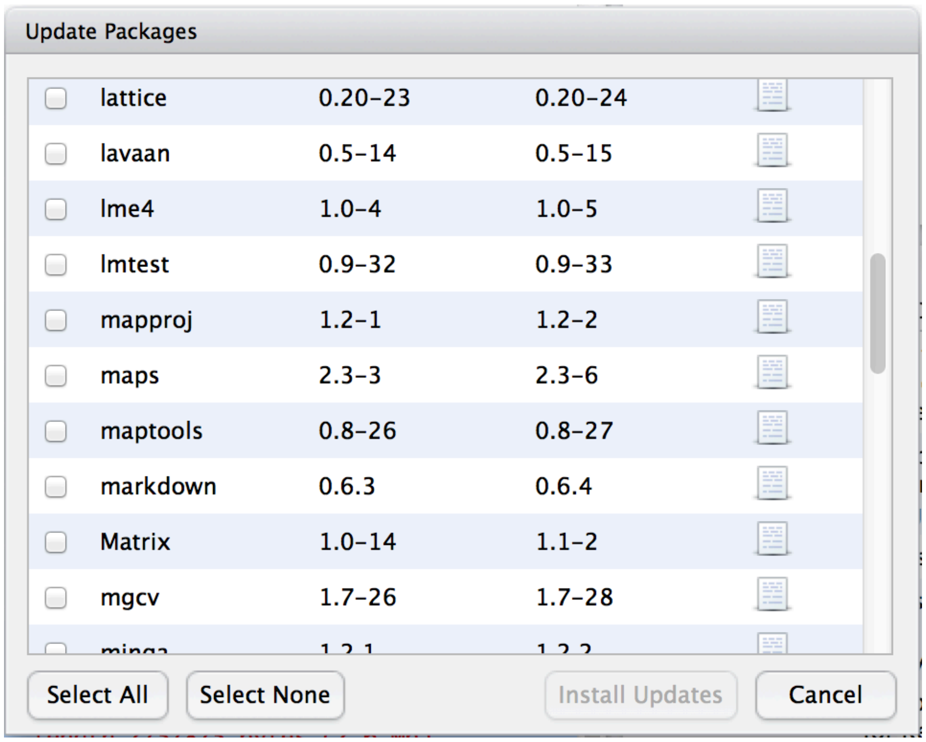
```
trying URL 'http://cran.rstudio.com/bin/macosx/contrib/3.0/psych_1.4.1.tgz'
Content type 'application/x-gzip' length 2737873 bytes (2.6 Mb)
opened URL
==================================================
downloaded 2.6 Mb


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.

2.15.6 Updating R and R packages

Every now and then the authors of packages release updated versions. The updated versions often add new functionality, fix bugs, and so on. It's generally a good idea to update your packages periodically. There's an `update.packages()` function that you can use to do this, but it's probably easier to stick with the RStudio tool. In the packages panel, click on the "Update Packages" button. This will bring up a window that looks like the one shown in Figure 2.10. 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



☐	lattice	0.20–23	0.20–24
☐	lavaan	0.5–14	0.5–15
☐	lme4	1.0–4	1.0–5
☐	lmtest	0.9–32	0.9–33
☐	mapproj	1.2–1	1.2–2
☐	maps	2.3–3	2.3–6
☐	maptools	0.8–26	0.8–27
☐	markdown	0.6.3	0.6.4
☐	Matrix	1.0–14	1.1–2
☐	mgcv	1.7–26	1.7–28
☐	minqa	1.2.1	1.2.2

Figure 2.10: The RStudio dialog box for updating packages

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.

2.15.7 What packages does this book use?

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

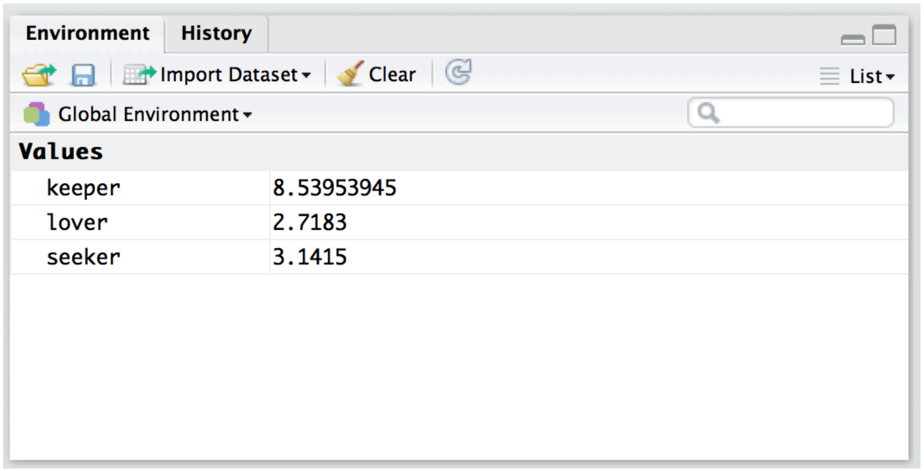
- `lsr`. This is the *Learning Statistics with R* package that accompanies this book. It doesn't have a lot of interesting high-powered tools: it's just a small collection of handy little things that I think can be useful to novice users. As you get more comfortable with R this package should start to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

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

2.16 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 2.14. 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.

2.16.1 Listing the contents of the workspace



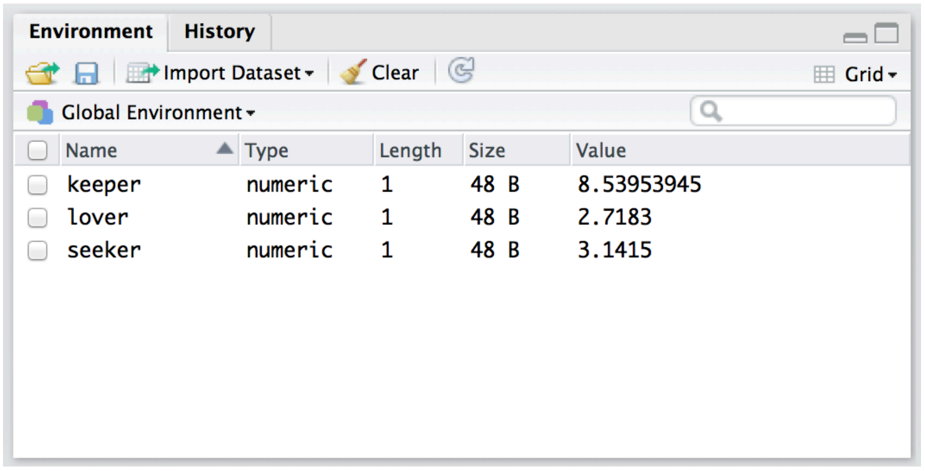
keeper	8.53953945
lover	2.7183
seeker	3.1415

Figure 2.11: The RStudio Environment panel shows you the contents of the workspace. The view shown above is the list view. To switch to the grid view, click on the menu item on the top right that currently reads list. Select grid from the dropdown menu, and then it will switch to a view like the one shown in the other workspace figure

The first thing that you need to know how to do is examine the contents of the workspace. If you're using RStudio, you will probably find that the easiest way to do this is to use the "Environment" panel in the top right hand corner. Click on that, and you'll see a list that looks very much like the one shown in Figures 2.11 and 2.12. If you're using the commmand line, then the `objects()` function may come in handy:

```
objects()
```

```
##  [1] "any.sales.this.month" "berkeley"             "berkeley.small"
##  [4] "coef"                 "days.per.month"       "february.sales"
##  [7] "greeting"             "is.the.Party.correct" "keeper"
```



Name	Type	Length	Size	Value
keeper	numeric	1	48 B	8.53953945
lover	numeric	1	48 B	2.7183
seeker	numeric	1	48 B	3.1415

Figure 2.12: The RStudio "Environment" panel shows you the contents of the workspace. Compare this "grid" view to the "list" earlier

```
## [10] "lover"                "months"               "profit"
## [13] "projecthome"          "revenue"              "royalty"
## [16] "sales"                "sales.by.month"       "seeker"
## [19] "simpson"              "stock.levels"         "x"
## [22] "xlu"
```

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

```
library(lsr)
```

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

```
who()
```

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

As you can see, the `who()` function lists all the variables and provides some basic information about what kind of variable each one is and how many elements it contains. Personally, I find this output much easier more useful than the very compact output of the `objects()` function, but less overwhelming than the extremely verbose `ls.str()` function. Throughout this book you'll see me using the `who()` function a lot. You don't have to use it yourself: in fact, I suspect you'll find it easier to look at the RStudio environment panel. But for the purposes of writing a textbook I found it handy to have a nice text based description: otherwise there would be about another 100 or so screenshots added to the book.[37]

2.16.2 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

[37]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!

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 2.18, 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 2.12), 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.[38] In any case, if you click "yes", that variable will disappear from the workspace: it will no longer appear in the environment panel, and it won't show up when you use the `who()` command.

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

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

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

```
who()
```

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

[38]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.

```
##    sales                 numeric      1
##    sales.by.month        numeric      12
##    simpson               matrix       6 x 5
##    stock.levels          character    12
##    x                     logical      3
##    xlu                   numeric      1
```

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

2.17 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 2.17.1. 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 2.17.2. 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.

2.17.1 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[39] 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:

[39]Well, the partition, technically.

Table 2.4: Basic arithmetic operations in R. These five operators are used very frequently throughout the text, so it's important to be familiar with them at the outset.

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

```
C:\Users\danRbook\LSR.pdf
```

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

```
/Users/dan/Rbook/LSR.pdf
```

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

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

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

```
~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.

2.17.2 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 2.17.4. 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.[40]

2.17.3 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 **??**). 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:

[40]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.

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

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

2.17.4 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 2.13 – 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 2.13. 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

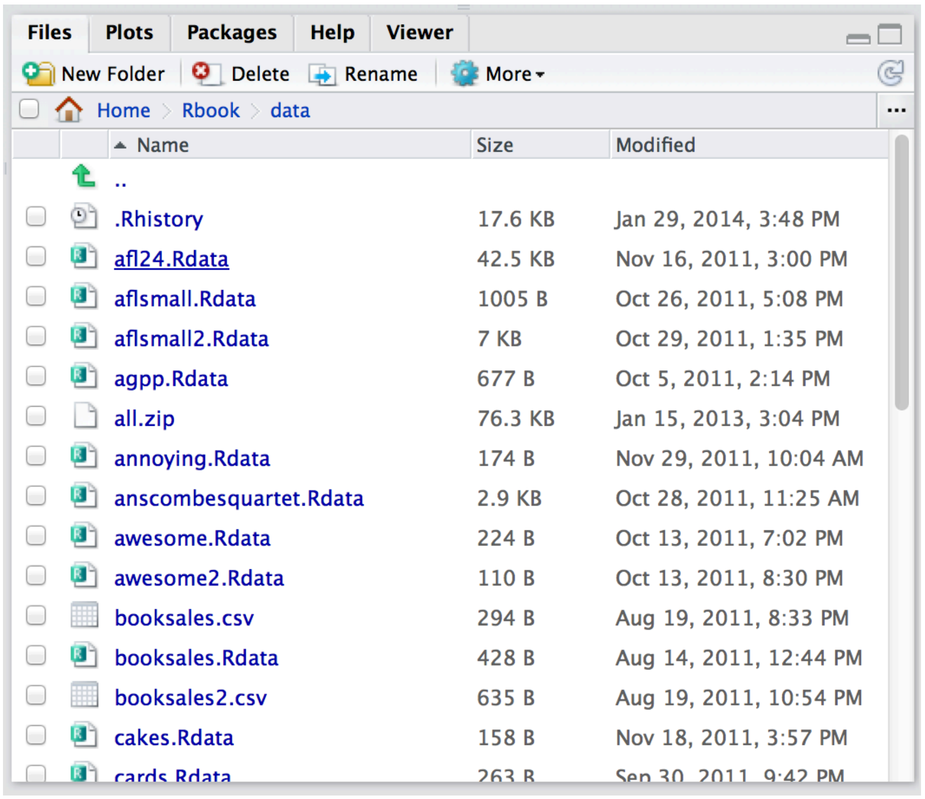
```
/Users/dan/Rbook/data
```

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



Name	Size	Modified
..		
.Rhistory	17.6 KB	Jan 29, 2014, 3:48 PM
afl24.Rdata	42.5 KB	Nov 16, 2011, 3:00 PM
aflsmall.Rdata	1005 B	Oct 26, 2011, 5:08 PM
aflsmall2.Rdata	7 KB	Oct 29, 2011, 1:35 PM
agpp.Rdata	677 B	Oct 5, 2011, 2:14 PM
all.zip	76.3 KB	Jan 15, 2013, 3:04 PM
annoying.Rdata	174 B	Nov 29, 2011, 10:04 AM
anscombesquartet.Rdata	2.9 KB	Oct 28, 2011, 11:25 AM
awesome.Rdata	224 B	Oct 13, 2011, 7:02 PM
awesome2.Rdata	110 B	Oct 13, 2011, 8:30 PM
booksales.csv	294 B	Aug 19, 2011, 8:33 PM
booksales.Rdata	428 B	Aug 14, 2011, 12:44 PM
booksales2.csv	635 B	Aug 19, 2011, 10:54 PM
cakes.Rdata	158 B	Nov 18, 2011, 3:57 PM
cards.Rdata	263 B	Sep 30, 2011, 9:42 PM

Figure 2.13: 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.

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.

2.18 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 **??**.

There are also several other types of file that R makes use of,[41] but they're not really all that central to our interests. There are also several other kinds of data file that you might want to import into R. For instance, you might want to open Microsoft Excel spreadsheets (.xlsx files), or data files that have been saved in the native file formats for other statistics software, such as SPSS, SAS, Minitab, Stata or Systat. Finally, you might have to handle databases. R tries hard to play nicely with other software, so it has tools that let you open and work with any of these and many others. I'll discuss some of these other possibilities elsewhere in this book (Section **??**), 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.

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

2.18.1 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 2.17.2), the output referred to a file called booksales.Rdata. Let's say I want to load the data from this file into my workspace. The way I do this is with the `load()` function. There are two arguments to this function, but the only one we're interested in is

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

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

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

but this is pretty lengthy. Given that the working directory (remember, we changed the directory at the end of Section 2.17.4) 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.

2.18.2 Loading workspace files using RStudio

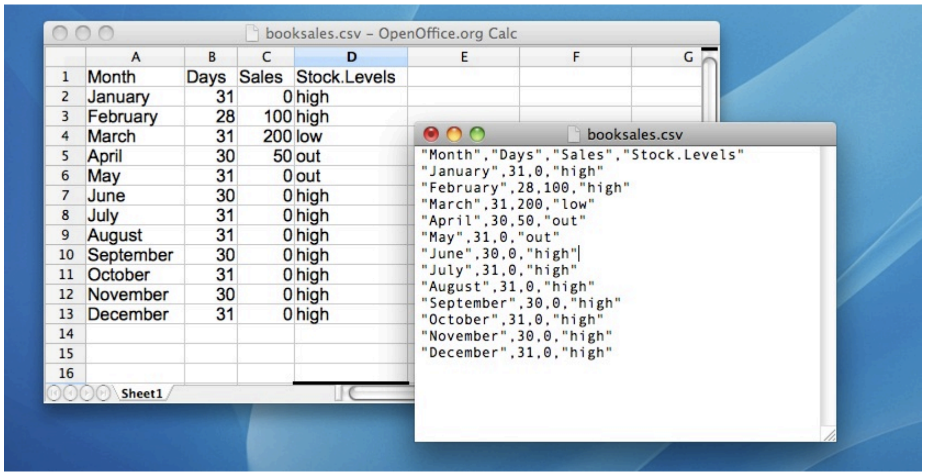
Okay, so how do we open an .Rdata file using the RStudio file panel? It's terribly simple. First, use the file panel to find the folder that contains the file you want to load. If you look at Figure 2.13, you can see that there are several .Rdata files listed. Let's say I want to load the `booksales.Rdata` 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.

2.18.3 Importing data from CSV files using loadingcsv

One quite commonly used data format is the humble "comma separated value" file, also called a CSV file, and usually bearing the file extension .csv. CSV files are just plain old-fashioned text files, and what they store is basically just a table of data. This is illustrated in Figure 2.14, 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.



	A	B	C	D	E	F	G
1	Month	Days	Sales	Stock.Levels			
2	January	31	0	high			
3	February	28	100	high			
4	March	31	200	low			
5	April	30	50	out			
6	May	31	0	out			
7	June	30	0	high			
8	July	31	0	high			
9	August	31	0	high			
10	September	30	0	high			
11	October	31	0	high			
12	November	30	0	high			
13	December	31	0	high			
14							
15							
16							

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

Figure 2.14: 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.

If RStudio were not available to you, the easiest way to open this file would be to use the `read.csv()` function.[42] This function is pretty flexible, and I'll talk a lot more about it's capabilities in Section **??** for more details, but for now there's only two arguments to the function that I'll mention:

[42]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.

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

```
print( books )
```

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

Clearly, it's worked, but the format of this output is a bit unfamiliar. We haven't seen anything like this before. What you're looking at is a *data frame*, which is a very important kind of variable in R, and one I'll discuss in Section 2.21. For now, let's just be happy that we imported the data and that it looks about right.

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

2.18.4 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 2.15. I'm assuming that you're familiar with your own computer, so you should have no problem finding the CSV file that you want to import! Find the one you want, then click on the "Open" button. When you do this, you'll see a window that looks like the one in Figure 2.16.

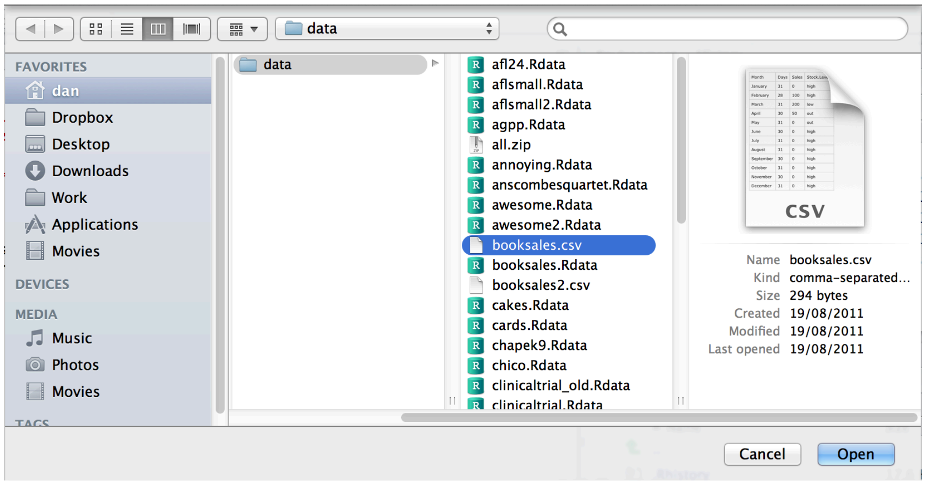
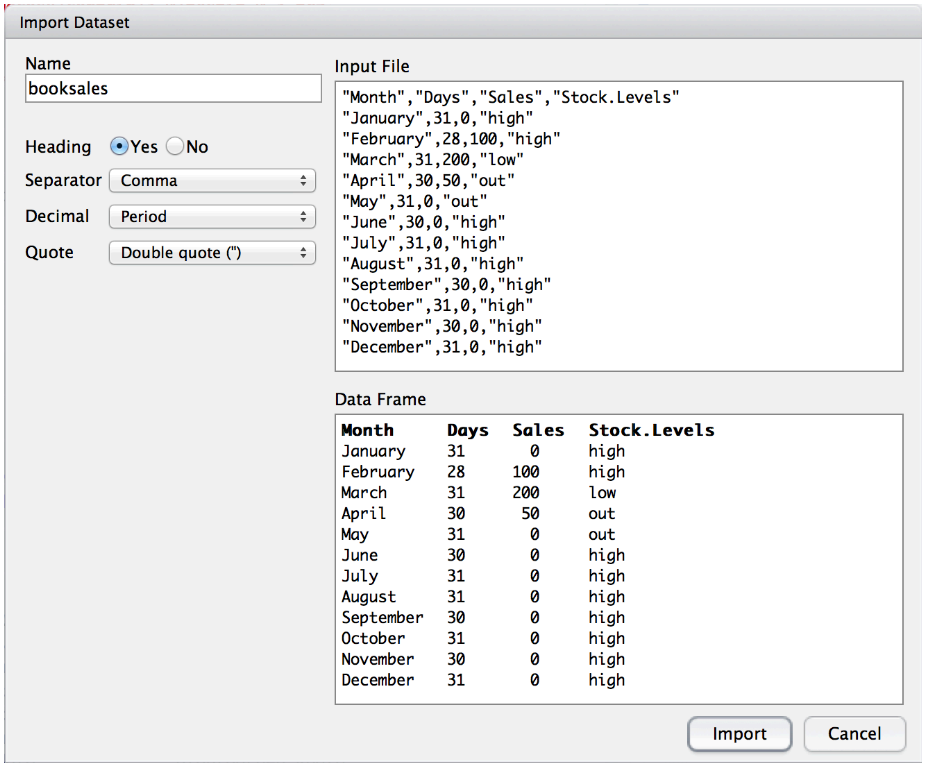


Figure 2.15: A dialog box on a Mac asking you to select the CSV file R should try to import. Mac users will recognise this immediately: it's the usual way in which a Mac asks you to find a file. Windows users won't see this: they'll see the usual explorer window that Windows always gives you when it wants you to select a file.

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".



Name: booksales

Heading [✓ Yes] No

Separator: Comma

Decimal: Period

Quote: Double quote (")

Input File

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

Data Frame

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

Figure 2.16: The RStudio window for importing a CSV file into R

- 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.

2.18.5 Saving a workspace file using `save`

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

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

Since `file` is the first argument, you can shorten this to `save.image("myfile.Rdata")`; and if you want to save to a different directory, then (as always) you need to be more explicit about specifying the path to the file, just as we discussed in Section 2.17. 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
```

2.18.6 Saving a workspace file using RStudio

RStudio allows you to save the workspace pretty easily. In the environment panel (Figures 2.11 and 2.12) 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.[44] 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 2.15). 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.

[44] 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 **??**.

2.18.7 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 **??**.

2.19 Useful things to know about variables

In Chapter 2 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.

2.19.1 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*.

2.19.2 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)`.

2.19.3 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.[45] 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:

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

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

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

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

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

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

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

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

1. The ***class*** of a variable is a "high level" classification, and it captures psychologically (or statistically) meaningful distinctions. For instance `"2011-09-12"` and `"my birthday"` are both text strings, but there's an important difference between the two: one of them is a date. So it would be nice if we could get R to recognise that `"2011-09-12"` is a date, and allow us to do things like add or subtract from it. The class of a variable is what R uses to keep track of things like that. Because the class of a variable is critical for determining what R can or can't do with it, the `class()` function is very handy.
2. The ***mode*** of a variable refers to the format of the information that the variable stores. It tells you whether R has stored text data or numeric data, for instance, which is kind of useful, but it only makes these "simple" distinctions. It can be useful to know about, but it's not the main thing we care about. So I'm not going to use the `mode()` function very much.[46]
3. The ***type*** of a variable is a very low level classification. We won't use it in this book, but (for those of you that care about these details) this is where you can see the distinction between integer data, double precision

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

numeric, etc. Almost none of you actually will care about this, so I'm not even going to bother demonstrating the `typeof()` function.

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

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

```
## [1] "character"
```

```
x <- TRUE     # x is logical
class(x)
```

```
## [1] "logical"
```

```
x <- 100     # x is a number
class(x)
```

```
## [1] "numeric"
```

Exciting, no?

2.20 Factors

Okay, it's time to start introducing some of the data types that are somewhat more specific to statistics. If you remember back to Chapter 1.6, when we assign numbers to possible outcomes, these numbers can mean quite different things depending on what kind of variable we are attempting to measure. In particular, we commonly make the distinction between *nominal*, *ordinal*, *interval* and *ratio* scale data. How do we capture this distinction in R? Currently, we only seem to have a single numeric data type. That's probably not going to be enough, is it?

A little thought suggests that the numeric variable class in R is perfectly suited for capturing ratio scale data. For instance, if I were to measure response time (RT) for five different events, I could store the data in R like this:

```
RT <- c(342, 401, 590, 391, 554)
```

where the data here are measured in milliseconds, as is conventional in the psychological literature. It's perfectly sensible to talk about "twice the response time", $2 \times \text{RT}$, or the "response time plus 1 second", $\text{RT} + 1000$, and so both of the following are perfectly reasonable things for R to do:

```
2 * RT
```

```
## [1]  684  802 1180  782 1108
```

```
RT + 1000
```

```
## [1] 1342 1401 1590 1391 1554
```

And to a lesser extent, the "numeric" class is okay for interval scale data, as long as we remember that multiplication and division aren't terribly interesting for these sorts of variables. That is, if my IQ score is 110 and yours is 120, it's perfectly okay to say that you're 10 IQ points smarter than me[47], but it's not okay to say that I'm only 92% as smart as you are, because intelligence doesn't have a natural zero.[48] We might even be willing to tolerate the use of numeric variables to represent ordinal scale variables, such as those that you typically get when you ask people to rank order items (e.g., like we do in Australian elections), though as we will see R actually has a built in tool for representing ordinal data (see Section **??**) However, when it comes to nominal scale data, it becomes completely unacceptable, because almost all of the "usual" rules for what you're allowed to do with numbers don't apply to nominal scale data. It is for this reason that R has ***factors***.

2.20.1 Introducing factors

Suppose, I was doing a study in which people could belong to one of three different treatment conditions. Each group of people were asked to complete the same task, but each group received different instructions. Not surprisingly, I might want to have a variable that keeps track of what group people were in. So I could type in something like this

[47] Taking all the usual caveats that attach to IQ measurement as a given, of course.

[48] Or, more precisely, we don't know how to measure it. Arguably, a rock has zero intelligence. But it doesn't make sense to say that the IQ of a rock is 0 in the same way that we can say that the average human has an IQ of 100. And without knowing what the IQ value is that corresponds to a literal absence of any capacity to think, reason or learn, then we really can't multiply or divide IQ scores and expect a meaningful answer.

```
group <- c(1,1,1,2,2,2,3,3,3)
```

so that `group[i]` contains the group membership of the `i`-th person in my study. Clearly, this is numeric data, but equally obviously this is a nominal scale variable. There's no sense in which "group 1" plus "group 2" equals "group 3", but nevertheless if I try to do that, R won't stop me because it doesn't know any better:

```
group + 2
```

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

Apparently R seems to think that it's allowed to invent "group 4" and "group 5", even though they didn't actually exist. Unfortunately, R is too stupid to know any better: it thinks that `3` is an ordinary number in this context, so it sees no problem in calculating `3 + 2`. But since *we're* not that stupid, we'd like to stop R from doing this. We can do so by instructing R to treat `group` as a factor. This is easy to do using the `as.factor()` function.[49]

```
group <- as.factor(group)
group
```

```
## [1] 1 1 1 2 2 2 3 3 3
## Levels: 1 2 3
```

It looks more or less the same as before (though it's not immediately obvious what all that `Levels` rubbish is about), but if we ask R to tell us what the class of the `group` variable is now, it's clear that it has done what we asked:

```
class(group)
```

```
## [1] "factor"
```

Neat. Better yet, now that I've converted `group` to a factor, look what happens when I try to add 2 to it:

```
group + 2
```

```
## Warning in Ops.factor(group, 2): '+' not meaningful for factors
```

```
## [1] NA NA NA NA NA NA NA NA NA
```

This time even R is smart enough to know that I'm being an idiot, so it tells me off and then produces a vector of missing values. (i.e., `NA`: see Section 2.19.1).

[49]Once again, this is an example of *coercing* a variable from one class to another. I'll talk about coercion in more detail in Section **??**.

2.20.2 Labelling the factor levels

I have a confession to make. My memory is not infinite in capacity; and it seems to be getting worse as I get older. So it kind of annoys me when I get data sets where there's a nominal scale variable called `gender`, with two levels corresponding to males and females. But when I go to print out the variable I get something like this:

```
gender
```

```
## [1] 1 1 1 1 1 2 2 2 2
## Levels: 1 2
```

Okaaaay. That's not helpful at all, and it makes me very sad. Which number corresponds to the males and which one corresponds to the females? Wouldn't it be nice if R could actually keep track of this? It's way too hard to remember which number corresponds to which gender. To fix this problem what we need to do is assign meaningful labels to the different *levels* of each factor. We can do that like this:

```
levels(group) <- c("group 1", "group 2", "group 3")
print(group)
```

```
## [1] group 1 group 1 group 1 group 2 group 2 group 2 group 3 group 3 group 3
## Levels: group 1 group 2 group 3
```

```
levels(gender) <- c("male", "female")
print(gender)
```

```
## [1] male   male   male   male   male   female female female female
## Levels: male female
```

That's much easier on the eye.

2.20.3 Moving on...

Factors are very useful things, and we'll use them a lot in this book: they're *the* main way to represent a nominal scale variable. And there are lots of nominal scale variables out there. I'll talk more about factors in Section **??**, but for now you know enough to be able to get started.

2.21 Data frames

It's now time to go back and deal with the somewhat confusing thing that happened in Section 2.18.3 when we tried to open up a CSV file. Apparently we succeeded in loading the data, but it came to us in a very odd looking format. At the time, I told you that this was a ***data frame***. Now I'd better explain what that means.

2.21.1 Introducing data frames

In order to understand why R has created this funny thing called a data frame, it helps to try to see what problem it solves. So let's go back to the little scenario that I used when introducing factors in Section 2.20. In that section I recorded the `group` and `gender` for all 9 participants in my study. Let's also suppose I recorded their ages and their `score` on "Dan's Terribly Exciting Psychological Test":

```
age <- c(17, 19, 21, 37, 18, 19, 47, 18, 19)
score <- c(12, 10, 11, 15, 16, 14, 25, 21, 29)
```

Assuming no other variables are in the workspace, if I type `who()` I get this:

```
who()
```

```
##    -- Name --              -- Class --   -- Size --
##    age                     numeric       9
##    any.sales.this.month    logical       12
##    berkeley                data.frame    39 x 3
##    berkeley.small          data.frame    46 x 2
##    coef                    numeric       2
##    days.per.month          numeric       12
##    february.sales          numeric       1
##    gender                  factor        9
##    greeting                character     1
##    group                   factor        9
##    is.the.Party.correct    logical       1
##    months                  character     12
##    projecthome             character     1
##    revenue                 numeric       1
##    royalty                 numeric       1
##    sales                   numeric       1
##    sales.by.month          numeric       12
##    score                   numeric       9
##    simpson                 matrix        6 x 5
##    stock.levels            character     12
##    xlu                     numeric       1
```

So there are four variables in the workspace, `age`, `gender`, `group` and `score`. And it just so happens that all four of them are the same size (i.e., they're all vectors with 9 elements). Aaaand it just so happens that `age[1]` corresponds to the age of the first person, and `gender[1]` is the gender of that very same person, etc. In other words, you and I both know that all four of these variables correspond to the *same* data set, and all four of them are organised in exactly the same way.

However, R *doesn't* know this! As far as it's concerned, there's no reason why the `age` variable has to be the same length as the `gender` variable; and there's no particular reason to think that `age[1]` has any special relationship to `gender[1]` any more than it has a special relationship to `gender[4]`. In other words, when we store everything in separate variables like this, R doesn't know anything about the relationships between things. It doesn't even really know that these variables actually refer to a proper data set. The data frame fixes this: if we store our variables inside a data frame, we're telling R to treat these variables as a single, fairly coherent data set.

To see how they do this, let's create one. So how do we create a data frame? One way we've already seen: if we import our data from a CSV file, R will store it as a data frame. A second way is to create it directly from some existing variables using the `data.frame()` function. All you have to do is type a list of variables that you want to include in the data frame. The output of a `data.frame()` command is, well, a data frame. So, if I want to store all four variables from my experiment in a data frame called `expt` I can do so like this:

```
expt <- data.frame ( age, gender, group, score )
expt
```

```
##   age gender   group score
## 1  17   male group 1    12
## 2  19   male group 1    10
## 3  21   male group 1    11
## 4  37   male group 2    15
## 5  18   male group 2    16
## 6  19 female group 2    14
## 7  47 female group 3    25
## 8  18 female group 3    21
## 9  19 female group 3    29
```

Note that `expt` is a completely self-contained variable. Once you've created it, it no longer depends on the original variables from which it was constructed. That is, if we make changes to the original `age` variable, it will *not* lead to any changes to the age data stored in `expt`.

2.21.2 Pulling out the contents of the data frame using `$`

At this point, our workspace contains only the one variable, a data frame called `expt`. But as we can see when we told R to print the variable out, this data frame contains 4 variables, each of which has 9 observations. So how do we get this information out again? After all, there's no point in storing information if you don't use it, and there's no way to use information if you can't access it. So let's talk a bit about how to pull information out of a data frame.

The first thing we might want to do is pull out one of our stored variables, let's say `score`. One thing you might try to do is ignore the fact that `score` is locked up inside the `expt` data frame. For instance, you might try to print it out like this:

```
score
```

```
## Error in eval(expr, envir, enclos): object 'score' not found
```

This doesn't work, because R doesn't go "peeking" inside the data frame unless you explicitly tell it to do so. There's actually a very good reason for this, which I'll explain in a moment, but for now let's just assume R knows what it's doing. How do we tell R to look inside the data frame? As is always the case with R there are several ways. The simplest way is to use the `$` operator to extract the variable you're interested in, like this:

```
expt$score
```

```
## [1] 12 10 11 15 16 14 25 21 29
```

2.21.3 Getting information about a data frame

One problem that sometimes comes up in practice is that you forget what you called all your variables. Normally you might try to type `objects()` or `who()`, but neither of those commands will tell you what the names are for those variables inside a data frame! One way is to ask R to tell you what the *names* of all the variables stored in the data frame are, which you can do using the `names()` function:

```
names(expt)
```

```
## [1] "age"    "gender" "group"  "score"
```

An alternative method is to use the `who()` function, as long as you tell it to look at the variables inside data frames. If you set `expand = TRUE` then it will not only list the variables in the workspace, but it will "expand" any data frames that you've got in the workspace, so that you can see what they look like. That is:

```
who(expand = TRUE)
```

```
##    -- Name --             -- Class --    -- Size --
##    any.sales.this.month   logical        12
##    berkeley               data.frame     39 x 3
##     $women.apply          numeric        39
##     $total.admit          numeric        39
##     $number.apply         numeric        39
##    berkeley.small         data.frame     46 x 2
##     $women.apply          numeric        46
##     $total.admit          numeric        46
##    coef                   numeric        2
##    days.per.month         numeric        12
##    expt                   data.frame     9 x 4
##     $age                  numeric        9
##     $gender               factor         9
##     $group                factor         9
##     $score                numeric        9
##    february.sales         numeric        1
##    greeting               character      1
##    is.the.Party.correct   logical        1
##    months                 character      12
##    projecthome            character      1
##    revenue                numeric        1
##    royalty                numeric        1
##    sales                  numeric        1
##    sales.by.month         numeric        12
##    simpson                matrix         6 x 5
##    stock.levels           character      12
##    xlu                    numeric        1
```

or, since `expand` is the first argument in the `who()` function you can just type `who(TRUE)`. I'll do that a lot in this book.

2.21.4 Looking for more on data frames?

There's a lot more that can be said about data frames: they're fairly complicated beasts, and the longer you use R the more important it is to make sure you really understand them. We'll talk a lot more about them in Chapter **??**.

2.22 Lists

The next kind of data I want to mention are ***lists***. Lists are an extremely fundamental data structure in R, and as you start making the transition from a novice to a savvy R user you will use lists all the time. I don't use lists very often in this book – not directly – but most of the advanced data structures in R are built from lists (e.g., data frames are actually a specific type of list). Because lists are so important to how R stores things, it's useful to have a basic understanding of them. Okay, so what is a list, exactly? Like data frames, lists are just "collections of variables." However, unlike data frames – which are basically supposed to look like a nice "rectangular" table of data – there are no constraints on what kinds of variables we include, and no requirement that the variables have any particular relationship to one another. In order to understand what this actually *means*, the best thing to do is create a list, which we can do using the `list()` function. If I type this as my command:

```
Dan <- list( age = 34,
            nerd = TRUE,
            parents = c("Joe","Liz")
)
```

R creates a new list variable called `Dan`, which is a bundle of three different variables: `age`, `nerd` and `parents`. Notice, that the `parents` variable is longer than the others. This is perfectly acceptable for a list, but it wouldn't be for a data frame. If we now print out the variable, you can see the way that R stores the list:

```
print( Dan )
```

```
## $age
## [1] 34
##
## $nerd
## [1] TRUE
##
## $parents
## [1] "Joe" "Liz"
```

As you might have guessed from those `$` symbols everywhere, the variables are stored in exactly the same way that they are for a data frame (again, this is not surprising: data frames *are* a type of list). So you will (I hope) be entirely unsurprised and probably quite bored when I tell you that you can extract the variables from the list using the `$` operator, like so:

```
Dan$nerd
```

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

If you need to add new entries to the list, the easiest way to do so is to again use `$`, as the following example illustrates. If I type a command like this

```
Dan$children <- "Alex"
```

then R creates a new entry to the end of the list called `children`, and assigns it a value of `"Alex"`. If I were now to `print()` this list out, you'd see a new entry at the bottom of the printout. Finally, it's actually possible for lists to contain other lists, so it's quite possible that I would end up using a command like `Dan$children$age` to find out how old my son is. Or I could try to remember it myself I suppose.

2.23 Formulas

The last kind of variable that I want to introduce before finally being able to start talking about statistics is the ***formula***. Formulas were originally introduced into R as a convenient way to specify a particular type of statistical model (see Chapter **??**) but they're such handy things that they've spread. Formulas are now used in a lot of different contexts, so it makes sense to introduce them early.

Stated simply, a formula object is a variable, but it's a special type of variable that specifies a relationship between other variables. A formula is specified using the "tilde operator" `~`. A very simple example of a formula is shown below:[50]

```
formula1 <- out ~ pred
formula1
```

```
## out ~ pred
```

The *precise* meaning of this formula depends on exactly what you want to do with it, but in broad terms it means "the `out` (outcome) variable, analysed in terms of the `pred` (predictor) variable". That said, although the simplest and most common form of a formula uses the "one variable on the left, one variable on the right" format, there are others. For instance, the following examples are all reasonably common

[50]Note that, when I write out the formula, R doesn't check to see if the `out` and `pred` variables actually exist: it's only later on when you try to use the formula for something that this happens.

```
formula2 <-  out ~ pred1 + pred2   # more than one variable on the right
formula3 <-  out ~ pred1 * pred2   # different relationship between predictors
formula4 <-  ~ var1 + var2         # a 'one-sided' formula
```

and there are many more variants besides. Formulas are pretty flexible things, and so different functions will make use of different formats, depending on what the function is intended to do.

2.24 Generic functions

There's one really important thing that I omitted when I discussed functions earlier on in Section 2.5, and that's the concept of a ***generic function***. The two most notable examples that you'll see in the next few chapters are `summary()` and `plot()`, although you've already seen an example of one working behind the scenes, and that's the `print()` function. The thing that makes generics different from the other functions is that their behaviour changes, often quite dramatically, depending on the `class()` of the input you give it. The easiest way to explain the concept is with an example. With that in mind, lets take a closer look at what the `print()` function actually does. I'll do this by creating a formula, and printing it out in a few different ways. First, let's stick with what we know:

```
my.formula <- blah ~ blah.blah    # create a variable of class "formula"
print( my.formula )               # print it out using the generic print() function
```

```
## blah ~ blah.blah
```

So far, there's nothing very surprising here. But there's actually a lot going on behind the scenes here. When I type `print( my.formula )`, what actually happens is the `print()` function checks the class of the `my.formula` variable. When the function discovers that the variable it's been given is a formula, it goes looking for a function called `print.formula()`, and then delegates the whole business of printing out the variable to the `print.formula()` function.[51] For what it's worth, the name for a "dedicated" function like `print.formula()` that exists only to be a special case of a generic function like `print()` is a ***method***, and the name for the process in which the generic function passes off all the hard work onto a method is called ***method dispatch***. You won't need to understand

[51]For readers with a programming background: what I'm describing is the very basics of how S3 methods work. However, you should be aware that R has two entirely distinct systems for doing object oriented programming, known as S3 and S4. Of the two, S3 is simpler and more informal, whereas S4 supports all the stuff that you might expect of a fully object oriented language. Most of the generics we'll run into in this book use the S3 system, which is convenient for me because I'm still trying to figure out S4.

the details at all for this book, but you do need to know the gist of it; if only because a lot of the functions we'll use are actually generics. Anyway, to help expose a little more of the workings to you, let's bypass the `print()` function entirely and call the formula method directly:

```
print.formula( my.formula )       # print it out using the print.formula() method

## Appears to be deprecated
```

There's no difference in the output at all. But this shouldn't surprise you because it was actually the `print.formula()` method that was doing all the hard work in the first place. The `print()` function itself is a lazy bastard that doesn't do anything other than select which of the methods is going to do the actual printing.

Okay, fair enough, but you might be wondering what would have happened if `print.formula()` didn't exist? That is, what happens if there isn't a specific method defined for the class of variable that you're using? In that case, the generic function passes off the hard work to a "default" method, whose name in this case would be `print.default()`. Let's see what happens if we bypass the `print()` formula, and try to print out `my.formula` using the `print.default()` function:

```
print.default( my.formula )      # print it out using the print.default() method
```

```
## blah ~ blah.blah
## attr(,"class")
## [1] "formula"
## attr(,".Environment")
## <environment: R_GlobalEnv>
```

Hm. You can kind of see that it is trying to print out the same formula, but there's a bunch of ugly low-level details that have also turned up on screen. This is because the `print.default()` method doesn't know anything about formulas, and doesn't know that it's supposed to be hiding the obnoxious internal gibberish that R produces sometimes.

At this stage, this is about as much as we need to know about generic functions and their methods. In fact, you can get through the entire book without learning any more about them than this, so it's probably a good idea to end this discussion here.

2.25 Getting help

The very last topic I want to mention in this chapter is where to go to find help. Obviously, I've tried to make this book as helpful as possible, but it's not even close to being a comprehensive guide, and there's thousands of things it doesn't cover. So where should you go for help?

2.25.1 How to read the help documentation

I have somewhat mixed feelings about the help documentation in R. On the plus side, there's a lot of it, and it's very thorough. On the minus side, there's a lot of it, and it's very thorough. There's so much help documentation that it sometimes doesn't help, and most of it is written with an advanced user in mind. Often it feels like most of the help files work on the assumption that the reader already understands everything about R except for the specific topic that it's providing help for. What that means is that, once you've been using R for a long time and are beginning to get a feel for how to use it, the help documentation is awesome. These days, I find myself really liking the help files (most of them anyway). But when I first started using R I found it very dense.

To some extent, there's not much I can do to help you with this. You just have to work at it yourself; once you're moving away from being a pure beginner and are becoming a skilled user, you'll start finding the help documentation more and more helpful. In the meantime, I'll help as much as I can by trying to explain to you what you're looking at when you open a help file. To that end, let's look at the help documentation for the `load()` function. To do so, I type either of the following:

```
?load
help("load")
```

When I do that, R goes looking for the help file for the "load" topic. If it finds one, Rstudio takes it and displays it in the help panel. Alternatively, you can try a fuzzy search for a help topic

```
??load
help.search("load")
```

This will bring up a list of possible topics that you might want to follow up in. Regardless, at some point you'll find yourself looking at an actual help file. And when you do, you'll see there's a quite a lot of stuff written down there, and it comes in a pretty standardised format. So let's go through it slowly, using the "`load`" topic as our example. Firstly, at the very top we see this:

> oad {base}
>
> R Documentation
>
> Reload Saved Datasets
>
> Description
>
> Reload datasets written with the function save.

Fairly straightforward. The next section describes how the function is used:

> sage

In this instance, the usage section is actually pretty readable. It's telling you that there are two arguments to the `load()` function: the first one is called `file`, and the second one is called `envir`. It's also telling you that there is a default value for the envir argument; so if the user doesn't specify what the value of envir should be, then R will assume that `envir = parent.frame()`. In contrast, the file argument has no default value at all, so the user must specify a value for it. So in one sense, this section is very straightforward.

The problem, of course, is that you don't know what the `parent.frame()` function actually does, so it's hard for you to know what the `envir = parent.frame()` bit is all about. What you could do is then go look up the help documents for the `parent.frame()` function (and sometimes that's actually a good idea), but often you'll find that the help documents for those functions are just as dense (if not more dense) than the help file that you're currently reading. As an alternative, my general approach when faced with something like this is to skim over it, see if I can make any sense of it. If so, great. If not, I find that the best thing to do is ignore it. In fact, the first time I read the help file for the load() function, I had no idea what any of the `envir` related stuff was about. But fortunately I didn't have to: the default setting here (i.e., `envir = parent.frame()`) is actually the thing you want in about 99% of cases, so it's safe to ignore it.

Basically, what I'm trying to say is: don't let the scary, incomprehensible parts of the help file intimidate you. Especially because there's often some parts of the help file that will make sense. Of course, I guarantee you that sometimes this strategy will lead you to make mistakes… often embarrassing mistakes. But it's still better than getting paralysed with fear.

So, let's continue on. The next part of the help documentation discusses each of the arguments, and what they're supposed to do:

> rguments
>
> file
>
> a (readable binary-mode) connection or a character string giving the name of the file to load (when tilde expansion is done).
>
> envir
>
> the environment where the data should be loaded.
>
> verbose
>
> should item names be printed during loading?

Okay, so what this is telling us is that the `file` argument needs to be a string (i.e., text data) which tells R the name of the file to load. It also seems to be hinting that there's other possibilities too (e.g., a "binary mode connection"), and you probably aren't quite sure what "tilde expansion" means[52]. But overall, the meaning is pretty clear.

Turning to the `envir` argument, it's now a little clearer what the Usage section was babbling about. The `envir` argument specifies the name of an environment (see Section 4.3 if you've forgotten what environments are) into which R should place the variables when it loads the file. Almost always, this is a no-brainer: you want R to load the data into the same damn environment in which you're invoking the `load()` command. That is, if you're typing `load()` at the R prompt, then you want the data to be loaded into your workspace (i.e., the global environment). But if you're writing your own function that needs to load some data, you want the data to be loaded inside that function's private workspace. And in fact, that's exactly what the `parent.frame()` thing is all about. It's telling the `load()` function to send the data to the same place that the `load()` command itself was coming from. As it turns out, if we'd just ignored the envir bit we would have been totally safe. Which is nice to know.

Moving on, next up we get a detailed description of what the function actually does:

> etails
>
> load can load R objects saved in the current or any earlier format. It can read a compressed file (see save) directly from a file or from a suitable connection (including a call to url).
>
> A not-open connection will be opened in mode "rb" and closed after use. Any connection other than a gzfile or gzcon connection will be wrapped in gzcon to allow compressed saves to be handled: note that this leaves the connection in an altered state (in particular, binary-only), and that it needs to be closed explicitly (it will not be garbage-collected).
>
> Only R objects saved in the current format (used since R 1.4.0) can be read from a connection. If no input is available on a connection a warning will be given, but any input not in the current format will result in a error.
>
> Loading from an earlier version will give a warning about the 'magic number': magic numbers 1971:1977 are from R < 0.99.0, and RD[ABX]1 from R 0.99.0 to R 1.3.1. These are all obsolete, and you are strongly recommended to re-save such files in a current format.
>
> The verbose argument is mainly intended for debugging. If it is TRUE, then as objects from the file are loaded, their names will be printed to the console. If verbose is set to an integer value greater than one, additional

[52]It's extremely simple, by the way. We discussed it in Section 4.4, though I didn't call it by that name. Tilde expansion is the thing where R recognises that, in the context of specifying a file location, the tilde symbol ~ corresponds to the user home directory (e.g., /Users/dan/).

> names corresponding to attributes and other parts of individual objects will also be printed. Larger values will print names to a greater depth.
>
> Objects can be saved with references to namespaces, usually as part of the environment of a function or formula. Such objects can be loaded even if the namespace is not available: it is replaced by a reference to the global environment with a warning. The warning identifies the first object with such a reference (but there may be more than one).

Then it tells you what the output value of the function is:

> alue
>
> A character vector of the names of objects created, invisibly.

This is usually a bit more interesting, but since the `load()` function is mainly used to load variables into the workspace rather than to return a value, it's no surprise that this doesn't do much or say much. Moving on, we sometimes see a few additional sections in the help file, which can be different depending on what the function is:

> arning
>
> Saved R objects are binary files, even those saved with ascii = TRUE, so ensure that they are transferred without conversion of end of line markers. load tries to detect such a conversion and gives an informative error message.
>
> load(<file>) replaces all existing objects with the same names in the current environment (typically your workspace, .GlobalEnv) and hence potentially overwrites important data. It is considerably safer to use envir = to load into a different environment, or to attach(file) which load()s into a new entry in the search path.
>
> Note
>
> file can be a UTF-8-encoded filepath that cannot be translated to the current locale.

Yeah, yeah. Warning, warning, blah blah blah. Towards the bottom of the help file, we see something like this, which suggests a bunch of related topics that you might want to look at. These can be quite helpful:

> ee Also
>
> save, download.file; further attach as wrapper for load().
>
> For other interfaces to the underlying serialization format, see unserialize and readRDS.

Finally, it gives you some examples of how to use the function(s) that the help file describes. These are supposed to be proper R commands, meaning that you should be able to type them into the console yourself and they'll actually work. Sometimes it can be quite helpful to try the examples yourself. Anyway, here they are for the "`load`" help file:

xamples

As you can see, they're pretty dense, and not at all obvious to the novice user. However, they do provide good examples of the various different things that you can do with the `load()` function, so it's not a bad idea to have a look at them, and to try not to find them too intimidating.

2.25.2 Other resources

- The Rseek website (www.rseek.org). One thing that I really find annoying about the R help documentation is that it's hard to search properly. When coupled with the fact that the documentation is dense and highly technical, it's often a better idea to search or ask online for answers to your questions. With that in mind, the Rseek website is great: it's an R specific search engine. I find it really useful, and it's almost always my first port of call when I'm looking around.
- The R-help mailing list (see http://www.r-project.org/mail.html for details). This is the official R help mailing list. It can be very helpful, but it's *very* important that you do your homework before posting a question. The list gets a lot of traffic. While the people on the list try as hard as they can to answer questions, they do so for free, and you *really* don't want to know how much money they could charge on an hourly rate if they wanted to apply market rates. In short, they are doing you a favour, so be polite. Don't waste their time asking questions that can be easily answered by a quick search on Rseek (it's rude), make sure your question is clear, and all of the relevant information is included. In short, read the posting guidelines carefully (http://www.r-project.org/posting-guide.html), and make use of the `help.request()` function that R provides to check that you're actually doing what you're expected.

2.26 Summary

This chapter continued where Chapter 2 left off. The focus was still primarily on introducing basic R concepts, but this time at least you can see how those concepts are related to data analysis:

- Installing, loading and updating packages. Knowing how to extend the functionality of R by installing and using packages is critical to becoming an effective R user
- Getting around. Section 2.16 talked about how to manage your workspace and how to keep it tidy. Similarly, Section 2.17 talked about how to get R to interact with the rest of the file system.
- Loading and saving data. Finally, we encountered actual data files. Loading and saving data is obviously a crucial skill, one we discussed in Section 2.18.
- Useful things to know about variables. In particular, we talked about special values, element names and classes.
- More complex types of variables. R has a number of important variable types that will be useful when analysing real data. I talked about factors in Section 2.20, data frames in Section 2.21, lists in Section 2.22 and formulas in Section 2.23.
- Generic functions. How is it that some function seem to be able to do lots of different things? Section 2.24 tells you how.
- Getting help. Assuming that you're not looking for counselling, Section 2.25 covers several possibilities. If you are looking for counselling, well, this book really can't help you there. Sorry.

Taken together, Chapters 2 and 2.13 provide enough of a background that you can finally get started doing some statistics! Yes, there's a lot more R concepts that you ought to know (and we'll talk about some of them in Chapters**??** and**??**), but I think that we've talked quite enough about programming for the moment. It's time to see how your experience with programming can be used to do some data analysis…

Chapter 3

Descriptive statistics

Text by Navarro (2018)

Any time that you get a new data set to look at, one of the first tasks that you have to do is find ways of summarising the data in a compact, easily understood fashion. This is what ***descriptive statistics*** (as opposed to inferential statistics) is all about. In fact, to many people the term "statistics" is synonymous with descriptive statistics. It is this topic that we'll consider in this chapter, but before going into any details, let's take a moment to get a sense of why we need descriptive statistics. To do this, let's load the `aflsmall.Rdata` file, and use the `who()` function in the `lsr` package to see what variables are stored in the file:

```
load( "./data/aflsmall.Rdata" )
library(lsr)
who()
```

```
##    -- Name --              -- Class --   -- Size --
##    afl.finalists           factor        400
##    afl.margins             numeric       176
##    any.sales.this.month    logical       12
##    berkeley                data.frame    39 x 3
##    berkeley.small          data.frame    46 x 2
##    coef                    numeric       2
##    Dan                     list          4
##    days.per.month          numeric       12
##    expt                    data.frame    9 x 4
##    february.sales          numeric       1
##    formula1                formula
##    formula2                formula
##    formula3              formula
##    formula4              formula
##    greeting              character     1
##    is.the.Party.correct  logical       1
##    months                character     12
##    my.formula            formula
##    projecthome           character     1
##    revenue               numeric       1
##    royalty               numeric       1
##    sales                 numeric       1
##    sales.by.month        numeric       12
##    simpson               matrix        6 x 5
##    stock.levels          character     12
##    xlu                   numeric       1
```

There are two variables here, `afl.finalists` and `afl.margins`. We'll focus a bit on these two variables in this chapter, so I'd better tell you what they are. Unlike most of data sets in this book, these are actually real data, relating to the Australian Football League (AFL) [1] The `afl.margins` variable contains the winning margin (number of points) for all 176 home and away games played during the 2010 season. The `afl.finalists` variable contains the names of all 400 teams that played in all 200 finals matches played during the period 1987 to 2010. Let's have a look at the `afl.margins` variable:

```
print(afl.margins)
```

```
##   [1]  56  31  56   8  32  14  36  56  19   1   3 104  43  44  72   9  28
##  [18]  25  27  55  20  16  16   7  23  40  48  64  22  55  95  15  49  52
##  [35]  50  10  65  12  39  36   3  26  23  20  43 108  53  38   4   8   3
##  [52]  13  66  67  50  61  36  38  29   9  81   3  26  12  36  37  70   1
##  [69]  35  12  50  35   9  54  47   8  47   2  29  61  38  41  23  24   1
##  [86]   9  11  10  29  47  71  38  49  65  18   0  16   9  19  36  60  24
## [103]  25  44  55   3  57  83  84  35   4  35  26  22   2  14  19  30  19
## [120]  68  11  75  48  32  36  39  50  11   0  63  82  26   3  82  73  19
## [137]  33  48   8  10  53  20  71  75  76  54  44   5  22  94  29   8  98
## [154]   9  89   1 101   7  21  52  42  21 116   3  44  29  27  16   6  44
## [171]   3  28  38  29  10  10
```

This output doesn't make it easy to get a sense of what the data are actually saying. Just "looking at the data" isn't a terribly effective way of understanding data. In order to get some idea about what's going on, we need to calculate some descriptive statistics (this chapter) and draw some nice pictures (Chapter

[1]Note for non-Australians: the AFL is an Australian rules football competition. You don't need to know anything about Australian rules in order to follow this section.

3.7. Since the descriptive statistics are the easier of the two topics, I'll start with those, but nevertheless I'll show you a histogram of the `afl.margins` data, since it should help you get a sense of what the data we're trying to describe actually look like. But for what it's worth, this histogram – which is shown in Figure 3.1 – was generated using the `hist()` function. We'll talk a lot more about how to draw histograms in Section 3.7.2. For now, it's enough to look at the histogram and note that it provides a fairly interpretable representation of the `afl.margins` data.

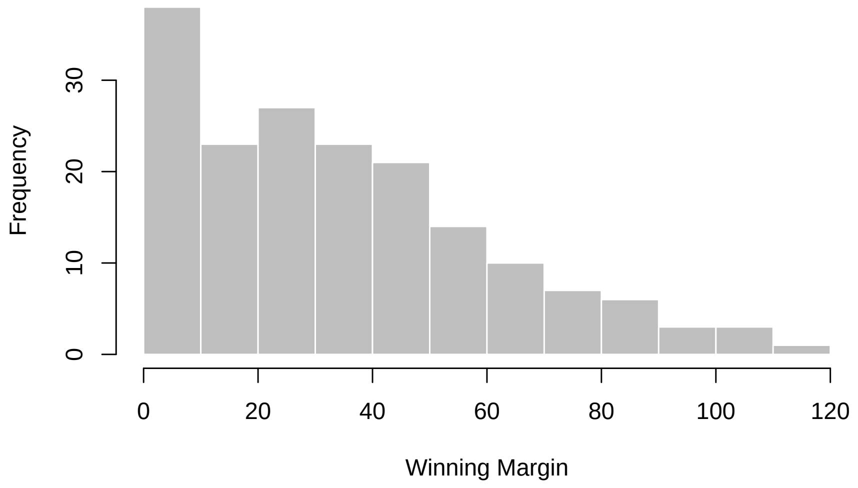


Figure 3.1: A histogram of the AFL 2010 winning margin data (the `afl.margins` variable). As you might expect, the larger the margin the less frequently you tend to see it.

3.1 Measures of central tendency

Drawing pictures of the data, as I did in Figure 3.1 is an excellent way to convey the "gist" of what the data is trying to tell you, it's often extremely useful to try to condense the data into a few simple "summary" statistics. In most situations, the first thing that you'll want to calculate is a measure of ***central tendency***. That is, you'd like to know something about the "average" or "middle" of your data lies. The two most commonly used measures are the mean, median and mode; occasionally people will also report a trimmed mean. I'll explain each of these in turn, and then discuss when each of them is useful.

3.1.1 The mean

The ***mean*** of a set of observations is just a normal, old-fashioned average: add all of the values up, and then divide by the total number of values. The first five AFL margins were 56, 31, 56, 8 and 32, so the mean of these observations is just:

$$\frac{56 + 31 + 56 + 8 + 32}{5} = \frac{183}{5} = 36.60$$

Of course, this definition of the mean isn't news to anyone: averages (i.e., means) are used so often in everyday life that this is pretty familiar stuff. However, since the concept of a mean is something that everyone already understands, I'll use this as an excuse to start introducing some of the mathematical notation that statisticians use to describe this calculation, and talk about how the calculations would be done in R.

The first piece of notation to introduce is N, which we'll use to refer to the number of observations that we're averaging (in this case $N = 5$). Next, we need to attach a label to the observations themselves. It's traditional to use X for this, and to use subscripts to indicate which observation we're actually talking about. That is, we'll use $X_1$ to refer to the first observation, $X_2$ to refer to the second observation, and so on, all the way up to X_N for the last one. Or, to say the same thing in a slightly more abstract way, we use X_i to refer to the i-th observation. Just to make sure we're clear on the notation, the following table lists the 5 observations in the `afl.margins` variable, along with the mathematical symbol used to refer to it, and the actual value that the observation corresponds to:

the observation	its symbol	the observed value
winning margin, game 1	$X_1$	56 points
winning margin, game 2	$X_2$	31 points
winning margin, game 3	$X_3$	56 points
winning margin, game 4	$X_4$	8 points
winning margin, game 5	$X_5$	32 points

Okay, now let's try to write a formula for the mean. By tradition, we use $\bar{X}$ as the notation for the mean. So the calculation for the mean could be expressed using the following formula:

$$\bar{X} = \frac{X_1 + X_2 + ... + X_{N-1} + X_N}{N}$$

This formula is entirely correct, but it's terribly long, so we make use of the ***summation symbol*** $\sum$ to shorten it.[2] If I want to add up the first five obser-

[2]The choice to use Σ to denote summation isn't arbitrary: it's the Greek upper case letter sigma, which is the analogue of the letter S in that alphabet. Similarly, there's an equivalent symbol used to denote the multiplication of lots of numbers: because multiplications are also called "products", we use the Π symbol for this; the Greek upper case pi, which is the analogue of the letter P.

vations, I could write out the sum the long way, $X_1 + X_2 + X_3 + X_4 + X_5$ or I could use the summation symbol to shorten it to this:

$$\sum_{i=1}^{5} X_i$$

Taken literally, this could be read as "the sum, taken over all i values from 1 to 5, of the value X_i". But basically, what it means is "add up the first five observations". In any case, we can use this notation to write out the formula for the mean, which looks like this:

$$\bar{X} = \frac{1}{N} \sum_{i=1}^{N} X_i$$

In all honesty, I can't imagine that all this mathematical notation helps clarify the concept of the mean at all. In fact, it's really just a fancy way of writing out the same thing I said in words: add all the values up, and then divide by the total number of items. However, that's not really the reason I went into all that detail. My goal was to try to make sure that everyone reading this book is clear on the notation that we'll be using throughout the book: $\bar{X}$ for the mean, $\sum$ for the idea of summation, X_i for the ith observation, and N for the total number of observations. We're going to be re-using these symbols a fair bit, so it's important that you understand them well enough to be able to "read" the equations, and to be able to see that it's just saying "add up lots of things and then divide by another thing".

3.1.2 Calculating the mean in R

Okay that's the maths, how do we get the magic computing box to do the work for us? If you really wanted to, you could do this calculation directly in R. For the first 5 AFL scores, do this just by typing it in as if R were a calculator…

```
(56 + 31 + 56 + 8 + 32) / 5
```

```
## [1] 36.6
```

… in which case R outputs the answer 36.6, just as if it were a calculator. However, that's not the only way to do the calculations, and when the number of observations starts to become large, it's easily the most tedious. Besides, in almost every real world scenario, you've already got the actual numbers stored in a variable of some kind, just like we have with the `afl.margins` variable. Under those circumstances, what you want is a function that will just add up all the values stored in a numeric vector. That's what the `sum()` function does.

If we want to add up all 176 winning margins in the data set, we can do so using the following command:[3]

```
sum( afl.margins )
```

```
## [1] 6213
```

If we only want the sum of the first five observations, then we can use square brackets to pull out only the first five elements of the vector. So the command would now be:

```
sum( afl.margins[1:5] )
```

```
## [1] 183
```

To calculate the mean, we now tell R to divide the output of this summation by five, so the command that we need to type now becomes the following:

```
sum( afl.margins[1:5] ) / 5
```

```
## [1] 36.6
```

Although it's pretty easy to calculate the mean using the `sum()` function, we can do it in an even easier way, since R also provides us with the `mean()` function. To calculate the mean for all 176 games, we would use the following command:

```
mean( x = afl.margins )
```

```
## [1] 35.30114
```

However, since `x` is the first argument to the function, I could have omitted the argument name. In any case, just to show you that there's nothing funny going on, here's what we would do to calculate the mean for the first five observations:

```
mean( afl.margins[1:5] )
```

```
## [1] 36.6
```

As you can see, this gives exactly the same answers as the previous calculations.

[3]Note that, just as we saw with the combine function `c()` and the remove function `rm()`, the `sum()` function has unnamed arguments. I'll talk about unnamed arguments later in Section **??**, but for now let's just ignore this detail.

3.1.3 The median

The second measure of central tendency that people use a lot is the ***median***, and it's even easier to describe than the mean. The median of a set of observations is just the middle value. As before let's imagine we were interested only in the first 5 AFL winning margins: 56, 31, 56, 8 and 32. To figure out the median, we sort these numbers into ascending order:

$$8, 31, \mathbf{32}, 56, 56$$

From inspection, it's obvious that the median value of these 5 observations is 32, since that's the middle one in the sorted list (I've put it in bold to make it even more obvious). Easy stuff. But what should we do if we were interested in the first 6 games rather than the first 5? Since the sixth game in the season had a winning margin of 14 points, our sorted list is now

$$8, 14, \mathbf{31}, \mathbf{32}, 56, 56$$

and there are *two* middle numbers, 31 and 32. The median is defined as the average of those two numbers, which is of course 31.5. As before, it's very tedious to do this by hand when you've got lots of numbers. To illustrate this, here's what happens when you use R to sort all 176 winning margins. First, I'll use the `sort()` function (discussed in Chapter **??**) to display the winning margins in increasing numerical order:

```
sort( x = afl.margins )
```

```
##   [1]   0   0   1   1   1   1   2   2   3   3   3   3   3   3   3   3   4
##  [18]   4   5   6   7   7   8   8   8   8   8   9   9   9   9   9   9  10
##  [35]  10  10  10  10  11  11  11  12  12  12  13  14  14  15  16  16  16
##  [52]  16  18  19  19  19  19  19  20  20  20  21  21  22  22  22  23  23
##  [69]  23  24  24  25  25  26  26  26  26  27  27  28  28  29  29  29  29
##  [86]  29  29  30  31  32  32  33  35  35  35  35  36  36  36  36  36  36
## [103]  37  38  38  38  38  38  39  39  40  41  42  43  43  44  44  44  44
## [120]  44  47  47  47  48  48  48  49  49  50  50  50  50  52  52  53  53
## [137]  54  54  55  55  55  56  56  56  57  60  61  61  63  64  65  65  66
## [154]  67  68  70  71  71  72  73  75  75  76  81  82  82  83  84  89  94
## [171]  95  98 101 104 108 116
```

The middle values are 30 and 31, so the median winning margin for 2010 was 30.5 points. In real life, of course, no-one actually calculates the median by sorting the data and then looking for the middle value. In real life, we use the median command:

```r
median( x = afl.margins )
```

```
## [1] 30.5
```

which outputs the median value of 30.5.

3.1.4 Mean or median? What's the difference?

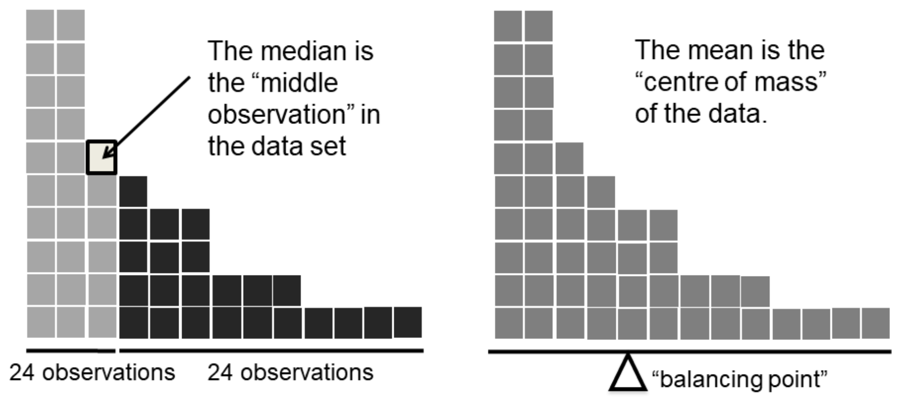

The median is the "middle observation" in the data set
24 observations
24 observations
The mean is the "centre of mass" of the data.
"balancing point"

Figure 3.2: An illustration of the difference between how the mean and the median should be interpreted. The mean is basically the "centre of gravity" of the data set: if you imagine that the histogram of the data is a solid object, then the point on which you could balance it (as if on a see-saw) is the mean. In contrast, the median is the middle observation. Half of the observations are smaller, and half of the observations are larger.

Knowing how to calculate means and medians is only a part of the story. You also need to understand what each one is saying about the data, and what that implies for when you should use each one. This is illustrated in Figure 3.2 the mean is kind of like the "centre of gravity" of the data set, whereas the median is the "middle value" in the data. What this implies, as far as which one you should use, depends a little on what type of data you've got and what you're trying to achieve. As a rough guide:

- If your data are nominal scale, you probably shouldn't be using either the mean or the median. Both the mean and the median rely on the idea that the numbers assigned to values are meaningful. If the numbering scheme is arbitrary, then it's probably best to use the mode (Section 3.1.7) instead.

- If your data are ordinal scale, you're more likely to want to use the median than the mean. The median only makes use of the order information in your data (i.e., which numbers are bigger), but doesn't depend on the precise numbers involved. That's exactly the situation that applies when your data are ordinal scale. The mean, on the other hand, makes use of the precise numeric values assigned to the observations, so it's not really appropriate for ordinal data.
- For interval and ratio scale data, either one is generally acceptable. Which one you pick depends a bit on what you're trying to achieve. The mean has the advantage that it uses all the information in the data (which is useful when you don't have a lot of data), but it's very sensitive to extreme values, as we'll see in Section 3.1.6.

Let's expand on that last part a little. One consequence is that there's systematic differences between the mean and the median when the histogram is asymmetric (skewed; see Section 3.3). This is illustrated in Figure 3.2 notice that the median (right hand side) is located closer to the "body" of the histogram, whereas the mean (left hand side) gets dragged towards the "tail" (where the extreme values are). To give a concrete example, suppose Bob (income \$50,000), Kate (income \$60,000) and Jane (income \$65,000) are sitting at a table: the average income at the table is \$58,333 and the median income is \$60,000. Then Bill sits down with them (income \$100,000,000). The average income has now jumped to \$25,043,750 but the median rises only to \$62,500. If you're interested in looking at the overall income at the table, the mean might be the right answer; but if you're interested in what counts as a typical income at the table, the median would be a better choice here.

3.1.5 A real life example

To try to get a sense of why you need to pay attention to the differences between the mean and the median, let's consider a real life example. Since I tend to mock journalists for their poor scientific and statistical knowledge, I should give credit where credit is due. This is from an excellent article on the ABC news website[4] 24 September, 2010:

> Senior Commonwealth Bank executives have travelled the world in the past couple of weeks with a presentation showing how Australian house prices, and the key price to income ratios, compare favourably with similar countries. "Housing affordability has actually been going sideways for the last five to six years," said Craig James, the chief economist of the bank's trading arm, CommSec.

This probably comes as a huge surprise to anyone with a mortgage, or who wants a mortgage, or pays rent, or isn't completely oblivious to what's been

[4] www.abc.net.au/news/stories/2010/09/24/3021480.htm

going on in the Australian housing market over the last several years. Back to the article:

> CBA has waged its war against what it believes are housing doomsayers with graphs, numbers and international comparisons. In its presentation, the bank rejects arguments that Australia's housing is relatively expensive compared to incomes. It says Australia's house price to household income ratio of 5.6 in the major cities, and 4.3 nationwide, is comparable to many other developed nations. It says San Francisco and New York have ratios of 7, Auckland's is 6.7, and Vancouver comes in at 9.3.

More excellent news! Except, the article goes on to make the observation that...

> Many analysts say that has led the bank to use misleading figures and comparisons. If you go to page four of CBA's presentation and read the source information at the bottom of the graph and table, you would notice there is an additional source on the international comparison – Demographia. However, if the Commonwealth Bank had also used Demographia's analysis of Australia's house price to income ratio, it would have come up with a figure closer to 9 rather than 5.6 or 4.3

That's, um, a rather serious discrepancy. One group of people say 9, another says 4-5. Should we just split the difference, and say the truth lies somewhere in between? Absolutely not: this is a situation where there is a right answer and a wrong answer. Demographia are correct, and the Commonwealth Bank is incorrect. As the article points out

> [An] obvious problem with the Commonwealth Bank's domestic price to income figures is they compare average incomes with median house prices (unlike the Demographia figures that compare median incomes to median prices). The median is the mid-point, effectively cutting out the highs and lows, and that means the average is generally higher when it comes to incomes and asset prices, because it includes the earnings of Australia's wealthiest people. To put it another way: the Commonwealth Bank's figures count Ralph Norris' multi-million dollar pay packet on the income side, but not his (no doubt) very expensive house in the property price figures, thus understating the house price to income ratio for middle-income Australians.

Couldn't have put it better myself. The way that Demographia calculated the ratio is the right thing to do. The way that the Bank did it is incorrect. As for why an extremely quantitatively sophisticated organisation such as a major bank made such an elementary mistake, well... I can't say for sure, since I have no special insight into their thinking, but the article itself does happen to mention the following facts, which may or may not be relevant:

> [As] Australia's largest home lender, the Commonwealth Bank has one of the biggest vested interests in house prices rising. It effectively owns a massive swathe of Australian housing as security for its home loans as well as many small business loans.

My, my.

3.1.6 Trimmed mean

One of the fundamental rules of applied statistics is that the data are messy. Real life is never simple, and so the data sets that you obtain are never as straightforward as the statistical theory says.[5] This can have awkward consequences. To illustrate, consider this rather strange looking data set:

$$-100, 2, 3, 4, 5, 6, 7, 8, 9, 10$$

If you were to observe this in a real life data set, you'd probably suspect that something funny was going on with the -100 value. It's probably an ***outlier***, a value that doesn't really belong with the others. You might consider removing it from the data set entirely, and in this particular case I'd probably agree with that course of action. In real life, however, you don't always get such cut-and-dried examples. For instance, you might get this instead:

$$-15, 2, 3, 4, 5, 6, 7, 8, 9, 12$$

The -15 looks a bit suspicious, but not anywhere near as much as that -100 did. In this case, it's a little trickier. It *might* be a legitimate observation, it might not.

When faced with a situation where some of the most extreme-valued observations might not be quite trustworthy, the mean is not necessarily a good measure of central tendency. It is highly sensitive to one or two extreme values, and is thus not considered to be a ***robust*** measure. One remedy that we've seen is to use the median. A more general solution is to use a "trimmed mean". To calculate a trimmed mean, what you do is "discard" the most extreme examples on both ends (i.e., the largest and the smallest), and then take the mean of everything else. The goal is to preserve the best characteristics of the mean and the median: just like a median, you aren't highly influenced by extreme outliers,

[5] Or at least, the basic statistical theory – these days there is a whole subfield of statistics called *robust statistics* that tries to grapple with the messiness of real data and develop theory that can cope with it.

but like the mean, you "use" more than one of the observations. Generally, we describe a trimmed mean in terms of the percentage of observation on either side that are discarded. So, for instance, a 10% trimmed mean discards the largest 10% of the observations *and* the smallest 10% of the observations, and then takes the mean of the remaining 80% of the observations. Not surprisingly, the 0% trimmed mean is just the regular mean, and the 50% trimmed mean is the median. In that sense, trimmed means provide a whole family of central tendency measures that span the range from the mean to the median.

For our toy example above, we have 10 observations, and so a 10% trimmed mean is calculated by ignoring the largest value (i.e., `12`) and the smallest value (i.e., `-15`) and taking the mean of the remaining values. First, let's enter the data

```
dataset <- c( -15,2,3,4,5,6,7,8,9,12 )
```

Next, let's calculate means and medians:

```
mean( x = dataset )
```

```
## [1] 4.1
```

```
median( x = dataset )
```

```
## [1] 5.5
```

That's a fairly substantial difference, but I'm tempted to think that the mean is being influenced a bit too much by the extreme values at either end of the data set, especially the -15 one. So let's just try trimming the mean a bit. If I take a 10% trimmed mean, we'll drop the extreme values on either side, and take the mean of the rest:

```
mean( x = dataset, trim = .1)
```

```
## [1] 5.5
```

which in this case gives exactly the same answer as the median. Note that, to get a 10% trimmed mean you write `trim = .1`, not `trim = 10`. In any case, let's finish up by calculating the 5% trimmed mean for the `afl.margins` data,

```
mean( x = afl.margins, trim = .05)
```

```
## [1] 33.75
```

3.1.7 Mode

The mode of a sample is very simple: it is the value that occurs most frequently. To illustrate the mode using the AFL data, let's examine a different aspect to the data set. Who has played in the most finals? The `afl.finalists` variable is a factor that contains the name of every team that played in any AFL final from 1987-2010, so let's have a look at it. To do this we will use the `head()` command. `head()` is useful when you're working with a data.frame with a lot of rows since you can use it to tell you how many rows to return. There have been a lot of finals in this period so printing afl.finalists using `print(afl.finalists)` will just fill us the screen. The command below tells R we just want the first 25 rows of the data.frame.

```
head(afl.finalists, 25)
```

```
##  [1] Hawthorn    Melbourne   Carlton     Melbourne   Hawthorn
##  [6] Carlton     Melbourne   Carlton     Hawthorn    Melbourne
## [11] Melbourne   Hawthorn    Melbourne   Essendon    Hawthorn
## [16] Geelong     Geelong     Hawthorn    Collingwood Melbourne
## [21] Collingwood West Coast  Collingwood Essendon    Collingwood
## 17 Levels: Adelaide Brisbane Carlton Collingwood Essendon ... Western Bulldogs
```

There are actually 400 entries (aren't you glad we didn't print them all?). We *could* read through all 400, and count the number of occasions on which each team name appears in our list of finalists, thereby producing a ***frequency table***. However, that would be mindless and boring: exactly the sort of task that computers are great at. So let's use the `table()` function (discussed in more detail in Section **??**) to do this task for us:

```
table( afl.finalists )
```

```
## afl.finalists
##         Adelaide         Brisbane          Carlton      Collingwood
##               26               25               26               28
##         Essendon          Fitzroy        Fremantle          Geelong
##               32                0                6               39
##         Hawthorn        Melbourne  North Melbourne    Port Adelaide
##               27               28               28               17
##         Richmond         St Kilda           Sydney       West Coast
##                6               24               26               38
## Western Bulldogs
##               24
```

Now that we have our frequency table, we can just look at it and see that, over the 24 years for which we have data, Geelong has played in more finals than any other team. Thus, the mode of the `finalists` data is `"Geelong"`. The core packages in R don't have a function for calculating the mode[6]. However, I've included a function in the `lsr` package that does this. The function is called `modeOf()`, and here's how you use it:

```
modeOf( x = afl.finalists )
```

```
## [1] "Geelong"
```

There's also a function called `maxFreq()` that tells you what the modal frequency is. If we apply this function to our finalists data, we obtain the following:

```
maxFreq( x = afl.finalists )
```

```
## [1] 39
```

Taken together, we observe that Geelong (39 finals) played in more finals than any other team during the 1987-2010 period.

One last point to make with respect to the mode. While it's generally true that the mode is most often calculated when you have nominal scale data (because means and medians are useless for those sorts of variables), there are some situations in which you really do want to know the mode of an ordinal, interval or ratio scale variable. For instance, let's go back to thinking about our `afl.margins` variable. This variable is clearly ratio scale (if it's not clear to you, it may help to re-read Section **??**), and so in most situations the mean or the median is the measure of central tendency that you want. But consider this scenario... a friend of yours is offering a bet. They pick a football game at random, and (without knowing who is playing) you have to guess the *exact* margin. If you guess correctly, you win $50. If you don't, you lose $1. There are no consolation prizes for "almost" getting the right answer. You have to guess exactly the right margin[7] For this bet, the mean and the median are completely useless to you. It is the mode that you should bet on. So, we calculate this modal value

```
modeOf( x = afl.margins )
```

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

[6]As we saw earlier, it *does* have a function called `mode()`, but it does something completely different.

[7]This is called a "0-1 loss function", meaning that you either win (1) or you lose (0), with no middle ground.

```
maxFreq( x = afl.margins )
```

```
## [1] 8
```

So the 2010 data suggest you should bet on a 3 point margin, and since this was observed in 8 of the 176 game (4.5% of games) the odds are firmly in your favour.

3.2 Measures of variability

The statistics that we've discussed so far all relate to *central tendency*. That is, they all talk about which values are "in the middle" or "popular" in the data. However, central tendency is not the only type of summary statistic that we want to calculate. The second thing that we really want is a measure of the ***variability*** of the data. That is, how "spread out" are the data? How "far" away from the mean or median do the observed values tend to be? For now, let's assume that the data are interval or ratio scale, so we'll continue to use the `afl.margins` data. We'll use this data to discuss several different measures of spread, each with different strengths and weaknesses.

3.2.1 Range

The ***range*** of a variable is very simple: it's the biggest value minus the smallest value. For the AFL winning margins data, the maximum value is 116, and the minimum value is 0. We can calculate these values in R using the `max()` and `min()` functions:

```
max( afl.margins )
```

```
## [1] 116
```

```
min( afl.margins )
```

```
## [1] 0
```

where I've omitted the output because it's not interesting. The other possibility is to use the `range()` function; which outputs both the minimum value and the maximum value in a vector, like this:

```
range( afl.margins )
```

```
## [1]   0 116
```

Although the range is the simplest way to quantify the notion of "variability", it's one of the worst. Recall from our discussion of the mean that we want our summary measure to be robust. If the data set has one or two extremely bad values in it, we'd like our statistics not to be unduly influenced by these cases. If we look once again at our toy example of a data set containing very extreme outliers...

$$-100, 2, 3, 4, 5, 6, 7, 8, 9, 10$$

... it is clear that the range is not robust, since this has a range of 110, but if the outlier were removed we would have a range of only 8.

3.2.2 Interquartile range

The ***interquartile range*** (IQR) is like the range, but instead of calculating the difference between the biggest and smallest value, it calculates the difference between the 25th quantile and the 75th quantile. Probably you already know what a ***quantile*** is (they're more commonly called percentiles), but if not: the 10th percentile of a data set is the smallest number x such that 10% of the data is less than x. In fact, we've already come across the idea: the median of a data set is its 50th quantile / percentile! R actually provides you with a way of calculating quantiles, using the (surprise, surprise) `quantile()` function. Let's use it to calculate the median AFL winning margin:

```
quantile( x = afl.margins, probs = .5)
```

```
##  50%
## 30.5
```

And not surprisingly, this agrees with the answer that we saw earlier with the `median()` function. Now, we can actually input lots of quantiles at once, by specifying a vector for the `probs` argument. So lets do that, and get the 25th and 75th percentile:

```
quantile( x = afl.margins, probs = c(.25,.75) )
```

```
##   25%   75%
## 12.75 50.50
```

And, by noting that $50.5 - 12.75 = 37.75$, we can see that the interquartile range for the 2010 AFL winning margins data is 37.75. Of course, that seems like too much work to do all that typing, so R has a built in function called `IQR()` that we can use:

```
IQR( x = afl.margins )
```

```
## [1] 37.75
```

While it's obvious how to interpret the range, it's a little less obvious how to interpret the IQR. The simplest way to think about it is like this: the interquartile range is the range spanned by the "middle half" of the data. That is, one quarter of the data falls below the 25th percentile, one quarter of the data is above the 75th percentile, leaving the "middle half" of the data lying in between the two. And the IQR is the range covered by that middle half.

3.2.3 Mean absolute deviation

The two measures we've looked at so far, the range and the interquartile range, both rely on the idea that we can measure the spread of the data by looking at the quantiles of the data. However, this isn't the only way to think about the problem. A different approach is to select a meaningful reference point (usually the mean or the median) and then report the "typical" deviations from that reference point. What do we mean by "typical" deviation? Usually, the mean or median value of these deviations! In practice, this leads to two different measures, the "mean absolute deviation (from the mean)" and the "median absolute deviation (from the median)". From what I've read, the measure based on the median seems to be used in statistics, and does seem to be the better of the two, but to be honest I don't think I've seen it used much in psychology. The measure based on the mean does occasionally show up in psychology though. In this section I'll talk about the first one, and I'll come back to talk about the second one later.

Since the previous paragraph might sound a little abstract, let's go through the ***mean absolute deviation*** from the mean a little more slowly. One useful thing about this measure is that the name actually tells you exactly how to calculate it. Let's think about our AFL winning margins data, and once again we'll start by pretending that there's only 5 games in total, with winning margins of 56, 31, 56, 8 and 32. Since our calculations rely on an examination of the deviation from some reference point (in this case the mean), the first thing we need to calculate is the mean, $\bar{X}$. For these five observations, our mean is $\bar{X} = 36.6$. The next step is to convert each of our observations X_i into a deviation score. We do this by calculating the difference between the observation X_i and the mean $\bar{X}$. That is, the deviation score is defined to be $X_i - \bar{X}$. For the first observation in our sample, this is equal to $56 - 36.6 = 19.4$. Okay, that's simple enough. The next step in the process is to convert these deviations to absolute deviations. As we discussed earlier when talking about the `abs()` function in R (Section 2.5), we do this by converting any negative values to positive ones. Mathematically, we would denote the absolute value of -3 as $|-3|$, and so we say that $|-3| = 3$. We use the absolute value function here because we don't really care whether the value is higher than the mean or lower than the mean, we're just interested in how *close* it is to the mean. To help make this process as obvious as possible, the table below shows these calculations for all five observations:

the observation	its symbol	the observed value
winning margin, game 1	X_1	56 points
winning margin, game 2	X_2	31 points
winning margin, game 3	X_3	56 points
winning margin, game 4	X_4	8 points
winning margin, game 5	X_5	32 points

Now that we have calculated the absolute deviation score for every observation in the data set, all that we have to do to calculate the mean of these scores. Let's do that:

$$\frac{19.4 + 5.6 + 19.4 + 28.6 + 4.6}{5} = 15.52$$

And we're done. The mean absolute deviation for these five scores is 15.52.

However, while our calculations for this little example are at an end, we do have a couple of things left to talk about. Firstly, we should really try to write down a proper mathematical formula. But in order do to this I need some mathematical notation to refer to the mean absolute deviation. Irritatingly, "mean absolute deviation" and "median absolute deviation" have the same acronym (MAD), which leads to a certain amount of ambiguity, and since R tends to use MAD to refer to the median absolute deviation, I'd better come up with something different for the mean absolute deviation. Sigh. What I'll do is use AAD instead, short for *average* absolute deviation. Now that we have some unambiguous notation, here's the formula that describes what we just calculated:

$$(X) = \frac{1}{N} \sum_{i=1}^{N} |X_i - \bar{X}|$$

The last thing we need to talk about is how to calculate AAD in R. One possibility would be to do everything using low level commands, laboriously following the same steps that I used when describing the calculations above. However, that's pretty tedious. You'd end up with a series of commands that might look like this:

```
X <- c(56, 31,56,8,32)    # enter the data
X.bar <- mean( X )        # step 1. the mean of the data
```

```
AD <- abs( X - X.bar )    # step 2. the absolute deviations from the mean
AAD <- mean( AD )         # step 3. the mean absolute deviations
print( AAD )              # print the results
```

```
## [1] 15.52
```

Each of those commands is pretty simple, but there's just too many of them. And because I find that to be too much typing, the `lsr` package has a very simple function called `aad()` that does the calculations for you. If we apply the `aad()` function to our data, we get this:

```
library(lsr)
aad( X )
```

```
## [1] 15.52
```

No suprises there.

3.2.4 Variance

Although the mean absolute deviation measure has its uses, it's not the best measure of variability to use. From a purely mathematical perspective, there are some solid reasons to prefer squared deviations rather than absolute deviations. If we do that, we obtain a measure is called the ***variance***, which has a lot of really nice statistical properties that I'm going to ignore,[8](X)$ and Var(Y) respectively. Now imagine I want to define a new variable Z that is the sum of the two, $Z = X + Y$. As it turns out, the variance of Z is equal to Var(X) + Var(Y). This is a *very* useful property, but it's not true of the other measures that I talk about in this section.] and one massive psychological flaw that I'm going to make a big deal out of in a moment. The variance of a data set X is sometimes written as Var(X), but it's more commonly denoted s^2 (the reason for this will become clearer shortly). The formula that we use to calculate the variance of a set of observations is as follows:

$$\text{Var}(X) = \frac{1}{N} \sum_{i=1}^{N} \left(X_i - \bar{X}\right)^2$$

$$\text{Var}(X) = \frac{\sum_{i=1}^{N} \left(X_i - \bar{X}\right)^2}{N}$$

[8]Well, I will very briefly mention the one that I think is coolest, for a very particular definition of "cool", that is. Variances are *additive*. Here's what that means: suppose I have two variables X and Y, whose variances are $Var

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.

Notation [English]	$i$ [which game]	$X_i$ [value]	$X_i - \bar{X}$ [deviation from mean]	$(X
	1	56	19.4	376
	2	31	-5.6	31.
	3	56	19.4	376
	4	8	-28.6	817
	5	32	-4.6	21.

As you can see, it's basically the same formula that we used to calculate the mean absolute deviation, except that instead of using "absolute deviations" we use "squared deviations". It is for this reason that the variance is sometimes referred to as the "mean square deviation".

Now that we've got the basic idea, let's have a look at a concrete example. Once again, let's use the first five AFL games as our data. If we follow the same approach that we took last time, we end up with the following table:

That last column contains all of our squared deviations, so all we have to do is average them. If we do that by typing all the numbers into R by hand…

```
( 376.36 + 31.36 + 376.36 + 817.96 + 21.16 ) / 5
```

```
## [1] 324.64
```

… we end up with a variance of 324.64. Exciting, isn't it? For the moment, let's ignore the burning question that you're all probably thinking (i.e., what the heck does a variance of 324.64 actually mean?) and instead talk a bit more about how to do the calculations in R, because this will reveal something very weird.

As always, we want to avoid having to type in a whole lot of numbers ourselves. And as it happens, we have the vector `X` lying around, which we created in the previous section. With this in mind, we can calculate the variance of `X` by using the following command,

```
mean( (X - mean(X) )^2)
```

```
## [1] 324.64
```

and as usual we get the same answer as the one that we got when we did everything by hand. However, I *still* think that this is too much typing. Fortunately, R has a built in function called `var()` which does calculate variances. So we could also do this...

```
var(X)
```

```
## [1] 405.8
```

and you get the same... no, wait... you get a completely *different* answer. That's just weird. Is R broken? Is this a typo? Is Dan an idiot?

As it happens, the answer is no.[9] It's not a typo, and R is not making a mistake. To get a feel for what's happening, let's stop using the tiny data set containing only 5 data points, and switch to the full set of 176 games that we've got stored in our `afl.margins` vector. First, let's calculate the variance by using the formula that I described above:

```
mean( (afl.margins - mean(afl.margins) )^2)
```

```
## [1] 675.9718
```

Now let's use the `var()` function:

```
var( afl.margins )
```

```
## [1] 679.8345
```

Hm. These two numbers are very similar this time. That seems like too much of a coincidence to be a mistake. And of course it isn't a mistake. In fact, it's very simple to explain what R is doing here, but slightly trickier to explain *why* R is doing it. So let's start with the "what". What R is doing is evaluating a slightly different formula to the one I showed you above. Instead of averaging the squared deviations, which requires you to divide by the number of data points N, R has chosen to divide by $N-1$. In other words, the formula that R is using is this one

$$\frac{1}{N-1} \sum_{i=1}^{N} \left(X_i - \bar{X} \right)^2$$

It's easy enough to verify that this is what's happening, as the following command illustrates:

[9]With the possible exception of the third question.

```
sum( (X-mean(X))^2 ) / 4
```

```
## [1] 405.8
```

This is the same answer that R gave us originally when we calculated `var(X)` originally. So that's the *what*. The real question is *why* R is dividing by $N-1$ and not by N. After all, the variance is supposed to be the *mean* squared deviation, right? So shouldn't we be dividing by N, the actual number of observations in the sample? Well, yes, we should. However, as we'll discuss in Chapter **??**, there's a subtle distinction between "describing a sample" and "making guesses about the population from which the sample came". Up to this point, it's been a distinction without a difference. Regardless of whether you're describing a sample or drawing inferences about the population, the mean is calculated exactly the same way. Not so for the variance, or the standard deviation, or for many other measures besides. What I outlined to you initially (i.e., take the actual average, and thus divide by N) assumes that you literally intend to calculate the variance of the sample. Most of the time, however, you're not terribly interested in the sample *in and of itself*. Rather, the sample exists to tell you something about the world. If so, you're actually starting to move away from calculating a "sample statistic", and towards the idea of estimating a "population parameter". However, I'm getting ahead of myself. For now, let's just take it on faith that R knows what it's doing, and we'll revisit the question later on when we talk about estimation in Chapter **??**.

Okay, one last thing. This section so far has read a bit like a mystery novel. I've shown you how to calculate the variance, described the weird "$N-1$" thing that R does and hinted at the reason why it's there, but I haven't mentioned the single most important thing... how do you *interpret* the variance? Descriptive statistics are supposed to describe things, after all, and right now the variance is really just a gibberish number. Unfortunately, the reason why I haven't given you the human-friendly interpretation of the variance is that there really isn't one. This is the most serious problem with the variance. Although it has some elegant mathematical properties that suggest that it really is a fundamental quantity for expressing variation, it's completely useless if you want to communicate with an actual human... variances are completely uninterpretable in terms of the original variable! All the numbers have been squared, and they don't mean anything anymore. This is a huge issue. For instance, according to the table I presented earlier, the margin in game 1 was "376.36 points-squared higher than the average margin". This is *exactly* as stupid as it sounds; and so when we calculate a variance of 324.64, we're in the same situation. I've watched a lot of footy games, and never has anyone referred to "points squared". It's *not* a real unit of measurement, and since the variance is expressed in terms of this gibberish unit, it is totally meaningless to a human.

3.2.5 Standard deviation

Okay, suppose that you like the idea of using the variance because of those nice mathematical properties that I haven't talked about, but – since you're a human and not a robot – you'd like to have a measure that is expressed in the same units as the data itself (i.e., points, not points-squared). What should you do? The solution to the problem is obvious: take the square root of the variance, known as the ***standard deviation***, also called the "root mean squared deviation", or RMSD. This solves out problem fairly neatly: while nobody has a clue what "a variance of 324.68 points-squared" really means, it's much easier to understand "a standard deviation of 18.01 points", since it's expressed in the original units. It is traditional to refer to the standard deviation of a sample of data as s, though "sd" and "std dev." are also used at times. Because the standard deviation is equal to the square root of the variance, you probably won't be surprised to see that the formula is:

$$s = \sqrt{\frac{1}{N} \sum_{i=1}^{N} \left(X_i - \bar{X} \right)^2}$$

and the R function that we use to calculate it is `sd()`. However, as you might have guessed from our discussion of the variance, what R actually calculates is slightly different to the formula given above. Just like the we saw with the variance, what R calculates is a version that divides by $N-1$ rather than N. For reasons that will make sense when we return to this topic in Chapter@ refch:estimation I'll refer to this new quantity as $\hat{\sigma}$ (read as: "sigma hat"), and the formula for this is

$$\hat{\sigma} = \sqrt{\frac{1}{N-1} \sum_{i=1}^{N} \left(X_i - \bar{X} \right)^2}$$

With that in mind, calculating standard deviations in R is simple:

```
sd( afl.margins )
```

```
## [1] 26.07364
```

Interpreting standard deviations is slightly more complex. Because the standard deviation is derived from the variance, and the variance is a quantity that has little to no meaning that makes sense to us humans, the standard deviation doesn't have a simple interpretation. As a consequence, most of us just rely on a simple rule of thumb: in general, you should expect 68% of the data to fall within 1 standard deviation of the mean, 95% of the data to fall within 2 standard deviation of the mean, and 99.7% of the data to fall within 3 standard deviations of the mean. This rule tends to work pretty well most of the time, but it's not exact: it's actually calculated based on an *assumption* that the histogram is symmetric and "bell shaped".[10] As you can tell from looking at the AFL winning margins histogram in Figure 3.1, this isn't exactly true of our data! Even so, the rule is approximately correct. As it turns out, 65.3% of the AFL margins data fall within one standard deviation of the mean. This is shown visually in Figure 3.3.

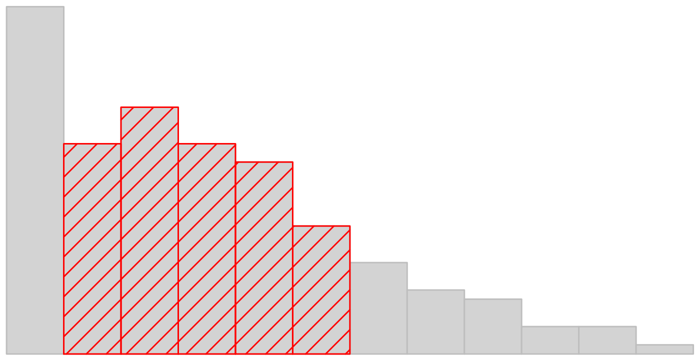


Figure 3.3: An illustration of the standard deviation, applied to the AFL winning margins data. The shaded bars in the histogram show how much of the data fall within one standard deviation of the mean. In this case, 65.3% of the data set lies within this range, which is pretty consistent with the "approximately 68% rule" discussed in the main text.

3.2.6 Median absolute deviation

The last measure of variability that I want to talk about is the ***median absolute deviation*** (MAD). The basic idea behind MAD is very simple, and is pretty much identical to the idea behind the mean absolute deviation (Section 3.2.3). The difference is that you use the median everywhere. If we were to frame this idea as a pair of R commands, they would look like this:

[10]Strictly, the assumption is that the data are *normally* distributed, which is an important concept that we'll discuss more in Chapter **??**, and will turn up over and over again later in the book.

```
# mean absolute deviation from the mean:
mean( abs(afl.margins - mean(afl.margins)) )
```

```
## [1] 21.10124
```

```
# *median* absolute deviation from the *median*:
median( abs(afl.margins - median(afl.margins)) )
```

```
## [1] 19.5
```

This has a straightforward interpretation: every observation in the data set lies some distance away from the typical value (the median). So the MAD is an attempt to describe a *typical deviation from a typical value* in the data set. It wouldn't be unreasonable to interpret the MAD value of 19.5 for our AFL data by saying something like this:

> The median winning margin in 2010 was 30.5, indicating that a typical game involved a winning margin of about 30 points. However, there was a fair amount of variation from game to game: the MAD value was 19.5, indicating that a typical winning margin would differ from this median value by about 19-20 points.

As you'd expect, R has a built in function for calculating MAD, and you will be shocked no doubt to hear that it's called `mad()`. However, it's a little bit more complicated than the functions that we've been using previously. If you want to use it to calculate MAD in the exact same way that I have described it above, the command that you need to use specifies two arguments: the data set itself `x`, and a `constant` that I'll explain in a moment. For our purposes, the constant is 1, so our command becomes

```
mad( x = afl.margins, constant = 1 )
```

```
## [1] 19.5
```

Apart from the weirdness of having to type that `constant = 1` part, this is pretty straightforward.

Okay, so what exactly is this `constant = 1` argument? I won't go into all the details here, but here's the gist. Although the "raw" MAD value that I've described above is completely interpretable on its own terms, that's not actually how it's used in a lot of real world contexts. Instead, what happens a lot is that the researcher *actually* wants to calculate the standard deviation. However, in the same way that the mean is very sensitive to extreme values, the standard deviation is vulnerable to the exact same issue. So, in much the same way that people sometimes use the median as a "robust" way of calculating "something that is like the mean", it's not uncommon to use MAD as a method for calculating "something that is like the standard deviation". Unfortunately, the *raw* MAD value doesn't do this. Our raw MAD value is 19.5, and our standard deviation was 26.07. However, what some clever person has shown is that, under certain assumptions[11], you can multiply the raw MAD value by 1.4826 and obtain a number that is directly comparable to the standard deviation. As a consequence, the default value of `constant` is 1.4826, and so when you use the `mad()` command without manually setting a value, here's what you get:

```
mad( afl.margins )
```

```
## [1] 28.9107
```

I should point out, though, that if you want to use this "corrected" MAD value as a robust version of the standard deviation, you really are relying on the assumption that the data are (or at least, are "supposed to be" in some sense) symmetric and basically shaped like a bell curve. That's really *not* true for our `afl.margins` data, so in this case I wouldn't try to use the MAD value this way.

3.2.7 Which measure to use?

We've discussed quite a few measures of spread (range, IQR, MAD, variance and standard deviation), and hinted at their strengths and weaknesses. Here's a quick summary:

- *Range.* Gives you the full spread of the data. It's very vulnerable to outliers, and as a consequence it isn't often used unless you have good reasons to care about the extremes in the data.
- *Interquartile range.* Tells you where the "middle half" of the data sits. It's pretty robust, and complements the median nicely. This is used a lot.
- *Mean absolute deviation.* Tells you how far "on average" the observations are from the mean. It's very interpretable, but has a few minor issues (not discussed here) that make it less attractive to statisticians than the standard deviation. Used sometimes, but not often.
- *Variance.* Tells you the average squared deviation from the mean. It's mathematically elegant, and is probably the "right" way to describe variation around the mean, but it's completely uninterpretable because it doesn't use the same units as the data. Almost never used except as a

[11]The assumption again being that the data are normally-distributed!

mathematical tool; but it's buried "under the hood" of a very large number of statistical tools.

- *Standard deviation.* This is the square root of the variance. It's fairly elegant mathematically, and it's expressed in the same units as the data so it can be interpreted pretty well. In situations where the mean is the measure of central tendency, this is the default. This is by far the most popular measure of variation.
- *Median absolute deviation.* The typical (i.e., median) deviation from the median value. In the raw form it's simple and interpretable; in the corrected form it's a robust way to estimate the standard deviation, for some kinds of data sets. Not used very often, but it does get reported sometimes.

In short, the IQR and the standard deviation are easily the two most common measures used to report the variability of the data; but there are situations in which the others are used. I've described all of them in this book because there's a fair chance you'll run into most of these somewhere.

3.3 Skew and kurtosis

There are two more descriptive statistics that you will sometimes see reported in the psychological literature, known as skew and kurtosis. In practice, neither one is used anywhere near as frequently as the measures of central tendency and variability that we've been talking about. Skew is pretty important, so you do see it mentioned a fair bit; but I've actually never seen kurtosis reported in a scientific article to date.

```
## [1] -0.915704
```

```
## [1] 0.007451671
```

```
## [1] 0.924245
```

Since it's the more interesting of the two, let's start by talking about the ***skewness***. Skewness is basically a measure of asymmetry, and the easiest way to explain it is by drawing some pictures. As Figure 3.4 illustrates, if the data tend to have a lot of extreme small values (i.e., the lower tail is "longer" than the upper tail) and not so many extremely large values (left panel), then we say that the data are *negatively skewed*. On the other hand, if there are more extremely large values than extremely small ones (right panel) we say that the data are *positively skewed*. That's the qualitative idea behind skewness. The actual formula for the skewness of a data set is as follows

$$\text{skewness}(X) = \frac{1}{N\hat{\sigma}^3} \sum_{i=1}^{N} (X_i - \bar{X})^3$$

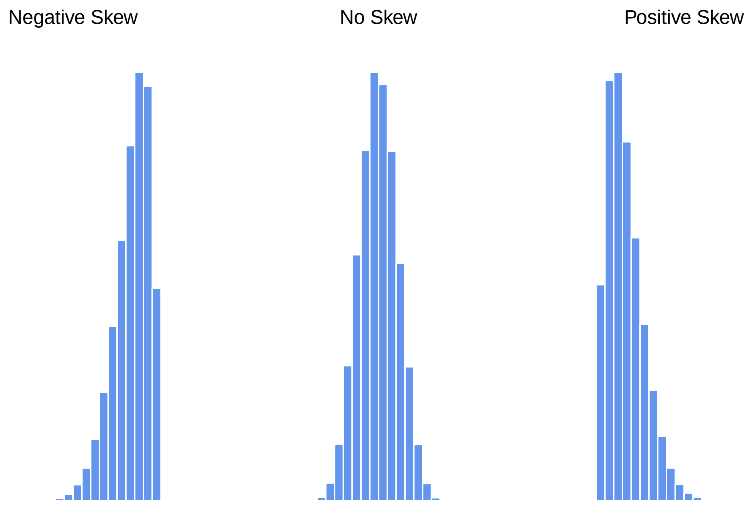


Figure 3.4: An illustration of skewness. On the left we have a negatively skewed data set (skewness $= -.93$), in the middle we have a data set with no skew (technically, skewness $= -.006$), and on the right we have a positively skewed data set (skewness $= .93$).

where N is the number of observations, $\bar{X}$ is the sample mean, and $\hat{\sigma}$ is the standard deviation (the "divide by $N-1$" version, that is). Perhaps more helpfully, it might be useful to point out that the `psych` package contains a `skew()` function that you can use to calculate skewness. So if we wanted to use this function to calculate the skewness of the `afl.margins` data, we'd first need to load the package

```
library( psych )
```

which now makes it possible to use the following command:

```
skew( x = afl.margins )
```

```
## [1] 0.7671555
```

Not surprisingly, it turns out that the AFL winning margins data is fairly skewed.

The final measure that is sometimes referred to, though very rarely in practice, is the ***kurtosis*** of a data set. Put simply, kurtosis is a measure of the "pointiness" of a data set, as illustrated in Figure 3.5.

```
## [1] -0.9520916
```

```
## [1] -0.01464293
```

```
## [1] 1.990522
```

By convention, we say that the "normal curve" (black lines) has zero kurtosis, so the pointiness of a data set is assessed relative to this curve. In this Figure, the data on the left are not pointy enough, so the kurtosis is negative and we call the data *platykurtic*. The data on the right are too pointy, so the kurtosis is positive and we say that the data is *leptokurtic*. But the data in the middle are just pointy enough, so we say that it is *mesokurtic* and has kurtosis zero. This is summarised in the table below:

informal term	technical name	kurtosis value
too flat	platykurtic	negative
just pointy enough	mesokurtic	zero
too pointy	leptokurtic	positive

The equation for kurtosis is pretty similar in spirit to the formulas we've seen already for the variance and the skewness; except that where the variance involved

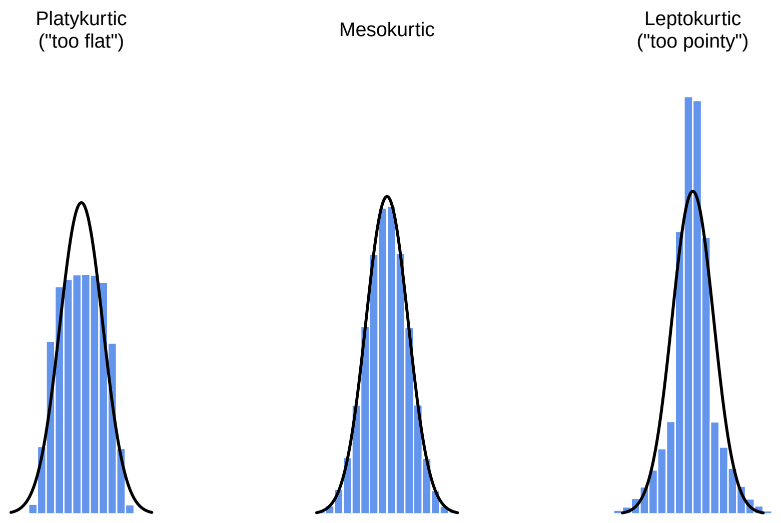


Figure 3.5: An illustration of kurtosis. On the left, we have a "platykurtic" data set (kurtosis = $-.95$), meaning that the data set is "too flat". In the middle we have a "mesokurtic" data set (kurtosis is almost exactly 0), which means that the pointiness of the data is just about right. Finally, on the right, we have a "leptokurtic" data set (kurtosis = 2.12) indicating that the data set is "too pointy". Note that kurtosis is measured with respect to a normal curve (black line)

squared deviations and the skewness involved cubed deviations, the kurtosis involves raising the deviations to the fourth power:[12]

$$\text{kurtosis}(X) = \frac{1}{N\hat{\sigma}^4} \sum_{i=1}^{N} \left(X_i - \bar{X} \right)^4 - 3$$

I know, it's not terribly interesting to me either. More to the point, the `psych` package has a function called `kurtosi()` that you can use to calculate the kurtosis of your data. For instance, if we were to do this for the AFL margins,

```
kurtosi( x = afl.margins )
```

```
## [1] 0.02962633
```

we discover that the AFL winning margins data are just pointy enough.

3.4 Getting an overall summary of a variable

Up to this point in the chapter I've explained several different summary statistics that are commonly used when analysing data, along with specific functions that you can use in R to calculate each one. However, it's kind of annoying to have to separately calculate means, medians, standard deviations, skews etc. Wouldn't it be nice if R had some helpful functions that would do all these tedious calculations at once? Something like `summary()` or `describe()`, perhaps? Why yes, yes it would. So much so that both of these functions exist. The `summary()` function is in the `base` package, so it comes with every installation of R. The `describe()` function is part of the `psych` package, which we loaded earlier in the chapter.

3.4.1 "Summarising" a variable

The `summary()` function is an easy thing to use, but a tricky thing to understand in full, since it's a generic function (see Section 2.24. The basic idea behind the `summary()` function is that it prints out some useful information about whatever object (i.e., variable, as far as we're concerned) you specify as the `object` argument. As a consequence, the behaviour of the `summary()` function differs quite dramatically depending on the class of the object that you give it. Let's start by giving it a *numeric* object:

[12]The "-3" part is something that statisticians tack on to ensure that the normal curve has kurtosis zero. It looks a bit stupid, just sticking a "-3" at the end of the formula, but there are good mathematical reasons for doing this.

```
summary( object = afl.margins )
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
##    0.00   12.75   30.50   35.30   50.50  116.00
```

For numeric variables, we get a whole bunch of useful descriptive statistics. It gives us the minimum and maximum values (i.e., the range), the first and third quartiles (25th and 75th percentiles; i.e., the IQR), the mean and the median. In other words, it gives us a pretty good collection of descriptive statistics related to the central tendency and the spread of the data.

Okay, what about if we feed it a logical vector instead? Let's say I want to know something about how many "blowouts" there were in the 2010 AFL season. I operationalise the concept of a blowout (see Chapter 1.6) as a game in which the winning margin exceeds 50 points. Let's create a logical variable `blowouts` in which the *i*-th element is `TRUE` if that game was a blowout according to my definition,

```
blowouts <-  afl.margins > 50
blowouts
```

```
##   [1]  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
##  [12]  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
##  [23] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE  TRUE FALSE FALSE
##  [34]  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [45] FALSE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE
##  [56]  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
##  [67]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
##  [78] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [89] FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
## [100] FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE  TRUE  TRUE  TRUE FALSE
## [111] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE
## [122]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE
## [133] FALSE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE
## [144]  TRUE  TRUE  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE
## [155]  TRUE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE
## [166] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

So that's what the `blowouts` variable looks like. Now let's ask R for a `summary()`

```
summary( object = blowouts )
```

```
##    Mode   FALSE    TRUE 
## logical     132      44
```

In this context, the `summary()` function gives us a count of the number of `TRUE` values, the number of `FALSE` values, and the number of missing values (i.e., the `NA`s). Pretty reasonable behaviour.

Next, let's try to give it a factor. If you recall, I've defined the `afl.finalists` vector as a factor, so let's use that:

```
summary( object = afl.finalists )
```

```
##          Adelaide          Brisbane           Carlton       Collingwood
##                26                25                26                28
##          Essendon           Fitzroy         Fremantle           Geelong
##                32                 0                 6                39
##          Hawthorn         Melbourne   North Melbourne     Port Adelaide
##                27                28                28                17
##          Richmond          St Kilda            Sydney        West Coast
##                 6                24                26                38
## Western Bulldogs
##                24
```

For factors, we get a frequency table, just like we got when we used the `table()` function. Interestingly, however, if we convert this to a character vector using the `as.character()` function (see Section **??**, we don't get the same results:

```
f2 <- as.character( afl.finalists )
summary( object = f2 )
```

```
##    Length     Class      Mode
##       400 character character
```

This is one of those situations I was referring to in Section 2.20, in which it is helpful to declare your nominal scale variable as a factor rather than a character vector. Because I've defined `afl.finalists` as a factor, R *knows* that it should treat it as a nominal scale variable, and so it gives you a much more detailed (and helpful) summary than it would have if I'd left it as a character vector.

3.4.2 "Summarising" a data frame

Okay what about data frames? When you pass a data frame to the `summary()` function, it produces a slightly condensed summary of each variable inside the data frame. To give you a sense of how this can be useful, let's try this for a new data set, one that you've never seen before. The data is stored in the `clinicaltrial.Rdata` file, and we'll use it a lot in Chapter **??** (you can find a complete description of the data at the start of that chapter). Let's load it, and see what we've got:

```
load( "./data/clinicaltrial.Rdata" )
who(TRUE)
```

```
##    -- Name --      -- Class --    -- Size --
##    clin.trial      data.frame     18 x 3
##     $drug          factor         18
##     $therapy       factor         18
##     $mood.gain     numeric        18
```

There's a single data frame called `clin.trial` which contains three variables, `drug`, `therapy` and `mood.gain`. Presumably then, this data is from a clinical trial of some kind, in which people were administered different drugs; and the researchers looked to see what the drugs did to their mood. Let's see if the `summary()` function sheds a little more light on this situation:

```
summary( clin.trial )
```

```
##        drug          therapy      mood.gain
##  placebo :6   no.therapy:9   Min.   :0.1000
##  anxifree:6   CBT       :9   1st Qu.:0.4250
##  joyzepam:6                  Median :0.8500
##                              Mean   :0.8833
##                              3rd Qu.:1.3000
##                              Max.   :1.8000
```

Evidently there were three drugs: a placebo, something called "anxifree" and something called "joyzepam"; and there were 6 people administered each drug. There were 9 people treated using cognitive behavioural therapy (CBT) and 9 people who received no psychological treatment. And we can see from looking at the summary of the `mood.gain` variable that most people did show a mood gain (mean = .88), though without knowing what the scale is here it's hard to say much more than that. Still, that's not too bad. Overall, I feel that I learned something from that.

3.4.3 "Describing" a data frame

The `describe()` function (in the `psych` package) is a little different, and it's really only intended to be useful when your data are interval or ratio scale. Unlike the `summary()` function, it calculates the same descriptive statistics for any type of variable you give it. By default, these are:

- `var`. This is just an index: 1 for the first variable, 2 for the second variable, and so on.

- `n`. This is the sample size: more precisely, it's the number of non-missing values.
- `mean`. This is the sample mean (Section 3.1.1).
- `sd`. This is the (bias corrected) standard deviation (Section 3.2.5).
- `median`. The median (Section 3.1.3).
- `trimmed`. This is trimmed mean. By default it's the 10% trimmed mean (Section 3.1.6).
- `mad`. The median absolute deviation (Section 3.2.6).
- `min`. The minimum value.
- `max`. The maximum value.
- `range`. The range spanned by the data (Section 3.2.1).
- `skew`. The skewness (Section 3.3).
- `kurtosis`. The kurtosis (Section 3.3).
- `se`. The standard error of the mean (Chapter **??**).

Notice that these descriptive statistics generally only make sense for data that are interval or ratio scale (usually encoded as numeric vectors). For nominal or ordinal variables (usually encoded as factors), most of these descriptive statistics are not all that useful. What the `describe()` function does is convert factors and logical variables to numeric vectors in order to do the calculations. These variables are marked with `*` and most of the time, the descriptive statistics for those variables won't make much sense. If you try to feed it a data frame that includes a character vector as a variable, it produces an error.

With those caveats in mind, let's use the `describe()` function to have a look at the `clin.trial` data frame. Here's what we get:

```
describe( x = clin.trial )
```

```
##           vars  n mean   sd median trimmed  mad min max range skew
## drug*        1 18 2.00 0.84   2.00    2.00 1.48 1.0 3.0   2.0 0.00
## therapy*     2 18 1.50 0.51   1.50    1.50 0.74 1.0 2.0   1.0 0.00
## mood.gain    3 18 0.88 0.53   0.85    0.88 0.67 0.1 1.8   1.7 0.13
##           kurtosis   se
## drug*        -1.66 0.20
## therapy*     -2.11 0.12
## mood.gain    -1.44 0.13
```

As you can see, the output for the asterisked variables is pretty meaningless, and should be ignored. However, for the `mood.gain` variable, there's a lot of useful information.

3.5 Descriptive statistics separately for each group

It is very commonly the case that you find yourself needing to look at descriptive statistics, broken down by some grouping variable. This is pretty easy to do in R, and there are three functions in particular that are worth knowing about: `by()`, `describeBy()` and `aggregate()`. Let's start with the `describeBy()` function, which is part of the `psych` package. The `describeBy()` function is very similar to the `describe()` function, except that it has an additional argument called `group` which specifies a grouping variable. For instance, let's say, I want to look at the descriptive statistics for the `clin.trial` data, broken down separately by `therapy` type. The command I would use here is:

```
describeBy( x=clin.trial, group=clin.trial$therapy )
```

```
##
##  Descriptive statistics by group
## group: no.therapy
##           vars n mean   sd median trimmed  mad min max range skew kurtosis
## drug*        1 9 2.00 0.87    2.0    2.00 1.48 1.0 3.0   2.0 0.00    -1.81
## therapy*     2 9 1.00 0.00    1.0    1.00 0.00 1.0 1.0   0.0  NaN      NaN
## mood.gain    3 9 0.72 0.59    0.5    0.72 0.44 0.1 1.7   1.6 0.51    -1.59
##             se
## drug*     0.29
## therapy*  0.00
## mood.gain 0.20
## --------------------------------------------------------
## group: CBT
##           vars n mean   sd median trimmed  mad min max range  skew
## drug*        1 9 2.00 0.87    2.0    2.00 1.48 1.0 3.0   2.0  0.00
## therapy*     2 9 2.00 0.00    2.0    2.00 0.00 2.0 2.0   0.0   NaN
## mood.gain    3 9 1.04 0.45    1.1    1.04 0.44 0.3 1.8   1.5 -0.03
##           kurtosis   se
## drug*        -1.81 0.29
## therapy*       NaN 0.00
## mood.gain    -1.12 0.15
```

As you can see, the output is essentially identical to the output that the `describe()` function produce, except that the output now gives you means, standard deviations etc separately for the `CBT` group and the `no.therapy` group. Notice that, as before, the output displays asterisks for factor variables, in order to draw your attention to the fact that the descriptive statistics that it has calculated won't be very meaningful for those variables. Nevertheless, this command has given us some really useful descriptive statistics `mood.gain` variable, broken down as a function of `therapy`.

A somewhat more general solution is offered by the `by()` function. There are three arguments that you need to specify when using this function: the `data` argument specifies the data set, the `INDICES` argument specifies the grouping variable, and the `FUN` argument specifies the name of a function that you want to apply separately to each group. To give a sense of how powerful this is, you can reproduce the `describeBy()` function by using a command like this:

```
by( data=clin.trial, INDICES=clin.trial$therapy, FUN=describe )
```

```
## clin.trial$therapy: no.therapy
##           vars n mean   sd median trimmed  mad min max range skew kurtosis
## drug*        1 9 2.00 0.87    2.0    2.00 1.48 1.0 3.0   2.0 0.00    -1.81
## therapy*     2 9 1.00 0.00    1.0    1.00 0.00 1.0 1.0   0.0  NaN      NaN
## mood.gain    3 9 0.72 0.59    0.5    0.72 0.44 0.1 1.7   1.6 0.51    -1.59
##             se
## drug*     0.29
## therapy*  0.00
## mood.gain 0.20
## --------------------------------------------------------
## clin.trial$therapy: CBT
##           vars n mean   sd median trimmed  mad min max range  skew
## drug*        1 9 2.00 0.87    2.0    2.00 1.48 1.0 3.0   2.0  0.00
## therapy*     2 9 2.00 0.00    2.0    2.00 0.00 2.0 2.0   0.0   NaN
## mood.gain    3 9 1.04 0.45    1.1    1.04 0.44 0.3 1.8   1.5 -0.03
##           kurtosis   se
## drug*        -1.81 0.29
## therapy*       NaN 0.00
## mood.gain    -1.12 0.15
```

This will produce the exact same output as the command shown earlier. However, there's nothing special about the `describe()` function. You could just as easily use the `by()` function in conjunction with the `summary()` function. For example:

```
by( data=clin.trial, INDICES=clin.trial$therapy, FUN=summary )
```

```
## clin.trial$therapy: no.therapy
##        drug         therapy    mood.gain
##  placebo :3   no.therapy:9   Min.   :0.1000
##  anxifree:3   CBT       :0   1st Qu.:0.3000
##  joyzepam:3                  Median :0.5000
##                              Mean   :0.7222
##                              3rd Qu.:1.3000
##                              Max.   :1.7000
## ------------------------------------------------------------
## clin.trial$therapy: CBT
##        drug          therapy    mood.gain
##  placebo :3   no.therapy:0   Min.   :0.300
##  anxifree:3   CBT       :9   1st Qu.:0.800
##  joyzepam:3                  Median :1.100
##                              Mean   :1.044
##                              3rd Qu.:1.300
##                              Max.   :1.800
```

Again, this output is pretty easy to interpret. It's the output of the `summary()` function, applied separately to `CBT` group and the `no.therapy` group. For the two factors (`drug` and `therapy`) it prints out a frequency table, whereas for the numeric variable (`mood.gain`) it prints out the range, interquartile range, mean and median.

What if you have multiple grouping variables? Suppose, for example, you would like to look at the average mood gain separately for all possible combinations of drug and therapy. It is actually possible to do this using the `by()` and `describeBy()` functions, but I usually find it more convenient to use the `aggregate()` function in this situation. There are again three arguments that you need to specify. The `formula` argument is used to indicate which variable you want to analyse, and which variables are used to specify the groups. For instance, if you want to look at `mood.gain` separately for each possible combination of `drug` and `therapy`, the formula you want is `mood.gain ~ drug + therapy`. The `data` argument is used to specify the data frame containing all the data, and the `FUN` argument is used to indicate what function you want to calculate for each group (e.g., the `mean`). So, to obtain group means, use this command:

```
aggregate( formula = mood.gain ~ drug + therapy,  # mood.gain by drug/therapy combina
           data = clin.trial,                     # data is in the clin.trial data fr
           FUN = mean                             # print out group means
)
```

```
##       drug    therapy mood.gain
## 1  placebo no.therapy  0.300000
## 2 anxifree no.therapy  0.400000
## 3 joyzepam no.therapy  1.466667
## 4  placebo        CBT  0.600000
## 5 anxifree        CBT  1.033333
## 6 joyzepam        CBT  1.500000
```

or, alternatively, if you want to calculate the standard deviations for each group, you would use the following command (argument names omitted this time):

```
aggregate( mood.gain ~ drug + therapy, clin.trial, sd )
```

```
##       drug    therapy mood.gain
## 1  placebo no.therapy 0.2000000
## 2 anxifree no.therapy 0.2000000
## 3 joyzepam no.therapy 0.2081666
## 4  placebo        CBT 0.3000000
## 5 anxifree        CBT 0.2081666
## 6 joyzepam        CBT 0.2645751
```

3.6 Good descriptive statistics are descriptive!

The death of one man is a tragedy. The death of millions is a statistic.

– Josef Stalin, Potsdam 1945

950,000 – 1,200,000

– Estimate of Soviet repression deaths, 1937-1938 (Ellman, 2002)

Stalin's infamous quote about the statistical character death of millions is worth giving some thought. The clear intent of his statement is that the death of an individual touches us personally and its force cannot be denied, but that the deaths of a multitude are incomprehensible, and as a consequence mere statistics, more easily ignored. I'd argue that Stalin was half right. A statistic is an abstraction, a description of events beyond our personal experience, and so hard to visualise. Few if any of us can imagine what the deaths of millions is "really" like, but we can imagine one death, and this gives the lone death its feeling of immediate tragedy, a feeling that is missing from Ellman's cold statistical description.

Yet it is not so simple: without numbers, without counts, without a description of what happened, we have *no chance* of understanding what really happened, no opportunity event to try to summon the missing feeling. And in truth, as I write this, sitting in comfort on a Saturday morning, half a world and a whole lifetime away from the Gulags, when I put the Ellman estimate next to the Stalin quote a dull dread settles in my stomach and a chill settles over me. The Stalinist repression is something truly beyond my experience, but with a combination of statistical data and those recorded personal histories that have come down to us, it is not entirely beyond my comprehension. Because what Ellman's numbers tell us is this: over a two year period, Stalinist repression wiped out the equivalent of every man, woman and child currently alive in the city where I live. Each one of those deaths had it's own story, was it's own tragedy, and only some of those are known to us now. Even so, with a few carefully chosen statistics, the scale of the atrocity starts to come into focus.

Thus it is no small thing to say that the first task of the statistician and the scientist is to summarise the data, to find some collection of numbers that can convey to an audience a sense of what has happened. This is the job of descriptive statistics, but it's not a job that can be told solely using the numbers. You are a data analyst, not a statistical software package. Part of your job is to take these *statistics* and turn them into a *description*. When you analyse data, it is not sufficient to list off a collection of numbers. Always remember that what you're really trying to do is communicate with a human audience. The numbers are important, but they need to be put together into a meaningful story that your audience can interpret. That means you need to think about framing. You need to think about context. And you need to think about the individual events that your statistics are summarising.

3.7 Drawing graphs

Above all else show the data.

–Edward Tufte[13]

Visualising data is one of the most important tasks facing the data analyst. It's important for two distinct but closely related reasons. Firstly, there's the matter of drawing "presentation graphics": displaying your data in a clean, visually appealing fashion makes it easier for your reader to understand what you're trying to tell them. Equally important, perhaps even more important, is the fact that drawing graphs helps *you* to understand the data. To that end, it's important to draw "exploratory graphics" that help you learn about the data as you go about analysing it. These points might seem pretty obvious, but I cannot count the number of times I've seen people forget them.

To give a sense of the importance of this chapter, I want to start with a classic illustration of just how powerful a good graph can be. To that end, Figure 3.6 shows a redrawing of one of the most famous data visualisations of all time: John Snow's 1854 map of cholera deaths. The map is elegant in its simplicity. In the background we have a street map, which helps orient the viewer. Over the top, we see a large number of small dots, each one representing the location of a cholera case. The larger symbols show the location of water pumps, labelled by name. Even the most casual inspection of the graph makes it very clear that the source of the outbreak is almost certainly the Broad Street pump. Upon viewing this graph, Dr Snow arranged to have the handle removed from the pump, ending the outbreak that had killed over 500 people. Such is the power of a good data visualisation.

[13]The origin of this quote is Tufte's lovely book *The Visual Display of Quantitative Information*.

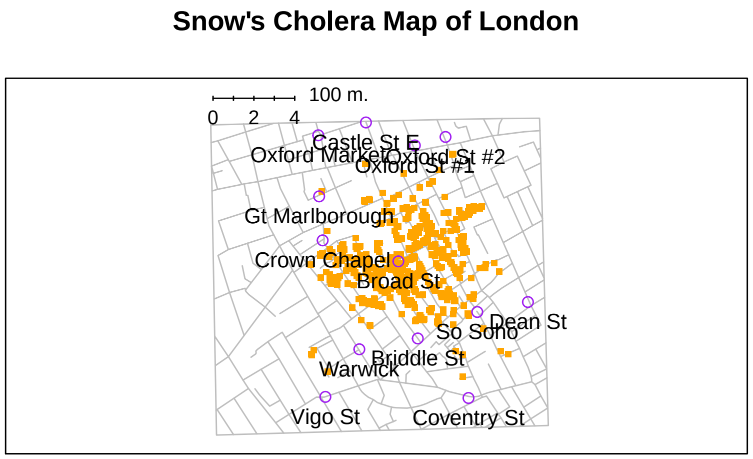


Figure 3.6: A stylised redrawing of John Snow's original cholera map. Each small dot represents the location of a cholera case, and each large circle shows the location of a well. As the plot makes clear, the cholera outbreak is centred very closely on the Broad St pump. This image uses the data from the `HistData` package, and was drawn using minor alterations to the commands provided in the help files. Note that Snow's original hand drawn map used different symbols and labels, but you get the idea.

The goals in this chapter are twofold: firstly, to discuss several fairly standard graphs that we use a lot when analysing and presenting data, and secondly, to show you how to create these graphs in R. The graphs themselves tend to be pretty straightforward, so in that respect this chapter is pretty simple. Where people usually struggle is learning how to produce graphs, and especially, learning how to produce good graphs.[14] Fortunately, learning how to draw graphs in R is reasonably simple, as long as you're not too picky about what your graph looks like. What I mean when I say this is that R has a lot of *very* good graphing functions, and most of the time you can produce a clean, high-quality graphic without having to learn very much about the low-level details of how R handles graphics. Unfortunately, on those occasions when you do want to do something non-standard, or if you need to make highly specific changes to the figure, you actually do need to learn a fair bit about the these details; and those details are both complicated and boring. With that in mind, the structure of this chapter is as follows: I'll start out by giving you a very quick overview of how graphics work in R. I'll then discuss several different kinds of graph and how to draw them, as well as showing the basics of how to customise these plots.

Dave note: Navarro (2018) included a fair amount of under-the-hood detail about how graphs are drawn in R. We do not need that level of theory for now. Instead, I'm including just the least you need to quickly visualize your data.

3.7.1 An introduction to plotting

Before I discuss any specialised graphics, let's start by drawing a few very simple graphs just to get a feel for what it's like to draw pictures using R. To that end, let's create a small vector `Fibonacci` that contains a few numbers we'd like R to draw for us. Then, we'll ask R to `plot()` those numbers. The result is Figure 3.7.

```
Fibonacci <- c( 1,1,2,3,5,8,13 )
plot( Fibonacci )
```

As you can see, what R has done is plot the *values* stored in the `Fibonacci` variable on the vertical axis (y-axis) and the corresponding *index* on the horizontal axis (x-axis). In other words, since the 4th element of the vector has a value of 3, we get a dot plotted at the location (4,3). That's pretty straightforward, and

[14]I should add that this isn't unique to R. Like everything in R there's a pretty steep learning curve to learning how to draw graphs, and like always there's a massive payoff at the end in terms of the quality of what you can produce. But to be honest, I've seen the same problems show up regardless of what system people use. I suspect that the hardest thing to do is to force yourself to take the time to think deeply about what your graphs are doing. I say that in full knowledge that only about half of my graphs turn out as well as they ought to. Understanding what makes a good graph is easy: actually designing a good graph is *hard*.

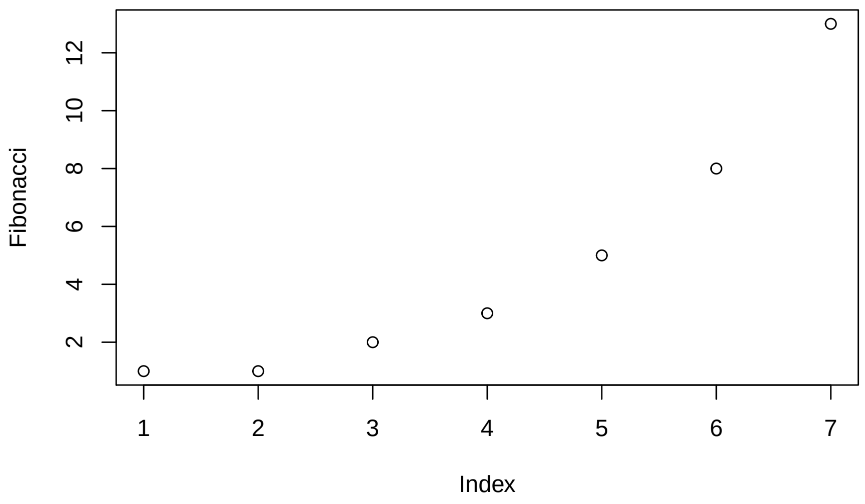


Figure 3.7: Our first plot

the image in Figure 3.7 is probably pretty close to what you would have had in mind when I suggested that we plot the `Fibonacci` data.

3.7.1.1 Customising the title and the axis labels

One of the first things that you'll find yourself wanting to do when customising your plot is to label it better. You might want to specify more appropriate axis labels, add a title or add a subtitle. The arguments that you need to specify to make this happen are:

- `main`. A character string containing the title.
- `sub`. A character string containing the subtitle.
- `xlab`. A character string containing the x-axis label.
- `ylab`. A character string containing the y-axis label.

These aren't graphical parameters, they're arguments to the high-level function. However, because the high-level functions all rely on the same low-level function to do the drawing[15] the names of these arguments are identical for pretty much every high-level function I've come across. Let's have a look at what happens

[15]The low-level function that does this is called `title()` in case you ever need to know, and you can type `?title` to find out a bit more detail about what these arguments do.

when we make use of all these arguments. Here's the command. The picture that this draws is shown in Figure 3.8.

```
plot( x = Fibonacci,
               main = "You specify title using the 'main' argument",
               sub = "The subtitle appears here! (Use the 'sub' argument for this)",
               xlab = "The x-axis label is 'xlab'",
                ylab = "The y-axis label is 'ylab'"
             )
```

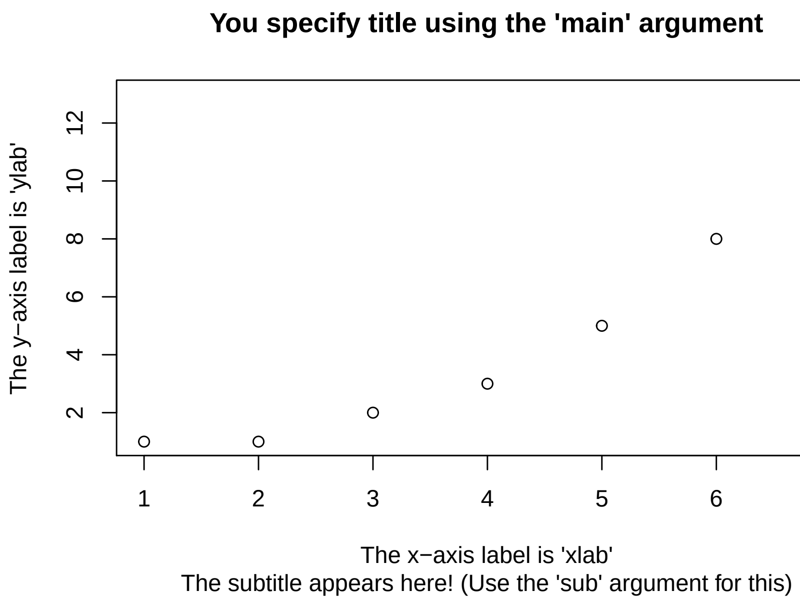


Figure 3.8: How to add your own title, subtitle, x-axis label and y-axis label to the plot.

It's more or less as you'd expect. The plot itself is identical to the one we drew in Figure 3.7, except for the fact that we've changed the axis labels, and added a title and a subtitle. Even so, there's a couple of interesting features worth calling your attention to. Firstly, notice that the subtitle is drawn below the plot, which I personally find annoying; as a consequence I almost never use subtitles. You may have a different opinion, of course, but the important thing is that you remember where the subtitle actually goes. Secondly, notice that R has decided to use boldface text and a larger font size for the title. This is one of my most hated default settings in R graphics, since I feel that it draws too much attention to the title. Generally, while I do want my reader to look at the title, I find that the R defaults are a bit overpowering, so I often like to change the settings. To that end, there are a bunch of graphical parameters that you can use to customise the font style:

- *Font styles:* `font.main`, `font.sub`, `font.lab`, `font.axis`. These four parameters control the font style used for the plot title (`font.main`), the subtitle (`font.sub`), the axis labels (`font.lab`: note that you can't specify separate styles for the x-axis and y-axis without using low level commands), and the numbers next to the tick marks on the axis (`font.axis`). Somewhat irritatingly, these arguments are numbers instead of meaningful names: a value of 1 corresponds to plain text, 2 means boldface, 3 means italic and 4 means bold italic.
- *Font colours:* `col.main`, `col.sub`, `col.lab`, `col.axis`. These parameters do pretty much what the name says: each one specifies a **col**our in which to type each of the different bits of text. Conveniently, R has a very large number of named colours (type `colours()` to see a list of over 650 colour names that R knows), so you can use the English language name of the colour to select it.[16] Thus, the parameter value here string like `"red"`, `"gray25"` or `"springgreen4"` (yes, R really does recognise four different shades of "spring green").
- *Font size:* `cex.main`, `cex.sub`, `cex.lab`, `cex.axis`. Font size is handled in a slightly curious way in R. The "cex" part here is short for "**c**haracter **ex**pansion", and it's essentially a magnification value. By default, all of these are set to a value of 1, except for the font title: `cex.main` has a default magnification of 1.2, which is why the title font is 20% bigger than the others.
- *Font family:* `family`. This argument specifies a font family to use: the simplest way to use it is to set it to `"sans"`, `"serif"`, or `"mono"`, corresponding to a san serif font, a serif font, or a monospaced font. If you want to, you can give the name of a specific font, but keep in mind that different operating systems use different fonts, so it's probably safest to keep it simple. Better yet, unless you have some deep objections to the R defaults, just ignore this parameter entirely. That's what I usually do.

To give you a sense of how you can use these parameters to customise your titles, the following command can be used to draw Figure 3.9:

```
plot( x = Fibonacci,                              # the data to plot
          main = "The first 7 Fibonacci numbers",  # the title
          xlab = "Position in the sequence",       # x-axis label
          ylab = "The Fibonacci number",           # y-axis
          font.main = 1,
          cex.main = 1,
```

[16]On the off chance that this isn't enough freedom for you, you can select a colour directly as a "red, green, blue" specification using the `rgb()` function, or as a "hue, saturation, value" specification using the `hsv()` function.

```
          font.axis = 2,
          col.lab = "gray50" )
```

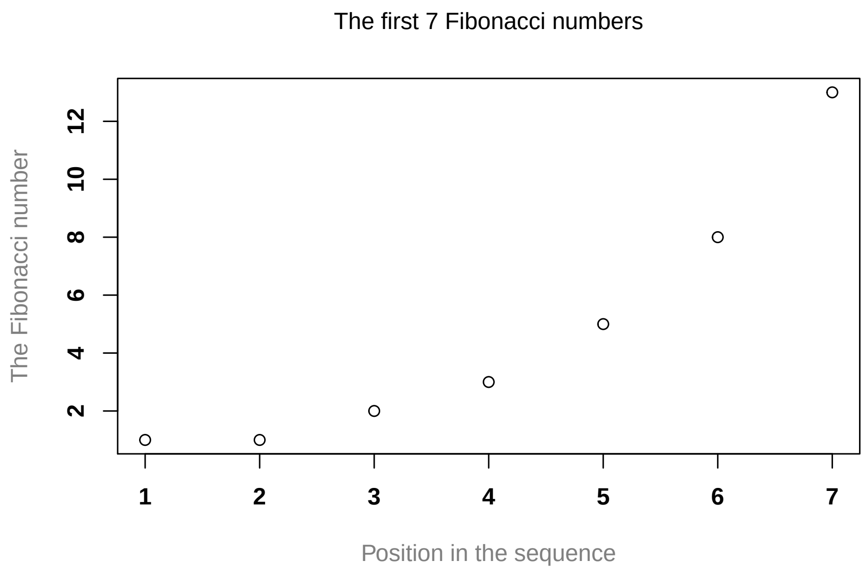


Figure 3.9: How to customise the appearance of the titles and labels.

Although this command is quite long, it's not complicated: all it does is override a bunch of the default parameter values. The only difficult aspect to this is that you have to remember what each of these parameters is called, and what all the different values are. And in practice I never remember: I have to look up the help documentation every time, or else look it up in this book.

3.7.2 Histograms

Now that we've tamed (or possibly fled from) the beast that is R graphical parameters, let's talk more seriously about some real life graphics that you'll want to draw. We begin with the humble ***histogram***. Histograms are one of the simplest and most useful ways of visualising data. They make most sense when you have an interval or ratio scale (e.g., the `afl.margins` data from Chapter 3 and what you want to do is get an overall impression of the data. Most of you probably know how histograms work, since they're so widely used, but for the sake of completeness I'll describe them. All you do is divide up the possible values into ***bins***, and then count the number of observations that fall within each bin. This count is referred to as the frequency of the bin, and is displayed as a bar: in the AFL winning margins data, there are 33 games in which the winning margin was less than 10 points, and it is this fact that is represented by the height of the leftmost bar in Figure 3.10. Drawing this histogram in R is pretty straightforward. The function you need to use is called `hist()`, and it has pretty reasonable default settings. In fact, Figure 3.10 is exactly what you get if you just type this:

```
hist( afl.margins )
```

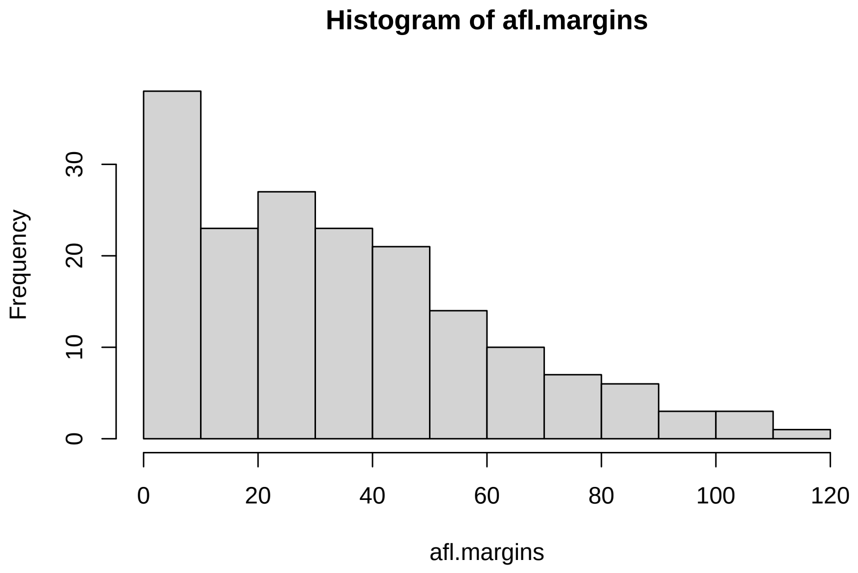


Figure 3.10: The default histogram that R produces

Although this image would need a lot of cleaning up in order to make a good presentation graphic (i.e., one you'd include in a report), it nevertheless does a pretty good job of describing the data. In fact, the big strength of a histogram is that (properly used) it does show the entire spread of the data, so you can get a pretty good sense about what it looks like. The downside to histograms is that they aren't very compact: unlike some of the other plots I'll talk about it's hard to cram 20-30 histograms into a single image without overwhelming the viewer. And of course, if your data are nominal scale (e.g., the `afl.finalists` data) then histograms are useless.

The main subtlety that you need to be aware of when drawing histograms is determining where the `breaks` that separate bins should be located, and (relatedly) how many breaks there should be. In Figure 3.10, you can see that R has made pretty sensible choices all by itself: the breaks are located at 0, 10, 20, ... 120, which is exactly what I would have done had I been forced to make a choice myself. On the other hand, consider the two histograms in Figure 3.11 and 3.12, which I produced using the following two commands:

```
hist( x = afl.margins, breaks = 3 )
```

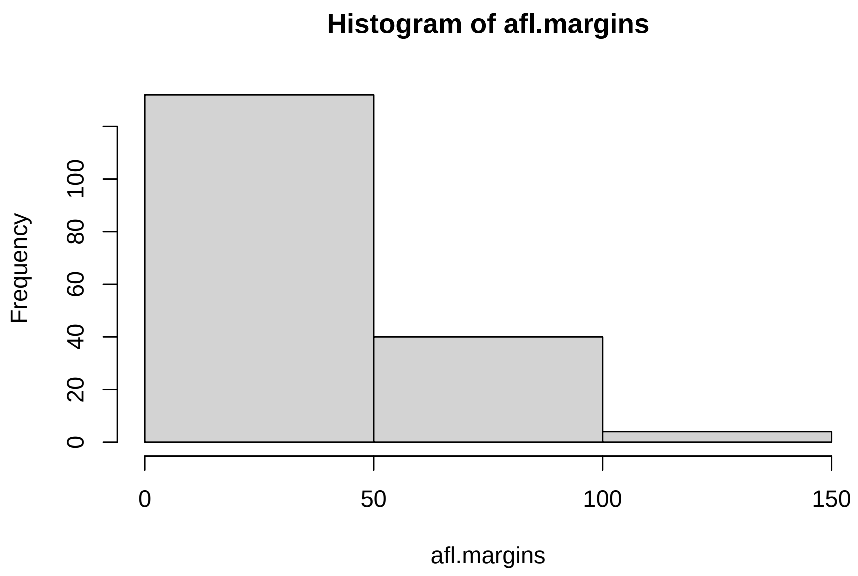


Figure 3.11: A histogram with too few bins

```
hist( x = afl.margins, breaks = 0:116 )
```

In Figure 3.12, the bins are only 1 point wide. As a result, although the plot is very informative (it displays the entire data set with no loss of information at all!) the plot is very hard to interpret, and feels quite cluttered. On the other hand, the plot in Figure 3.11 has a bin width of 50 points, and has the opposite problem: it's very easy to "read" this plot, but it doesn't convey a lot of information. One gets the sense that this histogram is hiding too much. In short, the way in which you specify the breaks has a big effect on what the histogram looks like, so it's important to make sure you choose the breaks sensibly. In general R does a pretty good job of selecting the breaks on its own, since it makes use of some quite clever tricks that statisticians have devised for automatically selecting the right bins for a histogram, but nevertheless it's usually a good idea to play around with the breaks a bit to see what happens.

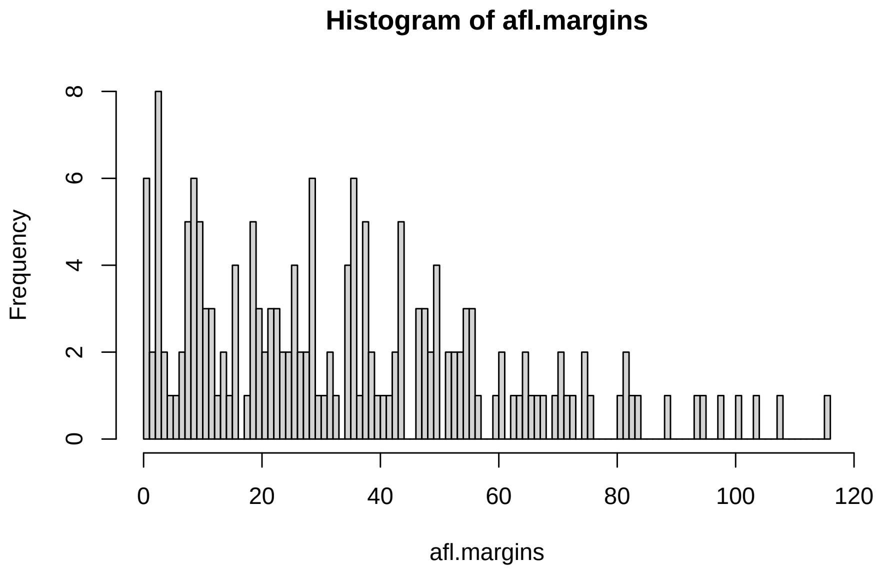


Figure 3.12: A histogram with too many bins

There is one fairly important thing to add regarding how the `breaks` argument works. There are two different ways you can specify the breaks. You can either specify *how many* breaks you want (which is what I did for panel b when I typed `breaks = 3`) and let R figure out where they should go, or you can provide a vector that tells R exactly where the breaks should be placed (which is what I did for panel c when I typed `breaks = 0:116`). The behaviour of the `hist()` function is slightly different depending on which version you use. If all you do is tell it *how many* breaks you want, R treats it as a "suggestion" not as a demand. It assumes you want "approximately 3" breaks, but if it doesn't think that this would look very pretty on screen, it picks a different (but similar) number. It does this for a sensible reason – it tries to make sure that the breaks are located at sensible values (like 10) rather than stupid ones (like 7.224414). And most of the time R is right: usually, when a human researcher says "give me 3 breaks", he or she really does mean "give me approximately 3 breaks, and don't put them in stupid places". However, sometimes R is dead wrong. Sometimes you really do mean "exactly 3 breaks", and you know precisely where you want them to go. So you need to invoke "real person privilege", and order R to do what it's bloody well told. In order to do that, you *have* to input the full vector that tells R exactly where you want the breaks. If you do that, R will go back to behaving like the nice little obedient calculator that it's supposed to be.

3.7.2.1 Visual style of your histogram

Okay, so at this point we can draw a basic histogram, and we can alter the number and even the location of the `breaks`. However, the visual style of the histograms shown in Figures 3.10, 3.11, and 3.12 could stand to be improved. We can fix this by making use of some of the other arguments to the `hist()` function. Most of the things you might want to try doing have already been covered in Section 3.7.1, but there's a few new things:

- *Shading lines*: `density`, `angle`. You can add diagonal lines to shade the bars: the `density` value is a number indicating how many lines per inch R should draw (the default value of `NULL` means no lines), and the `angle` is a number indicating how many degrees from horizontal the lines should be drawn at (default is `angle = 45` degrees).
- *Specifics regarding colours*: `col`, `border`. You can also change the colours: in this instance the `col` parameter sets the colour of the shading (either the shading lines if there are any, or else the colour of the interior of the bars if there are not), and the `border` argument sets the colour of the edges of the bars.
- *Labelling the bars*: `labels`. You can also attach labels to each of the bars using the `labels` argument. The simplest way to do this is to set `labels = TRUE`, in which case R will add a number just above each bar, that number being the exact number of observations in the bin. Alternatively, you can choose the labels yourself, by inputting a vector of strings, e.g., `labels = c("label 1","label 2","etc")`

Not surprisingly, this doesn't exhaust the possibilities. If you type `help("hist")` or `?hist` and have a look at the help documentation for histograms, you'll see a few more options. A histogram that makes use of the histogram-specific customisations as well as several of the options we discussed in Section 3.7.1 is shown in Figure 3.13. The R command that I used to draw it is this:

```r
hist( x = afl.margins,
      main = "2010 AFL margins", # title of the plot
      xlab = "Margin",           # set the x-axis label
      density = 10,              # draw shading lines: 10 per inch
      angle = 40,                # set the angle of the shading lines is 40 degrees
      border = "gray20",         # set the colour of the borders of the bars
      col = "gray80",            # set the colour of the shading lines
      labels = TRUE,             # add frequency labels to each bar
      ylim = c(0,40)             # change the scale of the y-axis
)
```

Overall, this is a much nicer histogram than the default ones.

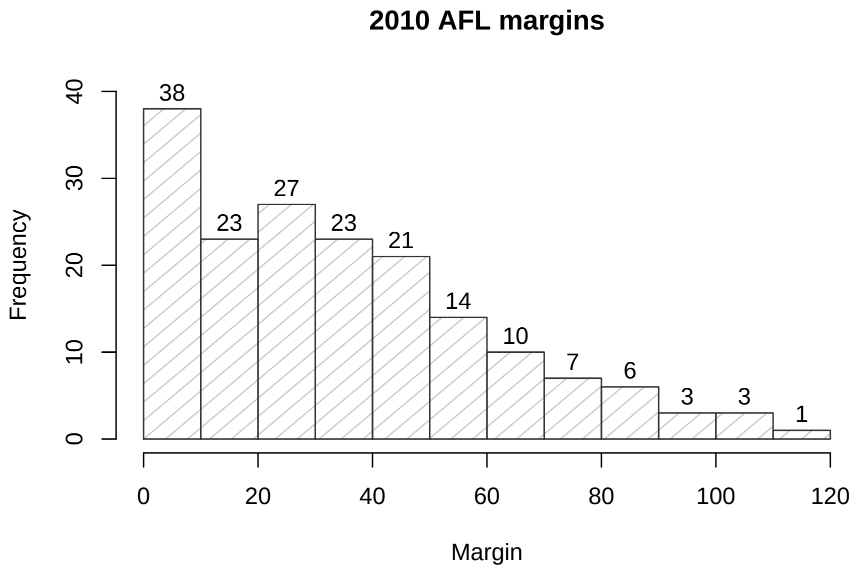


Figure 3.13: A histogram with histogram specific customisations

3.7.3 Boxplots

Another alternative to histograms is a ***boxplot***, sometimes called a "box and whiskers" plot. Like histograms, they're most suited to interval or ratio scale data. The idea behind a boxplot is to provide a simple visual depiction of the median, the interquartile range, and the range of the data. And because they do so in a fairly compact way, boxplots have become a very popular statistical graphic, especially during the exploratory stage of data analysis when you're trying to understand the data yourself. Let's have a look at how they work, again using the `afl.margins` data as our example. Firstly, let's actually calculate these numbers ourselves using the `summary()` function:[17]

```
summary( afl.margins )
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    0.00   12.75   30.50   35.30   50.50  116.00
```

So how does a boxplot capture these numbers? The easiest way to describe what a boxplot looks like is just to draw one. The function for doing this in R

[17]R being what it is, it's no great surprise that there's also a `fivenum()` function that does much the same thing.

is (surprise, surprise) `boxplot()`. As always there's a lot of optional arguments that you can specify if you want, but for the most part you can just let R choose the defaults for you. That said, I'm going to override one of the defaults to start with by specifying the `range` option, but for the most part you won't want to do this (I'll explain why in a minute). With that as preamble, let's try the following command:

```
boxplot( x = afl.margins, range = 100 )
```

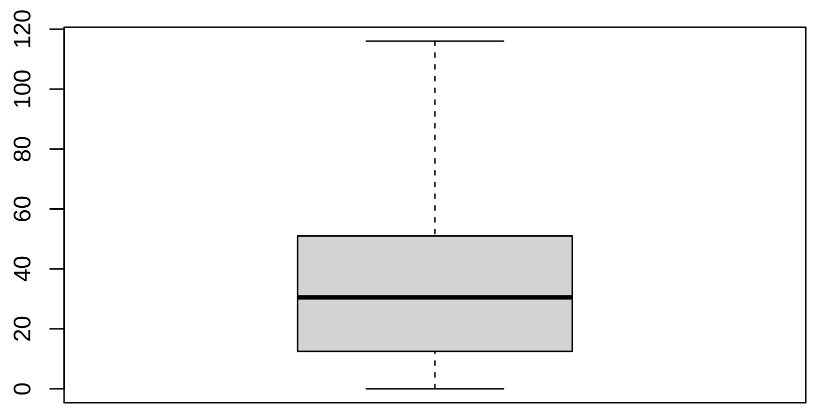


Figure 3.14: A basic boxplot

What R draws is shown in Figure 3.14, the most basic boxplot possible. When you look at this plot, this is how you should interpret it: the thick line in the middle of the box is the median; the box itself spans the range from the 25th percentile to the 75th percentile; and the "whiskers" cover the full range from the minimum value to the maximum value. This is summarised in the annotated plot in Figure 3.15.

In practice, this isn't quite how boxplots usually work. In most applications, the "whiskers" don't cover the full range from minimum to maximum. Instead, they actually go out to the most extreme data point that doesn't exceed a certain bound. By default, this value is 1.5 times the interquartile range, corresponding to a `range` value of 1.5. Any observation whose value falls outside this range is plotted as a circle instead of being covered by the whiskers, and is commonly referred to as an ***outlier***. For our AFL margins data, there is one observation

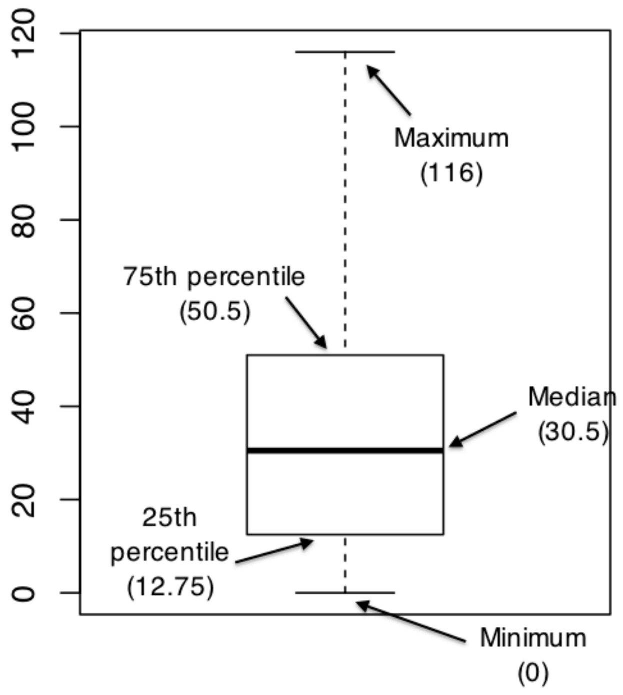


Figure 3.15: An annotated boxplot

(a game with a margin of 116 points) that falls outside this range. As a consequence, the upper whisker is pulled back to the next largest observation (a value of 108), and the observation at 116 is plotted as a circle. This is illustrated in Figure 3.16. Since the default value is `range = 1.5` we can draw this plot using the simple command

```
boxplot( afl.margins )
```

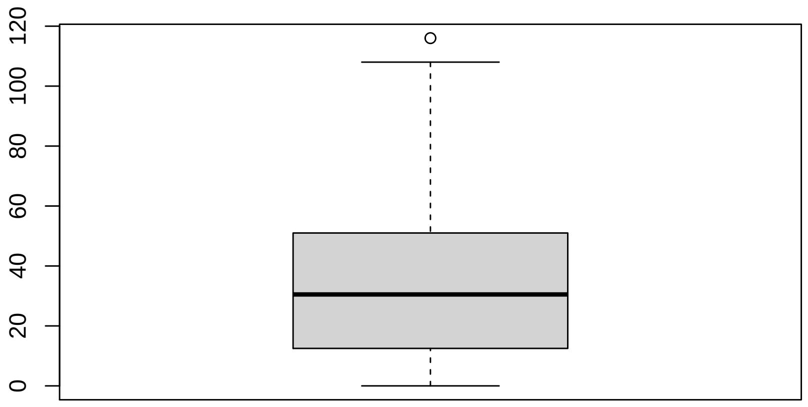


Figure 3.16: By default, R will only extent the whiskers a distance of 1.5 times the interquartile range, and will plot any points that fall outside that range separately

3.7.3.1 Visual style of your boxplot

I'll talk a little more about the relationship between boxplots and outliers in the Section 3.7.3.2, but before I do let's take the time to clean this figure up. Boxplots in R are extremely customisable. In addition to the usual range of graphical parameters that you can tweak to make the plot look nice, you can also exercise nearly complete control over every element to the plot. Consider the boxplot in Figure 3.17: in this version of the plot, not only have I added labels (`xlab`, `ylab`) and removed the stupid border (`frame.plot`), I've also dimmed all of the graphical elements of the boxplot except the central bar that plots the median (`border`) so as to draw more attention to the median rather than the rest of the boxplot.

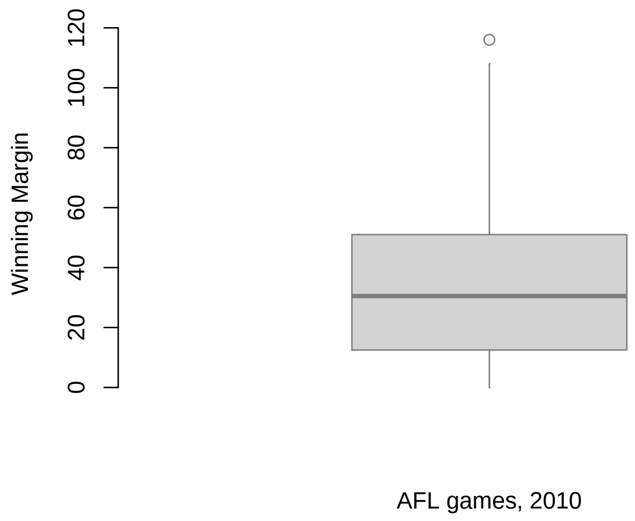


Figure 3.17: A boxplot with boxplot specific customisations

You've seen all these options in previous sections in this chapter, so hopefully those customisations won't need any further explanation. However, I've done two new things as well: I've deleted the cross-bars at the top and bottom of the whiskers (known as the "staples" of the plot), and converted the whiskers themselves to solid lines. The arguments that I used to do this are called by the ridiculous names of `staplewex` and `whisklty`,[18] and I'll explain these in a moment.

But first, here's the actual command I used to draw this figure:

```
boxplot( x = afl.margins,           # the data
         xlab = "AFL games, 2010",  # x-axis label
         ylab = "Winning Margin",   # y-axis label
         border = "grey50",         # dim the border of the box
         frame.plot = FALSE,        # don't draw a frame
         staplewex = 0,             # don't draw staples
```

[18]I realise there's a kind of logic to the way R names are constructed, but they still sound dumb. When I typed this sentence, all I could think was that it sounded like the name of a kids movie if it had been written by Lewis Carroll: "The frabjous gambolles of Staplewex and Whisklty" or something along those lines.

```
          whisklty = 1                # solid line for whisker
)
```

Overall, I think the resulting boxplot is a huge improvement in visual design over the default version. In my opinion at least, there's a fairly minimalist aesthetic that governs good statistical graphics. Ideally, every visual element that you add to a plot should convey part of the message. If your plot includes things that don't actually help the reader learn anything new, you should consider removing them. Personally, I can't see the point of the cross-bars on a standard boxplot, so I've deleted them.

3.7.3.2 Using box plots to detect outliers

Because the boxplot automatically (unless you change the `range` argument) separates out those observations that lie within a certain range, people often use them as an informal method for detecting ***outliers***: observations that are "suspiciously" distant from the rest of the data. Here's an example. Suppose that I'd drawn the boxplot for the AFL margins data, and it came up looking like Figure 3.18.

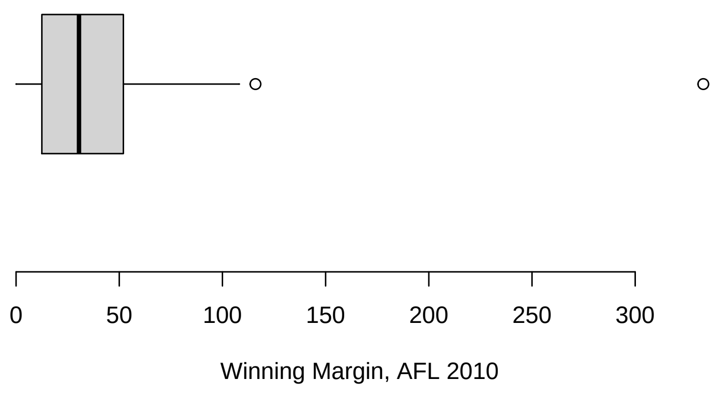


Figure 3.18: A boxplot showing one very suspicious outlier! I've drawn this plot in a similar, minimalist style to the one in Figure 3.17, but I've used the `horizontal` argument to draw it sideways in order to save space.

It's pretty clear that something funny is going on with one of the observations. Apparently, there was one game in which the margin was over 300 points! That doesn't sound right to me. Now that I've become suspicious, it's time to look a bit more closely at the data. One function that can be handy for this is the `which()` function; it takes as input a vector of logicals, and outputs the indices of the `TRUE` cases. This is particularly useful in the current context because it lets me do this:

```
suspicious.cases <- afl.margins > 300
which( suspicious.cases )
```

```
## [1] 137
```

although in real life I probably wouldn't bother creating the `suspicious.cases` variable: I'd just cut out the middle man and use a command like `which( afl.margins > 300 )`. In any case, what this has done is shown me that the outlier corresponds to game 137. Then, I find the recorded margin for that game:

```
afl.margins[137]
```

```
## [1] 333
```

Hm. That definitely doesn't sound right. So then I go back to the original data source (the internet!) and I discover that the actual margin of that game was 33 points. Now it's pretty clear what happened. Someone must have typed in the wrong number. Easily fixed, just by typing `afl.margins[137] <- 33`. While this might seem like a silly example, I should stress that this kind of thing actually happens a lot. Real world data sets are often riddled with stupid errors, especially when someone had to type something into a computer at some point. In fact, there's actually a name for this phase of data analysis, since in practice it can waste a huge chunk of our time: ***data cleaning***. It involves searching for typos, missing data and all sorts of other obnoxious errors in raw data files.[19]

What about the real data? Does the value of 116 constitute a funny observation not? Possibly. As it turns out the game in question was Fremantle v Hawthorn, and was played in round 21 (the second last home and away round of the season). Fremantle had already qualified for the final series and for them the outcome of the game was irrelevant; and the team decided to rest several

[19]Sometimes it's convenient to have the boxplot automatically label the outliers for you. The original `boxplot()` function doesn't allow you to do this; however, the `Boxplot()` function in the `car` package does. The design of the `Boxplot()` function is very similar to `boxplot()`. It just adds a few new arguments that allow you to tweak the labelling scheme. I'll leave it to the reader to check this out.

of their star players. As a consequence, Fremantle went into the game severely underpowered. In contrast, Hawthorn had started the season very poorly but had ended on a massive winning streak, and for them a win could secure a place in the finals. With the game played on Hawthorn's home turf[20] and with so many unusual factors at play, it is perhaps no surprise that Hawthorn annihilated Fremantle by 116 points. Two weeks later, however, the two teams met again in an elimination final on Fremantle's home ground, and Fremantle won comfortably by 30 points.[21]

So, should we exclude the game from subsequent analyses? If this were a psychology experiment rather than an AFL season, I'd be quite tempted to exclude it because there's pretty strong evidence that Fremantle weren't really trying very hard: and to the extent that my research question is based on an assumption that participants are genuinely trying to do the task. On the other hand, in a lot of studies we're actually interested in seeing the full range of possible behaviour, and that includes situations where people decide not to try very hard: so excluding that observation would be a bad idea. In the context of the AFL data, a similar distinction applies. If I'd been trying to make tips about who would perform well in the finals, I would have (and in fact did) disregard the Round 21 massacre, because it's way too misleading. On the other hand, if my interest is solely in the home and away season itself, I think it would be a shame to throw away information pertaining to one of the most distinctive (if boring) games of the year. In other words, the decision about whether to include outliers or exclude them depends heavily on *why* you think the data look they way they do, and what you want to use the data *for*. Statistical tools can provide an automatic method for suggesting candidates for deletion, but you really need to exercise good judgment here. As I've said before, R is a mindless automaton. It doesn't watch the footy, so it lacks the broader context to make an informed decision. You are *not* a mindless automaton, so you should exercise judgment: if the outlier looks legitimate to you, then keep it. In any case, I'll return to the topic again in Section **??**, so let's return to our discussion of how to draw boxplots.

3.7.3.3 Drawing multiple boxplots

One last thing. What if you want to draw multiple boxplots at once? Suppose, for instance, I wanted separate boxplots showing the AFL margins not just for

[20]Sort of. The game was played in Launceston, which is a de facto home away from home for Hawthorn.

[21]Contrast this situation with the next largest winning margin in the data set, which was Geelong's 108 point demolition of Richmond in round 6 at their home ground, Kardinia Park. Geelong have been one of the most dominant teams over the last several years, a period during which they strung together an incredible 29-game winning streak at Kardinia Park. Richmond have been useless for several years. This is in no meaningful sense an outlier. Geelong have been winning by these margins (and Richmond losing by them) for quite some time. Frankly I'm surprised that the result wasn't more lopsided: as happened to Melbourne in 2011 when Geelong won by a modest 186 points.

2010, but for every year between 1987 and 2010. To do that, the first thing we'll have to do is find the data. These are stored in the `aflsmall2.Rdata` file. So let's load it and take a quick peek at what's inside:

```
load( "aflsmall2.Rdata" )
who( TRUE )
#   -- Name --    -- Class --    -- Size --
#   afl2          data.frame     4296 x 2
#    $margin      numeric        4296
#    $year        numeric        4296
```

Notice that `afl2` data frame is pretty big. It contains 4296 games, which is far more than I want to see printed out on my computer screen. To that end, R provides you with a few useful functions to print out only a few of the row in the data frame. The first of these is `head()` which prints out the first 6 rows, of the data frame, like this:

```
head( afl2 )
```

```
##   margin year
## 1     33 1987
## 2     59 1987
## 3     45 1987
## 4     91 1987
## 5     39 1987
## 6      1 1987
```

You can also use the `tail()` function to print out the last 6 rows. The `car` package also provides a handy little function called `some()` which prints out a random subset of the rows.

In any case, the important thing is that we have the `afl2` data frame which contains the variables that we're interested in. What we want to do is have R draw boxplots for the `margin` variable, plotted separately for each separate `year`. The way to do this using the `boxplot()` function is to input a `formula` rather than a variable as the input. In this case, the formula we want is `margin ~ year`. So our boxplot command now looks like this. The result is shown in Figure 3.19.[22]

[22]Actually, there's other ways to do this. If the input argument `x` is a list object (see Section 2.22, the `boxplot()` function will draw a separate boxplot for each variable in that list. Relatedly, since the `plot()` function – which we'll discuss shortly – is a generic (see Section 2.24, you might not be surprised to learn that one of its special cases is a boxplot: specifically, if you use `plot()` where the first argument `x` is a factor and the second argument `y` is numeric, then the result will be a boxplot, showing the values in `y`, with a separate boxplot for each level. For instance, something like `plot(x = afl2\$year, y = afl2\$margin)` would work.

```
boxplot( formula = margin ~ year,
         data = afl2
)
```

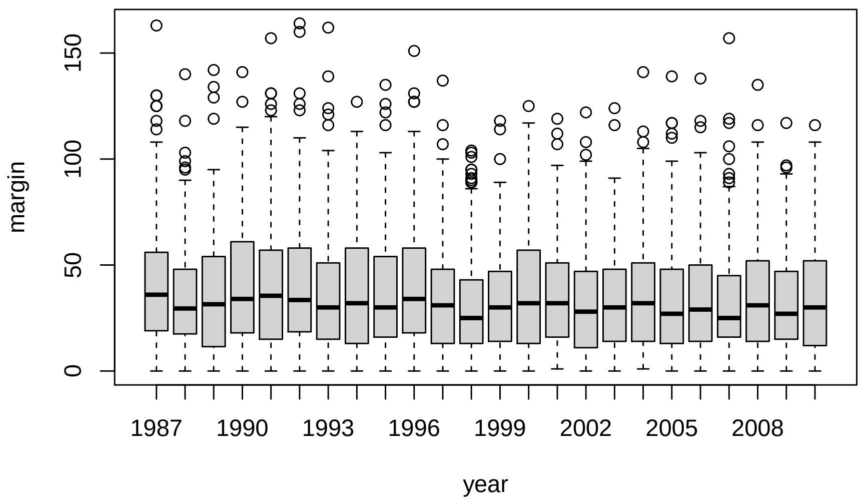


Figure 3.19: Boxplots showing the AFL winning margins for the 24 years from 1987 to 2010 inclusive. This is the default plot created by R, with no annotations added and no changes to the visual design. It's pretty readable, though at a minimum you'd want to include some basic annotations labelling the axes. Compare and contrast with Figure 3.20

Even this, the default version of the plot, gives a sense of why it's sometimes useful to choose boxplots instead of histograms. Even before taking the time to turn this basic output into something more readable, it's possible to get a good sense of what the data look like from year to year without getting overwhelmed with too much detail. Now imagine what would have happened if I'd tried to cram 24 histograms into this space: no chance at all that the reader is going to learn anything useful.

That being said, the default boxplot leaves a great deal to be desired in terms of visual clarity. The outliers are too visually prominent, the dotted lines look messy, and the interesting content (i.e., the behaviour of the median and the interquartile range across years) gets a little obscured. Fortunately, this is easy to fix, since we've already covered a lot of tools you can use to customise your output. After playing around with several different versions of the plot, the one

I settled on is shown in Figure 3.20. The command I used to produce it is long, but not complicated:

```
boxplot( formula =  margin ~ year,   # the formula
           data = afl2,                # the data set
           xlab = "AFL season",        # x axis label
           ylab = "Winning Margin",    # y axis label
           frame.plot = FALSE,         # don't draw a frame
           staplewex = 0,              # don't draw staples
           staplecol = "white",        # (fixes a tiny display issue)
           boxwex = .75,               # narrow the boxes slightly
           boxfill = "grey80",         # lightly shade the boxes
           whisklty = 1,               # solid line for whiskers
           whiskcol = "grey70",        # dim the whiskers
           boxcol = "grey70",          # dim the box borders
           outcol = "grey70",          # dim the outliers
           outpch = 20,                # outliers as solid dots
           outcex = .5,                # shrink the outliers
           medlty = "blank",           # no line for the medians
           medpch = 20,                # instead, draw solid dots
           medlwd = 1.5                # make them larger
 )
```

Of course, given that the command is that long, you might have guessed that I didn't spend ages typing all that rubbish in over and over again. Instead, I wrote a script, which I kept tweaking until it produced the figure that I wanted. We'll talk about scripts later in Section **??**, but given the length of the command I thought I'd remind you that there's an easier way of trying out different commands than typing them all in over and over.

3.7.4 Bar graphs

Another form of graph that you often want to plot is the ***bar graph***. The main function that you can use in R to draw them is the `barplot()` function.[23] And to illustrate the use of the function, I'll use the `finalists` variable that I introduced in Section 3.1.7. What I want to do is draw a bar graph that displays the number of finals that each team has played in over the time spanned by the `afl` data set. So, let's start by creating a vector that contains this information. I'll use the `tabulate()` function to do this (which will be discussed properly in Section **??**, since it creates a simple numeric vector:

[23]Once again, it's worth noting the link to the generic `plot()` function. If the `x` argument to `plot()` is a factor (and no `y` argument is given), the result is a bar graph. So you could use `plot( afl.finalists )` and get the same output as `barplot( afl.finalists )`.

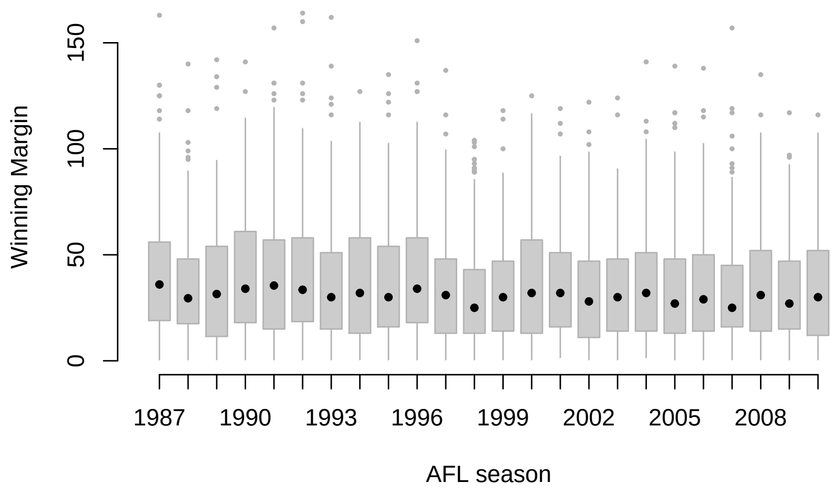


Figure 3.20: A cleaned up version of Figure 3.19. Notice that I've used a very minimalist design for the boxplots, so as to focus the eye on the medians. I've also converted the medians to solid dots, to convey a sense that year to year variation in the median should be thought of as a single coherent plot (similar to what we did when plotting the `Fibonacci` variable earlier). The size of outliers has been shrunk, because they aren't actually very interesting. In contrast, I've added a fill colour to the boxes, to make it easier to look at the trend in the interquartile range across years.

```
freq <- tabulate( afl.finalists )
print( freq )
```

```
##  [1] 26 25 26 28 32  0  6 39 27 28 28 17  6 24 26 38 24
```

This isn't exactly the prettiest of frequency tables, of course. I'm only doing it this way so that you can see the `barplot()` function in it's "purest" form: when the input is just an ordinary numeric vector. That being said, I'm obviously going to need the team names to create some labels, so let's create a variable with those. I'll do this using the `levels()` function, which outputs the names of all the levels of a factor (see Section 2.20:

```
teams <- levels( afl.finalists )
print( teams )
```

```
##  [1] "Adelaide"         "Brisbane"         "Carlton"
##  [4] "Collingwood"      "Essendon"         "Fitzroy"
##  [7] "Fremantle"        "Geelong"          "Hawthorn"
## [10] "Melbourne"        "North Melbourne"  "Port Adelaide"
## [13] "Richmond"         "St Kilda"         "Sydney"
## [16] "West Coast"       "Western Bulldogs"
```

Okay, so now that we have the information we need, let's draw our bar graph. The main argument that you need to specify for a bar graph is the `height` of the bars, which in our case correspond to the values stored in the `freq` variable:

```
barplot( height = freq )  # specifying the argument name
barplot( freq )           # the lazier version
```

Either of these two commands will produce the simple bar graph shown in Figure 3.21.

As you can see, R has drawn a pretty minimal plot. It doesn't have any labels, obviously, because we didn't actually tell the `barplot()` function what the labels are! To do this, we need to specify the `names.arg` argument. The `names.arg` argument needs to be a vector of character strings containing the text that needs to be used as the label for each of the items. In this case, the `teams` vector is exactly what we need, so the command we're looking for is:

```
    barplot( height = freq, names.arg = teams )
```

This is an improvement, but not much of an improvement. R has only included a few of the labels, because it can't fit them in the plot. This is the same behaviour

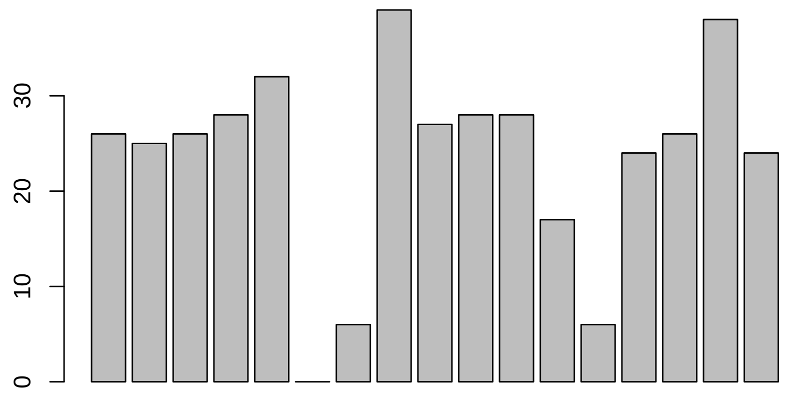


Figure 3.21: the simplest version of a bargraph, containing the data but no labels

we saw earlier with the multiple-boxplot graph in Figure 3.19. However, in Figure 3.19 it wasn't an issue: it's pretty obvious from inspection that the two unlabelled plots in between 1987 and 1990 must correspond to the data from 1988 and 1989. However, the fact that `barplot()` has omitted the names of every team in between Adelaide and Fitzroy is a lot more problematic.

The simplest way to fix this is to rotate the labels, so that the text runs vertically not horizontally. To do this, we need to alter set the `las` parameter, which I discussed briefly in Section 3.7.1. What I'll do is tell R to rotate the text so that it's always perpendicular to the axes (i.e., I'll set `las = 2`). When I do that, as per the following command...

```
barplot(height = freq,  # the frequencies
        names.arg = teams,  # the label
        las = 2)            # rotate the labels
```

... the result is the bar graph shown in Figure 3.23. We've fixed the problem, but we've created a new one: the axis labels don't quite fit anymore. To fix this, we have to be a bit cleverer again. A simple fix would be to use shorter names rather than the full name of all teams, and in many situations that's probably the right thing to do. However, at other times you really do need to create a bit more space to add your labels, so I'll show you how to do that.

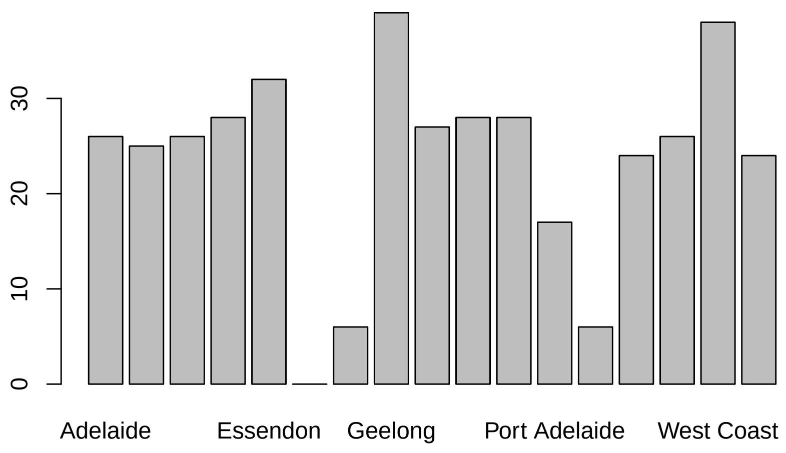


Figure 3.22: we've added the labels, but because the text runs horizontally R only includes a few of them

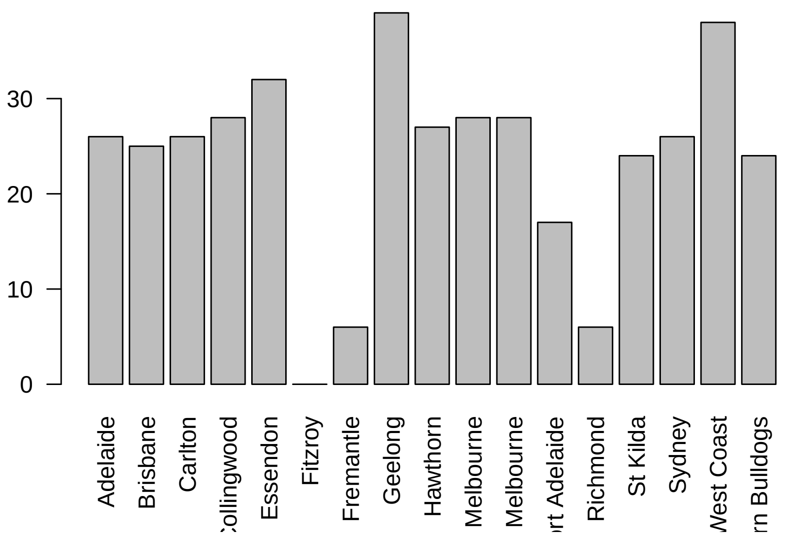


Figure 3.23: we've rotated the labels, but now the text is too long to fit

3.7.5 Saving image files using R and Rstudio

Hold on, you might be thinking. What's the good of being able to draw pretty pictures in R if I can't save them and send them to friends to brag about how awesome my data is? How do I save the picture? This is another one of those situations where the easiest thing to do is to use the RStudio tools.

If you're running R through Rstudio, then the easiest way to save your image is to click on the "Export" button in the Plot panel (i.e., the area in Rstudio where all the plots have been appearing). When you do that you'll see a menu that contains the options "Save Plot as PDF" and "Save Plot as Image". Either version works. Both will bring up dialog boxes that give you a few options that you can play with, but besides that it's pretty simple.

This works pretty nicely for most situations. So, unless you're filled with a burning desire to learn the low level details, feel free to skip the rest of this section.

3.7.6 Summary

Calculating some basic descriptive statistics is one of the very first things you do when analysing real data, and descriptive statistics are much simpler to understand than inferential statistics, so like every other statistics textbook I've started with descriptives. In this chapter, we talked about the following topics:

- *Measures of central tendency.* Broadly speaking, central tendency measures tell you where the data are. There's three measures that are typically reported in the literature: the mean, median and mode. (Section 3.1)
- *Measures of variability.* In contrast, measures of variability tell you about how "spread out" the data are. The key measures are: range, standard deviation, interquartile reange (Section 3.2)
- *Getting summaries of variables in R.* Since this book focuses on doing data analysis in R, we spent a bit of time talking about how descriptive statistics are computed in R. (Section 2.12 and 3.5)
- *Basic overview to R graphics.* In Section **??** we talked about how graphics in R are organised, and then moved on to the basics of how they're drawn in Section 3.7.1.
- *Common plots.* Much of the chapter was focused on standard graphs that statisticians like to produce: histograms (Section 3.7.2), boxplots (Section 3.7.3), and bar graphs (Section 3.7.4).

A traditional first course in statistics spends only a small proportion of the class on descriptive statistics, maybe one or two lectures at most. The vast majority of the lecturer's time is spent on inferential statistics, because that's where all the hard stuff is. That makes sense, but it hides the practical everyday importance of choosing good descriptives.

Dave note: With this chapter, I condensed two of Navarro (2018)'s chapters: descriptive stats and visualizations. For our course, the tools we need at this moment are the ones that help us describe single variables. We will introduce additional tools for summarizing and visualizing bivariate (two variable) data a bit later in the course, when we need them. And, if you do not feel comfortable with every customiziation option presented here, do not be concerned. For now, if you can generate histograms, bar charts, and the like, you have what you need. We will learn more about customizing these graphs for presentations later, as well.

Bibliography

Bickel, P. J., Hammel, E. A., and O'Connell, J. W. (1975). Sex bias in graduate admissions: Data from Berkeley. *Science*, 187:398–404.

Campbell, D. T. and Stanley, J. C. (1963). *Experimental and Quasi-Experimental Designs for Research*. Houghton Mifflin, Boston, MA.

Ellman, M. (2002). Soviet repression statistics: some comments. *Europe-Asia Studies*, 54(7):1151–1172.

Fox, J. and Weisberg, S. (2011). *An R Companion to Applied Regression*. Sage, Los Angeles, 2nd edition.

Navarro, D. (2018). *Learning statistics with R: A tutorial for psychology students and other beginners (version 0.6)*.

R Core Team (2013). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

Stevens, S. S. (1946). On the theory of scales of measurement. *Science*, 103:677–680.