Introduction to Data Technologies
WORKING DRAFT

Paul Murrell

May 19, 2008

Preface

The basic premise of this book is that scientists are required to perform many tasks with data other than statistical analyses. A lot of time and effort is usually invested in getting data ready for analysis: collecting the data, storing the data, transforming and subsetting the data, and transferring the data between different operating systems and applications.

Many scientists acquire data management skills in an ad hoc manner, as problems arise in practice. In most cases, skills are self-taught or passed down, guild-like, from master to apprentice. This book aims to provide a more structured and more complete introduction to the skills required for managing data.

The focus of this book is on computational tools that make the management of data faster, more accurate, and more efficient. The intention is to improve the awareness of what sorts of tasks can be achieved and to describe the correct approach to performing these tasks and there is an emphasis on working with data technologies via written computer languages.

This book will not turn the reader into a web designer, or a database administrator, or a software engineer. However, this book contains information on how to collect and publish information via the world wide web, how to access information stored in different formats, and how to write small programs to automate simple, repetitive tasks.

This book is intended to improve the work habits of individual researchers. It aims to provide a level of understanding that enables a scientist to access and interact with data sets no matter where or how they are stored.

This book is designed to be accessible and practical, with an emphasis on useful, applicable information. Each topic is covered in three different ways: initially, basic ideas are introduced, in an appropriate order, and using trivial examples, to give a quick, easy to read overview of the topic; this is followed by case studies which combine ideas and techniques together and provide demonstrations of more sophisticated and real-life use; finally, there are separate reference chapters, which contain almost no examples, just the bare information for easy look-up.

This book is written primarily for statisticians and this is reflected in the broad range of data sets used in the examples. However, the content should be relevant for anyone whose work involves the collection, preparation, or

analysis of data.

Writing code

The icon below was captured from the desktop of a computer running Microsoft Windows XP.

Is this document a Microsoft Office Excel spreadsheet?

Many computer users would say that it is. After all, it has got the little Excel image on it and it even says Microsoft Office Excel right below the name of the file. And if we double-clicked on this file, Excel would start up and open the file.

However, this file is not an Excel spreadsheet. It is a plain text file in a Comma-Separated Values (CSV) format. In fact the name of the file is not final, but final.csv. Excel can open this file, but so can thousands of other computer programs.

The computer protects us from this gritty detail by not showing the .csv suffix on the file name and it provides the convenience of automatically using Excel to open the file, rather than asking us what program to use.

Is this somehow a bad thing?

Yes, it is.

A computer user who only works with this sort of interface learns that this sort of file is only for use with Excel. The user becomes accustomed to the computer dictating what the user is able to do with a file.

It is important that users understand that they are able to dictate to the computer what should be done with a file. A CSV file can be viewed and modified using software as simple as Microsoft Notepad, but this does not occur to a user who is used to being told to use Excel.

For the majority of computer users, interaction with a computer is limited to clicking on web page hyperlinks, selecting menus, and filling in dialog boxes. The problem with this approach to computing is that it gives the impression that the user is controlled by the computer. The computer interface places limits on what the user can do.

The truth is of course exactly the opposite. It is the computer user who has control and can tell the computer exactly what to do. Learning to interact with a computer by writing computer code places users in their rightful position of power.

Computer code also has the huge advantage of providing an accurate record of the tasks that were performed. This serves both as a reminder of what was done and a recipe that allows others to replicate what was done.

For these reasons, this book focuses on computer languages as tools for data management.

Open standards and open source

This book almost exclusively describes technologies that are described by open standards or that are implemented in open source software, or both.[1]

For a technology to be an open standard, it must be described by a public document that provides enough information so that anyone can write software to work with technology. In addition, the description must not be subject to patents or other restrictions of use. Ideally, the document is published and maintained by an international, non-profit organisation. In practice, the important consequence is that the technology is not bound to a single software product.

This is in contrast to proprietary technologies, where the definitive description of the technology is not made available and is only supported by a single software product.

Open source software is software for which the source code is publicly available. This makes it possible, through scrutiny of the source code if necessary, to understand how a software product works. It also means that, if necessary, the behaviour of the software can be modified. In practice, the important consequence is that the software is not bound to a single software developer.

This is in contrast to proprietary software, where the software is only available from a single developer, the software is a "black-box", and changes, including corrections of errors, can only be made by the software developer.

The obvious advantage of using open standards and open source software is that the reader need not purchase any expensive proprietary software in order to benefit from the information in this book, but that is not the

[1] With one exception; see Section 7.6.

primary reason for this choice.

The main reason for selecting open standards and open source software is that this is the only way to ensure that we know where our data are on the computer and what happens to our data when we manipulate it with software, and it is the only way to guarantee that we can have free access to our data now and in the future.

The significance of these points is demonstrated by the growing list of governments and public institutions that are switching to open standards and open source software for storing and working with information.[2] In particular, for the storage of public records, it does not make sense to lock the information up in a format that cannot be accessed except by proprietary software. Similarly, for the dissemination and reproducibility of scientific research, it makes sense to fully disclose a complete description of how an analysis was conducted in addition to publishing the research results.

How to read this book

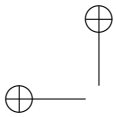
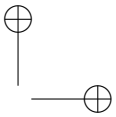
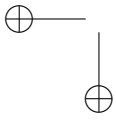
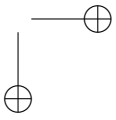
This book was developed from a set of lecture notes for a second-year course for statistics students. As a consequence, it is written to tell a story. There is an ordering of topics from data collection, through data storage and retrieval, to data processing. There is also a development from writing simple computer code with straightforward computer languages through to more complex tasks with more sophisticated languages. Furthermore, examples and case studies are carried over between different chapters in an attempt to illustrate how the different technologies need to be combined over the lifetime of a data set. In this way, the book is set up to be read in order from start to finish.

However, every effort has been made to ensure that individual chapters can be read on their own. Where necessary, figures are reproduced and descriptions are repeated so that it is not necessary to jump back and forth within the book in order to acquire a complete understanding of a particular section.

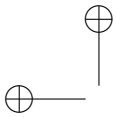
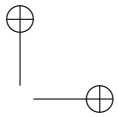
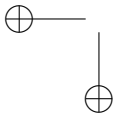
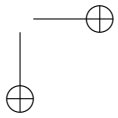
The addition of separate reference chapters is designed to allow the reader to quickly dip back into the book in order to refresh knowledge of a particular technology.

[2] High profile cases include: ...

Notation

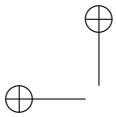
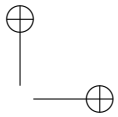
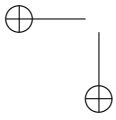
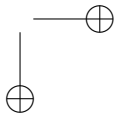





Brief Contents

Full Contents

List of Figures

List of Tables

1 Introduction

1.1 Case Study: Point Nemo

The Pacific Ocean is the largest body of water on Earth.[1]

The Live Access Server is one of many services provided by the National Aeronautics and Space Administration (NASA) for gaining access to their enormous repositories of atmospheric and astronomical data. The Live Access Server[2] provides access to atmospheric data from NASA's fleet of Earth-observing satellites, data that consists of coarsely gridded measurements of major atmospheric variables, such as ozone, cloud cover, pressure, and temperature. NASA provides a web site that allows researchers to select variables of interest, and geographic and temporal ranges, and then to download or view the relevant data (see Figure 1.1). Using this service, we can attempt to answer questions about atmospheric and weather conditions in different parts of the world.

The Pacific Pole of Inaccessibility is a location in the Southern Pacific Ocean that is recognised as one of the most remote locations on Earth. Also known as Point Nemo, it is the point on the ocean that is farthest from any land mass.[3] Its counterpart, the Eurasian Pole of Inaccessibility, in northern China, is the location on land that is farthest from any ocean.[4]

[1] Image source: Particle Dynamics Group, Department of Oceanography, Texas A&M University `http://www-ocean.tamu.edu/~pdgroup/jpegs/waves.jpg`
Used and distributed with permission.

[2] `http://mynasadata.larc.nasa.gov/LASintro.html`

[3] Longitude 123.4 west and latitude 48.9 south.

[4] Longitude 86.7 west and latitude 46.3 north.

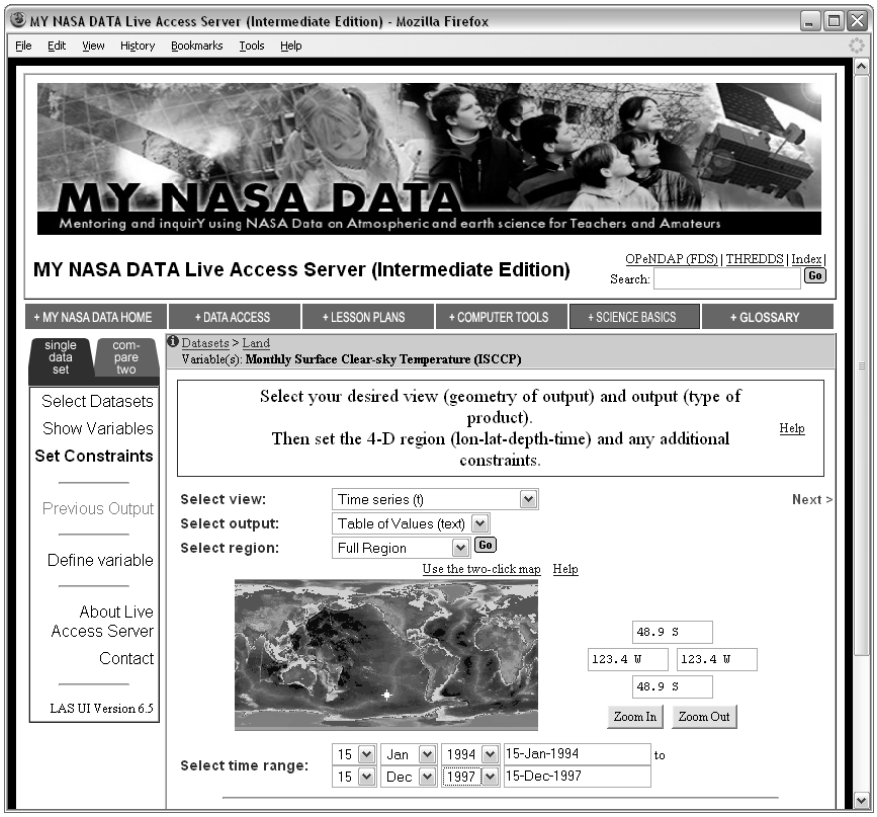


Figure 1.1: NASA's Live Access Server web site. On the map, the Pacific Pole of Inaccessibility is marked by a white plus sign.

These two geographical extremes—one in the southern hemisphere, over 2,500 km from the nearest land, and one in the northern hemisphere, over 2,500 km from the nearest ocean—are usually only of interest either to intrepid explorers or conspiracy theorists (a remote location is the perfect place to hide an important secret!). However, our interest will be to investigate the differences in weather conditions between these interesting geographical extremes by using NASA's Live Access Server.

To make our task a little more manageable, for now we will restrict our attention to a comparison of the surface temperatures at each of the Poles of Inaccessibility. To be precise, we will look at monthly average temperatures at these locations from January 1994 to December 1997.

In a book on data analysis, we would assume that the data are already in a form that can be conveniently loaded into statistical software, and the emphasis would be on how to analyse these data. However, that is not the focus of this book. Here, we are interested in all of the steps that must be taken *before* the data can be conveniently loaded into statistical software.

As anyone who has worked with data knows, it often takes more time and effort to get the data ready than it takes to perform the data analysis. And yet there are many more books on how to analyse data than there are on how to prepare data for analysis. This book aims to redress that balance.

In our example, the main data collection has already occurred; the data are measurements made by instruments on NASA satellites. However, we still need to collect the data from NASA's Live Access Server. We will do this initially by entering the appropriate parameters on the Live Access Server web site. Figure 1.2 shows the first few lines of data that the Live Access Server returns for the surface temperature at Point Nemo.

The first thing we should always do with a new data set is take a look at the **raw data**. Viewing the raw data is an important first step in becoming familiar with the data set. We should never automatically assume that the data are reliable or correct. We should always check with our own eyes. In this case, we are already in for a bit of a shock.

As an antipodean, I expect temperatures to be in degrees Celsius, so values like 278.9 make me break into a sweat. Even if we expect temperatures on the Fahrenheit scale, 278.9 is hotter than the average summer's day.

The problem of course is that these are scientific measurements, so the scale being used is Kelvin; the temperature scale where zero really means zero. 278.9 K is 5.8°C or 42°F, which is a cool, but entirely believable surface temperature value. When planning a visit to Point Nemo, it would be a good idea to pack a sweater.

```
             VARIABLE : Mean TS from clear sky composite (kelvin)
             FILENAME : ISCCPMonthly_avg.nc
             FILEPATH : /usr/local/fer_data/data/
             SUBSET   : 48 points (TIME)
             LONGITUDE: 123.8W(-123.8)
             LATITUDE : 48.8S
                        123.8W
                         23
 16-JAN-1994 00 /  1:  278.9
 16-FEB-1994 00 /  2:  280.0
 16-MAR-1994 00 /  3:  278.9
 16-APR-1994 00 /  4:  278.9
 16-MAY-1994 00 /  5:  277.8
 16-JUN-1994 00 /  6:  276.1
 ...
```

Figure 1.2: The first few lines of output from the Live Access Server for the surface temperature at Point Nemo.

Looking at the raw data, we also see a lot of other information besides the surface temperatures. There are longitude and latitude values, dates, and a description of the variable that has been measured, including the units of measurement. This **metadata** is very important because it provides us with a proper understanding of the data set. For example, the metadata makes it clear that the temperature values are on the Kelvin scale. The metadata also tells us that the longitude and latitude values, 123.8 W and 48.8 S are not exactly what we asked for. It turns out that the values provided by NASA in this data set have been averaged over a large area so this is as good as we are going to get.

We have established that the data seem credible, but do we have any reason to believe that the data are accurate or error-free? This comes down to how the data were **recorded**. In this case, the data were recorded by machines (satellites) and only ever existed in an **electronic format**. In particular, no humans got in the way, and there was no need for a **data entry** step. This increases our level of trust in the data.[5]

We should also notice that information does not just flow from the Live Access Server to us. Information is also sent from us to the Live Access Server to specify the exact data that we require. This is done via a web-based **electronic form**, which has several advantages: we can only ask for

[5]The fact that the data are provided by NASA also provides us with a certain level of confidence, although see Section 9.2.2 for an example of why we should always check the data no matter how much we trust the original source.

data that the Live Access Server is able to supply because there are only links for the available data; we can only ask for data that makes sense, for example the form does not allow us to enter latitudes beyond 90 north or 90 south; and we can get the data immediately—we do not have to wait for the postal service to deliver a paper form to NASA. These issues of how best to record data, and in particular how to get information into an electronic format are discussed in Chapter 5.

Before we go forward, we should take a step back and acknowledge the fact that we are able to read the data at all. This is a benefit of the **storage format** that the data are in; in this case, it is a **plain text** format. If the data had been in a more sophisticated **binary** format, we would need something more specialised than a common web browser to be able to view our data. In Chapter 7 we will spend a lot of time looking at the advantages and disadvantages of different data storage formats.

Having had a look at the raw data, the next step in familiarising ourselves with the data set should be to look at some numerical summaries and plots. The Live Access Server does not provide numerical summaries and, although it will produce some basic plots, we will need a bit more flexibility. So we will save the data to our own computer and load it into a statistical software package.

The first step is to save the data. The Live Access Server will provide an **ASCII file** for us, or we can just copy-and-paste the data into a **text editor** and save it from there. Again, we should appreciate the fact that this step is quite straightforward and is likely to work no matter what sort of computer or operating system we are using. This is another feature of having data in a plain text format.

Now we need to get the data into our statistical software. At this point, we encounter one of the disadvantages of a plain text format. Although we, as human readers, can see that the surface temperature values start on the ninth line and are the last value on each row (see Figure 1.2), there is no way that statistical software can figure this out on its own. We will have to describe the format of the data set to our statistical software.

In order to read the data into statistical software, we need to be able to express the following information: “skip the first 8 lines”; and “on each row, the values are separated by whitespace (one or more spaces or tabs)”; and “on each row, the date is the first value and the temperature is the last value (ignore the other three values)”. Here is one way to do this for the statistical software package R (Chapter 11 has much more to say about working with data in R):

```
read.table("PointNemo.txt", skip=8,
           colClasses=c("character",
                        "NULL", "NULL", "NULL",
                        "numeric"),
           col.names=c("date", "", "", "", "temp"))
```

This solution may appear complex, especially for anyone not experienced in writing computer code. Partly that is because this is complex information that we need to communicate to the computer and writing code is the best or even the only way to express that sort of information. However, the complexity of writing computer code gains us many benefits. For example, having written this piece of code to load in the data for the Pacific Pole of Inaccessibility, we can use it again, with only a change in the name of the file, to read in the data for the Eurasian Pole of Inaccessibility. That would look like this:

```
read.table("Eurasia.txt", skip=8,
           colClasses=c("character",
                        "NULL", "NULL", "NULL",
                        "numeric"),
           col.names=c("date", "", "", "", "temp"))
```

Imagine if we wanted to load in temperature data in this format from several hundred other locations around the world. Loading in such volumes of data would now be trivial and fast using code like this; performing such a task by selecting menus and filling in dialog boxes hundreds of times does not bear thinking about.

Going back a step, if we wanted to download the data for hundreds of locations around the world, would we want to fill in the Live Access Server web form hundreds of times? I most certainly would not. Here again, we can write code to make the task faster, more accurate, and more bearable.

As well as the web interface to the Live Access Server, it is also possible to make requests by **writing code** to communicate with the Live Access Server. Here is the code to ask for the temperature data from the Pacific Pole of Inaccessibility:

```
lasget.pl -x -123.4 -y -48.9 -t 1994-Jan-1:2001-Sep-30 \
          -f txt \
          http://mynasadata.larc.nasa.gov/las-bin/LASserver.pl \
          ISCCPMonthly_avg_nc ts
```

Again, that may appear complex, and there is a "start up" cost involved in learning how to write such code. However, this is the only sane method to obtain large amounts of data from the Live Access Server. Chapters 9 and 11 look at extracting data sets from complex systems and automating tasks that would otherwise be tedious or impossible if performed by hand.

Writing code, as we have seen above, is the only accurate method of communicating even mildly complex ideas to a computer; and even for very simple ideas, writing code is the most efficient method of communication. In this book, we will always communicate with the computer by **writing code**. In Chapter 2 we will discuss the basic ideas of how to write computer code properly and we will encounter a number of different computer languages throughout the remainder of the book.

At this point, we have the tools to access the Point Nemo data in a form that is convenient for conducting the data analysis, but, because this is not a book on data analysis, this is where we stop. The important points for our purposes are how the data are recorded, stored, accessed, and processed. These are the topics that will be expanded upon and discussed at length throughout the remainder of this book.

Summary

This book is concerned with the issues and technologies involved with the collection, storage, and handling of data sets.

We will focus on the ways in which these technologies can help us to perform tasks more efficiently and more accurately.

We will emphasise the appropriate use of these technologies; in particular, the importance of performing tasks by writing computer code.

2
Writing computer code

There are two aims for this chapter.

First, we need to learn how to write computer code. Most of the computer technologies that we encounter will consist of some sort of computer language, so it is essential that we learn from the start how to produce computer code in the right way.

Learning how to write code would be very dull if we only discussed code writing in abstract concepts, so the second aim of this chapter is to learn a computer language, with which to demonstrate good code writing.

The language that we will learn in this chapter is the Hypertext Markup Language, HTML. This language was chosen because it is a simple language, so it is a nice easy way to start writing computer code, and because it is a very useful language to know. Documents produced in HTML can be viewed on virtually any computer in the world because it is an open standard and all that is needed to view an HTML document is a web browser. For these reasons, HTML is an excellent format for disseminating scientific reports. It also makes sense to introduce HTML now, because we will make further use of it in Chapter 5 when discussing electronic forms.

2.1 Case study: Point Nemo (continued)

Figure 2.1 shows a simple web page that contains a very brief statistical report on the Poles of Inaccessibility data from Chapter 1. The report consists of text and an image (a plot), with different formatting applied to various parts of the text. The report heading is bold and larger than the other text, the table of numerical summaries is displayed with a monospace font,[1] and supplementary material is displayed in an italic font at the bottom of the page. In addition, the table of numerical summaries has a grey background and a border, while the image is horizontally-centred within the page. Part of the supplementary material at the bottom of the page also acts as a hyperlink; a mouse click on the underlined text will navigate

[1] In a monospace font, all characters have the same width.

to NASA's Live Access Server home page.

The entire web page in Figure 2.1 is described using HTML.

The HTML code that describes this web page is shown in Figure 2.2. We do not need to worry about the details of this code yet. What is important is to notice that all of this code is just text. Some of it is the text that makes up the content of the report, and other parts are special HTML **keywords** that describe how the content should be arranged and displayed; the latter can be distinguished as the parts surrounded by angled brackets `<like this>`. For example, the heading at the top of the page consists of two keywords, `<h3>` and `</h3>` surrounding the actual text of the heading (see lines 7 to 10).

There is also a clear structure to the code in Figure 2.2. For example, for every "opening tag" `<likeThis>`, there is a "closing tag" `</likeThis>`. This rigid structure is an important feature of computer languages. It is vital that we observe this structure, but the discipline required will help us to be accurate, clear, and logical in how we think about tasks and in how we communicate our instructions to the computer. As we will see later in this chapter, it is also important that we reflect this structure in the layout of our code (as has been done in Figure 2.2).

In this chapter we will learn the basics of HTML, with a focus on how the code itself is written and with an emphasis on the correct way to write computer code.

2.2 Syntax

The first thing we need to learn about a computer language is the correct **syntax** for the language. What is syntax? Consider the following sentence in the human language called English:

> The chicken iz to hot too eat!

This sentence has some technical problems. First of all it contains a spelling mistake—"iz" should be "is"—but even if we fix that, there are grammatical errors because the words are not in a valid order. The sentence still does not make sense:

> The chicken is to hot too eat!

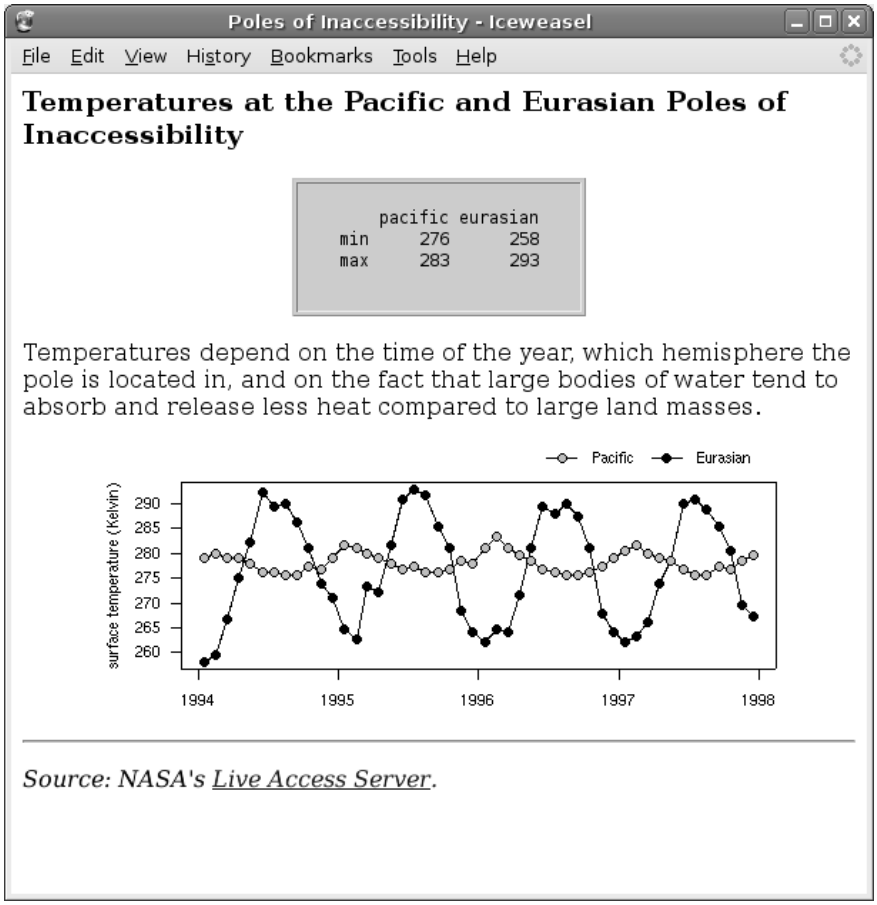


Figure 2.1: A simple web page as displayed by the Iceweasel browser on Debian Linux.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
    <head>
        <title>Poles of Inaccessibility</title>
    </head>
    <body>
        <h3>
        Temperatures at the Pacific and Eurasian Poles of
        Inaccessibility
        </h3>

        <center>
        <table border="1" bgcolor="#CCCCCC">
            <tr>
                <td>
                <pre>

        pacific eurasian
    min     276      258
    max     283      293
                </pre>
                </td>
            </tr>
        </table>
        </center>

        <p>
        Temperatures depend on the time of the year, which
        hemisphere the pole is located in, and on the fact
        that large bodies of water tend to absorb and release
        less heat compared to large land masses.
        </p>

        <center><img src="poleplot.png"></center>

        <hr>
        <p>
        <i>
        Source:  NASA's
        <a href="http://mynasadata.larc.nasa.gov/LASintro.html">
        Live Access Server</a>.
        </i>
        </p>
    </body>
</html>
```

Figure 2.2: The HTML code behind the web page in Figure 2.1. The line numbers (in grey) are just for reference.

The sentence is only a valid English sentence once both the spelling and grammar are correct:

> The chicken is too hot to eat!

When we write code in a computer language, we call these spelling or grammatical rules the **syntax** of the language.

It is very important to note that computers tend to be far less flexible than humans when it comes to comprehending language expressions. It is possible for a human to easily understand the original English sentence, even with spelling and grammatical errors. Computers are much more fussy and we will need to get the language syntax perfect before the computer will understand any code that we write.

The next section describes the basic syntax rules for the HTML language. This information will allow us to write HTML code that is correct in terms of spelling and grammar.

2.2.1 HTML syntax

HTML is nice because it is defined by a standard, so there is a single, public specification of HTML syntax. Unfortunately, as is often the case, it is actually defined by several standards. Put another way, there are several different **versions** of HTML, each with its own standard. We will focus on HTML 4.01 in this book.

HTML has a very simple syntax. HTML code consists of two basic components: **elements**, which are special HTML keywords, and **content**, which is just normal everyday text.

An element consists of a **start tag**, an **end tag** and some content in between. For example, the `title` element from Figure 2.2 is shown below:

```
<title>
    Poles of Inaccessibility
</title>
```

There is a start tag `<title>`, an end tag `</title>`, and plain text content.

The `title` element in a web page is usually displayed in the title bar of the web browser, as can be seen in Figure 2.1.

Some HTML elements may be "empty", which means that they only consist

of a start tag (no end tag and no content). An example is the `img` (short for "image") element from Figure 2.2, which inserts the plot in the web page.

```
<img src="poleplot.png">
```

The entire `img` element consists of this single tag.

There is a fixed set of valid HTML elements and only those elements can be used within HTML code. We will encounter several important elements in this chapter and a more comprehensive list is provided in Chapter 3.

HTML elements can have one or more **attributes**, which provide more information about the element. An attribute consists of the attribute name, an equals sign, and the attribute value, which is surrounded by quote marks. We have just seen an example in the `img` element above. The `img` element has an attribute called `src` that describes the location of a file containing the picture to be drawn on the web page. In the example above, the attribute is `src="poleplot.png"`.

Many attributes are optional and if they are not specified a default value is provided.

HTML tags must be ordered properly. All elements must nest cleanly and some elements are only allowed inside specific other elements. For example, a `title` element can only be used inside a `head` element, and the `title` element must start and end within the `head` element. The following HTML code is invalid because the `title` element does not finish within the `head` element:

```
<head>
    <title>
    Poles of Inaccessibility
</head>
    </title>
```

Finally, there are a few elements that *must* occur in an HTML document: there must be a `DOCTYPE` declaration, which states what computer language we are using; there must be a single `html` element, with a single `head` element and a single `body` element inside; and the `head` element must contain a single `title` element. Figure 2.3 shows a minimal HTML document.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
    <head>
        <title>A Minimal HTML Document</title>
    </head>
    <body>
        Your content goes here!
    </body>
</html>
```

Figure 2.3: A minimal HTML document.

2.2.2 Escape sequences

In all computer languages, certain words or characters have a special meaning within the language. These are sometimes called **reserved words** to indicate that they are reserved by the language for special use and cannot be used for their normal human-language purpose. This means that some words can never be used when writing in a computer language or, in other cases, a special code must be used instead. We will see reserved words and characters in all of the computer languages that we meet. This section describes some examples for HTML.

The content of an HTML element (whatever is written between the start and end tags) is mostly up to the author of the web page, but there are some characters that have a special meaning in the HTML language so these must be avoided. For example, the `<` character marks the start of an HTML tag, so this cannot be used for its normal meaning of "less than".

If we need to have a less-than sign within the content of an HTML element, we have to type `<` instead. This is an example of what is called an **escape sequence**.

Another special character in HTML is the greater-than sign, `>`. To produce one of these in the content of an HTML element, we must type `>`.

All HTML escape sequences are of this form: they start with an ampersand, `&`. This means of course that the ampersand is itself a special character, with its own escape sequence, `&`. A larger list of special characters and escape sequences in HTML is given in Section 3.1.2.

```
line 13 column 9 - Warning: <table> lacks "summary" attribute
line 34 column 17 - Warning: <img> lacks "alt" attribute
Info: Doctype given is "-//W3C//DTD HTML 4.01 Transitional//EN"
Info: Document content looks like HTML 4.01 Transitional
2 warnings, 0 errors were found!
```

Figure 2.4: Part of the output from running HTML Tidy on the HTML code in Figure 2.2.

2.2.3 Checking syntax

Knowing how to write the correct syntax for a computer language is not a guarantee that we *will* write the correct syntax for a particular piece of code. One way to check whether we have our syntax correct is to stare at it and try to see any errors. However, in this book, such a tedious, manual approach is discouraged; the computer is much better at this sort of task.

In general, we will enlist the help of computer software to check that the syntax of our code is correct. In the case of HTML code, there is a piece of software called HTML Tidy that can do the syntax checking.

2.2.4 Checking HTML code

HTML Tidy is a program for checking the syntax of HTML code. It can be downloaded from Source Forge[2] to run on your own computer and an online service is provided by the World Wide Web Consortium (W3C) at `http://cgi.w3.org/cgi-bin/tidy` or by Site Valet at `http://valet.htmlhelp.com/tidy/`.

HTML Tidy checks the syntax of HTML code, reports any problems that it finds, and produces a suggestion of what the correct code should look like. Figure 2.4 shows part of the output from running HTML Tidy on the simple HTML code in Figure 2.2.

An important skill to develop for writing computer code is the ability to decipher warning and error messages that the computer displays. In this case, there are no errors in the HTML code, which means that the syntax is correct. However, HTML Tidy has some warnings that include suggestions to make the code better.

[2] `http://tidy.sourceforge.net/`

2.2.5 Reading error information

The error (or warning) information provided by computer software is often very terse and technical. Reading error messages is a skill that improves with experience and it is important to seek out any piece of useful information in a message. Even if the message as a whole does not make sense, if the message can only point us to the correct area within the code, our mistake may become obvious.

In general, when the software checking our code encounters a series of errors, it is possible for the software to become confused. This can lead to more errors being reported than actually exist. It is *always* a good idea to tackle the first error first and it is usually a good idea to recheck code after fixing each error. Fixing the first error will sometimes eliminate or at least modify subsequent error messages.

The first warning from HTML Tidy in Figure 2.4 is this:

```
line 13 column 9 - Warning: <table> lacks "summary" attribute
```

To an experienced eye, the problem is clear, but this sort of message can be quite opaque for people who are new to writing computer code. A good first step is to make use of the information that is supplied about *where* the problem has occurred. In this case, we need to find the ninth character on line 13 of our code.

The line of HTML code in question is this:

```
        <table border="1" bgcolor="#CCCCCC">
```

Column 9 on this line is the `<` at the start of the HTML tag.

The warning message mentions `<table>`, so we might guess that we are dealing with the opening tag of a `table` element (which in this case just confirms that we are looking at the right place in our code). The message is complaining that this tag does not have an attribute called `summary`. Looking at the code, we see that there are two attributes, `border` and `bgcolor`, but nothing called `summary`. The solution clearly involves adding a `summary` attribute to this tag.

The second warning is similar. On line 34, there is an `<img>` tag that has a `src` attribute, but HTML Tidy would like us to add an `alt` attribute as well.

So far so good—we have determined the problem. Now all we have to do is find a solution. In both cases, we need to add an extra attribute, and we

even know the names of the attributes that we need to add. For example, the attribute that we need to add to the `<table>` tag will be something of the form: `summary="some value"`. However, it is not yet clear what the *value* of the attribute should be. This leads us to the next important skill to acquire for writing computer code. It is important to be able to read the **documentation** for a computer language (the manuals, help pages, online forums, etc).

2.2.6 Reading documentation

There are two main problems associated with learning a human language: the rules of grammar are usually totally inconsistent, with lots of exceptions and special cases; and there is an enormous vocabulary of different words to learn.

The nice thing about learning a computer language is that the rules of grammar are usually quite simple, there are usually very few of them, and they are usually very consistent.

Unfortunately, computer languages are similar to human languages in terms of vocabulary. The time-consuming part of learning a computer language involves learning all of the special words in the language and their meanings.

What makes this task worse is the fact that the reference material for computer languages, much like the error messages, can be terse and technical. As for reading error messages, practice and experience are the only known cures.

To continue with our HTML example, we want to find out more about the `summary` attribute of a `table` element. The source of definitive information on HTML is the W3C, the standards body that publishes the official "recommendations" that define a number of web-related computer languages.

The HTML 4.01 Specification[3] is an example of highly technical documentation. For example, Figure 2.5 shows an extract from the definition of a `table` element.

This is likely to appear quite intimidating for people who are new to writing computer code, although the information that we need is in there (the first of the "Attribute definitions" tells us about the `summary` attribute). However, one of the advantages of using established and open standards is that a lot of information for computer languages like HTML is available on the world-wide web and it is often possible to discover information that is at a more

[3] `http://www.w3.org/TR/html401`

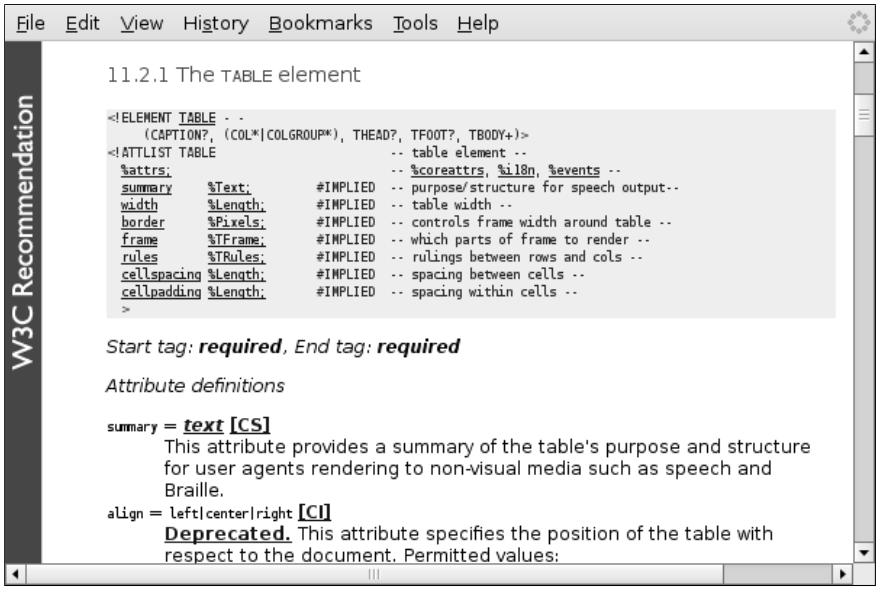


Figure 2.5: An extract from the W3C HTML 4.01 Specification for the `table` element.

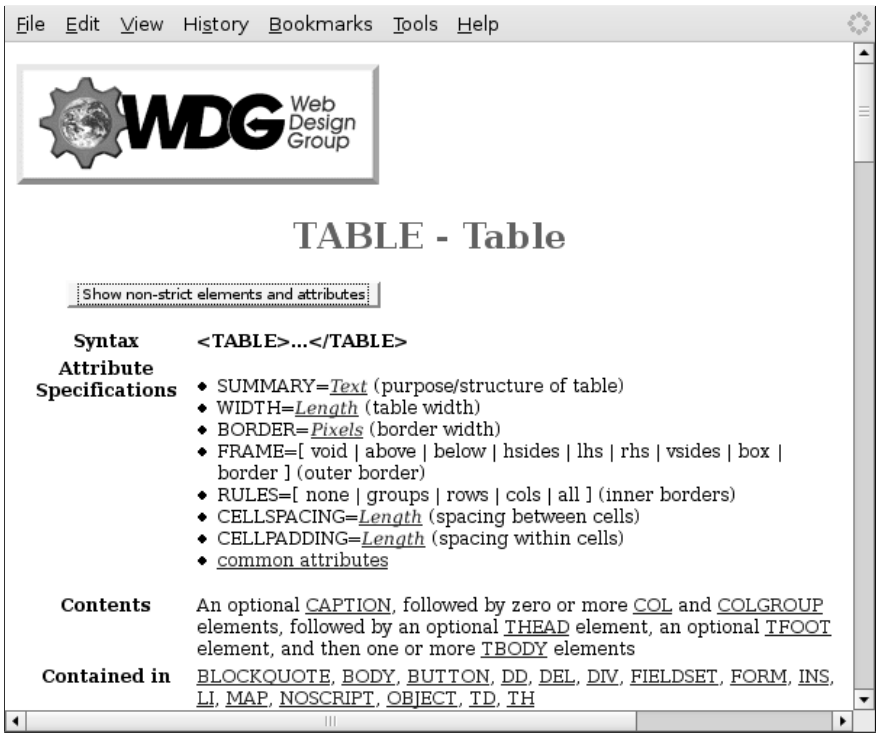


Figure 2.6: An extract from the WDG HTML 4.0 Reference for the `table` element.

introductory level. For HTML, a more accessible and discursive presentation of the information about the HTML language is provided by the Web Design Group[4] (WDG). Figure 2.6 shows part of the page containing the WDG description of the `table` element and Figure 2.7 shows the part of that page that is devoted to the `summary` attribute.

From these documents, we can see that the `summary` attribute for a `table` element is supposed to contain a description of the purpose and content of the table and that this information is especially useful for use with browsers that do not or cannot graphically display the table. For the HTML code that we are working with (Figure 2.2), we could modify line 13 like this:

```
<table border="1" bgcolor="#CCCCCC"
       summary="A simple border for numerical data">
```

[4] `http://htmlhelp.com/reference/html40/`

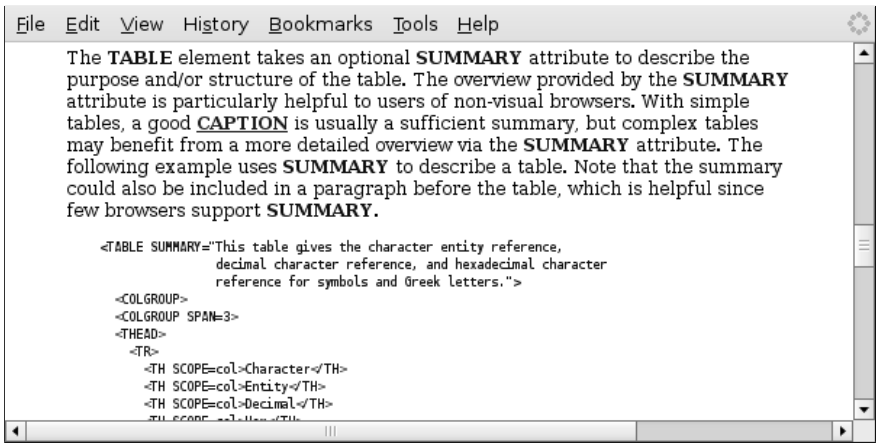


Figure 2.7: An extract from the WDG HTML 4.0 Reference for the `table` element, showing the discussion of the `summary` attribute.

2.3 Semantics

Consider the correct version of the english sentence we saw in Section 2.2:

> The chicken is too hot to eat!

This now has correct syntax, which means that there are no spelling errors or grammatical errors, which means that we should be able to successfully extract the meaning from the sentence. When we write code in a computer language, we call the meaning of the code—what the computer will do when the code is run—the **semantics** of the code. Computer code has no defined semantics until it has a correct syntax, so we should always check that our code is free of errors before worrying about whether it does what we want.

Look again at the english sentence; what does the sentence mean? One reading of the sentence suggests that a person has burnt his or her mouth with a piece of cooked chicken. However, there is another possible meaning; perhaps a chicken has lost its appetite because it has been out in the sun too long! One problem with human languages is that they tend to be very ambiguous.

Computer languages, by comparison, tend to be very precise, so as long as we get the syntax right, there should be a clear semantics for the code. This is important because it means that we can expect our code to produce the

same result on different computers and even with different software.

2.3.1 HTML semantics

The HTML 4.01 specification defines a fixed set of valid HTML elements and describes the meaning of each of those elements in terms of how they should be used to create a web page.

In this section, we will use the simple HTML page shown at the start of this chapter to demonstrate some of the basic HTML elements. Chapter 3 provides a larger list. Figure 2.8 and Figure 2.9 are reproductions of Figures 2.1 (what the web page looks like) and 2.2 (the HTML code) for convenient reference.

The main part of the HTML code in Figure 2.9 is contained within the `body` element (lines 6 to 44). This is the content of the web page; the information that will be displayed by the web browser.[5]

The first element we encounter within the `body` is an `h3` element (lines 7 to 10). The contents of this element provide a title for the page, which is indicated by drawing the relevant text bigger and bolder than normal text. There are several such heading elements in HTML, from `h1` to `h6`, where the number indicates the heading "level", with `1` being the top level (biggest and boldest) and `6` the lowermost level. Note that this element does two things: it describes the *structure* of the information in the web page and it controls the *appearance* for the information—how the text should be displayed. The structure of the document is what we should focus on; we will discuss the appearance of the web page in more depth in Section 2.5.1.

The next element in our code is a `center` element (lines 12 to 25). This only controls the appearance of its content, ensuring that the table within this element is horizontally centred.

Next up is the `table` element (lines 13 to 24). Within a `table` element, there must be a `tr` (table row) element for each row of the table, and within each `tr`, there must be a `td` (table data) element for each column of the table. In this case, the table consists of a single row (lines 14 to 23) and a single column (lines 15 to 22). The `border` attribute of the `table` element specifies that a border should be drawn around the outside of the table and the `bgcolor` attribute specifies a light grey background for the table.

The content of the table is a `pre` (preformatted text) element (lines 16 to 21). This element is used to display text exactly as it is entered, using a

[5] We will see important uses for the `head` element later in Section 2.5.1.

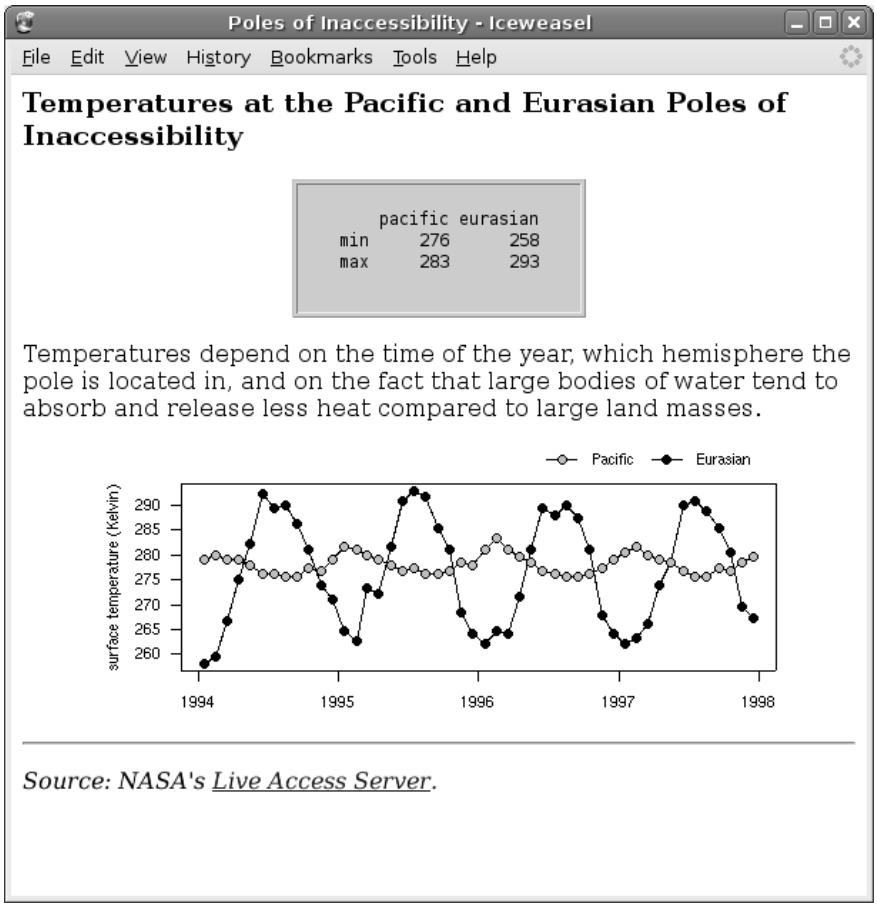


Figure 2.8: A simple web page as displayed by the Iceweasel browser on Debian Linux. This is a reproduction of Figure 2.1.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
    <head>
        <title>Poles of Inaccessibility</title>
    </head>
    <body>
        <h3>
        Temperatures at the Pacific and Eurasian Poles of
        Inaccessibility
        </h3>

        <center>
        <table border="1" bgcolor="#CCCCCC">
            <tr>
                <td>
                <pre>

        pacific eurasian
    min     276      258
    max     283      293
                </pre>
                </td>
            </tr>
        </table>
        </center>

        <p>
        Temperatures depend on the time of the year, which
        hemisphere the pole is located in, and on the fact
        that large bodies of water tend to absorb and release
        less heat compared to large land masses.
        </p>

        <center><img src="poleplot.png"></center>

        <hr>
        <p>
        <i>
        Source:  NASA's
        <a href="http://mynasadata.larc.nasa.gov/LASintro.html">
        Live Access Server</a>.
        </i>
        </p>
    </body>
</html>
```

Figure 2.9: The HTML code behind the web page in Figure 2.8. This is a reproduction of Figure 2.2.

monospace font, with all spaces faithfully reproduced. The point of this will become clear when we discuss the `p` element below.

The widths of the columns in an HTML table are automatically determined by the web browser to allow enough space for the contents of the table.

The next element in our code is a `p` (paragraph) element (lines 27 to 32). The important thing to notice about this element is that the contents are not displayed as they appear in the code (compare the line breaks in the HTML code within the `p` element in Figure 2.9 with the line breaks in the corresponding paragraph of text in Figure 2.8). The `p` tags indicate that the contents should be arranged as a paragraph of text.

After the paragraph, we have another `center` element which ensures the horizontal centering of the plot. The plot is generated by the `img` (image) element (line 34). The `src` attribute of the `img` element specifies the location of the file for the image.

At the bottom of the page we have an `hr` (horizontal rule) element (line 36), which produces a horizontal line across the page, followed by a final paragraph (`p` element) of text (lines 37 to 43). The text within this paragraph is italicized because it is contained within an `i` element (lines 38 to 42).

Part of the text in the final paragraph is also a hyperlink. The `a` (anchor) element around the text "Live Access Server" (lines 40 and 41) means that this text is highlighted and underlined. The `href` attribute specifies that when the text is clicked, the browser will navigate to the home page of NASA's Live Access Server.

These are some of the simple HTML elements that can be used to create web pages. There are many more elements that can be used to create a variety of different effects (Chapter 3 describes a few more and we will meet some special, interactive HTML elements in Chapter 5), but the basic pattern is the same: the code consists of HTML tags surrounding the important information that is to be displayed.

2.3.2 Running code

Having explained what the code in Figure 2.9 is supposed to do, how do we actually make it produce a web page like the one in Figure 2.8? This section looks at how to **run** computer code to get it to perform a task.

Just because there is a clear meaning for a piece of code does not mean that a human reading the code, even the human who wrote the code, will interpret the meaning of the code correctly. The only way to find out what

a piece of code really means is to run the code and see what it does.

As with syntax checking, we need to use software to run the code that we have written.

In the case of HTML, there are many software programs that will run the code, but the most common type is a web browser, such as Internet Explorer or Firefox.

2.3.3 Running HTML code

All that we need to do to run our HTML code is to open the file containing our code with a web browser. We can then see whether the code has produced the result that we want.

Web browsers tend to be very lenient with HTML syntax. If there is a syntax error in HTML code, most web browsers try to figure out what the code should do (rather than reporting an error). Unfortunately, this can lead to problems where two different web browsers will produce different results from exactly the same code.

Another problem arises because most web browsers do not completely implement the HTML standards. This means that some HTML code will not run correctly on some browsers.

The solution to these problems, for this book, has two parts: we will not use a browser to check our HTML syntax (we will use HTML Tidy instead, see Section 2.2.3); and we will use a single browser (Firefox) to define what a piece of HTML code should do. We will only be using a simple subset of the HTML language so the chance of encountering ambiguous behaviour is small anyway.

If we run HTML code and the result is what we want, we are almost, but not quite finished. In this case, we have code that has the correct syntax and the correct semantics, but we must also worry about whether our code has the correct aesthetics. It is important for code to be at least tidy and this topic is addressed in Section 2.4.

The next section looks at the situation where we run our code and the result is *not* what we want.

2.3.4 Debugging code

When we have code that has correct syntax and runs, but does not behave correctly, we say that there is a **bug** in our code. The process of fixing our code so that it does what we want it to is called **debugging** the code.

It is often the case that debugging code takes much longer than writing the code in the first place, so it is an important skill to acquire.

The source of common bugs varies enormously with different computer languages, but there are some common steps we should take when fixing any sort of code:

Do not blame the computer.
: There are two possible sources of problems: our code is wrong or the computer (or software used to run our code) is wrong. It will almost always be the case that our code is wrong. If we are completely convinced that our code is correct and the computer is wrong, we should go home, have a sleep and come back the next day. The problem in our code will usually then become apparent.

Recheck the syntax.
: Whenever a change is made to the code, check the syntax again before trying to run the code again. If the syntax is wrong, there is no hope that the code will run correctly.

Develop code incrementally.
: Do not try to write an entire web page at once. Write a small piece of code and get that to work, then add more code and get that to work. When things stop working, it will be obvious which bit of code is broken.

Apply the scientific method.
: Do not make many changes to the code at once to try to fix it. Even if the problem is cured, it will be difficult to determine which change made the difference. A common problem is introducing new problems as part of a fix for an old problem. Make one change to the code and see if that corrects the behaviour, then revert that change before trying something else.

Read the documentation.
: For all of the computer languages that we will deal with in this book, there are official documents plus many tutorials and online forums that contain information and examples for writing code. Find them and read them.

Ask for help.
In addition to the copious manuals and tutorials on the web, there are many forums for asking questions about computer languages. The friendliness of theses forums varies and it is important to read the documentation before taking this step.

Chapter 3 provides some basic information about common HTML elements and Section 3.3 provides some good starting points for detailed documentation about HTML.

We will discuss specific debugging tools for some languages as we meet them.

2.4 Writing for an audience

Up to this point, we have focused on making sure that the code we write works correctly; that it has no syntax errors and that it does what we want it to do.

However, having code that successfully performs a task is not the end of our journey. It is also important that we make our code neat and tidy. Just like owning a car, if we want our code to keep working tomorrow and the next day and the day after that, it is important that we maintain our code and keep it clean. This section discusses the proper way to write code so that it will last.

There are two important audiences to consider when writing computer code. The obvious one is the computer; it is vitally important that the computer understands what we are trying to tell it to do. This is mostly a matter of getting the syntax of our code right.

The other audience for code consists of humans. While it is important that code works (that the computer understands it), it is also essential that the code is comprehensible to people. And this does not just apply to code that is shared with others, because the most important person who needs to understand a piece of code is the original author of the code! It is very easy to underestimate the probability of having to reuse a piece of code weeks, months, or even years after it was initially written, and in such cases it is common for the code to appear much less obvious on a second viewing, even to the original author.

It is also easy to underestimate the likelihood that other people will get to view a piece of code. For example, our scientific peers should *want* to see our code so that they know what we did to our data. All code should be treated as if it is for public consumption.

2.4.1 Layout of code

One simple, but important way that code can be improved for a human audience is to format the code so that it is easy to read and easy to navigate.

Consider the following two code chunks. They contain identical HTML code, as far as the computer's understanding is concerned, but they are vastly different in terms of how easy it is for a human reader to comprehend them. Try finding the "title" part of the code. Even without knowing anything about HTML, this is a ridiculously easy task in the second layout, and annoyingly difficult in the first.

```
<html><head><title>A Minimal HTML
Document</title></head><body>
Your content goes here!</body>
```

```
<html>
    <head>
        <title>A Minimal HTML Document</title>
    </head>
    <body>
        Your content goes here!
    </body>
```

This demonstrates the basic idea behind laying out code. The changes are entirely cosmetic, but they are extremely effective. It also demonstrates one important layout technique: indenting.

2.4.2 Indenting code

The idea of indenting code is to expose the **structure** of the code. What "structure" means will vary between computer languages, but in most cases the language will contain **blocks** of the form:

```
BEGIN
    body line
    body line
END
```

As demonstrated, a simple indenting rule is always to indent the "body" of a block of code. This is very easy to demonstrate using HTML, where code blocks are formed by start and end tags. Here is a simple example:

```
<head>
    <title>A Minimal HTML Document</title>
</head>
```

The amount of indenting is a personal choice. The examples here have used 4 spaces, but 2 spaces or even 8 space are also common. Whatever indentation is chosen, it is essential that the indenting rule is applied consistently, especially when more than one person might modify the same piece of code.

Exposing structure of code by indenting is important because it makes it easy for someone reading the code to navigate within the code. It is easy to identify different parts of the code, which makes it easier to see what the code is doing.

Another useful result of indenting is that it provides a basic check on the correctness of code. Look again at the simple HTML code example. Does anything look wrong?

```
<html>
    <head>
        <title>A Minimal HTML Document</title>
    </head>
    <body>
        Your content goes here!
    </body>
```

Even without knowing anything about HTML, the lack of symmetry in the layout suggests that there is something missing at the bottom of this piece of code. In this case, indenting has alerted us to the fact that there is no end `</html>` tag.

2.4.3 Long lines of code

Another situation where indenting should be applied is when a line of computer code becomes very long. It is a bad idea to have a single line of code that is wider than the screen on which the code is being viewed (so that we have to scroll across the window to see all of the code). When this happens, the code should be split across several lines (most computer languages do not notice the difference). Here is an example of a line of HTML code that is too long.

```
<table border="1" bgcolor="#CCCCCC" summary="A simple border for numerical data">
```

Here is the code again, split across several lines. It is important that the subsequent lines of code are indented so that they are visually grouped with the first line.

```
<table border="1" bgcolor="#CCCCCC"
       summary="A simple border for numerical data">
```

In the case of a long HTML element, a reasonable approach is to left-align the start of all attributes within the same tag (as shown above).

2.4.4 White space

White space refers to empty gaps in code, which are important for making code easy for humans to read. Wouldyouwriteinyournativelanguagewithoutputtingspacesbetweenthewords?

Indenting is a form of white space that always appears at the start of a line, but white space is effective *within* and *between* lines of code as well. For example, the following code is too dense and therfore is difficult to read.

```
<table border="1"width="100%"bgcolor="#CCCCCC">
```

This modification of the code, with extra spaces, is much easier on the eye.

```
<table border="1" width="100%" bgcolor="#CCCCCC">
```

The two code chunks below demonstrate the usefulness of blank lines between code blocks to help expose the structure, particularly in large pieces of code.

```
<html>
<head>
    <title>
    A Minimal HTML Document
    </title>
</head>
<body>
    Your content goes here!
</body>
</html>
```

```
<html>

<head>
    <title>
    A Minimal HTML Document
    </title>
</head>

<body>
    Your content goes here!
</body>

</html>
```

Again, exactly when to use spaces or blank lines depends on personal style.

2.4.5 Documenting code

In Section 2.2.6, we discussed the importance of being able to *read* documentation about a computer language. In this section, we consider the task of *writing* documentation for our own code.

As with the layout of code, the purpose of documentation is to communicate. The obvious target of this communication is other people, so that they know what we did. A less obvious, but no less important, target is the code author. It is essential that when we return to a task days, weeks, or even months after we first performed the task, we are able to pick up the task again, and pick it up quickly.

Most of what we will have to say about documentation will apply to writing **comments**—messages written in plain language, embedded in the code, and which the computer ignores.

2.4.6 HTML comments

Here is how to include a comment within HTML code.

```
<!-- This is a comment -->
```

Anything between the opening `<!--` and closing `-->`, including HTML tags, is completely ignored by the computer. It is only there to edify a human reader.

```
R> htmlcss <- readLines(file.path("HTML", "htmlcss.html"))
R> commentstart <- grep("<!--", htmlcss)
R> commentend <- grep("-->", htmlcss)
```

There is an example of the use of a comment in HTML code on lines 6 and 7 in Figure 2.10 (see 36)).

How many comments?

Having no comments in code is generally a bad idea and it is usually the case that people do not add enough comments to their code. However,

it can also be a problem if there are too many comments.[6] Comments should not just be a repetition of the code. Good uses of comments include: providing a conceptual summary of a block of code; explaining a particularly complicated piece of code; and explaining arbitrary constant values.

2.5 The DRY principle

One of the purposes of this book is to introduce and explain various technologies for working with data. We have already met one such technology, HTML, for producing reports on the world-wide web.

Another purpose of this book is to promote the correct approach, or "best practice", for using these technologies. An example of this is the emphasis on writing code using computer languages rather than learning to use dialog boxes and menus in a software application.

In this section, we will look at another example of best practice called the **DRY principle**,[7] which has important implications for how we manage the code that we write.

DRY stands for **Don't Repeat Yourself** and the principle is that there should only ever be *one copy* of any important piece of information.

The reason for this principle is that one copy is much easier to maintain than multiple copies; if the information needs to be changed, there is only one place to change it. In this way the principle promotes efficiency. Furthermore, if we lapse and allow several copies of a piece of information, then it is possible for the copies to diverge or for one copy to get out of date. From this perspective, having only one copy improves our accuracy.

To understand the DRY principle, consider what happens when we move house to a new address. One of the many inconveniences of shifting house involves letting everyone know our new address. We have to alert schools, banks, insurance companies, doctors, friends, etc. The DRY principle suggests that we should have only one copy of our address stored somewhere (e.g., at the post office) and everyone else should refer to that address. That way, if we shift house, we only have to tell the post office the new address and everyone will see the change. In the current situation, where there are multiple copies of our address, it is easy for us to forget to update one of the copies when we change address. For example, we might forget to tell

[6] If there are too many comments, it can become a burden to ensure that the comments are all correct if the code is ever modified. It can even be argued that too many comments make it hard to see the actual code!

[7] Doff cap to Andy Hunt and Dave Thomas, the "Pragmatic Programmers".

the bank, so all our bank correspondence will be sent to the wrong address!

The DRY principle will be very important when we discuss the storage of data (Chapter 7), but it can also be applied to computer code that we write. In the next section, we will look at one example of applying the DRY principle to writing computer code.

2.5.1 Cascading Style Sheets

Cascading Style Sheets (CSS) is a language that is used to describe how to display information. It is commonly used with HTML to control the appearance of a web page. In fact, the preferred way to produce a web page is to use HTML to indicate the *structure* of the information and CSS to specify the *appearance*. One of the reasons that this is preferred is due to the DRY principle.

CSS syntax

CSS code consists of a series of **rules** and each rule comprises a **selector** and a set of **properties**. The selector specifies what sort of HTML element the rule applies to and the properties specify how that element should be displayed. An example of a CSS rule is shown below.

```
table {
    border-width: thin;
    border-style: solid;
}
```

In this rule, the selector is `table` so this rule will apply to all HTML `table` elements.

There are two properties in this rule, both relating to the border that is drawn around the table. The properties specify that a thin, solid border should be drawn around tables.

CSS is a completely separate language from HTML, but the rules in CSS code can be combined with HTML to control the appearance of a web page.

CSS code can be associated with HTML code in several ways, but the best way is to create a separate file for the CSS code. In this way, a web page consists of two files: one file contains the main content with HTML code to specify the structure and another file contains the CSS code to specify the appearance. The HTML file is linked to the CSS file via a special element

in the HTML code.

To demonstrate the combination of CSS and HTML code, and to describe more about CSS rules, we will reproduce the web page from the start of this chapter using CSS.

Figure 2.10 shows HTML code for a simple statistical report. This is very similar code to that in Figure 2.2, but several attributes have been removed and others have been added. For example, the `table` element used to have attributes controlling its appearance (this code is from line 13 in Figure 2.2):

```
<table border="1" bgcolor="#CCCCCC">
```

Now, the `table` element has only a `class` attribute (this code is from line 18 in Figure 2.10):

```
<table class="figure">
```

We will see an explanation of the `class` attribute soon.

The most important difference in the HTML code is the addition of a `link` element within the `head` element (lines 9 and 10 in Figure 2.10). The relevant line is this:

```
<link rel="stylesheet" href="csscode.css" type="text/css">
```

This piece of HTML code specifies that there is a file called `csscode.css` that contains CSS rules and that those rules should be applied to the HTML elements in this (HTML) file. The contents of the file `csscode.css` are shown in Figure 2.11.

There are several rules in the CSS code and they demonstrate several different ways of specifying CSS rule selectors as well as several different CSS properties.

The first rule is this:

```
table {
    background-color: #CCCCCC;
    border-width: thin;
    border-style: solid;
    white-space: pre;
    font-family: monospace;
}
```

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
    <head>
        <title>Poles of Inaccessibility</title>

        <!-- The appearance of the web page is controlled by
             CSS code rather than by HTML attributes -->

        <link rel="stylesheet" href="csscode.css"
              type="text/css">
    </head>
    <body>
        <h3>
        Temperatures at the Pacific and Eurasian Poles of
        Inaccessibility
        </h3>

        <table class="figure">
            <tr>
                <td>
        pacific eurasian
    min     276      258
    max     283      293
                </td>
            </tr>
        </table>

        <p>
        Temperatures depend on the time of the year, which
        hemisphere the pole is located in, and on the fact
        that large bodies of water tend to absorb and release
        less heat compared to large land masses.
        </p>

        <img class="figure" src="poleplot.png">

        <hr>
        <p class="footer">
        Source:  NASA's
        <a href="http://mynasadata.larc.nasa.gov/LASintro.html">
        Live Access Server</a>.
        </p>
    </body>
</html>
```

Figure 2.10: The HTML code behind the web page in Figure 2.12. This is similar to the code in Figure 2.2 except that it combines HTML and CSS code, rather than using pure HTML.

```
table {
    background-color: #CCCCCC;
    border-width: thin;
    border-style: solid;
    white-space: pre;
    font-family: monospace;
}

.figure {
    margin-left: auto;
    margin-right: auto;
}

img {
    display: block;
}

p.footer {
    font-style: italic;
}
```

Figure 2.11: The CSS code behind the appearance of the web page in Figure 2.12. The CSS rules in this code are applied to the HTML elements in Figure 2.10.

This rule has a simple selector, `table`, so it will apply to all `table` elements. The first three properties specify that the background colour for the table should be a light grey and that there should be a thin, solid border.[8] The last two properties control the appearance of the text within the table and specify that a monospace font should be used and that the text should be displayed exactly as it appears in the HTML code (in particular, all spaces should be displayed).

The second rule demonstrates a different sort of CSS rule selector, called a **class selector**:

```
.figure {
    margin-left: auto;
    margin-right: auto;
}
```

The . (full stop) at the start of the selector name is important. It indicates that what follows is the name of a CSS **class**. This rule will be applied to any HTML element that has a `class` attribute with the value `"figure"`. Looking at the HTML code in Figure 2.10, we can see that this rule will apply to both the `table` element and the `img` element (both of these elements have attributes `class="figure"`; see lines 18 and 35).

The properties in this second CSS rule are also a little more complicated. These control the margins (empty space) to the left and right of the relevant HTML element. By specifying `auto` margins we are letting these margins grow to be as large as they can, but still leave enough room for the HTML element. This is how we can centre an HTML element on the page using CSS.

The third CSS rule again has a straightforward selector, `img`. This rule will apply to all `img` elements. The CSS property `display: block` makes the `img` element behave like a paragraph of its own,[9] which is necessary to make the previous rule centre the image.

```
img {
    display: block;
}
```

The final rule controls the formatting of the source comment at the bottom

[8] This is the one aspect of the appearance that is not identical to the original HTML-only appearance; unfortunately, it is not possible to replicate the default appearance of a table border using CSS.

[9] Normally, `img` elements behave like a word in a sentence; in HTML terminology, `img` elements are **inline** elements, rather than **block-level** elements.

of the web page:

```
p.footer {
    font-style: italic;
}
```

The selector in this case combines both an HTML element name and a CSS class name. This rule will affect all `p` elements that have a `class` attribute with the value `"footer"`. The effect of this rule is to make the text italic, but only in the `p` element at the bottom of the HTML code (lines 38 to 42). This rule does not affect the other `p` element within the HTML code (lines 28 to 33).

The final result of this combination of HTML and CSS code is shown in Figure 2.12.

The only difference in the appearance between Figure 2.12 and Figure 2.1 is the border around the numeric results, but the underlying code is quite different.

The point of this example, other than to introduce another simple computer language, CSS, that provides the "correct" way to control the appearance of web pages, is to show the DRY principle in action.

There are two ways that the use of CSS demonstrates the DRY principle. First of all, if we want to produce another HTML report and we want that report to have the same appearance as the report we have just produced, we can simply make use of the same CSS code to control the appearance of the new report. In other words, with CSS, we can have a single copy of the code that controls the web page appearance and this code can be reused for different web page content.

The alternative would be to include HTML attributes in both the original report and the new report to control the appearance. That would involve typing more code and, more importantly, it would mean more than one copy of the important information about the appearance of the web pages. If we wanted to change the appearance of the reports, we would have to change both reports. With a single CSS file, any changes in the appearance only need to be made to the CSS file and that will update both reports.

Another advantage of using CSS code can be seen if we want to produce the same report in two different styles. For example, we might produce one version for people to view on a screen and a different version for people to print out.

The wrong way to create two different versions of the web page would be to

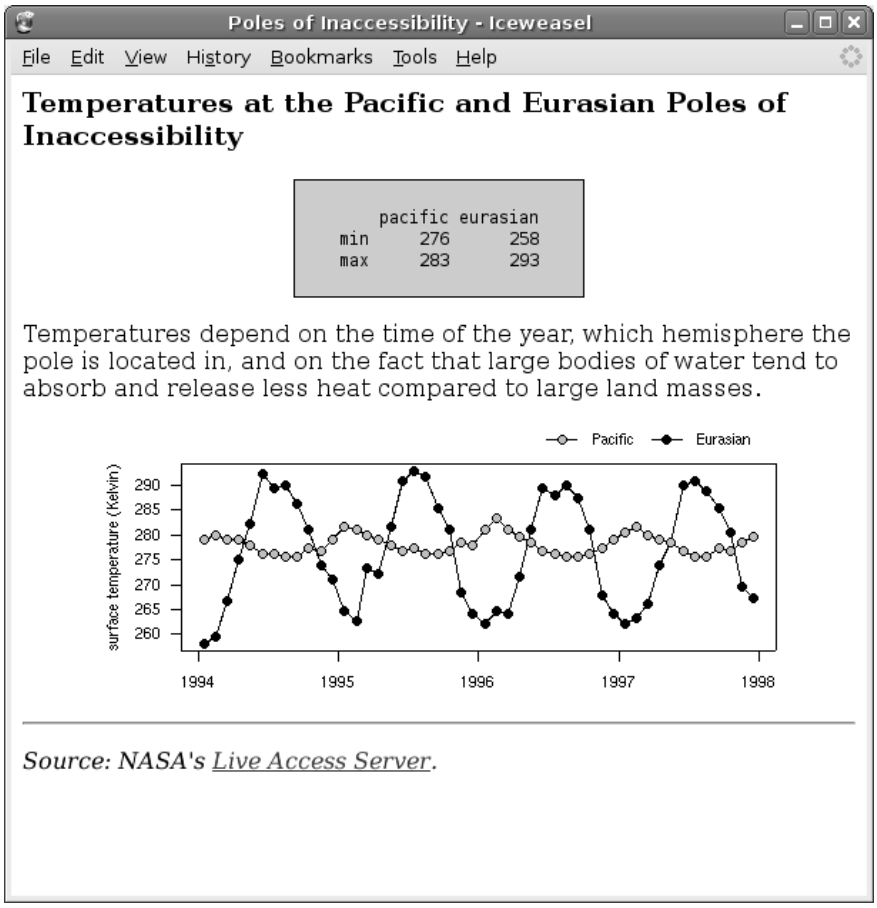


Figure 2.12: A simple web page as displayed by the Iceweasel browser on Debian Linux. This is very similar to Figure 2.1, but it has been produced using a combination of HTML and CSS code, rather than pure HTML.

have two copies of the HTML code, with different appearance information in each one. That approach would violate the DRY principle. What we want is to only have one copy of the HTML code.

This is possible using CSS. We can have two separate CSS files, each containing different rules, and swap between them to produce different views of the same HTML code. We will develop a new set of CSS rules in a file called `csscode2.css`. Then all we have to do is change the `link` element in the HTML code to refer to this new CSS code. The new line of HTML code looks like this:

```
<link rel="stylesheet" href="csscode2.css" type="text/css">
```

Figure 2.13 shows the result we will end up with. This should be compared with Figure 2.12, which is the same HTML code, just with different CSS code controlling the appearance.

Figure 2.14 shows the file containing the new set of CSS rules (compare these with the CSS rules in Figure 2.11).

In this new CSS code, there is a new rule for the `body` element to produce margins around the whole page (lines 1 to 5). There is also a new rule for the `hr` element that means that the horizontal line is not drawn at all (`display: none`; lines 29 to 31) and one for the `a` element so that it is drawn in white (lines 35 to 37). Another significant change is the new property for `table` elements, so that the numeric results "float" to the right of the page, with the text wrapped around it to the left (line 14).

The new CSS code also demonstrates another type of CSS selector:

```
h3, p.footer {
    background-color: black;
    color: white;
    padding: 5px;
}
```

The selector for this rule specifies a **group** of elements. This rule is applied to all `h3` elements *and* to all `p` elements that have a `class` attribute with the value `"footer"`. This is the rule that displays the heading and the source information in white text on a black background.

The code also demonstrates how to write comments in CSS. Lines 33 and 34 are a CSS comment:

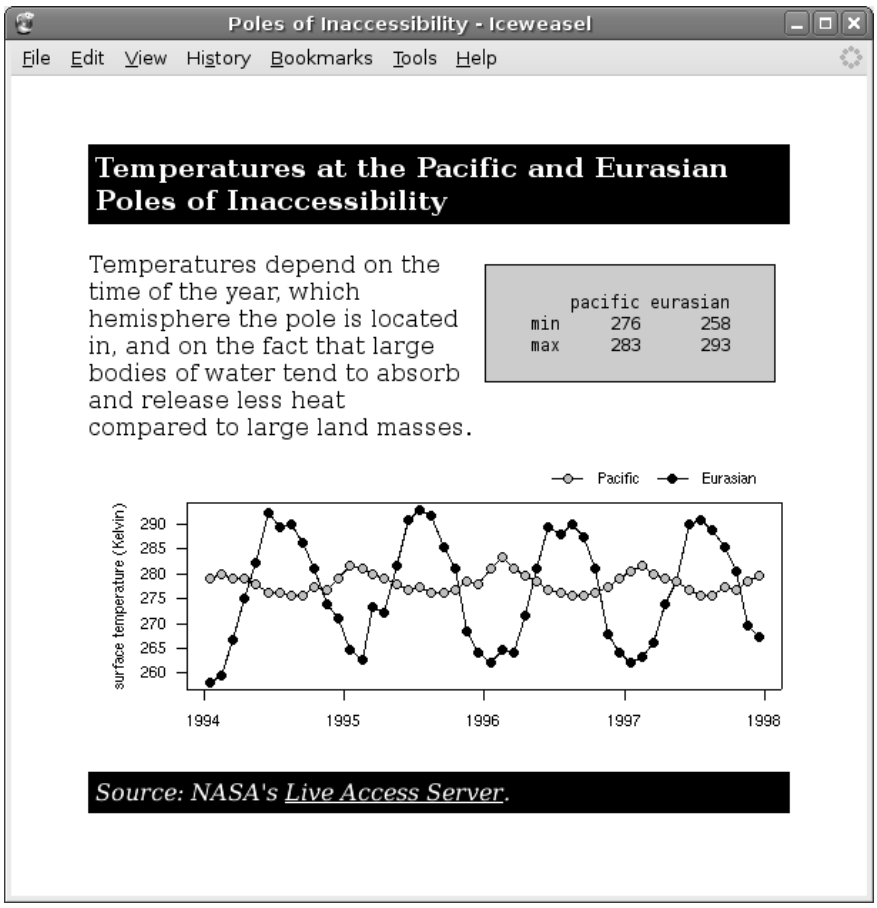


Figure 2.13: A simple web page as displayed by the Iceweasel browser on Debian Linux. This is an alternative presentation of the content of Figure 2.12 using different CSS rules with the same HTML elements.

```
body {
    margin-top: 8%;
    margin-left: 9%;
    margin-right: 9%;
}

h3, p.footer {
    background-color: black;
    color: white;
    padding: 5px;
}

table {
    float: right;
    margin: 2%;
    background-color: #CCCCCC;
    border-width: thin;
    border-style: solid;
    white-space: pre;
    font-family: monospace;
}

img {
    display: block;
    margin-left: auto;
    margin-right: auto;
}

hr {
    display: none;
}

/* Anchors are blue by default which is not
   very visible on a black background */
a {
    color: white;
}

p.footer {
    font-style: italic;
}
```

Figure 2.14: The CSS code behind the appearance of the web page in Figure 2.13. The CSS rules in this code are applied to the HTML elements in Figure 2.10.

```
/* Anchors are blue by default which is not
   very visible on a black background */
```

In CSS, a comment is anything enclosed between an opening `/*` and a closing `*/`.

2.6 Text editors

The act of writing code is itself dependent on computer tools. We use software to record and manage our keystrokes in an effective manner. This section discusses what sort of tool should be used to write computer code effectively.

An important feature of computer code is that it is just plain text. There are many software packages that allow us to enter text, but some are more appropriate than others.

2.6.1 Text editors are not word processors

For many people, the most obvious software program for entering text is a word processor, such as Microsoft Word or Open Office Writer. These programs are *not* a good choice for editing computer code. A word processor is a good program for making text look pretty with lots of fancy formatting and wonderful fonts. However, these are not things that we want to do with our raw computer code.

The programs that we use to run our code expect to encounter only plain text, so we must use software that creates only text documents, which means we must use a **text editor**.[10]

2.6.2 Important features of a text editor

For many people, the most obvious program for entering just text is Microsoft Notepad. This program has the nice feature that it saves just text, but its usefulness ends there.

When we write computer code, a good choice of text editor can make us much more accurate and efficient.

[10] Software that is designed not just for writing text, but specifically for writing computer code may also be called a **code editor**.

The following facilities are particularly useful for writing computer code:

atomatic indenting
: As we saw in Section 2.4.1, it is important to arrange code in a neat fashion. A text editor that helps to indent code (place empty space at the start of a line) makes this easier and faster.

parenthesis matching
: Many computer languages use special symbols, e.g., `{` and `}`, to mark the beginning and end of blocks of code. Some text editors provide feedback on such matching pairs, which makes it easier to write code correctly.

syntax highlighting
: All computer languages have special **keywords** that have a special meaning for the language (e.g., anything `<likeThis>` in HTML). Many text editors automatically colour such keywords, which makes it easier to read code and easier to spot simple mistakes.

line numbering
: Some text editors automatically number each line of computer code (and in some cases each column or character as well) and this makes navigation within the code much easier. This is particularly important when trying to find errors in the code (see Section 2.2.3).

2.6.3 Text editor software

In the absence of everything else, Notepad is better than using a word processor. However, many useful (and free) text editors exist that do a much better job. Some examples are Crimson Editor on Windows, and Kate on Linux. The ultimate text editor is a cross-platform software package called Emacs, which is extremely flexible and powerful, but it does have a steeper learning curve.

Professional code writers will often use an **integrated development environment** (IDE). These provide even greater support for writing code, but they tend to focus on a single computer language. An exception is the Eclipse package, which can be customized for many different languages, but again, the learning curve is steeper with this sort of software.

2.7 Further reading

"Code Complete"
by Steve McConnell]
2nd edition (2004) Microsoft Press.
Exhaustive discussion of ways of whys of writing good computer code. Includes languages and advanced topics way beyond the scope of this book.

Summary

Writing computer code should be performed with a text editor to produce a plain text file.

Code should first be checked for correct syntax (spelling and grammar).

Code that has correct syntax can then be run to determine whether it performs as intended.

Code should be written for human consumption as well as for correctness.

Comments should be included in code and the code should be arranged neatly so that the structure of the code is obvious to human eyes.

HTML is a simple language for describing the structure of the content of web pages. It is a useful cross-platform format for producing reports.

CSS is a language for controlling the appearance of the content of web pages.

The separation of code for a web page into HTML and CSS helps to avoid duplication of code (an example of the DRY principle in action).

HTML and CSS code can be run in any web browser.

3
HTML Reference

HTML is a computer language used to create web pages. HTML code can be run by opening the file containing the code with any web browser.

The information in this chapter describes HTML 4.01, which is a W3C Recommendation.

3.1 HTML syntax

HTML code consists of HTML **elements**.

An element consists of an opening **tag**, followed by the element **content**, followed by a closing tag. An opening tag is of the form `<elementName>` and a closing tag is of the form `</elementName>`. The example code below shows a `title` element; the opening tag is `<title>`, the closing tag is `</title>` and the content is the text: `Poles of Inaccessibility`.

```
<title>
    Poles of Inaccessibility
</title>
```

Some elements are **empty**, which means that they consist of only an opening tag (no content and no closing tag). The following code shows an `hr` element, which is an example of an empty element.

```
<hr>
```

An element may have one or more **attributes**, which are of the form `attributeName="attributeValue"`. Attributes appear in the opening tag. The code below shows the opening tag for a `table` element, with an attribute called `border`. The value of the attribute in this example is `"1"`.

```
<table border="1">
```

There is a fixed set of valid HTML elements (Section 3.2.1 provides a list of some common elements) and each element has its own set of possible

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
    <head>
        <title>A Minimal HTML Document</title>
    </head>
    <body>
        Your content goes here!
    </body>
</html>
```

Figure 3.1: A minimal HTML document.

attributes.

Certain HTML elements are compulsory. An HTML document must include a `DOCTYPE` declaration and a single `html` element. Within the `html` element there must be a single `head` element and a single `body` element. Within the `head` element there must be a `title` element. Figure 3.1 shows a minimal piece of HTML code.

Section 3.2.1 describes each of the common elements in a little more detail, including any important attributes, and which elements may be placed inside which other elements.

3.1.1 HTML comments

Comments in HTML code are anything within an opening `<!--` and a closing `-->`. All characters, including HTML tags, lose their special meaning within an HTML comment.

3.1.2 HTML entities

The less-than and greater-than characters used in HTML tags are special characters and must be escaped to obtain their literal meaning. The **escape sequences** are called **entities**. All entities start with an ampersand so the ampersand is also special and must be escaped. Entities provide a way to include some other special characters and symbols within HTML code as well. Table 3.1 shows some common HTML entities.

Table 3.1: Some common HTML entities.

Character	Description	Entity
<	less-than sign	`<`
>	greater-than sign	`>`
&	ampersand	`&`
π	greek letter pi	`&pi;`
μ	greek letter mu	`&mu;`
€	Euro symbol	`&euro;`
£	British pounds	`&pound;`
©	copyright symbol	`&copy;`

3.2 HTML semantics

The primary purpose of HTML tags is to specify the *structure* of a web page.

Elements are either **block-level** or **inline**. A block-level element is like a paragraph; it is a container that can be filled with other elements. Most block-level elements can contain any other sort of element. An inline element is like a word within a paragraph; it is a small component that is arranged with other components inside a container. An inline element usually only contains text.

The content of an element may be other elements or plain text. There is a limit on which elements may be nested within other elements (see Section 3.2.1).

3.2.1 Common HTML elements

This section briefly describes the important behaviour, attributes, and rules for each of the common HTML elements.

`<html>`
: Must have exactly one `head` element followed by exactly one `body` element. No attributes of interest.[1]

`<head>`
: Only allowed within the `html` element. Must have exactly one `title`.

[1] Except for some allowing internationalisation features such as language settings and direction of flow of text.

May also contain `link` elements to refer to external CSS files and/or `style` elements for inline CSS rules. No attributes of interest.

`<title>`
: Must go in the `head` element and must only contain text. Information for the computer to use to identify the web page rather than for display, though it is often displayed in the title bar of the browser window. No attributes.

`<link>`
: An empty element that must go in the `head` element. Important attributes are: `rel`, which should have the value `"stylesheet"`; `href`, which specifies the location of a file containing CSS code (can be a URL); `type`, which should have the value `"text/css"`. The `media` attribute may also be used to distinguish between a style sheet for display on `"screen"` as opposed to display in `"print"`.

  An example of a `link` element is shown below.

```
<link rel="stylesheet" href="csscode.css"
      type="text/css">
```

  Other sorts of links are also possible, but are beyond the scope of this book.

`<body>`
: Only allowed within the `html` element. Should only contain one or more block-level elements, but most browsers will also allow inline elements. Various appearance-related attributes are possible, but CSS should be used instead.

`<p>`
: A block-level element that can appear within most other block-level elements. Should only contain inline elements (words and images). Automatically typesets the contents as a paragraph (i.e., automatically decides where to break lines). May have the common attributes `class` and `style`.

`<img>`
: An empty, inline element (i.e., images are treated like words in a sentence). Can be used within almost any other element. Important attributes are `src`, to specify the file containing the image (this may be a URL, i.e., an image anywhere on the web), and `alt` to specify alternative text for non-graphical browsers.

`<a>`
: Known as an **anchor**. An inline element that can go inside any other

element. It can contain any other inline element (except another anchor). Important attributes are: `href`, which means that the anchor is a hypertext link and the value of the attribute specifies a destination (when the content of the anchor is clicked on, the browser navigates to this destination); `name`, which means that the anchor is the destination for a hyperlink.

The value of an `href` attribute can be: a URL, which specifies a separate web page to navigate to; something of the form `#target`, which specifies an anchor within the same document that has an attribute `name="target"`; or a combination, which specifies an anchor within a separate document. For example,

```
http://www.w3.org/TR/html401/
```

specifies the top of the W3C page for HTML 4.01 and

```
http://www.w3.org/TR/html401/#minitoc
```

specifies the table of contents further down that web page.

`<h1>` ... `<h6>`
Block-level elements that denote that the contents are a section heading. Can appear within almost any other block-level element, but can only contain inline elements. No attributes of interest. These should be used to indicate the section structure of a document, not for their default display properties. CSS should be used to achieve the desired weight and size of the text in headings.

`<table>`, `<tr>`, and `<td>`
A `table` element contains one or more `tr` elements, each of which contains one or more `td` elements (so `td` elements can only appear within `tr` elements, which can only appear within `table` elements). A `table` element may appear within almost any other block-level element. In particular, a table can be nested within the `td` element of another table.

The `table` element has a `summary` attribute to describe the table for non-graphical browsers. There are also attributes to control borders, background colours, and widths of columns, but CSS is the preferred way to control these features.

The `tr` element has attributes for the alignment of the contents of columns, including aligning numeric values on decimal points. There are no corresponding CSS properties.

The `td` element also has alignment attributes for the contents of a column for one specific row, but these can be handled via CSS instead. However, there are several attributes specific to `td` elements, in

particular, `rowspan` and `colspan` which allow a single cell to spread across more than one row or column.

Unless explicit dimensions are given, the table rows and columns are automatically sized to fit their contents.

It is tempting to use tables to arrange content on a web page, but it is recommended to use CSS for this purpose instead. Unfortunately, the support for CSS in web browsers tends to be worse for CSS than it is for `table` elements, so it may not always be possible to use CSS for arranging content. This warning also applies to controlling borders and background colours via CSS.

An example of a table with three rows and three columns:

```
<table>
    <tr>
        <td></td> <td>pacific</td> <td>eurasian</td>
    </tr>
    <tr>
        <td>min</td> <td>276</td> <td>258</td>
    </tr>
    <tr>
        <td>max</td> <td>283</td> <td>293</td>
    </tr>
</table>
```

It is also possible to construct more complex tables with separate `thead`, `tbody`, and `tfoot` elements to group rows within the table (i.e., these three elements can go inside a `table` element, with `tr` elements inside them).

`<hr>`

An empty element that produces a horizontal line. It can appear within almost any block-level element. No attributes of interest. This entire element can be replaced by CSS control of borders.

`<br>`

An empty element that forces a new line or line-break. It can be put anywhere. No attributes of interest. This element should be used sparingly. In general, text should be broken into lines by the browser to fit the available space.

`<ul>`, `<ol>`, and `<li>`

A `ul` or `ol` element contains one or more `li` elements. Anything can go inside an `li` element (i.e., you can make a list of text descriptions, a list of tables, or even a list of lists). The former case produces

a bullet-point list and the latter produces a numbered list. These elements have no attributes of interest. CSS can be used to control the style of the bullets or numbering and the spacing between items in the list.

It is also possible to produce "definition" lists, where each item has a heading. Use a `dl` element for the overall list with a `dt` element to give the heading and a `dd` element to give the definition for each item.

`<pre>`
: Block-level element that displays any text content exactly as it appears in the source code. Good for displaying computer code. It is possible to have other elements within a `pre` element. No attributes of interest. Like the `hr` element, this element can usually be replaced by CSS styling.

`<div>` and `<span>`
: Generic block-level and inline elements (respectively). No attributes of interest. These can be used as "blank" elements with no predefined appearance properties. Their appearance can then be fully specified via CSS. In theory, any other HTML element can be emulated using one of these elements and appropriate CSS properties. In practice, the standard HTML elements are more convenient for their default behaviour and these elements are used for more exotic situations.

3.2.2 Common HTML attributes

Almost all elements may have a `class` attribute, so that a CSS style specified in the `head` element can be associated with that element. Similarly, all elements may have an `id` attribute, which can be used to associate a CSS style. The value of all `id` attributes within a piece of HTML code must be unique.

All elements may also have a `style` attribute, which allows "inline" CSS rules to be specified within the element's opening tag.

3.3 Further reading

The W3C HTML 4.01 Specification
: `http://www.w3.org/TR/html401/`
The formal and official definition of HTML. Quite technical.

Getting started with HTML
by Dave Raggett
`http://www.w3.org/MarkUp/Guide/`
An introductory tutorial to HTML by one of the original designers of the language. A bit dated, but still a good starting point.

The Web Design Group's HTML 4 web site
`http://htmlhelp.com/reference/html40/`
A more friendly user-oriented description of HTML.

The w3schools HTML Tutorial
`http://www.w3schools.com/html/`
Quick, basic tutorial-based introduction to HTML.

4
CSS Reference

Cascading Style Sheets (CSS) is a language used to specify the appearance of web pages—fonts, colours, and how the material is arranged on the page.

CSS is run when it is linked to some HTML code (see Section 4.4) and that HTML code is run.

The information in this chapter describes CSS level 1, which is a W3C Recommendation.

4.1 CSS syntax

CSS code consists of one or more **rules**.

Each CSS rule consists of a **selector** and, within brackets, one or more **properties**.

The selector specifies which HTML elements the rule applies to and the properties control the way that those HTML elements are displayed. An example of a CSS rule is shown below:

```
table {
    border-width: thin;
    border-style: solid;
}
```

The code `table` is the selector and there are two properties, `border-width` and `border-style`, with values `thin` and `solid`, respectively.

4.2 CSS selectors

Within a CSS rule, the selector specifies which HTML elements will be affected by the rule. There are several ways to specify a CSS selector:

Element selectors:

The selector is just the name of an HTML element. All elements of this type in the linked HTML code will be affected by the rule. An example is show below:

```
a {
    color: white;
}
```

This rule will apply to *all* anchor (`a`) elements within the linked HTML code.

Class selectors:

The selector contains a full stop (`.`) and the part after the full stop describes the name of a **class**. All elements that have a `class` attribute with the appropriate value will be affected by the rule. An example is shown below:

```
p.footer {
    font-style: italic;
}
```

This rule will apply to any paragraph (`p`) element that has the attribute `class="footer"`. It will *not* apply to other `p` elements. It will not apply to other HTML elements, even if they have the attribute `class="footer"`.

If no HTML element name is specified, the rule will apply to *all* HTML elements with the appropriate class. An example is shown below:

```
.figure {
    margin-left: auto;
    margin-right: auto;
}
```

This rule will apply to any HTML element that has the attribute `class="figure"`.

ID selectors:

The selector contains a hash character (`#`). The rule will apply to all elements that have an appropriate `id` attribute. This type of rule can be used to control the appearance of exactly one element. An example is shown below:

```
p#footer {
    font-style: italic;
}
```

This rule will apply to the paragraph (`p`) element that has the attribute `id="footer"`. There can only be one such element within a piece of HTML code because the `id` attribute must be unique for all elements. This means that the HTML element name is redundant and can be left out. The rule below has the same effect as the previous rule:

```
#footer {
    font-style: italic;
}
```


4.3 CSS properties

This section describes some of the common CSS properties, including the values that each property can take.

`font-family:`
: Controls the overall font family (the general style) for text within an element. The value can be a generic font type, for example, `monospace` or `serif`, or it can be a specific font family name, for example, `Courier` or `Times`. If a specific font is specified, it is usually a good idea to also include (after a comma) a generic font as well in case the person viewing the result does not have the specific font on their computer. An example is shown below:

  ```
  font-family:  Times, serif
  ```

  This means that a Times font will be used if it is available, otherwise the browser will choose a serif font that is available.

`font-style:`, `font-weight:`, and `font-size:`
: Control the detailed appearance of text. The style can be `normal` or `italic`, the weight can be `normal` or `bold`, and the size can be `large` or `small`.

  There are a number of relative values for size (they go down to `xx-small` and `xx-large`), but it is also possible to specify an absolute size, such as `24pt`.

`color:` and `background-color:`
: Control the foreground colour (e.g., for displaying text), and the background colour for an element.

  For specifying the colour value, there are a few basic colour names, e.g., `black`, `white`, `red`, `green`, and `blue`, but for anything else it is

necessary to specify a red-green-blue (RGB) triplet. This consists of an amount of red, an amount of green, and an amount of blue. The amounts can be specified as percentages so that, for example, `rgb(0%, 0%, 0%)` is black and `rgb(100%, 100%, 100%)` is white, and Ferrari red is `rgb(83%, 13%, 20%)`.[1]

`text-align:`
: Controls the alignment of text within an element, with possible values `left`, `right`, `center`, or `justify`. This property only makes sense for block-level elements.

`width:` and `height:`
: Allows explicit control of the width or height of an element. By default, these are the amount of space required for the element. For example, a paragraph of text expands to fill the width of the page and uses as many lines as necessary, while an image has an instrinsic size (number of pixels in each direction).

  Explicit widths or heights can be either percentages (of the parent element) or an absolute value. Absolute values must include a unit, e.g., `in` for inches, `cm` for centimetres, or `px` for pixels. For example, within a web page that is 800 pixels wide on a screen that has a resolution of 100 dots-per-inch (dpi), to make a paragraph of text half the width of the page, the following three specifications are identical:

```
p { width: 50% }

p { width: 4in }

p { width: 400px }
```

`border-width:`, `border-style:`, and `border-color:`
: Control the appearance of borders around an element. Borders are only drawn if the `border-width` is greater than zero. Valid border styles include `solid`, `double`, and `inset` (which produces a fake 3D effect).

  These properties affect all borders, but there are other properties that affect only the top, left, right, or bottom border of an element. For example it is possible to produce a horizontal line at the top of a paragraph by using just the `border-top-width` property.

`margin:`
: Controls the space around the outside of the element (between this

[1] According to the COLOURlovers web site `http://www.colourlovers.com/color/D32232/Ferrari_Red`.

element and neighbouring elements). The size of margins can be expressed using units, as for the `width` and `height` properties.

This property affects all margins (top, left, right, and bottom). There are properties, e.g., `margin-top`, for controlling individual margins instead.

`padding:`
Controls the space between the border of the element and the element's contents. Values are specified as they are for margins. There are also specific properties, e.g., `padding-top`, for individual control of the padding on each side of the element.

`display:`
Controls how the element is arranged relative to other elements. A value of `block` means that the element is like a self-contained paragraph (typically, with an empty line before it and an empty line after it). A value of `inline` means that the element is just placed beside whatever was the previous element (like words in a sentence). The value `none` means that the element is not displayed at all.

Most HTML elements are either intrinsically block-level or inline, so some uses of this property will not make sense.

`whitespace:`
Controls how whitespace in the content of an element is treated. By default, any amount of whitespace in HTML code collapses down to just a single space when displayed, and the browser decides when a new line is required. A value of `pre` for this property forces all whitespace within the content of an element to be displayed (especially all spaces and all new lines).

`float:`
Can be used to allow text (or other inline elements) to wrap around another element (such as an image). The value `right` means that the element (e.g., image) "floats" to the right of the web page and other content (e.g., text) will fill in the gap to the left. The value `left` works analogously.

`clear:`
Controls whether floating elements are allowed beside an element. The value `both` means that the element will be placed below any previous floating elements. This can be used to have the effect of turning off text wrapping.

4.4 Linking CSS to HTML

CSS code can be linked to HTML code in one of three ways:

External CSS:
The CSS code can be in a separate file and the HTML code can include a `link` element within its `head` element that specifies the location of the CSS code file. An example is shown below:

```
<link rel="stylesheet" href="csscode.css"
      type="text/css">
```

This line would go within a file of HTML code and it refers to CSS code within a file called `csscode.css`.

Embedded CSS:
It is also possible to include CSS code within a `style` element within the `head` element of HTML code. An example of this is shown below:

```
<html>
    <head>
        <style>
            p.footer {
                font-style: italic;
            }
        </style>
    ...
```

This approach is not recommended because any reuse of the CSS code with other HTML code requires copying the CSS code (which violates the DRY principle).

Inline CSS:
It is also possible to include CSS code within the `style` attribute of an HTML element. An example is shown below:

```
<p style="font-style: italic">
```

This approach is actively discouraged because it leads to many copies of the same CSS code within a single piece of HTML code.

4.5 CSS tips and tricks

4.6 Further reading

The W3C CSS level 1 Specification
`http://www.w3.org/TR/CSS1`
The formal and official definition of CSS (level 1). Quite technical.

Adding a touch of style
by Dave Raggett
`http://www.w3.org/MarkUp/Guide/Style.html`
An introductory tutorial to CSS by one of the original designers of HTML. A bit dated, but still a good starting point.

The Web Design Group's CSS web site
`http://htmlhelp.com/reference/css/`
A more friendly user-oriented description of CSS.

The w3schools CSS Tutorial
`http://www.w3schools.com/css/`
A tutorial-based introduction to CSS.

5
Data Entry

This chapter is concerned with how computer technologies can assist in the task of recording information.

Many large data sets, such as bank transactions, computer network activity, and satellite weather recordings are collected directly by machines. In those cases, the data recording is about as good as it gets. The data are immediately electronic and the data that are recorded are a faithful representation of what the recording mechanism "sees".

However, a huge amount of data is still collected by humans via interview or written surveys and this sort of data collection requires a **data entry** step in order to get the information into an electronic format.

There are good reasons for using humans to collect data. For example, humans are better than computers at interviewing other human subjects, at explaining procedures and answering arbitrary questions. Humans are also better at tasks like observing animal behaviour in the field, at judging criteria such as "level of aggressiveness", and assessing variables that are hard to define or capture precisely in a mechanical manner. However, there are serious disadvantages to having humans collect data.

When it comes to recording or copying information, humans are slow and incaccurate, so there are efficiencies to be gained from ensuring that the data are recorded only once and that the data are checked as they are entered.

In this chapter, we will look at **electronic forms**, a computer technology for assisting data entry.

5.1 Case study: I-94W

Temporary entry to the United States of America is very straightforward for citizens of countries that are involved in the visa waiver program, *unless* you have been involved in various undesirable activities ...

Temporary visitors to the United States of America may enter without a valid visa if they carry the passport of a country involved in the US visa waiver programme. Such visitors must fill out a form called I-94W.

This form requests standard demographic information followed by a series of questions regarding engagement in various criminal activities. The latter list of questions is shown in Figure 5.1.

Figure 5.2 shows an **electronic form** version of some of these questions. More specifically, it shows an HTML form; a web page containing interactive components that allow us to enter information.

One advantages of the electronic form should be immediately obvious: the form only allows the user to select one of the countries currently participating in the visa waiver program. This improvement in the accuracy of data collection is one of the prime reasons for using an electronic form.

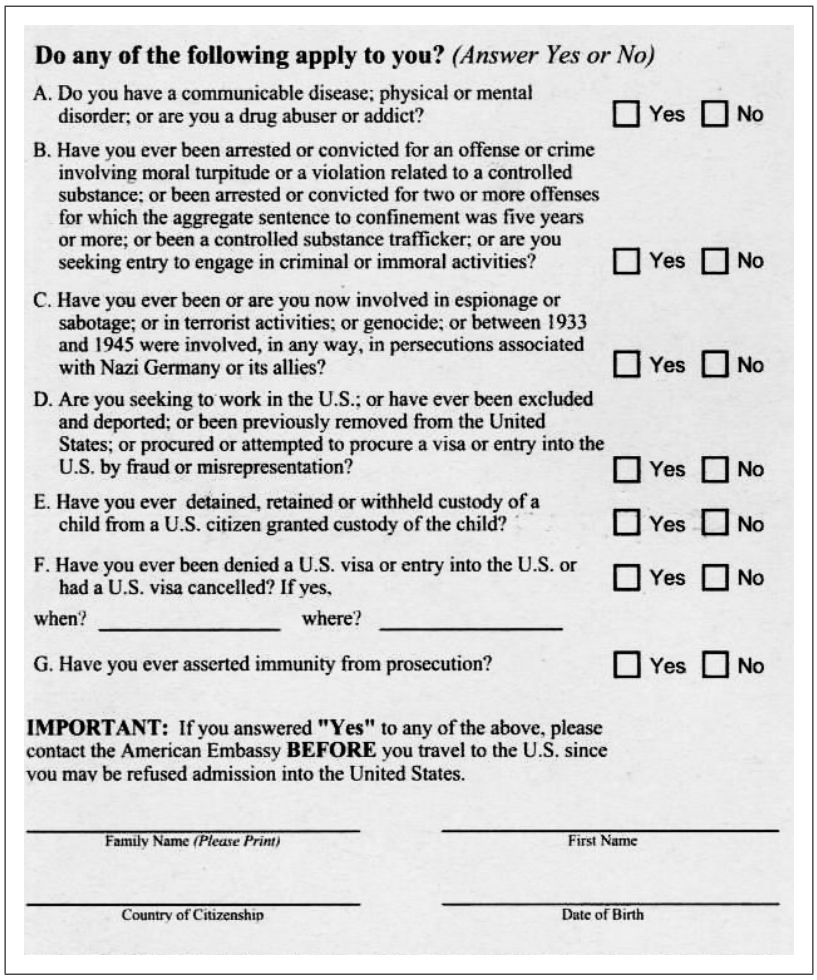
In this chapter we will discuss the advantages and disadvantages of using electronic forms for questionnaires like I-94W and for data entry in general. We will also learn how to create an electronic form by writing HTML code.

5.2 Electronic forms

An electronic form is a graphical user interface (GUI) for entering data into a computer. The main advantage of using such a form for data entry is straightforward: information can be checked for accuracy and validity at the time that it is entered.

An electronic form makes it easy to constrain the input to be one of a fixed set of valid responses, which improves the accuracy with which data are collected Section 5.3 describes the standard electronic form elements and how they can be used to limit user input.

There are also ways to perform higher-level checks on the consistency of



Do any of the following apply to you? *(Answer Yes or No)*

A. Do you have a communicable disease; physical or mental disorder; or are you a drug abuser or addict? ☐ Yes ☐ No

B. Have you ever been arrested or convicted for an offense or crime involving moral turpitude or a violation related to a controlled substance; or been arrested or convicted for two or more offenses for which the aggregate sentence to confinement was five years or more; or been a controlled substance trafficker; or are you seeking entry to engage in criminal or immoral activities? ☐ Yes ☐ No

C. Have you ever been or are you now involved in espionage or sabotage; or in terrorist activities; or genocide; or between 1933 and 1945 were involved, in any way, in persecutions associated with Nazi Germany or its allies? ☐ Yes ☐ No

D. Are you seeking to work in the U.S.; or have ever been excluded and deported; or been previously removed from the United States; or procured or attempted to procure a visa or entry into the U.S. by fraud or misrepresentation? ☐ Yes ☐ No

E. Have you ever detained, retained or withheld custody of a child from a U.S. citizen granted custody of the child? ☐ Yes ☐ No

F. Have you ever been denied a U.S. visa or entry into the U.S. or had a U.S. visa cancelled? If yes, ☐ Yes ☐ No

when? _____ where? _____

G. Have you ever asserted immunity from prosecution? ☐ Yes ☐ No

IMPORTANT: If you answered **"Yes"** to any of the above, please contact the American Embassy **BEFORE** you travel to the U.S. since you may be refused admission into the United States.

_____ Family Name *(Please Print)*

_____ First Name

_____ Country of Citizenship

_____ Date of Birth

Figure 5.1: The top half of the back side of USCIS form I-94W, an application for a temporary VISA waiver for visitors to the United States.

Figure 5.2: An electronic form version of USCIS form I-94W (see Figure 5.1).

information that is entered, for example, to eliminate the possibility of including in a data set male subjects who claim to have given birth. In Section 5.4, we will discuss various ways to validate user input.

In addition to the benefits of accuracy and validity-checking, electronic forms provide a number of advantages for the collection of information, when compared to a pen-and-paper format. For example, using electronic forms can reduce the cost of administering a survey because there are no printing or postage costs. On the other hand, problems such as nonresponse bias can be worse with electronic forms because people are unable or simply prefer not to interact with a computer.[1]

Although electronic forms are not perfect in all ways, they are clearly an important additional tool in the process of data entry. In the rest of this chapter, we will discuss the creation and use of electronic forms and we will see how to create electronic forms in web pages using HTML.

5.2.1 HTML forms

There are many software systems for creating electronic forms. For example, forms can be created within Microsoft Excel to validate data that is entered into a spreadsheet. Many database management systems, such as Oracle and Microsoft Access, also provide facilities for generating forms for data entry.

HTML forms are a good platform for creating forms for several reasons. First, HTML is an open standard and HTML forms should work with any web browser. This makes it a very accessible technology for creating electronic forms and it is also an advantage for deploying electronic forms because it does not impose high expectations on the users of the form in terms of their technical ability or in terms of their computing facilities.

Another advantage is that HTML forms are based on a text description (HTML code) so we can create a form just by writing computer code. Last, but not least, because we have already learnt a little about HTML, we are in a good position to press on with creating forms using HTML.

Figure 5.3 shows an excerpt from the code behind the I-94W electronic form in Figure 5.2.

The HTML code for the I-94W electronic form is larger and contains new elements compared to the plain HTML examples in Chapter 2, but the basic structure is the same: HTML tags surround text content.

[1] For further discussion of these issues see the Exploring Online Research Methods web

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
    <title>I-94W online</title>
    <style>
        body { background-color: #9CF2CA; padding-top: 20px;
               padding-left: 5%; padding-right: 5%; }
    </style>
</head>
<body>
    <form action="http://www.formbuddy.com/cgi-bin/form.pl"
          method="post">
    <p>
    <b>Do any of the following apply to you?</b>
    </p>

    <table>
    <tr><td valign="top">A.</td>
        <td valign="top"><input type="checkbox" name="ill"></td>
        <td>Do you have a communicable disease; physical or
            mental disorder; or are you a drug abuser or
            addict?</td></tr>
    <tr><td valign="top">F.</td>
        <td valign="top"><input type="checkbox" name="visa"></td>
        <td>Have you ever been denied a U.S. visa or entry into
            the U.S or had a U.S. visa cancelled?</td></tr>
    <tr><td></td>
        <td></td>
        <td>If yes,
            when?  <input type="text" name="when" size=10>
            where? <input type="text" name="where" size=10></td>
    </tr>
    </table>

    <p>
    Country of Citizenship:
    </p>
    <table>
    <tr><td width=25%>
            <input type="radio" name="country" value="1">
            Andorra</td>
        <td width=25%>
            <input type="radio" name="country" value="2">
            Iceland</td>
```

Figure 5.3: An extract from the HTML code behind the electronic form version of I-94W (see Figure 5.2). This code shows that an HTML Form consists of plain HTML elements (as discussed in Chapter 1) and special form-related elements such as `form` and `input`.

There are the standard `html`, `head`, `title`, and `body` elements (lines 1 to 10), plus an example of embedded CSS rules (lines 5 to 8). The most important new elements are a `form` element (starting on line 11) and several `input` elements (lines 13 and 14), which create the radio buttons and the Submit button on the web page.

This demonstrates that creating an HTML form is simply a matter of learning a few more HTML elements. We will look at these new elements in detail in Section 5.3.

5.2.2 Other uses of electronic forms

In addition to their role in data entry, electronic forms are also useful in a general sense as a graphical user interface. For example, a web page containing interactive elements such as buttons and checkboxes is essentially a dialog box. One application of this idea is to use an electronic form as a user-friendly interface to a data analysis program. This idea will be demonstrated later in the book in Section 11.12.

5.3 Electronic form components

The fundamental feature of electronic forms is that they contain **interactive components** for entering data. Information is entered not just by typing text, but also by selecting menu items, or checking boxes.

This section describes the standard set of electronic form components and when they are typically used. We will also see how to create the components using HTML.

5.3.1 HTML form elements

The primary HTML element used for creating forms is, appropriately, the `form` element. An example can be seen on lines 11 and 12 in Figure 5.3. This element does not display anything in a browser, but it provides a container for all other form elements.

The `form` element dictates how and where the information that is entered into the form gets submitted. The important attributes that control these aspects of a form are discussed later in Section 5.5.

---

site `http://www.geog.le.ac.uk/orm/site/home.htm`.

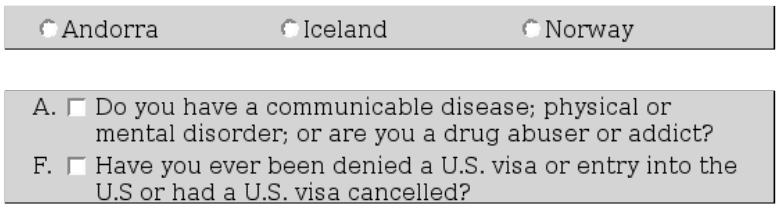


Figure 5.4: Examples of radio buttons (top) and check boxes (bottom) in an electronic form.

All other form components are described by elements nested within the `form` element and will be addressed in the appropriate section below.

5.3.2 Radio buttons

Radio buttons allow a single response to be chosen from a fixed set of valid responses. The advantage of this sort of component is that it restricts the input to only valid responses.

Radio buttons can be used to enter data for a categorical variable. A single variable, e.g., gender, requires a group of radio buttons, with one radio button for each possible value of the categorical variable, e.g., one button for male and one button for female.

Radio buttons are not appropriate when the list of possible responses is long, such as when selecting a year of birth. In cases like this, a menu or drop-down list is more appropriate (see Section 5.3.5).

An example of the use of radio buttons in the I-94W form is shown in Figure 5.4. The typical display of a radio button is a small circle, with a dot drawn within the circle to show that the radio button has been selected.

In HTML, radio buttons are generated by `input` elements. The code used to produce one of the radio buttons in Figure 5.4 is shown below:

```
<input type="radio" name="country" value="1" />
```

The `input` element can be used to produce several different form components via the `type` attribute. In this case, a radio button is produced by specifying `type="radio"`.

The other important attributes are `name` and `value`. The `value` specifies what data value will be recorded if this radio button is selected (selecting

this radio button will record the value `1`). The `value` can be any text so, for example, an alternative would be to specify `value="Andorra"` because this radio button correponds to the Andorra option on the form. That would make it easier to know what this radio button is for, but would be less efficient in terms of storing the information.

The `name` attribute provides a label for the information that is to be stored. It corresponds to the name of a variable in a data set. The important thing to remember when creating radio buttons in an HTML form is that all radio buttons within the same set of answers must have the same `name` attribute. This is necessary so that only one of the radio can be selected at once, which is the characteristic behaviour of radio buttons. For example, the code below shows the HTML behind another of the radio buttons from the same question in Figure 5.4:

```
<input type="radio" name="country" value="2">
```

The `value` for this radio button is `2` (which corresponds to an answer of Iceland), but the `name` is `country`, which is the same as for the other radio button. This means that only one of these two radio buttons can be selected at once.

Only one value will be stored from a group of radio buttons so it is important that the `value` attributes are all different for all radio buttons with the same `name`.

It is important to note that each `input` element only produces a little round radio button on the web page. All of the text labels are produced by plain text within the web page and it is up to the page designer to make sure that the text for questions and the text for answers is appropriately arranged relative to the form components. We will discuss this further in Section 5.3.8.

For radio buttons, the `input` element has a `checked` attribute that can be used to make one of the radio buttons within a group selected by default when the form is first displayed.

5.3.3 Check boxes

Check boxes allow zero or more responses to be chosen from a fixed set of options. The difference between check boxes and radio buttons is that, with check boxes, it is valid to select none of the options *and* it is valid to select more than one option at the same time.

Check boxes can be used to enter data for yes/no questions. Each check box corresponds to a different variable.

An example of the use of check boxes in the I-94W electronic form is shown in Figure 5.4. The typical display of check boxes is a square, with a tick or a cross within the square to indicate that the check box has been selected.

In HTML, check boxes are another variation of the `input` element, this time with `type="checkbox"`. The code below shows the HTML behind the two check boxes in Figure 5.4:

```
<input type="checkbox" name="ill">

<input type="checkbox" name="visa">
```

The `value` attribute is less important with check box components because there are only two possible values for a check box: selected or unselected. What is more important is the component `name`, which corresponds to the name of the variable that is being recorded. Every check box in an electronic form should have a unique name.

Like for radio buttons, the `input` element for check boxes has a `checked` attribute that can be used to make any checkbox selected by default when the form is first displayed.

5.3.4 Text fields

Text fields allow open-ended responses. It is much harder to constrain text input to valid values—for example, it may be hard to ensure that a telephone number has the correct format—but if the set of possible answers is unknown or infinite, for example, when asking for a person's name, then this sort of form component is the only option. Sophisticated validation of text responses can still be done, but it requires knowledge of other technologies (see Section 5.4 and Section 11.8.5).

A text field usually corresponds to data entry for a single variable.

Common uses of text fields are to allow the respondent to expand upon a previous response to a question or to allow open-ended feedback at the end of a survey. Examples of these uses of text fields are shown in Figure 5.5.

In HTML, there are two types of text form component, one for entering small amounts of text (a single word or expression) and one for entering large amounts of text. The code below shows the HTML behind the first small text field in Figure 5.5. This is another variation on the `input` element.

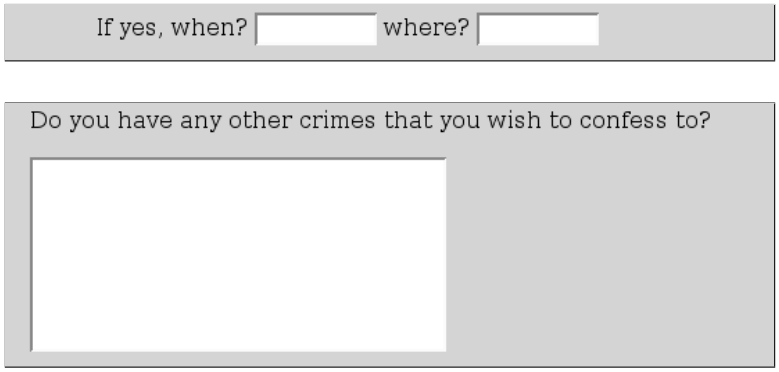


Figure 5.5: Examples of text fields in an electronic form, with input restricted (top) and open-ended (bottom).

```
<input type="text" name="when" size=10>
```

Again, the `name` attribute is the most important so that different text input components within a form can be distinguished from each other. The information that is recorded for this form component is whatever the user types into the text field. The `value` attribute just provides default text (i.e., this is what is shown when the form is first viewed).

This type of `input` element also allows a `maxlength` attribute, which can be used to limit the number of characters that the user can type into the field. There is also a `size` attribute to control how wide the text field appears on screen (as a number of characters of text).

For entering large blocks of text, there is a separate, `textarea`, element. The HTML code for the large text field in Figure 5.5 is shown below:

```
<textarea name="confession" rows="8" cols="40">
</textarea>
```

The `rows` and `cols` attributes of this element are used to control how many rows and columns of text are displayed on screen (i.e., how big the text field is on screen). The content of the element is used as default text when the form is first displayed. Care should be taken with this content because all white space is preserved, so any spaces or new lines show up in the default text. In the example, the default is to show a blank text field.

Figure 5.6: An example of a menu form component. The bottom of the menu has been cropped in this image, but the advantage of a menu component is that it can include a large number of options without taking up too much space on the screen when the user is not interacting with this component.

5.3.5 Menus

A menu or drop-down list allows a single response to be chosen from a fixed set of options. This type of form component is very similar to a radio button, but, because a menu generally takes up less space on screen, it is more appropriate when the list of available options is large.

Menus can be used to record data for a single variable.

The I-94W question on country of citizenship (shown in Figure 5.2) could be presented as a menu. A possible menu presentation for this question is shown in Figure 5.6.

In HTML, a menu is created using a `select` element for the overall menu with `option` elements inside to specify the individual menu options. The HTML code for the menu in Figure 5.6 is shown below:

```
<select name="country">
    <option>Andorra</option>
    <option>Iceland</option>
    <option>Norway</option>

⋮
    <option>Germany</option>
    <option>New Zealand</option>
    <option>United Kingdom</option>
</select>
```

Some of the code has been left out, but the pattern should be clear.

The value that is recorded from the menu component is the value of the

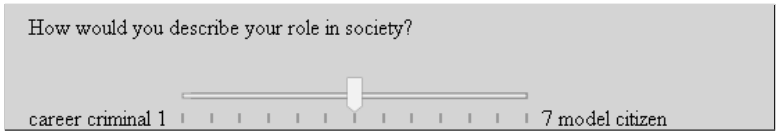


Figure 5.7: An example of a question that asks the subject to provide a rating, implemented as a slider.

`option` element that is selected by the user. By default, the value of an `option` element is the content of the element, but there is a `value` attribute to allow a different value to be specified. For example, in the following code, the value of the element would be `NZL` not `New Zealand`:

```
<option value="NZL">New Zealand</option>
```

The `option` element also has a `selected` attribute that can be used to set the default menu option when the form is first displayed.

5.3.6 Sliders

A slider allows a response along a continuous or fine-grained scale. The response is constrained by minimum and maximum values, but remains free within that range.

A slider is often used to enter data for a rating variable. Figure 5.7 shows an example of a slider input element.

Unfortunately, HTML has no form element corresponding to a slider component. Section 5.4.2 describes some alternative technologies that could be considered if this type of form component is important to a survey.

5.3.7 Buttons

A very important component in any form is the button at the bottom that lets the user submit the information that has been entered.

In HTML, a submit button can be produced either using a `button` element or as yet another variation on the `input` element. The following code shows the HTML behind the submit button on the I-94W form in Figure 5.2:

```
<input type="submit" value="submit">
```

The `value` attribute can be used to specify the text that is displayed on the button. The next piece of code is equivalent, but uses a `button` element instead:

```
<button value="submit" name="submit" type="submit">
```

What happens when the submit button is clicked will be discussed further in Section 5.5.

5.3.8 Labels

As mentioned in Section 5.3.2, the arrangement of text labels relative to the actual form components is entirely the responsibility of the designer of the form. Care must be taken to make sure that the text label beside, say, a radio button corresponds to the value that is recorded for the radio button.

For example, the following HTML code is perfectly valid syntax, but it is semantically-challenged:

```
Select your gender:
  male <input type="radio" name="gender" value="female">
female <input type="radio" name="gender" value="male">
```

It is possible to use a `label` element within an HTML form to explicitly relate a piece of text to a form component, but even then the semantic content of the text is still the responsibility of the form author.

One advantage of using the `label` element is that it can provide assistance for using a form with non-graphical browsers and it can enhance the behaviour of the components in on-screen interactions. For example, if a `label` element is used to relate a text label with a check box or radio button, then a click on the text will select the corresponding check box or radio button.

A `label` element is related to a form component either by placing the form component within the `label` element, or by using the `for` attribute of the `label` element. The code below shows a `label` element being used to relate the text label to the first radio button in Figure 5.4:

```
<input type="radio" name="country" value="1"
       id="COUNTRY01">
<label for="COUNTRY01">
    Andorra
</label>
```

The value of the `for` attribute of the `label` element refers to the value of the `id` attribute of the radio button element. The following code is equivalent, but uses the approach of embedding the radio button element within the `label` element:

```
<label>
    <input type="radio" name="country" value="1" />
    Andorra
</label>
```

The use of `label` elements is also an example of good documentation technique. Even if there is no effect in terms of the behaviour of the form, there is a gain in terms of the clarity and maintainability of the underlying code.

5.4 Validating input

The previous section described how electronic form components such as radio buttons, check boxes, and menus ensure that only valid data can be entered into a form. However, form components such as text fields allow unconstrained input. Furthermore, there may be relationships between different form elements that need to be maintained. For example, if a person has greater than zero grandchildren, that person should have at least one child.

For these reasons, it is useful to be able to enforce rules or constraints on the values that are entered into form components such as text fields.

There are two basic approaches to checking form data: **client-side**, where the checking is done by the web browser *before* the form data is submitted; and **server-side**, where the checking is done on a web server *after* the form data has been submitted.

In this section, we will look at several approaches to **client-side validation** of the form data. Section 5.5 contains a discussion of server-side validation.

```
 1 <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
 2 <html>
 3 <head>
 4     <title>A Minimal Script</title>
 5     <script type="text/ecmascript">
 6         alert("Welcome to my web page!")
 7     </script>
 8 </head>
 9 <body>
10 </body>
11 </html>
```

Figure 5.8: A minimal JavaScript.

5.4.1 JavaScript

In addition to the HTML code that describes content and structure, and the CSS code that describes layout and appearance, a web page may also contain **scripts**—code in a scripting language, that describes dynamic behaviour for the web page.

Within the `head` element of a web page, it is possible to include one or more `script` elements, which either contain script code or provide a reference to a separate file containing script code.

There are a number of scripting languages that can be used within web pages, but the major, cross-platform, standardised language is JavaScript.[2]

Figure 5.8 shows a minimal web page with a tiny script that pops up an annoying message when the web page is loaded. Figure 5.9 shows the result when the web page is viewed in a browser.

The `script` element (lines 5 to 7) has a `type` attribute that is used to specify which scripting language is being used. As with a `style` element that contains CSS code, the content of a `script` element is scripting code, *not* HTML, so the rules of syntax within a `script` element are completely different to the rest of the document.

A complete discussion of JavaScript is beyond the scope of this book (though we will discuss writing general-purpose scripts in a different language in Chapter 11). However, it is possible to make use of scripts without having to write them, so we will now describe some simple scripts that can be used to perform basic checking of form input.

[2]The language standard is actually called ECMAScript. The standard is implemented as JavaScript in Mozilla-based browsers (e.g., Firefox) and as JScript in Internet Explorer.

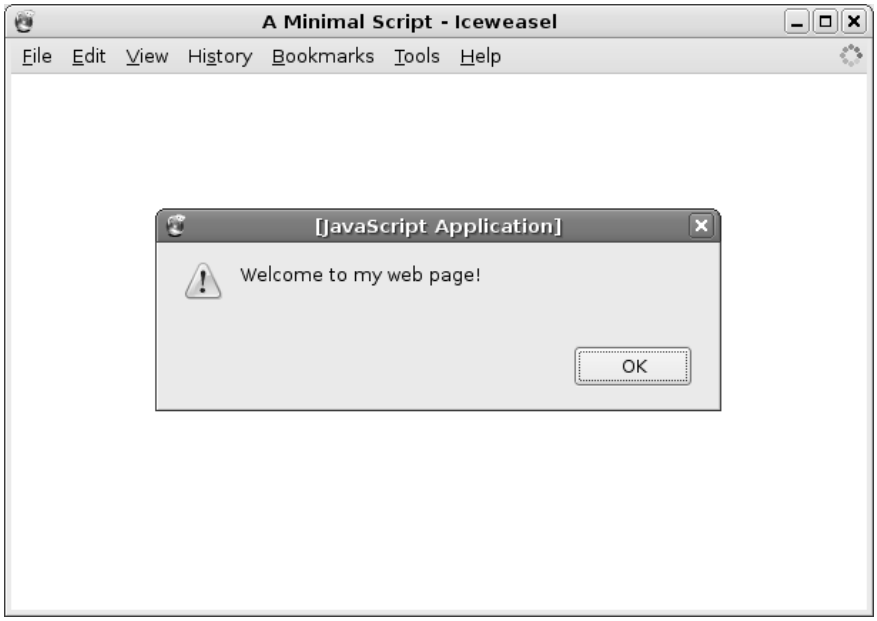


Figure 5.9: A minimal web page containing JavaScript as displayed by the Iceweasel browser on Debian Linux.

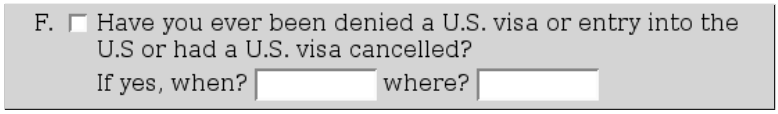


Figure 5.10: An example of a text field in the I-94W form.

The code in a script is run when a web page is loaded. In the example shown in Figure 5.8, the code does something immediately—it pops up a dialog box. However, it is more common for script code to define **functions**, which are parcels of code that will be run later, when the user interacts with the web page.

HTML form components have attributes that can be used to specify that a script function should be run when something happens to the form component. For example, there is an `onfocus` attribute that is run when the user interacts with a form component (e.g., when the user clicks in a text field or clicks on a radio button). Similarly, there is an `onblur` attribute that is run when a form control loses focus (i.e., when the user clicks somewhere else).

As an example, consider again the text field in the I-94W form for recording when access to the U.S. or a visa application had been denied. This was originally shown in Figure 5.5, but it is reproduced in Figure 5.10 for convenience and to show the form elements in their full context within the survey.

Suppose that we would like to limit the response to the "when?" question to be a valid year. That is, we would like the user only to be able to enter a four-digit number between 1776 (just to be safe) and 2007.

The use of an `input` element with `type="text"` and `maxlength="4"` could be used to restrict input to four characters. However, it would be much better if we could force the user to enter *exactly* four characters (no fewer), *and* if we could force the user to enter only digits (i.e., no letters or symbols).

We can use JavaScript to perform this additional checking via the following steps:

1. Add a `script` element to the HTML code that loads a set of JavaScript functions for checking form input. The script element must go within the `head` element and should look like this:

```
<script type="text/javascript" src="validate.js">
</script>
```

The file containing the JavaScript code, `validate.js`, can be downloaded from the book website[3] and should be placed in the same directory as the file containing the HTML code.[4]

2. Add an `onblur` attribute to the `input` element, so that when the user enters data into the text field, a JavaScript function will be run to check that the code is valid. In this example, we want to check that the input is a four-digit number, so we use the `hasIntegerLength` function. The `input` element for the text field should now look like this:

```
<input type="text" name="when" size="4" maxlength="4"
       onblur="hasIntegerLength(this, 4)">
```

   If a non-number is entered, or a number with any number of digits other than four is entered, or the field is left blank, an error message will appear and the text field will be selected so that the value can be corrected. Figure 5.11 shows the error message from an invalid data entry.

   This is just one example of the sort of check that we might make on an electronic form component. The JavaScript code in the file `validate.js` provides several other functions like this for checking the input. The full list of functions is provided in Chapter 6.

3. Add an `onsubmit` attribute to the `form` element that calls a `validateAll` script function. This attribute runs the script function before submitting the form data. It will check that all of the fields that have `onblur` attributes are checked again before the form is submitted.

   This is an important step because a form can be submitted without the user visiting all of the fields within a form. This step ensures that all fields are validated when the user attempts to submit the form.

   The `form` element in this example now looks like this:

```
<form action="http://www.formbuddy.com/cgi-bin/form.pl"
      method="post" onsubmit="return(validateAll())">
```

   If *any* of the validation checks fail, the form data is not submitted.

This series of steps can be used to add simple validation to an electronic form. More complex validation requires knowledge of JavaScript and/or server-side technologies (see Section 5.5 and Section 6.5).

[3] `http://www.stat.auckland.ac.nz/~paul/ItDT/`

[4] Alternatively, the `src` attribute of the `script` element can contain the full URL to the JavaScript file so that the file of JavaScript code is downloaded whenever the HTML form is loaded into a browser.

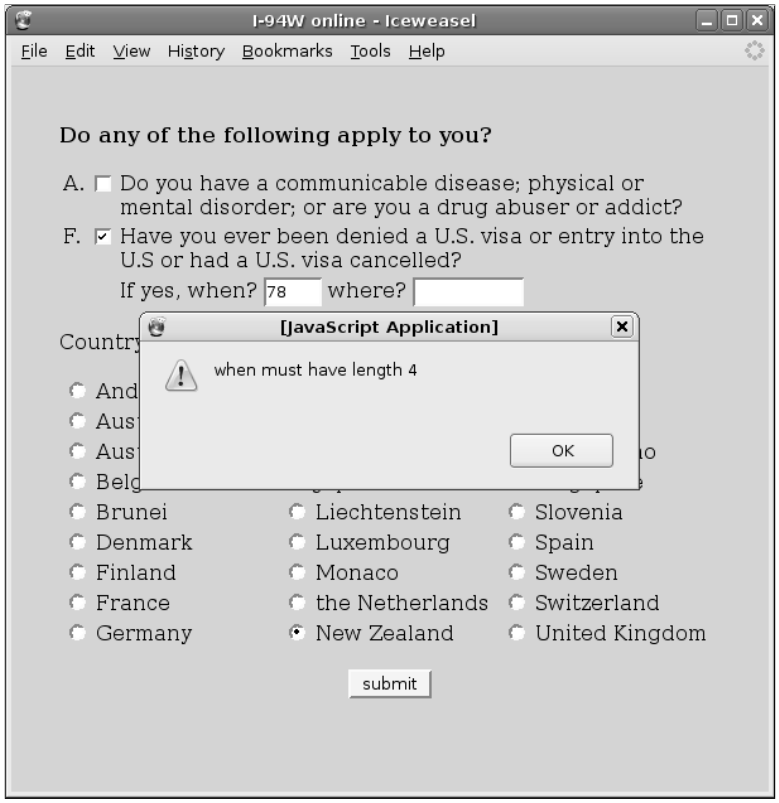


Figure 5.11: An example of an error message resulting from invalid input to a text field.

5.4.2 Other electronic forms technologies

The lack of facilities for validating input in pure HTML is one of the motivations behind more recent projects aimed at developing standard languages for describing electronic forms.

XForms is a language that is much more complex than HTML forms, but provides sophisticated facilities for constraining the input values for a form. The slider in Figure 5.7 was produced using XForms and Figure 5.12 shows the XForms code behind that web page.

Much of this code should be familiar; a lot of it is standard HTML elements. The differences are the use of the `range` element (lines 26 to 29), which generates the slider on screen, and the `model` element (lines 10 to 17), which specifies how the form data is structured and what type of input each form component can enter (in this case, the slider records a decimal value).

XForms is a very powerful language that allows for very precise control over the data that is recorded by an electronic form, however, it quickly becomes quite complex.

Another electronic form technology that is being developed is **Web Forms 2**. This is a less radical departure from HTML forms and mostly just extends the standard with new elements to allow for other input components (e.g., sliders, dates, ...) and new attributes to allow more constraints to be applied to the input data (e.g., ranges of values).

Figure 5.13 shows Web Forms 2 code for a slider like that in Figure 5.7 and the resulting web page is shown in Figure 5.14. The Web Forms 2 code is much more like regular HTML code, with the slider generated by an `input` element with `type="range"` (line 18).

The major problem with these newer technologies is that there is less software available to support them. A number of programs implement XForms, including an add-on for the Firefox browser, and the Opera web browser has started an implementation of Web Forms 2, but these programs are either incomplete or not widely available.

The most recent development is the start of work on HTML 5 by the W3C, which is intended to incorporate many of the Web Forms 2 ideas (among other things) into the official HTML specification. At the time of writing, however, this work is at a very early stage.

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:xf="http://www.w3.org/2002/xforms"
      xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <head>
    <style>
        body { background-color: #9CF2CA;
               padding-top: 20px;
               padding-left: 5%; padding-right: 5%; }
    </style>
    <xf:model>
      <xf:instance>
        <root xmlns="">
          <role>4</role>
        </root>
      </xf:instance>
      <xf:bind nodeset="role" type="xs:decimal"/>
    </xf:model>
  </head>
  <body>
      <p>
      How would you describe your role in society?
      </p>

      <p>
      career criminal
      <xf:range ref="role"
                start="1" end="7" step=".5">
          <xf:label />
      </xf:range>
      model citizen
      </p>
  </body>
</html>
```

Figure 5.12: XHTML and XForms code for a slider form component.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <title>Web Forms 2 Slider</title>
    <style>
        body { background-color: #9CF2CA;
               padding-top: 20px;
               padding-left: 5%; padding-right: 5%; }
    </style>
  </head>
  <body>
      <p>
      How would you describe your role in society?
      </p>

      <p>
      career criminal
          <input type="range" value="4" min="1" max="7" step=".5">
      model citizen
      </p>
  </body>
</html>
```

Figure 5.13: HTML and Web Forms 2 code for a slider form component.

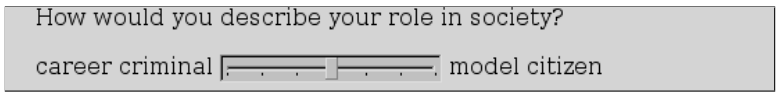


Figure 5.14: A slider form component described by Web Forms 2 code (see Figure 5.13) and viewed in the Opera web browser on Debian Linux.

5.5 Submitting input

To this point, we have only addressed how to create an electronic form and how to check the data that is entered into the form. We will now look at the final, very important, step of recording the information from the form.

5.5.1 HTML form submission

Most HTML forms have a submit button at the bottom of the form. Once all of the fields in the form have been filled in, the user clicks on the submit button to record the form data. The standard behaviour is to gather all of the data that were entered into the form and send it to another program to be processed.

The `form` element has two important attributes that together control what happens when the submit button is pressed. The first is the `action` attribute. This attribute specifies the program that the form data will be send to. It is usually a URL, which means that the program to process the data can reside anywhere on the world-wide web.

The `method` attribute of the `form` element controls *how* the information is sent. The value of this attribute is either `"get"` or `"post"`.

The code below shows the opening tag of the `form` element for the I-94W form.

```
<form action="http://www.formbuddy.com/cgi-bin/form.pl"
      method="post">
```

The data from the form is sent using the `post` method and the data is sent to a program called `form.pl`. This form uses a remotely-hosted form processing service called formbuddy.[5] A number of similar services are provided for free by various web site.[6]

5.5.2 Local HTML form submission

The problem with using `action` and `method` attributes is that the program specified by the `action` needs to be on a web server. Setting up web servers is beyond the scope of this book, so this section describes an alternative

[5] `http://www.formbuddy.com/`

[6] A list of services is maintained at
`http://cgi.resourceindex.com/Remotely_Hosted/Form_Processing/`

set up that will allow us to at least demonstrate the recording of form data using just a web browser.

The following steps can be used to set up a form so that the form data are recorded locally within the web browser.

1. Add a `script` element to the `head` element of the form to load the file `echo.js` containing JavaScript code.

   The `validate.js` code from Section 5.4 should also be loaded, so the `head` element of the HTML code should include the following lines:

   ```
   <script type="text/javascript" src="validate.js">
   </script>
   <script type="text/javascript" src="echo.js">
   </script>
   ```

   The file `echo.js` is available from the same location as the file `validate.js` (i.e., the web site for this book).

2. Add an `onsubmit` attribute to the `form` element that calls the `saveform()` function (from the `echo.js` file):

   ```
   <form onsubmit="return(saveform())">
   ```

   With this local submission approach, there is no need for an `action` attribute or a `method` attribute.

   The call to `saveForm()` *replaces* the call to the `validateAll()` function from `validate.js`, but the `saveform()` function calls `validateAll()` itself, so validation will still occur.

   Each time the form is submitted, the `saveform()` function will save the form data within the browser.

3. Add a new button at the bottom of the form using the following code:

   ```
   <input type="button" value="display"
          onclick="echoform()">
   ```

   This creates a new button, labelled display, in addition to the submit button that is already in the form.

   When the new display button is clicked, all of the form data that has been saved so far will be displayed at the bottom of the web form. This data can then be copied and pasted to a file for permanent storage.

Figure 5.15 shows what the I-94W form from Figure 5.10 looks like with these modifications (the only visible difference is the new display button at the bottom).

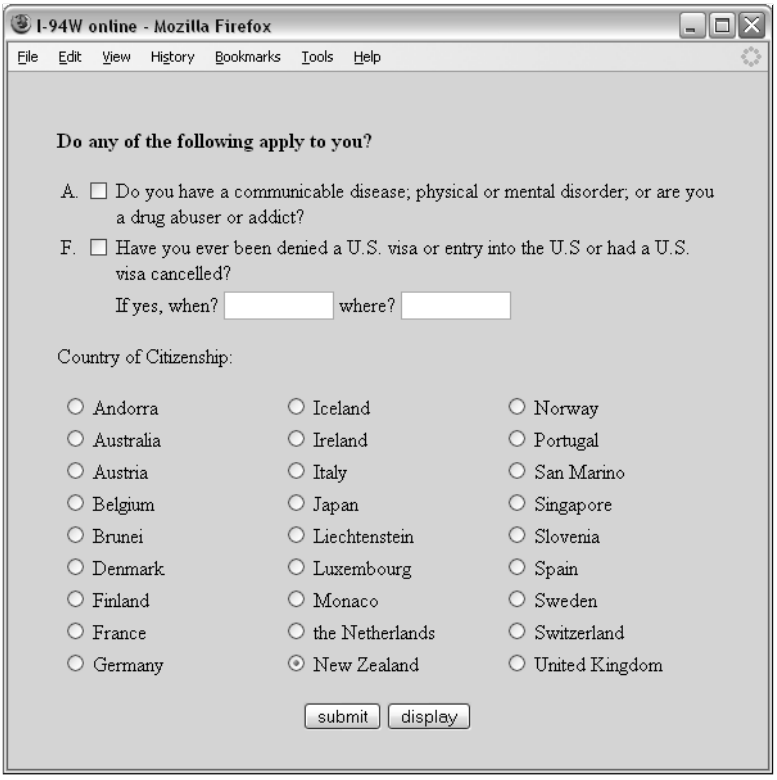


Figure 5.15: The form in Figure 5.10 set up for local submission.

Figure 5.16 shows the result of clicking on the display button after entering several sets of data.

Summary

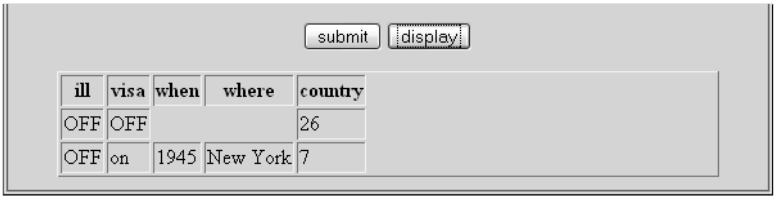


Figure 5.16: Displayed data from the I-94W form following a click on the display button.

6
HTML Forms Reference

HTML can be used to create electronic forms containing interactive elements, such as buttons and menus, for entering data. Figure 6.1 shows the typical form components available within an HTML form.

6.1 HTML form syntax

HTML form elements are just normal HTML elements, so all of the normal rules of HTML syntax apply (see Chapter 3).

All electronic form components must appear within a `form` element and the `form` element must appear within the `body` of the HTML code. Figure 6.2 shows the HTML code behind the electronic form in Figure 6.1.

6.2 HTML form semantics

6.2.1 Common attributes

All form elements that create form components have a `name` attribute and a `value` attribute. The `value` attribute specifies the data value that will be recorded from the form element. The `name` is important as a label for the data that is recorded from the form element; this corresponds to the name of a variable.

6.2.2 HTML form elements

`<input type="radio">`

Creates a radio button. Radio buttons are used where there is a single question with a fixed and finite set of possible answers and only one answer can be selected. Usually, one radio button is provided for each possible answer and only one of the radio buttons can be selected at once. In order to achieve this behaviour, all of the radio buttons

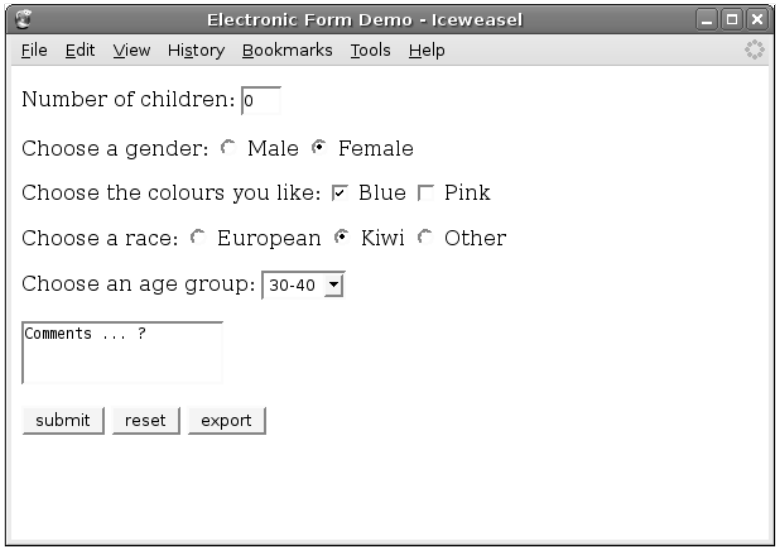


Figure 6.1: A demonstration of an HTML form, showing buttons, a menu, text fields, radio buttons, and check boxes.

for a single question *must* have the same `name` attribute. Conversely, each radio button for a single question *must* have a different `value` attribute.

An example of the use of radio buttons is shown on lines 18 and 19 of Figure 6.2 and in Figure 6.1.

`<input type="checkbox">`

Creates a check box. Each check box corresponds to a yes/no question, so each check box should have a unique `name` attribute. When a form is submitted, information is only sent for check boxes that have been selected, so the program processing the data must know about all check boxes in order to record a missing value or "off" value for check boxes that were not selected.

An example of the use of check boxes is shown on lines 23 and 24 of Figure 6.2 and in Figure 6.1.

`<input type="text">`

Creates a region for entering small amounts of text. A default value can be supplied via the `value` attribute. The maximum number of characters that can be entered can be controlled via the `maxlength` attribute, but otherwise the value entered by the user is unconstrained.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
    <title>Electronic Form Demo</title>
    <script type="text/ecmascript" src="validate.js"></script>
    <script type="text/ecmascript" src="echo.js"></script>
</head>
<body>
    <form onsubmit="return(saveform())">
    <p>
        Number of children:
        <input type="text" name="numChildren" value="0"
               size="2" maxlength="2"
               onblur="isIntegerRange(this, 0, 15)">
    </p>
    <p>
        Choose a gender:
        <input type="radio" name="gender" value="male" checked> Male
        <input type="radio" name="gender" value="female"> Female
    </p>
    <p>
        Choose the colours you like:
        <input type="checkbox" name="colourBlue" checked> Blue
        <input type="checkbox" name="colourPink"> Pink
    </p>
    <p>
        Choose a race:
        <input type="radio" name="race" value="euro"> European
        <input type="radio" name="race" value="kiwi" checked> Kiwi
        <input type="radio" name="race" value="other"> Other
    </p>
    <p>
        Choose an age group:
        <select name="agegp">
            <option>0-10</option>
            <option>10-20</option>
            <option>20-30</option>
            <option selected>30-40</option>
            <option>40-50</option>
            <option>50-60</option>
        </select>
    </p>
    <p>
        <textarea name="openText">Comments ... ?</textarea>
    </p>
    <p>
        <input type="submit" value="submit">
        <input type="reset" value="reset">
        <button onclick="echoform()">export</button>
    </p>
        <input type="hidden" id="echoFormFormat" value="html">
    </form>
</body>
</html>
```

Figure 6.2: HTML code for example form components.

See Section 6.4.

An example of the use of a text component is shown on lines 12 to 14 of Figure 6.2 and in Figure 6.1.

`<textarea>`
: Creates a region for entering large amounts of text. The size of the region that is displayed on the web page is controlled via `rows` and `cols` attributes. The values of these attributes are taken to be a number of lines of text and a number of characters of text, respectively. A default value can be supplied as the contents of the `textarea` element (white space is literal within this content).

  An example of the use of a text region is shown on line 44 of Figure 6.2 and in Figure 6.1.

`<input type="password">`
: Creates a region for entering a password; key strokes are echoed by printing dots or stars so that the actual value being entered is not visible on screen.

`<select>` and `<option>`
: The `select` element creates a selection menu. The options on the menu are created by `option` elements within the `select` element. The content of the `option` element is displayed on the menu. The data value that is recorded for each `option` element is also taken from the content of the `option` element *unless* the `option` element has a `value` attribute. The label for the data value comes from the `name` attribute of the `select` element (the `option` elements have no `name` attribute).

  An example of the use of a selection menu is shown on lines 34 to 41 of Figure 6.2 and in Figure 6.1.

`<input type="submit">`
: Creates a submit button. The label on the button can be controlled via the `value` attribute. Clicking this button will send the form data to a program for processing (see Section 6.3).

  An example of the use of a submit button is shown on line 47 of Figure 6.2 and in Figure 6.1.

`<input type="reset">`
: Creates a reset button. Clicking this button will reset the values of all controls to their default values.

  An example of the use of a reset button is shown on line 48 of Figure 6.2 and in Figure 6.1.

`<button>`
: Creates a button. This can be used as an alternative way to create a button for submitting the form, resetting the form, or, more generally, for associating a mouse click with a script action (see Section 6.4). The label on the button is taken from the contents of the `button` element (so this allows for fancier button labels, including images).

  An example of the use of a generic button is shown on line 49 of Figure 6.2 and in Figure 6.1.

`<label>`
: Associates a text label with a form component. Useful for non-graphical browsers and just as a discipline for documenting code. For radio buttons and check boxes this means that a click on the text will select the corresponding radio button or check box.

`<input type="hidden">`
: Generates an invisible form data value. This can be used to include information with the form data, but have nothing displayed on the web page.

  An example of the use of a hidden element is shown on line 51 of Figure 6.2. The purpose of this element is described in Section 6.4.2.

6.3 HTML form submission

In standard usage, HTML forms are a "client-server" technology. The "client" is a web browser, like Firefox, which is responsible for displaying the electronic form and allowing the user to enter data into the form. The "server" is a web server (e.g., Apache), and it is responsible for checking the data that was entered into the form and for storing the data somewhere (e.g., in a database). The web server is typically on a different computer from the browser, either on a local network, or somewhere on the world-wide-web.

The `action` attribute of the `form` element is used to specify the program to which the form data is sent. This can be just the name of a program (commonly, the name of a script in a language such as Perl or Python or PHP), in which case the program must reside in the same directory as the file of HTML code that describes the form, or it can be a full URL that provides a path to such a program (i.e., the HTML code and the processing code can reside on separate computers).

The `method` attribute of the `form` element is used to specify what format the form data are sent in. This attribute can have the value `"get"` or `"post"`.

For online surveys or data entry, `"post"` is probably the more appropriate option, though the `action` program may dictate which method to use.

The disadvantage of the standard usage of HTML forms is that it requires a separate computer and it requires web server software. The set up of a web server is beyond the scope of this book, but Section 6.4.2 describes one way to achieve input checking and data storage without the need for external programs (i.e., just with a browser).

The "local submission" described in Section 6.4.2 can be used for learning how to create HTML forms and for testing an HTML form, but for serious survey administration and data entry it will be necessary to set up a proper web server and data processing program. These services are provided by several online sites if the necessary expertise is not locally available.

6.4 HTML form scripts

It is possible to specify **script** code that should be run when the user interacts with the electronic form. The most common language used for scripts is JavaScript (officially ECMAScript, also known as JScript).

JavaScript code can be included as the content of a `script` element in the `head` of the HTML code, or the JavaScript can be in a separate file and just referred to via a `script` element.

This section does not provide a reference for the JavaScript language (see Section 6.5); it is only a guide to the use of some predefined scripts that are made available with this book.

The files containing JavaScript are called `validate.js` and `echo.js` and are available from the book web site.

6.4.1 Validation scripts

The file `validate.js` contains JavaScript code for performing simple checks on the values entered for text fields. These functions can be used by adding an `onblur` attribute to the element to be checked, so that the code is run whenever the user interacts with that element. Typically, these will only be necessary with `text` elements or `textarea` elements.

In all cases, these functions will open a dialog containing an error message if the check fails. The user must click on OK in the dialog in order to continue *and* the content of the offending element will be highlighted when control

returns to the form (i.e., no other data can be entered until the element has a valid value).

`notEmpty(this)`
: An error message will appear if the value is left blank. The code below shows an example of the use of this function to ensure that a name is entered into a text component:

```
Surname <text onblur="notEmpty(this)">
```

`isInteger(this)`
: The value must not be blank *and* the value must be a whole number. It is valid to enter a value that starts with a number, but the value will be truncated to just the number in that case.

`isIntegerRange(this, min, max)`
: The value must not be blank, the value must be a whole number, *and* the value must be between *min* and *max*.

`isReal(this)`
: The value must not be blank *and* the value must be a number (can include a decimal point). It is valid to enter a value that starts with a number, but the value will be truncated to just the number in that case.

`isRealRange(this, min, max)`
: The value must not be blank, the value must be a number, *and* the value must be between *min* and *max*.

`hasLength(this, n)`
: The value must be *n* characters long.

`hasIntegerLength(x, n)`
: The value must be a whole number *and* the value must have *n* digits.

`matchesPattern(this, pattern)`
: The value is checked against the regular expression in *pattern*. See Section 11.8.2 and Chapter 13 for information about regular expressions. The regular expressions in JavaScript correspond to Perl-style regular expressions.

6.4.2 Submission scripts

The file `validate.js` contains an JavaScript function called `validateAll()`. This script function can be used to check all form elements that have `onblur`

attributes just *before* the form is submitted. The effect is to force all validation checks to be performed when the form is submitted *and* submission will not proceed if any of the checks fail. This should be added as the `onsubmit` attribute of the `form` element as shown below:

```
<form onsubmit="return(validateAll())">
```

Local form submission

The file `echo.js` contains JavaScript functions for performing "local" submission of form data. This means that, when the submit button is clicked, the data is not sent to an external program for processing, but is saved within the browser instead.

In order to perform local submission, the `saveform()` function should be added to the `onsubmit` attribute of the `form` element as shown below:

```
<form onsubmit="return(saveform())">
```

In addition, an extra button should be added to the form using the code shown below (the `echoform()` function is provided within `echo.js`):

```
<button onclick="echoform()">display</button>
```

When this button is clicked, the data that has been entered so far will be displayed at the bottom of the web page. The data can then be copied and pasted elsewhere for permanent storage.

Navigation away from the form page (or closing the browser), without clicking the export button, will result in the loss of all of the submitted form data.

The `saveform()` function calls the `validateAll()` function from `validate.js` to perform validation checks *before* saving the form data within the browser.

Local form submission parameters

The functions in `echo.js` behave differently, depending on the value of certain parameters. For example, the format in which the form data is stored is controlled by a parameter called `echoFormFormat`. The parameters all have default values, but new values can be specified by including hidden elements in the HTML form (see Section 6.2.2).

An example is shown on line 51 of Figure 6.2 (reproduced below):

```
<input type="hidden" id="echoFormFormat" value="html">
```

This code sets the `echoFormFormat` parameter to the value `"html"`, which means that the form data are stored as an HTML table. This parameter can also take the value `"text"`, in which case the data are stored in a plain text format (see Section 7.3).

Other possible parameters and their values are:

`echoFormHeaders`
: This can take the values `true` (the default) or `false`. If it is `true` then variable names (taken from the `name` attributes of the form elements) are stored with the form data.

`echoFormFieldSep`
: If the format for saving data is plain text, this parameter controls what sort of character is placed between fields. The default is to use a comma.

`echoFormQuote`
: This can take the values `true` (the default) or `false`. If the format for saving data is plain text, and this parameter is `true`, then any data values containing the `echoFormFieldSep` are quoted (a double-quote character, `"`, is placed at either end of the data value). Any double-quote characters are also replaced with a double double-quote.

`echoFormReset`
: This can take the values `true` or `false` (the default). If it is true, then all form components are reset to their default values after the form is submitted.

The default values for these parameters mean that the default plain text format is a CSV format (see Section 7.3.5).

6.5 Further reading

The Exploring Online Research Methods Technical Guide
: `http://www.geog.le.ac.uk/orm/technical/techcontents.htm`
A comprehensive resource for creating electronic forms using HTML, CSS, and Javascript.

The W3C HTML 4.01 Specification
: `http://www.w3.org/TR/html401/`
The formal and official definition of HTML. Quite technical.

The Web Design Group's HTML 4 web site
`http://htmlhelp.com/reference/html40/`
A more friendly user-oriented description of HTML.

The w3schools HTML Tutorial
`http://www.w3schools.com/html/html_forms.asp`
The HTML forms section of the basic tutorial-based introduction to HTML.

The w3schools JavaScript Tutorial
`http://www.w3schools.com/js/`
A tutorial-based introduction to JavaScript.

The ECMAScript Language Specification
`http://www.ecma-international.org/publications/standards/Ecma-262.htm`
The ECMA-262 Standard document. Very technical.

7
Data Storage

The previous chapter focused on how to improve the process of recording information at the interface between humans and computers. In this chapter, we will look in depth at what form the information should take once it resides on a computer.

There are several good reasons why researchers need to know about data storage options. We may not have control over the format in which data is given to us. For example, data from NASA's Live Access Server is in a format decided by NASA and we are unlikely to be able to convince them to provide it in a different format. This says that we must know about different formats in order to gain access to data.

Another common situation is that We may have to transfer data between different applications or between different operating systems. This effectively involves temporary data storage so it is useful to understand how to select an appropriate storage format.

It is also possible to be involved in deciding the format for archiving a data set.

In this chapter, we will see a number of different data storage options and we will discuss the strengths and weaknesses of each.

7.1 Case study: YBC 7289

YBC 7289. Photo by Bill Casselman.[1]

Some of the earliest known examples of recorded information come from Mesopotamia, which roughly corresponds to modern-day Iraq, and date from around the middle of the fourth millenium BC. The writing is called *cuneiform*, which refers to the fact that marks were made in wet clay with a wedge-shaped stylus.

A particularly famous mathematical example of cuneiform is the clay tablet known as YBC 7289.

This tablet is inscribed with a set of numbers using the Babylonian sexagesimal (base-60) system. In this system, a symbol resembling a less-than sign (we will use `<`) represents the value 10 and a symbol resembling a tall narrow triangle, with the tip pointing down, represents the value 1 (we will use `|`). For example, the value 30 is written (roughly) like this: `<<<`. This value can be seen along the top-left edge of YBC 7289 (see Figure 7.1).

The number along the central horizontal line on YBC 7289 has four digits: `|`, which is 1; `<<||||`, which is 24; `<<<<<|`, which is 51; and `<`, which is 10. Historians have determined that there is an unwritten "decimal place" after the 1 and this means that the decimal value of this number is $1 + \frac{24}{60} + \frac{51}{3600} + \frac{10}{216000} = 1.41421296$ (to 9 significant decmial digits). For around 1600 BC, this value is ridiculously close to the true value of the length of the diagonal of a unit square ($\sqrt{2} = 1.41421356$).

The value at the bottom of the tablet has three digits—42, 25, and 35—and corresponds to the value $42 + \frac{25}{60} + \frac{35}{3600} = 30 \times \sqrt{2}$ (i.e., the length of the

[1] Source: Wikimedia Commons
`http://upload.wikimedia.org/wikipedia/commons/0/0b/Ybc7289-bw.jpg`
This image is available under a Creative Commons Attribution 2.5 licence.

Figure 7.1: Clay tablet YBC 7289 showing cunieform inscriptions that demonstrate the derivation of the square root of 2.

diagonal for a square with sides of length 30).

What we are going to do with this remarkable piece of mathematical history is to treat it as information that needs to be stored electronically.

The choice of a clay tablet for recording the information on YBC 7289 was obviously a good one in terms of the durability of the storage medium. Very few electronic media today have an expected lifetime of several thousand years. However, electronic media do have many other advantages.

The most obvious advantage of an electronic medium is that it is very easy to make copies. The curators in charge of YBC 7289 would no doubt love to be able to make identical copies of such a precious artifact, but truly identical copies are only really possible for electronic information.

This leads us to the problem of how we produce an electronic record of the tablet YBC 7289. We will consider a number of possibilities in order to introduce some of the issues that will be important when discussing various data storage alternatives throughout this chapter.

A straightforward approach to storing the information on this tablet would be to write a simple textual description of the tablet.

```
YBC 7289 is a clay tablet with various cuneiform marks on it
that describe the relationship between the length of the
diagonal and the length of the sides of a square.
```

This approach has the advantages that it is easy to create and it is easy for a human to access the information. However, when we store electronic information, we should also be concerned about whether the information is easily accessible for computer software. This essentially means that we should supply clear labels so that individual pieces of information can be retrieved easily. For example, the label of the tablet is something that might be used to identify this tablet from all other cuneiform artifacts, so the label information should be clearly identified.

```
label: YBC 7289
description: A clay tablet with various cuneiform marks on it
that describe the relationship between the length of the
diagonal and the length of the sides of a square.
```

Thinking about what sorts of questions will be asked of the data is a good way to guide the design of data storage. Another sort of information that people might go looking for is the set of cuneiform markings that occur on the tablet.

The markings on the tablet are numbers, but they are also symbols, so it would probably be best to record both numeric and textual representations. There are three sets of markings, and three values to record for each set; a common way to record this sort of information is with a row of information per set of markings, with three columns of values on each row.

```
   <<<                    30           30
     | <<|||| <<<<<| <     1 24 51 10    2.41421296
<<<<|| <<||||| <<<|||||    42 25 35      42.4263889
```

When storing the lines of symbols and numbers, we have spaced out the information so that it is easy, for a human, to see where one sort of value ends and another begins. Again, this information is even more important for the computer. Another option is to use a special character, such as a comma, to indicate the start/end of separate values.

```
values:
cuneiform,sexagesimal,decimal
<<<,30,30
| <<|||| <<<<<| <,1 24 51 10,2.41421296
<<<<|| <<||||| <<<|||||,42 25 35,42.4263889
```

Something else we should add is information about how the values relate to each other. Someone who is unfamiliar with Babylonian history may have difficulty realising how the three values on each line actually correspond to each other. This sort of encoding information is essential **metadata**—information about the data values.

```
encoding: In cuneiform, a '<' stands for 30 and
a '|' stands for 1.  Sexagesimal values are base 60, with
a sexagesimal point after the first digit;  the first digit
represents ones, the second digit is sixtieths, the third
is three-thousand six-hundredths, and the fourth is two
hundred and sixteen thousandths.
```

The position of the markings on the tablet, and the fact that there is also a square, with its diagonals inscribed, are all important information that contribute to a full understanding of the tablet. The best way to capture this information is with a photograph.

In many fields, data consist not just of numbers, but also pictures, sounds, and video. This sort of information creates additional files that are not easily incorporated together with textual or numerical data. The problem becomes not only how to store each individual representation of the information, but also how to *organise* the information in a sensible way. Something that we could do in this case in include a pointer to a file containing a photograph of the tablet.

```
photo: ybc7289.png
```

Information about the source of the data may also be of interest. For example, the tablet has been dated to sometime between 1800 BC and 1600 BC. Little is known of its rediscovery, except that it was acquired in 1912 AD by an agent of J. P. Morgan, who subsequntly bequeathed it to Yale University. This sort of **metadata** is easy to record as a textual description.

```
medium: clay tablet
history: Created between 1800 BC and 1600 BC, purchased by
J.P. Morgan 1912, bequeathed to Yale University.
```

The YBC in the tablet's label stands for the Yale Babylonian Collection. This tablet is just one item within one of the largest collections of cuneiforms in the world. In other words, there are a lot of other sources of data very similar to this one.

This has several implications for how we should store information about YBC 7298. First of all, we should store the same sort of information as is stored for other tablets in the collection so that, for example, a researcher can search for all tablets created in a certain time period. We should also think about the fact that some of the information that we have stored for YBC 7289 is very likely to be in common with all items in the collection. For example, the explanation of the sexagesimal system will be the same for other tablets from the same era. With this in mind, it does not make sense to record the encoding information for every single tablet. It would make sense to record the encoding information once, perhaps in a separate file, and just refer to the appropriate encoding information within the record for an individual tablet.

A complete version of the information that we have recorded so far might look like this:

```
label: YBC 7289
description: A clay tablet with various cuneiform marks on it
that describe the relationship between the length of the
diagonal and the length of the sides of a square.
photo: ybc7289.png
medium: clay tablet
history: Created between 1800 BC and 1600 BC, purchased by
J.P. Morgan 1912, bequeathed to Yale University.
encoding:  sexagesimal.txt
values:
cuneiform,sexagesimal,decimal
<<<,30,30
| <<|||| <<<<<| <,1 24 51 10,2.41421296
<<<<|| <<||||| <<<|||||,42 25 35,42.4263889
```

Is this the best possible way to store information about YBC 7289? Almost certainly not. Some problems with this approach include the fact that storing information as text is often not the most efficient approach and the fact that it would be difficult and slow for a computer to extract individual pieces of information from a free-form text format like this. However, the choice of an appropriate format also depends on how the data will be used.

The options discussed so far have only considered a couple of the possible *text* representations of the data. Another whole set of options to consider

are binary formats, for example, the photograph and the text and numeric information could all be included in a single file. The most likely solution in practice is that this information resides in a database of information that describes the entire Yale Babylonian Collection.

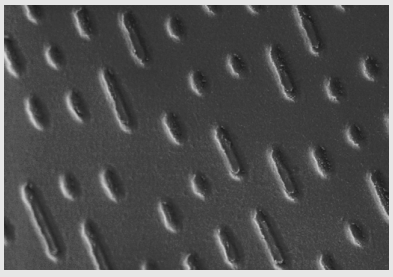
This chapter will look at the decisions involved in choosing a format for storing information, we will discuss a number of standard data storage formats, and we will acquire the technical knowledge to be able to work with the different formats.

7.2 Computer Memory

Given that we are always going to store our data on a computer, it makes sense for us to first find out a little bit about how that information is stored. How does a computer store the letter 'A' on a hard drive? What about the value $\frac{1}{3}$?

Knowledge of how information is stored digitally will allow us to reason about the amount of space that will be required to store a data set, which in turn will allow us to determine what software or hardware we will need to be able to work with a data set, and to decide upon an appropriate storage format. We will also look at some important *limitations* on how well information can be stored on a computer.

7.2.1 Bits, bytes, and words

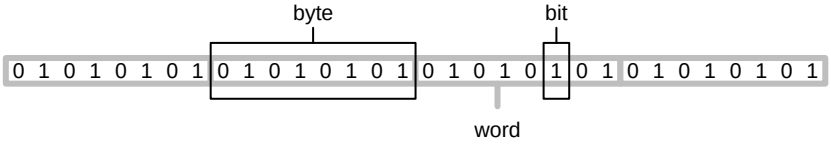


The surface of a CD magnified many times to show the pits in the surface that encode information.[2]

The most fundamental unit of computer memory is the **bit**. A bit can be

[2]Source: The University of Warwick, Department of Physics, Centre for Advanced Materials, Image Gallery
`http://physweb.spec.warwick.ac.uk/advmat/Imagegallery/CD1.htm`
Used and redistributed with permission.

a tiny magnetic region on a hard disk, a tiny dent in the reflective material on a CD or DVD, or a tiny transistor on a memory stick. Whatever the physical implementation, the important thing to know about a bit is that, like a switch, it can only take one of two values: it is either "on" or "off".



A collection of 8 bits is called a **byte** and (on the majority of computers today) a collection of 4 bytes, or 32 bits, is called a **word**. Each individual data value in a data set is usually stored using one or more bytes of memory, but at the lowest level, any data stored on a computer is just a large collection of bits. For example, the first 256 bits (32 bytes) of the electronic format of this book are shown below. At the lowest level, a data set is just a series of zeroes and ones like this.

```
00100101 01010000 01000100 01000110 00101101 00110001
00101110 00110100 00001010 00110101 00100000 00110000
00100000 01101111 01100010 01101010 00001010 00111100
00111100 00100000 00101111 01010011 00100000 00101111
01000111 01101111 01010100 01101111 00100000 00101111
01000100 00100000
```

The number of bytes and words used for an individual data value will vary depending on the storage format, the operating system, and even the computer hardware, but in many cases, a single letter or character of text takes up one byte and an integer, or whole number, takes up one word. A real or decimal number takes up one or two words depending on how it is stored.

For example, the text "hello" would take up 5 bytes of storage, one per character. The text "12345" would also require 5 bytes. The integer 12,345 would take up 4 bytes (1 word), as would the integers 1 and 12,345,678. The real number 123.45 would take up 4 or 8 bytes, as would the values 0.00012345 and 12345000.0.

7.2.2 Binary, Octal, and Hexadecimal

A piece of computer memory can be represented by a series of 0's and 1's, with one digit for each bit of memory; the value 1 represents an "on" bit

and a 0 represents an "off" bit. This notation is described as **binary** form. For example, below is a single byte of memory that contains the letter 'A' (ASCII code 65; binary 1000001).

```
01000001
```

A single word of memory contains 32 bits, so it requires binary 32 digits to represent a word in binary form. A more convenient notation is **octal**, where each digit represents a value from 0 to 7. Each octal digit is the equivalent of 3 binary digits, so a byte of memory can be represented using only 3 octal digits.

Binary values are pretty easy to spot, but octal values are much harder to distinguish from normal decimal values, so when writing octal values, it is common to precede the digits by a special character, such as a leading '0'.

As an example of octal form, the binary code for the character 'A' splits into triplets of binary digits (from the right) like this: `01 000 001`. So the octal digits are `101`, commonly written `0101` to emphasize the fact that these are octal digits.

An even more efficient way to represent memory is **hexadecimal** form. Here, each digit represents a value between 0 and 16, with values greater than 9 replaced with the characters `a` to `f`. A single hexadecimal digit corresponds to 4 bits, so each byte of memory requires only 2 hexadecimal digits. As with octal, it is common to precede hexdecimal digits with a special character, e.g., `0x` or `#`. The binary form for the character 'A' splits into two quadruplets: `0100 0001`. The hexadecimal digits are `41`, commonly written `0x41` or `#41`.

Another standard practice is to write hexadecimal representations as pairs of digits, corresponding to a single byte, separated by spaces. For example, the memory storage for the text "just testing" (12 bytes) could be represented as follows:

```
6a 75 73 74 20 74 65 73 74 69 6e 67
```

When displaying a block of computer memory, another standard practice is to present three columns of information: the left column presents an **offset**, a number indicating which byte is shown first on the row; the middle column shows the actual memory contents, typically in hexadecimal form; and the right column shows an interpretation of the memory contents (either characters, or numeric values). For example, the test "just testing" is shown below complete with offset and character display columns.

```
0  :  6a 75 73 74 20 74 65 73 74 69 6e 67  |  just testing
```

We will use this format for displaying raw blocks of memory throughout this section.

7.2.3 Numbers

Recall that the most basic unit of memory, the bit, has two possible states, "on" or "off". If we used one bit to store a number, we could use each different state to represent a different number. For example, a bit could be used to represent the numbers 0, when the bit is off, and 1, when the bit is on.

We will need to store numbers much larger than 1—and to do that we need more bits.

If we use two bits together to store a number, each bit has two possible states, so there are four possible combined states: both bits off, first bit off and second bit on, first bit on and second bit off, or both bits on. Again using each state to represent a different number, we could store four numbers using two bits: 0, 1, 2, and 3.

In general, if we use k bits, each bit has two possible states, and the bits combined can represent 2^k possible states, so with k bits, we could represent the numbers 0, 1, 2 up to $2^k - 1$.

Integers

Integers are commonly stored using a word of memory, which is 4 bytes or 32 bits, so integers from 0 up to 4,294,967,295 ($2^{32} - 1$) can be stored. Below are the integers 1 to 5 stored as four-byte values (each row represents one integer).

```
 0  :  00000001 00000000 00000000 00000000  |  1
 4  :  00000010 00000000 00000000 00000000  |  2
 8  :  00000011 00000000 00000000 00000000  |  3
12  :  00000100 00000000 00000000 00000000  |  4
16  :  00000101 00000000 00000000 00000000  |  5
```

This may look a little strange; within each byte (each block of eight bits), the bits are written from right to left like we are used to in normal decimal notation, but the bytes themselves are written left to right! It turns out

that the computer does not mind which order the bytes are used (as long as we tell the computer what the order is) and most software uses this left to right order for bytes.[3]

Two problems should immediately be apparent: using all available bits for positive integer values does not allow for any negative integer values; and very large integers, 2^{32} or greater, cannot be stored in a word of memory.

In practice, the first problem is solved by sacrificing one bit to indicate whether the number is positive or negative, so the range becomes -2,147,483,647 to 2,147,483,647 ($\pm 2^{31} - 1$).

The second problem, that we cannot store very large integers, is an inherent limit to storing information on a computer (in finite memory) and is worth remembering when working with very large values. Solutions include: using more memory to store integers, e.g., two words per integer, which uses up more memory, so is less memory-efficient; storing integers as real numbers, which can introduce inaccuracies (see below); or using arbitrary precision arithmetic, which uses as much memory per integer as is needed, but makes calculations with the values slower.

Depending on the computer language, it may also be possible to specify that only positive (unsigned) integers are required (i.e., reclaim the sign bit), in order to gain a greater upper limit. Conversely, if only very small integer values are needed, it may be possible to use a smaller number of bytes or even to work with only a couple of bits (less than a byte).

Real numbers

Real numbers (and rationals) are much harder to store digitally than integers.

Recall that k bits can represent 2^k different states. For integers, the first state can represent 0, the second state can represent 1, the third state can represent 2, and so on. We can only go as high as the integer $2^k - 1$, but at least we know that we can account for all of the integers up to that point.

Unfortunately, we cannot do the same thing for reals. We could say that the first state represents 0, but what does the second state represent? 0.1? 0.01? 0.00000001? Suppose we chose 0.01, so the first state represents 0, the second state represents 0.01, the third state represents 0.02, and so on. We can now only go as high as $0.01 \times (2^k - 1)$, *and* we have missed all of

[3]The order of the bytes is called the **endianness**; left to right is **little endian**, because the least significant byte, the byte representing the smallest part of the number, comes first. Right-to-left ordering is called **big endian**.

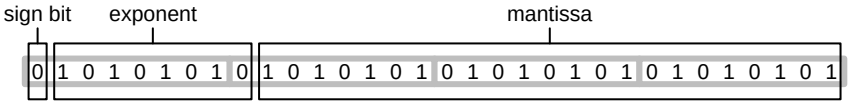
the numbers between 0.01 and 0.02 (and all of the numbers between 0.02 and 0.03, and infinitely many others).

This is another important limitation of storing information on a computer: there is a limit to the **precision** that we can achieve when we store real numbers. Most real values cannot be stored exactly on a computer. Examples of this problem include not only exotic values such as transcendental numbers (e.g., π and e), but also very simple everyday values such as $\frac{1}{3}$ or even 0.1. This is not as dreadful as it sounds, because even if the exact value cannot be stored, a value very very close to the true value can be stored. For example, if we use eight bytes to store a real number then we can store the distance of the earth from the sun to the nearest millimetre.

The limitation on numerical accuracy rarely has an effect on stored values because it is very hard to obtain a scientific measurement with this level of precision. However, when performing many calculations, even tiny errors in stored values can accumulate and result in significant problems. We will revisit this issue in Chapter 11. Solutions to storing real values with full precision include: using even more memory per value, especially in working, (e.g., 80 bits instead of 64) and using arbitrary-precision arithmetic.

We will now look at the details of how real numbers are stored digitally. What follows is some quite technical information. It is not important to memorize this information, but a basic understanding of the approach is useful to have.

A real number is stored as a **floating-point** number, which means that it is stored as two values: a **mantissa**, m, and an **exponent**, e, in the form $m \times 2^e$. When a single word is used to store a real number, a typical arrangement[4] uses 8 bits for the exponent and 23 bits for the mantissa (plus 1 bit to indicate the sign of the number).



The exponent mostly dictates the range of possible values. Eleven bits allows for a range of integers from -127 to 127, which means that it is possible to store numbers as small as 10^{-39} (2^{-127}) and as large as 10^{38} (2^{127}).[5]

[4] IEEE 754 standard for single-precision floating-point values.

[5] The limits in practice are a little narrower than this because of implementation details such as the need to be able to code special values like $\pm\infty$.

The mantissa dictates the precision with which values can be represented. The issue here is not the magnitude of a value (whether it is very large of very small), but the amount of precision that can be represented. With 23 bits, it is possible to represent 2^{23} different real values, which is a lot of values, but still leaves a lot of gaps. For example, if we are dealing with values in the range 0 to 1, we can take steps of $\frac{1}{2^{23}} \approx 0.0000001$, which means that we cannot represent any of the values between 0.0000001 and 0.0000002. In other words, we cannot distinguish between numbers that differ from each other by less than 0.0000001. If we deal with values in the range 0 to $10,000,000$, we can only take steps of $\frac{10,000,000}{2^{23}} \approx 1$, so we cannot distinguish between values that differ from each other by less than 1.

Below are the real values 1.0 to 5.0 stored as four-byte values (each row represents one real value). Remember that the bytes are ordered from left to right so the most important byte (containing the sign bit and most of the exponent) is the one on the right. The first bit of the byte second from the right is the last bit of the mantissa.

```
 0  :  00000000 00000000 10000000 00111111  |  1
 4  :  00000000 00000000 00000000 01000000  |  2
 8  :  00000000 00000000 01000000 01000000  |  3
12  :  00000000 00000000 10000000 01000000  |  4
16  :  00000000 00000000 10100000 01000000  |  5
```

For example, the exponent for the first value is `0111111 1`, which is 127. These exponents are "biased" by 127 so to get the final exponent we subtract 127 to get 0. The mantissa has an implicit value of 1 plus, for bit i, the value 2^{-i}. In this case, the entire mantissa is zero, so the mantissa is just the (implicit) value 1. The final value is $2^0 \times 1 = 1$.

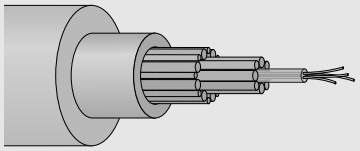
For the last value, the exponent is `1000000 1`, which is 129, less 127 is 2. The mantissa is `01` followed by 21 zeroes, which represents a value of (implicit) $1 + 2^{-2} = 1.25$. The final value is $2^2 \times 1.25 = 5$.

When real numbers are stored using two words instead of one, the range of possible values and the precision of stored values increases enormously, but there are still limits.

```
1156748010.47817 60
1156748010.47865 1254
1156748010.47878 1514
1156748010.4789 1494
1156748010.47892 114
1156748010.47891 1514
1156748010.47903 1394
1156748010.47903 1514
1156748010.47905 60
1156748010.47929 60
...
```

Figure 7.2: Several lines of network packet data as a plain text file. The number on the left is the number of seconds since January 1st 1970 and the number on the right is the size of the packet (in bytes).

7.2.4 Case study: Network traffic



The Southern Cross Cable provides the main internet connection between New Zealand, Australia, and the USA. Some parts of the cable lie over 7km beneath the ocean's surface.[6]

The central IT department of the University of Auckland has been collecting network traffic data since 1970. Measurements have been made on each packet of information that passed through a certain location on the network. These measurements include the time at which the packet reached the network location and the size of the packet.

The time measurements are the time elapsed, in seconds, since January 1st 1970 and the measurements are extremely accurate, being recorded to the nearest 10,000th of a second. Over time, this has resulted in numbers that are both very large (there are 31,536,000 seconds in a year) and very precise. Figure 7.2 shows several lines of the data stored as plain text.

[6]Image source: Wikimedia Commons
http://commons.wikimedia.org/wiki/Image:Southern_Cross_Cable_cross_section.svg
This image is in the public domain.

By the middle of 2007, the measurements were approaching the limits of precision for floating point values.

The data were analysed in a system that used 8 bytes per floating point number (i.e., 64-bit floating-point values). The IEEE standard for 64-bit or "double-precision" floating-point values uses 52 bits for the mantissa. This allows for approximately 2^{52} different real values.[7] In the range 0 to 1, this allows for values that differ by as little as $\frac{1}{2^{52}} \approx 0.0000000000000002$, but when the numbers are very large, for example on the order of 1,000,000,000, it is only possible to store values that differ by $1,000,000,000 \times \frac{1}{2^{52}} \approx 0.0000002$. In other words, double-precision floating-point values can be stored with up to only 16 significant digits.

The time measurements for the network packets differ by as little as 0.00001 seconds. Put another way, the measurements have 15 significant digits, which means that it is possible to store them with full precision as 64-bit floating-point values, but only just.

Furthermore, with values so close to the limits of precision, arithmetic performed on these values can become inaccurate. This story is taken up again in Section 11.5.7.

7.2.5 Text

Text is stored on a computer by first converting each character to an integer and then storing the integer. For example, to store the letter 'A', we will actually store the number 65; 'B' is 66, 'C' is 67, and so on.

A letter is usually stored using a single byte (8 bits). Each letter is assigned an integer number and that number is stored. For example, the letter 'A' is the number 65, which looks like this in binary format: `01000001`. The text "hello" (104, 101, 108, 108, 111) would look like this: `01101000 01100101 01101100 01101100 01101111`

The conversion of letters to numbers is called an **encoding**. The encoding used in the examples above is called ASCII[8] and is great for storing (American) English text. Other languages require other encodings in order to allow non-English characters, such as 'ö'.

ASCII only uses 7 of the 8 bits in a byte, so a number of other encodings are just extensions of ASCII where any number of the form `0xxxxxxx` matches

[7] The exact calculations require taking into account the fact that the mantissa is encoded with an implicit leading one and that certain bit patterns are reserved for special values such as infinity.

[8] American Standard Code for Information Interchange.

the ASCII encoding and the numbers of the form `1xxxxxxx` specify different characters for a specific set of languages. Some common encodings of this form are the ISO 8859 family of encodings, such as ISO-8859-1 or Latin-1 for West European languages, and ISO-8859-2 or Latin-2 for East European languages.

Even using all 8 bits of a byte, it is only possible to encode 256 (2^8) different characters. Several Asian and middle-Eastern countries have written languages that use several thousand different characters (e.g., Japanese Kanji ideographs). In order to store text in these languages, it is necessary to use a **multi-byte** encoding scheme where more than one byte is used to store each character.

UNICODE is an attempt to provide an encoding for all of the characters in all of the languages of the World. Every character has its own number, often written in the form `U+xxxxxx`. For example, the letter 'A' is `U+000041`[9] and the letter 'ö' is `U+0000F6`. UNICODE encodes for many thousands of characters, so requires more than one byte to store each character. On Windows, UNICODE text will typically use two bytes per character; on Linux, the number of bytes will vary depending on which characters are stored (if the text is only ASCII it will only take one byte per character).

For example, the text "just testing" is shown below saved via Microsoft's Notepad in three different encodings: ASCII, UNICODE, and UTF-8.

```
0  :  6a 75 73 74 20 74 65 73 74 69 6e 67  |  just testing
```

The ASCII format contains exactly one byte per character. The fourth byte, with the value `74`, is hexadecimal code for the decimal value 116, which is the ASCII code for the letter 't'. We can see this byte pattern several more times, whereever there is a 't' in the text.

```
 0  :  ff fe 6a 00 75 00 73 00 74 00 20 00  |  ..j.u.s.t. .
12  :  74 00 65 00 73 00 74 00 69 00 6e 00  |  t.e.s.t.i.n.
24  :  67 00                                |  g.
```

The UNICODE format differs from the ASCII format in two ways. For every byte in the ASCII file, there are now two bytes, one containing the binary code we saw before followed by a byte containing all zeroes. There are also two additional bytes at the start. These are called a byte order mark (**BOM**) and indicate the order (endianness) of the two bytes that make up each letter in the text.

[9] The numbers are written in hexadecimal (base 16) format; the decimal number 65 is 41 in hexadecimal.

```
 0  :  ef bb bf 6a 75 73 74 20 74 65 73 74  |  ...just test
12  :  69 6e 67                             |  ing
```

The UTF-8 format is mostly the same as the ASCII format; each letter has only one byte, with the same binary code as before because these are all common english letters. The difference is that there are three bytes at the start to act as a BOM.[10]

7.2.6 Data with units or labels

When storing values with a known range, it can be useful to take advantage of that knowledge. For example, suppose we want to store information on gender. There are (usually) only two possible values: `male` and `female`. One way to store this information would be as text: "male" and "female". However, that approach would take up at least 4 to 6 bytes per observation. We could do better by storing the information as an integer, with 1 representing male and 2 representing female, thereby only using as little as one byte per observation. We could do even better by using just a single bit per observation, with "on" representing male and "off" representing female.

On the other hand, storing "male" is much less likely to lead to confusion than storing 1 or by setting a bit to "on"; it is much easier to remember or intuit that "male" corresponds to `male`. This leads us to an ideal solution where only a number is stored, but the encoding relating "male" to 1 is also stored.

Dates

Dates are commonly stored as either text, such as `Feb 1 2006`, or as a number, for example, the number of days since 1970. A number of complications arise due to a variety of factors:

language and cultural one problem with storing dates as text is that the format can differ between different countries. For example, the second month of the year is called February in English-speaking countries, but something else in other countries. A more subtle and dangerous problem arises when dates are written in formats like this: `01/03/06`. In some countries, that is the first of March 2006, but in other countries it is the third of January 2006.

[10] Notepad writes a BOM at the start of UTF-8 files, but not all software does this.

time zones Dates (a particular day) are usually distinguished from **datetimes**, which specify not only a particular day, but also the hour, second, and even fractions of a second within that day. Datetimes are more complicated to work with because they depend on location; midday on the first of March 2006 happens at different times for different countries (in different time zones). Daylight saving just makes things worse.

changing calendars The current international standard for expressing the date is the Gregorian Calendar. Issues can arise because events may be recorded using a different calendar (e.g., the Islamic calendar or the Chinese calendar) or events may have occurred prior to the existence of the Gregorian (pre sixteenth century).

The important point is that we need to think about how we store dates, how much accuracy we should retain, and we must ensure that we store dates in an unambiguous way (for example, including a time zone or a locale). We will return to this issue later when we discuss the merits of different standard storage formats.

Money

There are two major issues with storing monetary values. The first is that the currency should be recorded; NZ$1.00 is very different from US$1.00. This issue applies of course to any value with a unit, such as temperature, weight, distances, etc.

The second issue with storing monetary values is that values need to be recorded exactly. Typically, we want to keep values to exactly two decimal places at all times. This is sometimes solved by using **fixed-point** representations of numbers rather than floating-point; the problems of lack of precision do not disappear, but they become predictable so that they can be dealt with in a rational fashion (e.g., rounding schemes).

In practice, most data values are recorded as numbers, which means that metadata containing the units or meaning of the numbers is essential.

7.3 Plain text files

The previous section looked at how individual data values are recorded in a digital format. We now turn our attention to how entire data sets are stored, starting with a discussion of the different varieties of **file formats**.

```
             VARIABLE : Mean TS from clear sky composite (kelvin)
             FILENAME : ISCCPMonthly_avg.nc
             FILEPATH : /usr/local/fer_data/data/
             SUBSET   : 48 points (TIME)
             LONGITUDE: 123.8W(-123.8)
             LATITUDE : 48.8S
                        123.8W
                         23
 16-JAN-1994 00 /  1:  278.9
 16-FEB-1994 00 /  2:  280.0
 16-MAR-1994 00 /  3:  278.9
 16-APR-1994 00 /  4:  278.9
 16-MAY-1994 00 /  5:  277.8
 16-JUN-1994 00 /  6:  276.1
 ...
```

Figure 7.3: The first few lines of the plain text output from the Live Access Server for the surface temperature at Point Nemo. This is a reproduction of Figure 1.2.

The simplest way to store information is in a plain text file. In this format, everything, including numeric values, is stored as a series of characters.

7.3.1 Case study: Point Nemo (continued)

A good example of this sort of plain text data is the surface temperature data for the Pacific Pole of Inaccessibility (see Section 1.1; Figure 7.3 reproduces Figure 1.2 for convenience).

Plain text files can be thought of as the lowest common denominator of storage formats; they might not be the most efficient or sophisticated solution, but we can be fairly certain that they will get the job done.

7.3.2 Flat files

A plain text file containing a data set may be referred to as a **flat file**. The basic characteristics of a flat file are that the data are stored as plain text, even the numbers are plain text, and that each line of the file contains one record or case in the data set.

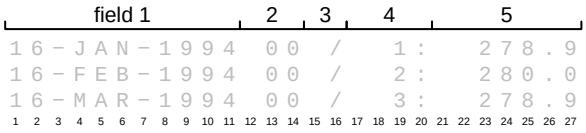
There may be a **header** at the start of the file containing general information, such as names of varaibles. In Figure 7.3, the first 8 lines are header information.

For each line in a flat file (each case in the data set), there are usually several **fields** containing the values for the different variables in the data set. There are two main approaches to differentiating fields within each line of the file:

Delimited format: Fields within a record are separated by a special character, or **delimiter**. For example, it is possible to view the file in Figure 7.3 as a delimited format, where each line after the header consists of two fields separated by a colon (the character ':' is the delimiter). Alternatively, if we used "white space" (one or more spaces or tabs) as the delimiter, there would be five fields, as shown below.

Using a delimited format can cause problems if it is possible for one of the values in the data set to contain the delimiter. The typical solution to this problem is to allow **quoting** of values or **escape sequences**. A fairly complete delimited format that has rules to account for these problems is the **comma-separated values** (CSV) format, which is described in more detail in Section 7.3.5.

Fixed-width format: Each field is allocated a fixed number of characters. For example, it is possible to view the file in Figure 7.3 as a fixed-width format, where the first field uses the first 20 characters and the second field uses the next 8 characters. Alternatively, there are five fields using 11, 3, 2, 4, and 8 characters respectively.



With a fixed width format we can have any character as part of the value of a field, but enforcing a fixed length for fields can be a problem if we do not know the maximum possible length for all variables. Also, if the values for a variable can have very different lengths, a fixed-width format can be inefficient because we store lots of empty space for short values.

All text file formats have the advantage that text is easy for humans to read and it is easy to write software to read text, but fixed-width formats are exceptional in that they are especially easy for humans to read because of the regular arrangement of values.

7.3.3 Advantages of plain text

The main advantage of plain text formats is their simplicity: we do not require complex software to create or view a text file and we do not need esoteric skills beyond being able to type on a keyboard, which means that it is easy for people to view and modify the data.

Virtually all software packages can read and write text files and plain text files are portable across different computer platforms.

7.3.4 Disadvantages of plain text

The main disadvantage of plain text formats is their simplicity. The simple one-row-per-case and one-column-per-variable format can be very inefficient and inappropriate for data sets with any sort of complex structure.

Consider a data set collected on two families, as depicted in Figure 7.4. What would this look like as a flat file, with one row for all of the information about each person in the data set? One possible fixed-width format is shown below:

```
                John      33  male
                Julia     32  female
John    Julia   Jack       6  male
John    Julia   Jill       4  female
John    Julia   John jnr   2  male
                David     45  male
                Debbie    42  female
David   Debbie  Donald    16  male
David   Debbie  Dianne    12  female
```

This format for storing these data is not ideal for two reasons. Firstly, it is not efficient; the parent information is repeated over and over again. This repetition is also undesirable because it creates opportunities for errors and inconsistencies to creep in. Ideally, each individual piece of information would be stored exactly once; if more than one copy exists then it is possible for the copies to disagree. The DRY principle (Section 2.5) applies to data

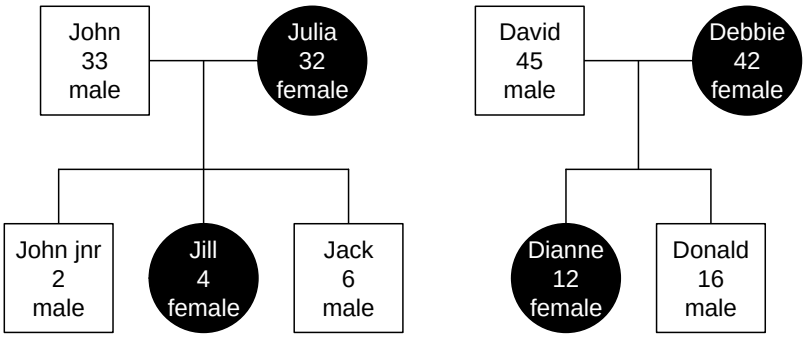


Figure 7.4: A family tree containing data on parents and children. An example of hierarchical data.

as well as code.

The other problem is not as obvious, but is arguably much more important. The fundamental structure of the flat file format means that each line of the file contains exactly one record or case in the data set. This works well when a data set only contains information about one type of object, or, put another way, when the data set itself has a flat structure.

The data set of family members does not have a flat structure. There is information about two different types of object, parents and children, and these objects have a definite relationship between them. We can say that the data set is **hierarchical** or **multi-level** or **stratified** (as is obvious from the view of the data in Figure 7.4). Any data set that is obtained using a non-trivial study design is likely to have a hierarchical structure like this.

In other words, a flat file format does not allow for sophisticated "data models" (see Section 7.7.6). A flat file is unable to provide an appropriate representation of a complex data structure.

The other major weakness of free-form text files is the lack of rules and standards. In the absence of a formally-declaration of the structure of a file, it is impossible for software to determine where the data values reside within the file. There is no standard way to specify the special character being used in delimited files, there is no way to specify the widths of fields (within the file itself) for fixed-width formats.

7.3.5 CSV files

Although not a formal standard, **comma-separated value** (CSV) files are very common and are a quite reliable plain text delimited format that at least solves the problem of a lack of rules and standards. This is a common way to export data from a spreadsheet.

The main rules for the CSV format are:

- Each field is separated by a comma (i.e., the character '`,`' is the delimiter).
- Fields containing commas must be surrounded by double quotes (i.e., the '`"`' character is special).
- Fields containing double quotes must be surrounded by double quotes *and* each embedded double quote must be represented using two double quotes (i.e., '`""`' is an escape sequence for a literal double quote).
- There can be a single header line containing the names of the fields.

7.3.6 Case Study: The Data Expo

The TIROS Operational Vertical Sounder (TOVS) instruments have been used to collect atmospheric data aboard National Oceanic and Atmospheric Administration (NOAA) satellites since 1978.[11]

The American Statistical Association (ASA) holds an annual conference called the Joint Statistical Meetings (JSM). One of the events sometimes held at this conference is a Data Exposition, where contestants are provided with a data set and must produce a poster demonstrating a comprehensive analysis of the data. For the Data Expo at the 2006 JSM the data were geographic and atmospheric measures obtained from NASA's Live Access Server (see Section 1.1).

The variables in the data set are: elevation, temperature (surface and air), ozone, air pressure, and cloud cover (low, mid, and high). With the excep-

[11] Image source: Wikimedia Commons
`http://commons.wikimedia.org/wiki/Image:Orbital_Planes.svg`
This image is in the public domain.

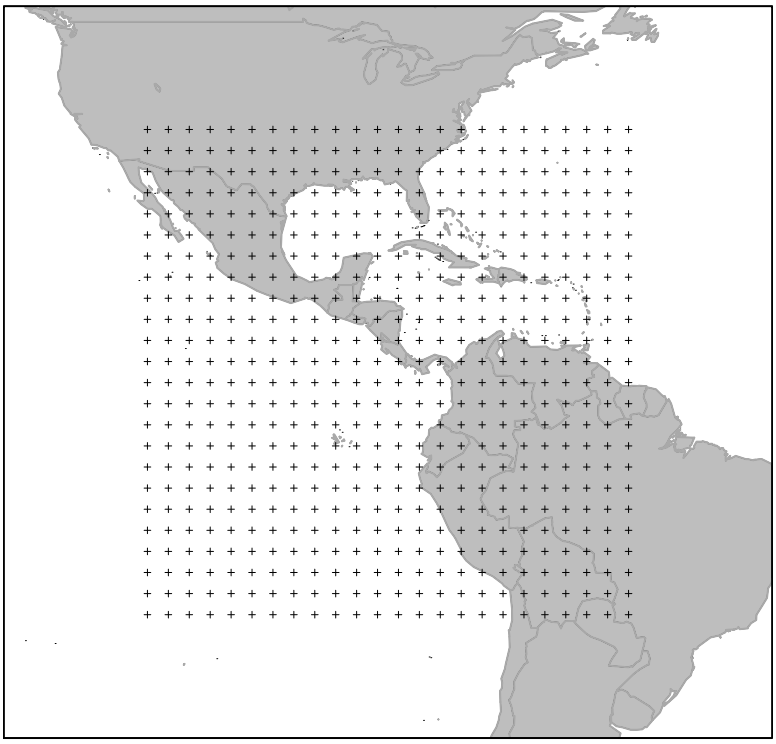


Figure 7.5: The geographic locations at which Live Access Server atmospheric data were obtained for the 2006 JSM Data Expo.

tion of elevation, all variables are monthly averages, with observations for January 1995 to December 2000. The data are measured at evenly-spaced geographic locations on a very coarse 24 by 24 grid covering Central America (see Figure 7.5).

The data were downloaded from the Live Access Server in a plain text format with one file for each variable, for each month; this produced 72 files per atmospheric variable, plus 1 file for elevation, for a total of 505 files. Figure 7.6 shows the start of one of the surface temperature files.

This data set demonstrates a number of advantages and limitations of a plain text format for storing data. First of all, the data is very straightforward to access because it does not need sophisticated software. It is also easy for a human to view the data and understand what is in each file.

```
             VARIABLE : Mean TS from clear sky composite (kelvin)
             FILENAME : ISCCPMonthly_avg.nc
             FILEPATH : /usr/local/fer_dsets/data/
             SUBSET   : 24 by 24 points (LONGITUDE-LATITUDE)
             TIME     : 16-JAN-1995 00:00
              113.8W 111.2W 108.8W 106.2W 103.8W 101.2W 98.8W  ...
               27     28     29     30     31     32     33    ...
36.2N / 51:  272.7  270.9  270.9  269.7  273.2  275.6  277.3  ...
33.8N / 50:  279.5  279.5  275.0  275.6  277.3  279.5  281.6  ...
31.2N / 49:  284.7  284.7  281.6  281.6  280.5  282.2  284.7  ...
28.8N / 48:  289.3  286.8  286.8  283.7  284.2  286.8  287.8  ...
26.2N / 47:  292.2  293.2  287.8  287.8  285.8  288.8  291.7  ...
23.8N / 46:  294.1  295.0  296.5  286.8  286.8  285.2  289.8  ...
...
```

Figure 7.6: The first few lines of output from the Live Access Server for the surface temperature of the Earth for January 1995, over a coarse 24 by 24 grid of locations covering central America.

However, the file format provides a classic demonstration of the typical lack of standardised structure in plain text files. For example, the raw data values only start on the eighth line of the file, but there is no indication of that fact within the file itself. This is not an issue for a human viewing the file, but a computer has no chance of detecting this structure automatically. Second, the raw data are arranged in a matrix, corresponding to a geographic grid of locations, but again there is no inhereht indication of this structure. For example, only a human can tell that the first 11 characters on each line of raw data are row labels describing latitude.

The "header" part of the file (the first seven lines) contains metadata, including information about what variable is recorded in the file and the units used for those measurements. This is very important and useful information, but again it is not obvious (for a computer) which bits are labels and which bits are information, let alone what sort of information is in each bit.

Finally, there is the fact that the data reside in 505 separate files. This is essentially an admission that plain text files are not suited to data sets with anything beyond a simple two-dimensional matrix-like structure. In this case, the temporal dimension—the fact that data are recorded at multiple time points—and the multivariate nature of the data—the fact that multiple variables are recorded—leads to there being separate files for each variable and for each time point. Having the data spread across many files creates issues in terms of the naming of files, for example, to ensure that all files from the same date, but containing different variables can be easily located.

There is also a reasonable amount of redundancy, with metadata and labels repeated many times over in different files.

7.4 XML

The **eXtensible Markup Language**, XML, is a language that is used for storing information. It is particularly appropriate for data that is to be shared between several different software systems.

Figure 7.7 shows the surface temperature data at the Pacific Pole of Inaccessibility (see Section 1.1) in two different formats: the original plain text and an XML format.

One fundamental similarity between these formats is that they are both just text. This is an important and beneficial property of XML; we can read it and manipulate it without any special skills or any specialized software.

There are many advantages to storing information in a plain text format, mostly related to the simplicity of plain text files. However, that same simplicity also creates problems because, for example, it is difficult to efficiently store complex data (see Section 7.3.3).

XML is a storage format that is still based on plain text, but does not suffer from many of the problems of plain text files because it adds flexibility, rigour, and standardization.

7.4.1 XML syntax

We will use the XML format for the Point Nemo temperature data (Figure 7.7) to demonstrate some of the basic rules of XML syntax.

The XML format of the data consists of two parts: XML **mark up** and the actual data itself. For example, the information about the latitude at which these data were recorded is stored with XML **tags**, `<latitude>` and `</latitude>`, surrounding the latitude value. The combination of tags and content are together described as an XML **element**.

```
<latitude>48.8S</latitude>
```

Each temperature measurement is contained within a `case` element, with the date and temperature data recorded as **attributes** of the element.

```
<?xml version="1.0"?>
<temperatures>
    <variable>Mean TS from clear sky composite (kelvin)</variable>
    <filename>ISCCPMonthly_avg.nc</filename>
    <filepath>/usr/local/fer_dsets/data/</filepath>
    <subset>93 points (TIME)</subset>
    <longitude>123.8W(-123.8)</longitude>
    <latitude>48.8S</latitude>
    <case date="16-JAN-1994" temperature="278.9" />
    <case date="16-FEB-1994" temperature="280" />
    <case date="16-MAR-1994" temperature="278.9" />
    <case date="16-APR-1994" temperature="278.9" />
    <case date="16-MAY-1994" temperature="277.8" />
    <case date="16-JUN-1994" temperature="276.1" />

    ...

</temperatures>
```

```
            VARIABLE : Mean TS from clear sky composite (kelvin)
            FILENAME : ISCCPMonthly_avg.nc
            FILEPATH : /usr/local/fer_dsets/data/
            SUBSET   : 93 points (TIME)
            LONGITUDE: 123.8W(-123.8)
            LATITUDE : 48.8S
                      123.8W
                       23
16-JAN-1994 00 /  1:  278.9
16-FEB-1994 00 /  2:  280.0
16-MAR-1994 00 /  3:  278.9
16-APR-1994 00 /  4:  278.9
16-MAY-1994 00 /  5:  277.8
16-JUN-1994 00 /  6:  276.1

...
```

Figure 7.7: The first few lines of the surface temperature at Point Nemo in two formats: plain text and XML.

```
<case date="16-JAN-1994" temperature="278.9" />
```

This should look very familiar because these are exactly the same notions of elements and attributes that we saw in HTML documents (see Chapter 2).

7.4.2 Advantages and disadvantages

In what ways is the XML format better or worse than the typical unstructured plain text format?

A self-describing format

The core advantage of an XML document is that it is **self-describing**.

The tags in an XML document provide information about where the data is stored within the document. This is an advantage because it means that humans can find information within the file easily. That is true of any plain text file, but it is especially true of XML files because the tags essentially provide a level of documentation for the human reader. For example, a line like this ...

```
<latitude>48.8S</latitude>
```

... not only makes it easy to determine that the value `48.8S` constitutes a single data value within the file, but it also makes it clear that this value is a north-south geographic location.

The fact that an XML document is self-describing is an even greater advantage from the perspective of the computer. An XML document provides enough information for software to determine how to read the file, without any further human intervention. Looking again at the line containing latitude information ...

```
<latitude>48.8S</latitude>
```

... there is enough information for the computer to be able to detect the value `48.8S` as a single data value, and the computer can also record the `latitude` label so that if a human user requests the information on latitude, the computer knows what to provide.

One consequence of this feature that may not be immediately obvious is that it is much easier to modify the structure of data within an XML document compared to a plain text file. The location of information within an XML document is not so much dependent on where it occurs

within the file, but where the tags occur within the file. As a trivial example, consider swapping the following lines in the Point Nemo XML file ...

```
    <longitude>123.8W(-123.8)</longitude>
    <latitude>48.8S</latitude>
```

... so that they look like this instead ...

```
    <latitude>48.8S</latitude>
    <longitude>123.8W(-123.8)</longitude>
```

The information is now at a different location within the file, but the task of retrieving the information on latitude is exactly the same. This can be a huge advantage if larger modifications need to be made to a data set, such as adding an entire new variable.

Representing complex data structures

The second main advantage of the XML format is that it can accommodate complex data strucutures. Consider the hierarchical data set in Figure 7.4. Because XML elements can be nested within each other, this sort of data set can be stored in a sensible fashion with families grouped together to make parent-child relations implicit and avoid repetition of the parent data. The plain text representation of these data are reproduced from page 121 below along with a possible XML representation.

```
                John      33  male
                Julia     32  female
John   Julia    Jack       6  male
John   Julia    Jill       4  female
John   Julia    John jnr   2  male
                David     45  male
                Debbie    42  female
David  Debbie   Donald    16  male
David  Debbie   Dianne    12  female
```

```
<family>
  <parent gender="male" name="John" age="33" />
  <parent gender="female" name="Julia" age="32" />
  <child gender="male" name="Jack" age="6" />
  <child gender="female" name="Jill" age="4" />
  <child gender="male" name="John jnr" age="2" />
</family>
<family>
  <parent gender="male" name="David" age="45" />
  <parent gender="female" name="Debbie" age="42" />
  <child gender="male" name="Donald" age="16" />
  <child gender="female" name="Dianne" age="12" />
</family>
```

The XML format is superior in the sense that the information about each person is only recorded once. Another advantage is that it would be very easy to represent a wider range of situations using the XML format. For example, adding a third step-parent to a family would be straightforward in XML, but it would be much more awkward in the fixed rows-and-columns plain text format.

Data integrity

Another important advantage of the XML format is that it provides some level of checking on the correctness of the data file (a check on the **data integrity**). We will discuss this more when we look at XML schema (see Section 7.4.5), but even just the fact that an XML document must obey the rules of XML means that we can use a computer to check that an XML document at least has a sensible structure.

Verbosity

The major disadvantage of XML is that it generates large files. With it being a plain text format, it is not memory efficient to start with, then with all of the additional tags around the actual data, files can become extremely large. In many cases, the tags can take up more room than the actual data!

These issues can be particularly acute for scientific data sets, where the structure of the data may be quite straightforward. For example, geographic data sets containing many observations at fixed locations naturally form a 3-dimensional array of values, which can be represented very simply and efficiently in a plain text or binary format. In

such cases, having highly repetitive XML tags around all values can be very inefficient indeed.

The verbosity of XML is also a problem for entering data into an XML format. It is just too laborious for a human to enter all of the tags by hand, so, in practice, it is only sensible to have a computer generate XML documents.

Costs of complexity

It should also be acknowledged that the additional sophistication of XML creates additional costs. Users have to be more educated and the software has to be more complex (which makes compatible software more rare).

The fact that computers can read XML easily and effectively, plus the fact that computers can produce XML rapidly (verbosity is less of an issue for a computer), means that XML is an excellent format for transferring information between different software programs. XML is a good language for computers to use to talk to each other, with the added bonus that humans can still easily eavesdrop on the conversation.

7.4.3 More XML syntax

As mentioned in the previous section, one of the advantages of XML is that the computer can perform checks on the correctness of an XML document, which provides at least some checks the correctness of the data that are stored in the document (also see Section 7.4.5). This section provides a bit more information on the basic rules that an XML document must obey. Much of the list is just a slightly stricter version of the HTML syntax rules that we met back in Section 2.2.1.

The first line of the document should declare that it is an XML document and the XML version being used. For example:

```
<?xml version="1.0"?>
```

The XML document must have a single **root** element.

Every element must have a start tag and an end tag. Empty elements must have the form `<name />`.

Elements must nest cleanly.

Attribute values must be within quotes.

Element and attribute names are case-sensitive.

Escape sequences

As with HTML, the characters `<`, `>`, and `&` (among others) are special and must be replaced with special escape sequences, `<`, `>`, and `&` respectively.

These escape sequences can be very inconvenient when storing data values, so it is also possible to mark an entire section of an XML document as "plain text" by placing it within a `CDATA` section, as follows:

```
<myxmlelement>
  <![CDATA[
    Lots of "<"s, ">"s, "&"s, "'"s and """s
  ]]>
</myxmlelement>
```

7.4.4 XML design

Though it is important to understand why XML documents are used, designing an XML document may be a rare event for a scientist. Nevertheless, we have something to gain from a brief consideration of how data might be organised in an XML file.

The first point is that there are many ways that a data set could be stored within an XML document. XML is actually a **meta-language**; it is a language for defining languages. In other words, we get to decide the structure for an XML document and XML is a language for describing the structure that we choose.

So what structure should we choose? We will look at some issues that might influence the design of an XML document. These will be useful in understanding why an XML document that we encounter has a particular structure and they will be useful as an introduction to similar ideas that we will discuss when we get to relational databases (Section 7.7).

Marking up data

The first XML design issue is to make sure that each value within a data set can be clearly identified. In other words, it should be trivial for a computer to extract each individual value. This means that every single value should be either the content of an element or the value of an attribute. The XML document shown in Figure 7.7 demonstrates this idea.

Figure 7.8 shows two other possible XML representations of the Pacific Pole of Inaccessibility temperature data. The example at the top of the figure demonstrates that it is very easy to create an XML document that follows the rules of XML, but provides no benefits over the original plain text format.

The example at the bottom of Figure 7.8 is more interesting. In this case, the irregular and one-off metadata values are individually identified within elements or attributes, but the regular and repetitive raw data values are not. This is not ideal from the point of view of the file being self-describing, but it may be a viable option when the raw data values have a very simple format (e.g., comma-delimited) and the data set is very large (so avoiding lengthy tags and attribute names is a major saving).

Things and measurements on things

When a data set has a non-rectangular structure, such as the family tree in Figure 7.4, an XML document can be designed to store the information more efficiently and more appropriately. The main idea here is to avoid repeating values.

When presented with a data set, the following questions should guide the design of the XML format:

- What sorts objects or "things" have been measured?
- What measurements have been made on each object or "thing"?

The rule of thumb is then to have an element for each object in the data set (and a different *type* of element for each different type of object) and then have an attribute for each measurement in the data set. Simple relationships between objects can sometimes be expressed by nesting elements.

For example, in the family tree data set, there are obviously measurements taken on people, those measurements being names and ages and genders. We could distinguish between parent objects and child objects, so we have elements like these:

```
<?xml version="1.0"?>
<temperatures>
            VARIABLE : Mean TS from clear sky composite (kelvin)
            FILENAME : ISCCPMonthly_avg.nc
            FILEPATH : /usr/local/fer_dsets/data/
            SUBSET   : 93 points (TIME)
            LONGITUDE: 123.8W(-123.8)
            LATITUDE : 48.8S
                       123.8W
                        23
16-JAN-1994 00 /  1:  278.9
16-FEB-1994 00 /  2:  280.0
16-MAR-1994 00 /  3:  278.9
16-APR-1994 00 /  4:  278.9
16-MAY-1994 00 /  5:  277.8
16-JUN-1994 00 /  6:  276.1

...

</temperatures>
```

```
<?xml version="1.0"?>
<temperatures>
    <variable>Mean TS from clear sky composite (kelvin)</variable>
    <filename>ISCCPMonthly_avg.nc</filename>
    <filepath>/usr/local/fer_dsets/data/</filepath>
    <subset>93 points (TIME)</subset>
    <longitude>123.8W(-123.8)</longitude>
    <latitude>48.8S</latitude>
    <cases>
16-JAN-1994 00 /  1:  278.9
16-FEB-1994 00 /  2:  280.0
16-MAR-1994 00 /  3:  278.9
16-APR-1994 00 /  4:  278.9
16-MAY-1994 00 /  5:  277.8
16-JUN-1994 00 /  6:  276.1

    ...

    </cases>
</temperatures>
```

Figure 7.8: The first few lines of the surface temperature at Point Nemo in two possible XML formats with differing levels of sophistication and appropriateness. These should be compared with the "complete" XML solution in Figure 7.7.

```
  <parent gender="female" name="Julia" age="32" />
  <child gender="male" name="Jack" age="6" />
```

There are two distinct families of people, so we could have elements to represent the different families and nest the relevant people within the appropriate family element to represent membership of a family.

```
<family>
  <parent gender="male" name="John" age="33" />
  <parent gender="female" name="Julia" age="32" />
  <child gender="male" name="Jack" age="6" />
  <child gender="female" name="Jill" age="4" />
  <child gender="male" name="John jnr" age="2" />
</family>
<family>
  <parent gender="male" name="David" age="45" />
  <parent gender="female" name="Debbie" age="42" />
  <child gender="male" name="Donald" age="16" />
  <child gender="female" name="Dianne" age="12" />
</family>
```

Elements versus attributes

Another decision to make is whether to store data values as the value of attributes of XML elements, or as the content of XML elements. For example, when storing the Point Nemo temperature data, we could store the temperature values as attributes of a `case` element ...

```
<case temperature="278.9" />
```

... or as the contents of a `temperature` element:

```
<temperature>278.9</temperature>
```

As demonstrated so far, the easiest solution is to store all measurements as the values of attributes. However, this is not always possible or appropriate. A measurement may have to be stored as the content of a separate element, rather than as the value of an attribute in the following cases:

- when the measurement is a lot of information, such as a general comment consisting of paragraphs of text.

- when the measurement contains lots of special characters, which would require a lot of escape sequences.
- when the order of the measurements matters (the order of attributes is arbitrary).
- when the measurements are not "simple" values. In other words, when a measurement is actually a series of measurements on a different sort of object (e.g., information about a room within a building). This is another way of saying that the value of an attribute cannot be an XML element. Data values that are not atomic (a single value) will generate an entire XML element, which must be stored as the content of a parent XML element.

7.4.5 XML Schema

We have already discussed the fact that an XML document can provide some checks that a data set is correct because the XML document must obey the basic rules of XML (elements must nest, attribute values must be surrounded by quotes, etc). While this sort of checking is better than nothing, the checks are very basic. It is much more useful to be able to perform more advanced checks such as whether necessary data values are included in a document, whether elements contain the correct sort of data value, and so on. With a little more work, XML provides these more advanced checking features as well.

The way that this extra information can be specified is by creating a **schema** for an XML document, which is a description of the structure of the document. A number of technologies exist for specifying XML schema, but we will focus on the **Document Type Definition** (DTD) language.

A DTD is a set of rules for an XML document. It contains **element type declarations** that describe what elements are permitted within the XML document, in what order, and how they may be nested within each other. The DTD also contains **attribute list declarations** that describe what attributes an element can have, whether attributes are optional or not, and what sort of values may be specified for each attribute.

7.4.6 Case study: Point Nemo (continued)

Figure 7.9 shows the temperature data at Point Nemo in an XML format (this is a reproduction of Figure 7.7 for convenience).

```
<?xml version="1.0"?>
<temperatures>
    <variable>Mean TS from clear sky composite (kelvin)</variable>
    <filename>ISCCPMonthly_avg.nc</filename>
    <filepath>/usr/local/fer_dsets/data/</filepath>
    <subset>93 points (TIME)</subset>
    <longitude>123.8W(-123.8)</longitude>
    <latitude>48.8S</latitude>
    <case date="16-JAN-1994" temperature="278.9" />
    <case date="16-FEB-1994" temperature="280" />
    <case date="16-MAR-1994" temperature="278.9" />
    <case date="16-APR-1994" temperature="278.9" />
    <case date="16-MAY-1994" temperature="277.8" />
    <case date="16-JUN-1994" temperature="276.1" />

    ...

</temperatures>
```

Figure 7.9: The first few lines of the surface temperature at Point Nemo in an XML format.

The structure of this XML document is as follows: there is a single overall `temperatures` element that contains all other elements. There are several elements containing various sorts of metadata: a `variable` element containing a description of the variable that has been measured; a `filename` element and a `filepath` element containing information about the file from which these data were extracted; and three elements, `subset`, `longitude`, and `latitude`, that together decribe the temporal and spatial limits of this subset of the original data. Finally, there are a number of `case` elements that contain the raw temperature data and each `case` element contains a temperature measurement and the date of the measurement as attributes.

A DTD describing this structure is shown in Figure 7.10.

For each type of element, there has to be an `<!ELEMENT>` declaration. The simplest example is for `case` elements because they are empty (they have no content), as indicated by the keyword `EMPTY`. Most other elements are similarly straightforward because their contents are just text, as indicated by the `#PCDATA` keyword. The `temperatures` element is more complex because it can contain other elements. The specification given in Figure 7.10 states that six elements (`variable` to `latitude`) must be present, and they must occur in the given order. There may also be zero or more `case` elements (the `*` means "zero or more").

```
<!ELEMENT temperatures (variable,
                        filename,
                        filepath,
                        subset,
                        longitude,
                        latitude,
                        case*)>
<!ELEMENT variable (#PCDATA)>
<!ELEMENT filename (#PCDATA)>
<!ELEMENT filepath (#PCDATA)>
<!ELEMENT subset (#PCDATA)>
<!ELEMENT longitude (#PCDATA)>
<!ELEMENT latitude (#PCDATA)>
<!ELEMENT case EMPTY>

<!ATTLIST case
    date        ID    #REQUIRED
    temperature CDATA #IMPLIED>
```

Figure 7.10: A DTD for the XML format used to store the surface temperature at Point Nemo (see Figure 7.9).

The `case` elements also have attributes, so there is an `<!ATTLIST>` declaration as well. This says that there must (`#REQUIRED`) be a `date` attribute and that each `date` value must be unique (`ID`). The `temperature` attribute is optional (`#IMPLIED`) and, if it occurs, the value can be any text (`CDATA`).

Section 8.2 describes the syntax and semantics of DTD files in more detail.

The rules given in a DTD are associated with an XML document using a **Document Type Declaration** as the second line of the XML document. This can have one of two forms:

DTD inline:
: The DTD can be included within the XML document. In the Point Nemo example, it would look like this:

```
<?xml version="1.0"?>
<!DOCTYPE temperatures [
    DTD code here
 ]>
<temperatures>
    ...
```

External DTD

The DTD can be in an external file, say `nemo.dtd`, and the XML document can refer to that file:

```
<?xml version="1.0"?>
<!DOCTYPE temperatures SYSTEM "nemo.dtd">
<temperatures>
    ...
```

The DRY principle suggests that an external DTD is by far the most sensible approach.

If an XML document is well-formed—it obeys the basic rules of XML syntax—*and* it obeys the rules given in a DTD, then the document is said to be **valid**. A validated XML document has the advantage that we can be sure that all of the necessary information for a data set has been included and has the correct structure, and that all data values have the correct sort of value.

The use of a DTD has some shortcomings, such as a lack of support for precisely specifying the data type of attribute values or the contents of elements. For example, it is not possible to specify that an attribute value must be an integer value between 0 and 100. There is also the difficulty that the DTD language is completely different from XML, so there is another technology to learn. XML Schema is an XML-based technology for specifying the design of XML documents that solves both of those problems, but it comes at the cost of much greater complexity. This complexity has lead to the development of further technologies that simplify the XML Schema syntax, such as Relax NG. The interested reader is referred to Section 8.3 for further pointers.

7.4.7 XML design for complex relationships

Another way to represent relationships between objects (elements) in an XML document is to use the special `ID` and `IDREF` (or `IDREFS`) attributes. An example of their use is given in the following variation on the storage of the hierarchical family data.

```
<family id="family1" />
<family id="family2" />

<parent gender="male" name="John" age="33"
        family="family1" />
<parent gender="female" name="Julia" age="32"
        family="family1" />
<child gender="male" name="Jack" age="6"
       family="family1" />
<child gender="female" name="Jill" age="4"
       family="family1" />
<child gender="male" name="John jnr" age="2"
       family="family1" />

<parent gender="male" name="David" age="45"
        family="family2" />
<parent gender="female" name="Debbie" age="42"
        family="family2" />
<child gender="male" name="Donald" age="16"
       family="family2" />
<child gender="female" name="Dianne" age="12"
       family="family2" />
```

The `family` elements have an `id` attribute and each `parent` or `child` element has a `family` attribute that refers to the `id` attribute value of one of the `family` elements. In the DTD, the `id` attribute is of type `ID` and the `family` attribute is of type `IDREF`.

When these attributes are used, additional data integrity checks can be enforced, because the DTD rules state that an `IDREF` attribute must have a value that matches the value of an `ID` element somewhere within the same XML document.

This sort of design problem arises because some situations cannot be represented by nesting elements within each other. These sorts of situations are discussed in more detail in Section 7.7.6 on database design.

7.5 Binary files

As we saw in Section 7.2, all electronic information, regardless of the format, is ultimately stored in a binary form—as a series of bits (zeroes and ones). However, the same data value can be recorded as a binary value in a number

of different ways.

For example, given the number `12345`, we could store it as individual characters `1`, `2`, `3`, `4`, and `5`, using one byte for each character:

```
00110001 00110010 00110011 00110100 00110101
```

Alternatively, we could store the number as a four-byte integer (see Section 7.2.3):

```
00111001 00110000 00000000 00000000
```

When we store information as individual one-byte characters, the result is a plain text file. This tends to be a less efficient method because it tends to consume more memory, but it has the advantage that the file has a very simple structure. A simple structure means that it is relatively easy to write software to read the file because we know that each byte just needs to be converted to a character. There may be problems determining data values from the individual characters (see Section 7.3), but the process of reading the basic unit of information (a character) from the file is straightforward.

For the purposes of this book, a **binary format** is just any format that is *not* plain text.

The characteristic feature of a binary format is that there is *not* a simple rule for determining how many bits or how many bytes constitute a basic unit of information. Given a series of, say, four bytes, we cannot assume that these correspond to four characters, or a single four-byte integer, or half of an eight-byte floating-point value (see Section 7.2.3). It is necessary for there to be a description of the rules for the format (we will look at one example soon) that state what information is stored and how many bits or bytes are used for each piece of information.

Binary formats are consequently much harder to write software for, which results in there being less software available to do the job.

However, some binary formats are easier to read than others. Given that a description is necessary to have any chance of reading a binary file, proprietary formats, where the file format description is kept private, are extremely difficult to deal with. Open standards become more important than ever.

Another advantage of binary formats is an improvement in terms of speed of access. While the basic unit of information is very straightforward in a plain text file (one byte equals one character), finding the actual data values is often much harder. For example, in order to find the third data value on the tenth row of a CSV file, the reader software must keep reading bytes

until nine end-of-line characters have been found and then two delimiter characters have been found. This means that, with text files, it is usually necessary to read the entire file in order to find any particular value.

For binary formats, some sort of format description, or map, is required to be able to find the location (and meaning) of any value in the file. However, the advantage of having such a map is that any value within the file can be found without having to read the entire file.

As a typical example, a standard feature of binary files is the inclusion of some sort of **header information**, both for the overall file, and for subsections within the file. This header information contains information such as the byte location within the file where a set of values begins (a *pointer*), the number of bytes used for each data value (the data *size*), plus the number of data values. It is then very simple to find, for example, the third data value within a set of values via straightforward arithmetic: $pointer + 2 \times size$.

More information is required in order to locate values within a binary format, but once that information is available, navigation within the file is faster and more flexible.

7.5.1 Case study: Point Nemo (continued)

Figure 7.11 shows the start of a binary file representation of the Point Nemo temperature data (see Section 1.1). The format is called netCDF and it has a specific structure that combines text and numeric values.

We will learn more about this format below. For now, the only point to make is that, while we can see some bits of text in the file (the right-hand column in Figure 7.11 shows ASCII interpretations of the bytes), it is not clear from this raw display of the data, where the text data starts and ends, where the numeric values are, and what each value is for.

Compared to a plain text file, this is a complete mess and we need software that understands the netCDF format in order to extract useful values from the file.

7.5.2 NetCDF

In order to provide a better understanding of binary formats, we will look at a specific example of an open standard binary format called network

```
  0  :  43 44 46 01 00 00 00 00 00 00 00  |  CDF........
 11  :  0a 00 00 00 01 00 00 00 04 54 69  |  .........Ti
 22  :  6d 65 00 00 00 30 00 00 00 00 00  |  me...0.....
 33  :  00 00 00 00 00 00 0b 00 00 00 02  |  ...........
 44  :  00 00 00 04 54 69 6d 65 00 00 00  |  ....Time...
 55  :  01 00 00 00 00 00 00 00 0c 00 00  |  ...........
 66  :  00 01 00 00 00 05 75 6e 69 74 73  |  ......units
 77  :  00 00 00 00 00 00 02 00 00 00 1f  |  ...........
 88  :  6e 75 6d 62 65 72 20 6f 66 20 64  |  number of d
 99  :  61 79 73 20 73 69 6e 63 65 20 31  |  ays since 1
110  :  39 37 30 2d 30 31 2d 30 31 00 00  |  970-01-01..
121  :  00 00 06 00 00 01 80 00 00 00 ec  |  ...........
132  :  00 00 00 0b 54 65 6d 70 65 72 61  |  ....Tempera
143  :  74 75 72 65 00 00 00 00 01 00 00  |  ture.......
154  :  00 00 00 00 00 0c 00 00 00 02 00  |  ...........
165  :  00 00 05 75 6e 69 74 73 00 00 00  |  ...units...
176  :  00 00 00 02 00 00 00 06 4b 65 6c  |  ........Kel
187  :  76 69 6e 00 00 00 00 00 0d 6d 69  |  vin......mi
198  :  73 73                             |  ss
```

Figure 7.11: The first 200 bytes of the netCDF format representation of the surface temperatures at Point Nemo. The data are shown here as unstructured bytes to demonstrate that, without knowledge of the structure of the binary file, there is no way to determine where the different data values reside or what format the values have been stored in (apart from the fact that there is obviously some textual data in the file).

Common Data Form (netCDF).[12]

A netCDF file contains three sorts of information: dimensions, variables, and attributes. Dimensions correspond to variables that are under experimental control, such as the geographic locations at which measurements are taken, the time points at which measurements are taken, or the levels of a drug that are to be administered. These are sometimes called "independent variables". In the case of the Point Nemo data set, there is one dimension: the months for which we have temperature records (from January 1994 to December 1997).

Variables correspond to measurements that are recorded for each case or subject. These are sometimes called "dependent variables". In the Point Nemo example, the variable is surface temperature.

Attributes are used to record metadata, such as the units of measurement in the Point Nemo data set.

The structure of a netCDF file consists of header information, which includes how many dimensions and variables are in the data set, followed by the raw data itself. Figure 7.12 shows a structured view of the start of the netCDF format for the Point Nemo temperature data. We will use this display to explain some of the header information within a netCDF file.

The first four bytes within a netCDF file contain information on the type of the file. The three characters `CDF` indicate that this is a netCDF file and the last byte specifies which version of netCDF is being used. The value `01` indicates that this is a version 1, or "classic", netCDF file.

The next four bytes in the netCDF format indicate how many "records" are stored in the file. This is only used when the size of the data set can grow, so in this case, where there is a fixed size to the data set, the value is just 0.

The third set of four bytes in the netCDF file represent a **flag**. The value of these four bytes is an integer that indicates what sort of information is to follow. In this case, the value `11` states that this is the beginning of a list of the dimensions that have been stored in the file.

The next four bytes also represent an integer, this time indicating how many dimensions are in the file. This file contains a single dimension.

At this point in the file, we start to see information about each of the dimensions. The first piece of information is the name of the dimension. This is character information so it comes in the form of an integer that says

[12] `http://www.unidata.ucar.edu/software/netcdf/`

```
========magicNumber
  0   :  43 44 46 01                          |  CDF.
========numRecords
  4   :  00 00 00 00                          |  0
========dimensionArrayFlag
  8   :  00 00 00 0a                          |  10
========dimensionArraySize
 12   :  00 00 00 01                          |  1
========dim1NameSize
 16   :  00 00 00 04                          |  4
========dim1Name
 20   :  54 69 6d 65                          |  Time
```

Figure 7.12: The start of the header information in the netCDF file that contains the surface temperatures at Point Nemo, with the structure of the netCDF format revealed so that separate data fields can be observed. The first three bytes are characters, the fourth byte is a number, the next four bytes are an integer value, and so on. This structured display shows just the first 24 bytes of the 200 unstructured bytes shown in Figure 7.11.

how many characters there are, followed by that many bytes containing the actual text. We can see that in the case of the Point Nemo file, the name of the single dimension is "Time", which is four characters long.

This small example demonstrates the classic structure of binary files. Information of different sorts is packed into the file right next to each other with lots of signs and labels to provide information about where the each piece starts and ends.

We will now look a bit further into the file to see a few more examples of how binary files are structured. Figure 7.13 shows two more chunks of the netCDF form of the Point Nemo data that relate to the storage of the surface temperature values within the file.

The top block in Figure 7.13 shows a part of the header of the netCDF file that contains information about the variables contained within the file. The first four bytes shown represent an integer value that indicates what data type is used to store the temperature values. The value `5` is used in netCDF to indicate that the values are stored as single-precision real values; each value is stored as a floating-point value using four bytes. The next four bytes also represent an integer, this time the total number of bytes used to store the values. This corresponds to the total number of values multiplied by the amount of memory used for each value; $48 \times 4 = 192$. The final four

```
========vble2Type
224  :  00 00 00 05                          |  5
========vble2Size
228  :  00 00 00 c0                          |  192
========vble2Offset
232  :  00 00 02 6c                          |  620
```

```
========temperatures
620  :  43 8b 73 33 43 8c 00 00  |  278.9 280.0
628  :  43 8b 73 33 43 8b 73 33  |  278.9 278.9
636  :  43 8a e6 66 43 8a 0c cd  |  277.8 276.1
644  :  43 8a 0c cd 43 89 cc cd  |  276.1 275.6
652  :  43 89 cc cd 43 8a a6 66  |  275.6 277.3
660  :  43 8a 59 9a 43 8b 73 33  |  276.7 278.9
668  :  43 8c cc cd 43 8c 8c cd  |  281.6 281.1
676  :  43 8c 00 00 43 8b 73 33  |  280.0 278.9
684  :  43 8a e6 66 43 8a 59 9a  |  277.8 276.7
692  :  43 8a a6 66 43 8a 0c cd  |  277.3 276.1
700  :  43 8a 0c cd 43 8a 59 9a  |  276.1 276.7
708  :  43 8b 33 33 43 8a e6 66  |  278.4 277.8
716  :  43 8c 8c cd 43 8d 99 9a  |  281.1 283.2
724  :  43 8c 8c cd 43 8b c0 00  |  281.1 279.5
732  :  43 8b 33 33 43 8a 59 9a  |  278.4 276.7
740  :  43 8a 0c cd 43 89 cc cd  |  276.1 275.6
748  :  43 89 cc cd 43 8a 0c cd  |  275.6 276.1
756  :  43 8a a6 66 43 8b 73 33  |  277.3 278.9
764  :  43 8c 40 00 43 8c cc cd  |  280.5 281.6
772  :  43 8c 00 00 43 8b 73 33  |  280.0 278.9
780  :  43 8b 33 33 43 8a 59 9a  |  278.4 276.7
788  :  43 89 cc cd 43 89 cc cd  |  275.6 275.6
796  :  43 8a a6 66 43 8a 59 9a  |  277.3 276.7
804  :  43 8b 33 33 43 8b c0 00  |  278.4 279.5
```

Figure 7.13: Two blocks of bytes within the netCDF file that contains the surface temperatures at Point Nemo. The top block, starting at byte 224, shows header information that specifies where the temperature values are stored within the file and how much memory is used for each value. These are all four-byte integer values. The bottom block, starting at byte 620, shows the memory block that stores the actual temperature values. These are all four-byte floating-point values.

bytes in this block again represent an integer, but this value is an **offset** that indicates the location of the actual temperature values within the file. This value says that the temperature values start at byte 620 within the file.

The bottom block in Figure 7.13 shows the 192 bytes within the netCDF file, starting at byte 620, with every four bytes interpreted as a single-precision floating-point value. These are the temperature values from the Point Nemo data set in a binary format.

It is not necessary to deal with binary files at this level of detail in order to access data. The appropriate software takes care of all of the details. The purpose of taking this closer look is to demonstrate the difference between plain text files and binary file formats. This example also provides a demonstration of how binary formats are set up with formats that describe the structure of the file itself. This means that, although software has to be given a lot of information about the netCDF format in order to read a netCDF file, once that information has been provided, the software can read *any* netCDF file. This is in contrast to plain text files; any software can read the characters within a plain text file, but interpreting the characters as data is difficult or impossible without information about the specific structure of the file.

7.6 Spreadsheets

To this point, we have discussed data storage options in terms of various **file formats**. We have been concerned with low-level information about how data is organised within a digital file. In this section, and in the following section on databases, we will be discussing higher-level interfaces to the data.

We are no longer concerned with the details of how the information is stored within a file. Instead we will work with a more **conceptual data model**, leaving the details of file formats to software.

For many people, the term **spreadsheet** is synonymous with Microsoft Excel. This section is *not* about Microsoft Excel, nor does it concern the suite of features of any particular spreadsheet software such as Gnumeric or Open Office Calc.

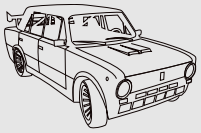
This section is concerned with the general concepts of spreadsheets and with how appropriate they are for data storage.

7.6.1 The structure of spreadsheets

The fundamental structure of a spreadsheet is a grid or table of cells, which are arranged neatly in rows and columns.

Spreadsheets are, by default, unstructured, with each cell capable of containing any type of information, independently of all other cells.

7.6.2 Case study: Over the limit



The Lada Riva is one of the highest-selling car models of all time and the only model to be found on every continent in the world.[13]

A study conducted by researchers from the Psychology Department at the University of Auckland[14] looked at whether informative road signs had any effect on the speed at which vehicles travelled along a busy urban road in Auckland.

Data were collected for several days during a baseline period and for several days when each of five different signs were erected beside the road. At each stage, the vehicle speeds were also collected for traffic travelling in the opposite direction along the road to provide a control set of observations.

The data were collected by the Waitakere City Council via detectors buried in the road and were delivered to the researchers in the form of Excel spreadsheets. Figure 7.14 shows a section of one of these spreadsheets and we will use this to demonstrate some of the advantages and disadvantages of using spreadsheets to store data.

One of the major attractions of using a spreadsheet is the fact that spreadsheet software displays the spreadsheet cells in a rectangular grid so that it is convenient and fast to enter, view, and modify the raw values in the data set. The arrangement of cells into distinct columns shares the same benefits as fixed-width format text files: it makes it very easy for a human to view and navigate within the data.

[13] Image source: The Open Clip Art Library
`http://openclipart.org/people/KlausGena/KlausGena_Tuned_Lada_VAZ_2101.svg`
This image is in the public domain.

[14] Wrapson, W., Harré, N, Murrell, P. (2006) Reductions in driver speed using posted feedback of speeding information: Social Comparison or Implied Surveillance? *Accident Analysis and Prevention.* **38**, 1119-1126.

	A	B	C	D	E	F	G	H	I	J	K
1	Parrs Cross Road from Seymour Road, Daily Speed										
2	Monday 13/03/00										
3	Speed (KPH)										
4	Hour End	0 – 30	30 – 40	40 – 50	50 – 60	60 – 70	70 – 80	80 – 90	90–100	100–110	110–200
5											
6	1:00	0	1	14	26	15	2	0	0	0	0
7	2:00	0	1	7	13	5	2	0	0	0	0
8	3:00	0	0	1	2	5	1	2	0	0	0
9	4:00	0	0	3	2	2	2	1	0	0	0
10	5:00	0	0	5	4	2	0	1	0	0	0
11	6:00	0	2	8	28	17	3	0	0	0	0
12	7:00	0	4	45	110	39	3	1	0	0	0
13	7:15	2	2	37	78	17	1	0	0	0	0
14	7:30	0	2	65	84	26	0	0	0	0	0
15	7:45	1	0	53	160	16	0	0	0	0	0
16	8:00	2	13	43	125	45	2	0	0	0	0
17	7– 8	5	17	198	447	104	3	0	0	0	0
18	8:15	0	20	44	151	35	1	0	0	0	0
19	8:30	0	3	69	154	28	0	0	0	0	0
20	8:45	3	4	81	164	12	0	0	0	0	0
21	9:00	2	7	106	160	9	0	0	0	0	0
22	8– 9	5	34	300	629	84	1	0	0	0	0
23	10:00	0	12	225	460	80	6	0	0	0	1
24	11:00	3	13	128	313	68	4	1	2	0	0
25	12:00	6	18	180	353	62	1	0	0	0	1
26	13:00	6	11	133	383	84	3	0	0	1	0
27	14:00	12	16	196	329	55	3	0	0	0	0
28	15:00	5	28	156	351	74	1	0	0	0	0

Figure 7.14: A section from the vehicle speed data as it was delivered in a spreadsheet format.

This is not to suggest that data validation should be performed by eyeballing the data set in a spreadsheet. That should be performed using plots and tools that produce simple numerical summaries, examples of which we will meet in Chapter 11. However, taking a look at the raw values within a data set is never a bad thing.

Figure 7.14 also shows that it is straightforward to include metadata in a spreadsheet (cells A1 to A3) because each cell in the spreadsheet can contain any sort of value.

Most spreadsheet software also allows for multiple sheets within a document, effectively providing a 3-dimensional cube of data cells. The vehicle speed study made use of this feature to store the data from each day on a separate sheet. Each condition of the study was stored in a separate spreadsheet file. Figure 7.15 shows three sheets, representing three days' worth of data, within one of the spreadsheet files.

In summary, spreadsheets are easy to use and very convenient. Not surprisingly, they are a very common choice for storing and sharing data sets.

However, there are some dangers to using spreadsheets for data storage, most of which stem from these same desirable features of simplicity and

	A	B	C	D	E	F	G	H	I	J	K
1	Parrs Cross Road from Seymour Road, Daily Speed										
2	Monday 13/03/00										
3	Speed (KPH)										
4	Hour End	0 – 30	30 – 40	40 – 50	50 – 60	60 – 70	70 – 80	80 – 90	90–100	100–110	110–200
5											
6	1:00										
7	2:00										
8	3:00										
9	4:00										
10	5:00										
11	6:00										
12	7:00										
13	7:15										
14	7:30										
15	7:45										
16	8:00										
17	7– 8										
18	8:15										
19	8:30										
20	8:45										
21	9:00										
22	8– 9										
23	10										
24	11										
25	12										
26	13										
27	14										
28	15										

	A	B	C	D	E	F	G	H	I	J	K
1	Parrs Cross Road from Seymour Road, Daily Speed										
2	Tuesday 14/03/00										
3	Speed (KPH)										
4	Hour End	0 – 30	30 – 40	40 – 50	50 – 60	60 – 70	70 – 80	80 – 90	90–100	100–110	110–200
5											
6	1:00										
7	2:00										
8	3:00										
9	4:00										
10	5:00										
11	6:00										
12	7:00										
13	7:15										
14	7:30										
15	7:45										
16	8:00										
17	7– 8										
18	8:15										
19	8:30										
20	8:45										
21	9:00										
22	8– 9										
23	10										
24	11										
25	12										
26	13										
27	14										
28	15										

	A	B	C	D	E	F	G	H	I	J	K
1	Parrs Cross Road from Seymour Road, Daily Speed										
2	Wednesday 15/03/00										
3	Speed (KPH)										
4	Hour End	0 – 30	30 – 40	40 – 50	50 – 60	60 – 70	70 – 80	80 – 90	90–100	100–110	110–200
5											
6	1:00	1	0	17	42	11	1	0	1	0	0
7	2:00	0	1	12	14	5	0	1	0	0	0
8	3:00	0	1	2	11	0	1	0	0	0	0
9	4:00	0	0	3	5	3	0	1	0	0	0
10	5:00	0	1	7	6	6	1	0	0	0	0
11	6:00	1	3	7	28	17	2	1	0	0	0
12	7:00	0	2	55	124	25	3	0	0	0	0
13	7:15	0	0	51	89	13	0	0	1	0	0
14	7:30	0	2	77	116	8	0	0	0	0	0
15	7:45	0	6	113	118	7	0	0	0	0	0
16	8:00	1	15	72	130	17	0	0	0	0	0
17	7– 8	1	23	313	453	45	0	0	1	0	0
18	8:15	0	21	117	130	5	0	0	0	0	0
19	8:30	0	8	121	117	11	3	0	0	0	0
20	8:45	19	42	122	74	9	0	0	0	0	0
21	9:00	56	64	118	82	5	0	0	0	0	0
22	8– 9	75	135	478	403	30	3	0	0	0	0
23	10:00	70	30	304	364	35	1	0	0	0	0
24	11:00	4	45	228	305	33	1	0	0	0	0
25	12:00	5	29	210	346	35	0	0	0	0	0
26	13:00	2	9	227	360	72	6	0	0	0	0
27	14:00	2	11	128	402	94	11	0	0	0	0
28	15:00	5	40	221	370	46	1	0	0	0	0

Figure 7.15: Sections from three of the sheets in the vehicle speed data spreadsheet.

ease of use.

One important problem is that spreadsheet cells are able to act independently of each other. Although the data in Figure 7.14 appear to have a useful structure, with each data value clearly associated with a time period and a speed range, this is in fact an illusion. The spreadsheet software does not, by default, place any significance on the fact that the values in row 6 all correspond to the time period from midnight to 1:00 a.m. Every cell in the spreadsheet is free to take any value regardless of which row or column it resides in.

This problem can be seen by looking at the time values in column A. To the human observer, it is clear that this column of values (apart from the first three rows) corresponds to time intervals. However, the spreadsheet data model does not enforce any such constraint on the data, as the value in row 17 clearly shows. All of the values up to that point (rows 6 through 16) have been time values, but row 17 contains the value 7-8. To human eyes this is clearly the time period 7:00 a.m. to 8:00 a.m., but any software trying to read this column of values will almost certainly fail to make this intuitive leap.

This particular problem is a feature of this particular data set, but the general problem pertaining to all spreadsheets is that the flexible value-per-cell data model allows this sort of thing to happen and the consequence is that additional data cleaning is necessary before the raw data in the spreadsheet can be used for analysis.

An example of where the lack of association between values that lie on the same row of a spreadsheet becomes dangerous is when people make use of the sorting facilities that are provided by most spreadsheet software. For example, it is a simple matter in most spreadsheet software to sort the values within a column, but doing this without simultaneously sorting all other columns in a data set leads to complete corruption of the data set.

Another major disadvantage of storing data in a spreadsheet is that the data are then only accessible using spreadsheet software. This is partially alleviated by the fact that some spreadsheet software is ubiquitous and the fact that all spreadsheet software can export spreadsheets in many different formats, including as plain text files. However, it is still necessary to have the spreadsheet software in order to read the data in first place, before being able to export it in a new format.

This is more of a problem for some spreadsheet software packages than others. For example, Microsoft Excel spreadsheets are stored using a closed proprietary format, making it essential to have Excel in order to access the

data.[15] By contrast, the Open Office Calc software uses the Open Document format, which is an open standard format based on XML. This means that Calc documents can be accessed by a number of different software packages; it is even possible (if dangerous) to view and modify Calc documents using a text editor.

A final disadvantage with some spreadsheet software is that, at least in some major spreadsheet software, there are fixed limits on the number of columns and on the number of rows in a spreadsheet, which means that some data sets simply cannot be stored in a spreadsheet format.

Many of the dangers associated with spreadsheets boil down to the fact that spreadsheets are easy to use, which means that many people use them, and spreadsheets are very flexible and quite powerful, which means that, without careful discipline, many people use them quite poorly. This is not directly the fault of spreadsheet software, which is designed for uses far beyond simple data storage, but it is a caution that spreadsheets should be used with care and discipline.

7.6.3 Flashback: Spreadsheets and data entry

The convenient table-of-cells arrangement of spreadsheets, plus the ability to dictate the type of the data that is stored in each cell, makes spreadsheets a useful and popular tool for data entry. Data are automatically stored in separate columns and it is possible to set up validity checks as data are entered, which was one of the major reasons for discussing electronic forms back in Chapter 5.

The unstructured format of spreadsheets means that it is very easy to end up with a poor data entry scheme, so if data is to be entered directly into a spreadsheet, it is important to follow some basic guidelines:

Keep the spreadsheet design simple
: Use a separate column for each variable and include all values for each case on a single row. The layout of the spreadsheet should be decided *before* any data entry occurs. The simple spreadsheet interface makes it tempting to enter data in an ad hoc fashion, but such an approach will only lead to problems later on.

  A simple design will not be the most efficient way to store the data (in terms of the amount of memory required), but it will reduce the

[15] In practice, a number of different software products have managed to "reverse engineer" the Excel format so that Excel documents can, for example, be accessed using Open Office Calc.

chances of errors in the spreadsheet.

If efficiency of design is important, then a more sophisticated storage format, such as a database, should be considered (see Section 7.7).

Enter all data on one spreadsheet
: This is an extension of the keep-it-simple guideline and is important for being able to successfully transfer the data to other systems. It is tempting to split data across spreadsheets, especially when the data set contains many measurement variables or when a study or experimental design is multi-leveled. Some problems with this approach are that it requires great discipline to ensure that multiple sheets maintain the same structure and it is harder to work with multiple spreadsheets (compared to dealing with multiple plain text files). When the data become large or complex a database solution is much more appropriate (see Section 7.7).

  One exception to this rule is the inclusion of metadata. It makes sense to store metadata on a separate spreadsheet. This means that the metadata is kept in the same file as the raw data, but keeps the raw data format clean and simple.

Add validity checking
: It is possible to specify the type (format) of the data in a cell, but this does not force any checks on the value that is entered into a cell.

  Spreadsheets provide additional **validity checking** to enforce specific data types and even specific data ranges on the data being entered into a cell. These facilities should be used.

7.7 Databases

When a data set becomes very large, or even just very complex in its structure, the ultimate storage solution is a **database**.

7.7.1 Some terminology

The term "database" can be used generally to describe any collection of information. In this section, the term "database" means a **relational** database, which is a a collection of data that is organised in a particular way.

Other types of database exist, such as hierarchical databases, network databases, and, more recently, object-oriented databases and XML databases, but relational databases are by far the most common.

The actual physical storage mechanism for a database–whether binary files or text files are used, whether one file or many files are used—will not concern us. We will only be concerned with the high-level, conceptual organisation of the data and will rely on software to decide how best to store the information in files.

The software that handles the physical representation, and allows us to work at a conceptual level, is called a **database management system** (**DBMS**), or in our case, more specifically a *relational* database management system (**RDBMS**).

7.7.2 The structure of a database

A relational database consists of a set of **tables**, where a table is conceptually just like a plain text file or a spreadsheet: a set of values arranged in rows and columns. The difference is that there are usually several tables in a single database, and the tables in a database have a much more formal structure than a plain text file or a spreadsheet.

For example, below we have a simple database table containing information on books. The table has three columns—the ISBN of the book, the title of the book, and the author of the book—and four rows, with each row representing one book.

```
ISBN        title                       author
----------  --------------------------  ----------------
0395193958  The Hobbit                  J. R. R. Tolkien
0836827848  Slinky Malinki              Lynley Dodd
0393310728  How to Lie with Statistics  Darrell Huff
0908783116  Mechanical Harry            Bob Kerr
```

Each table in a database has a unique name and each column in a table has a unique name.

Each column in a database table also has a data type associated with it, so all values in a single column are the same sort of data. In the book database example, all three columns are text or character values. The ISBN is not stored as an integer because it is a sequence of 10 digits (as opposed to a decimal value). For example, if we stored the ISBN as an integer, we would lose the leading `0`.

Each table in a database has a **primary key**. The primary key must be unique for every row in a table. In the book table, the ISBN provides a perfect primary key because every book has a different ISBN.

It is possible to create a primary key by combining the values of two or more columns. This is called a **composite primary key**. A table can only have one primary key, but the primary key may be composed from more than one column.

A database containing information on books might also contain information on book publishers. Below we show another table *in the same database* containing information on publishers.

```
ID  name               city        country
--  -----------------  ----------  -----------
1   Mallinson Rendel   Wellington  New Zealand
2   W. W. Norton       New York    USA
3   Houghton Mifflin   Boston      USA
```

Tables within the same database are related to each other using **foreign keys**. These are columns in one table that specify a value from the primary key in another table. For example, we can relate each book in the `book_table` to a publisher in the `publisher_table` by adding a foreign key to the `book_table`. This foreign key consists of a column, `pub`, containing the appropriate publisher `ID`. The `book_table` now looks like this:

```
ISBN        title                       author            pub
----------  --------------------------  ----------------  ---
0395193958  The Hobbit                  J. R. R. Tolkien  3
0836827848  Slinky Malinki              Lynley Dodd       1
0393310728  How to Lie with Statistics  Darrell Huff      2
0908783116  Mechanical Harry            Bob Kerr          1
```

Notice that two of the books have the same publisher.

7.7.3 Data integrity

As shown in the previous section, databases have a much more formal structure than any other storage option that we have encountered so far. This is an important advantage because this structure enforces constraints on the data in a database, which means that there are checks on the accuracy and consistency of data that is stored in a database. In other words, databases ensure better **data integrity**.

For example, the database structure ensures that all values in a single column of a table are of the same data type (e.g., they are all numbers). It is

possible, when setting up a database, to enforce quite specific constraints on what values can appear in a particular column of a table. We will not discuss the creation of databases in this chapter, but Section 10.3 provides some information on this topic.

Another important structural feature of databases is the existence of foreign keys and primary keys. Database software will enforce the rule that a primary key must be unique for every row in a table and it will enforce the rule that the value of a foreign key must refer to an existing primary key value. This idea is discussed further on page 160.

All of these checks are helpful in reducing the number of errors in a data set.

7.7.4 Advantages and disadvantages

A database compared to a flat text file is like a Ferrari compared to a horse and cart: *much* more expensive!

A database is of course also far more sophisticated than a flat text file, not to mention faster, more agile, and so on, but the cost is worth keeping in mind because a database is not always the best option. It is also worth noting that the cost is not just the software—there are several open source (free) database management systems—there is also the cost of acquiring or hiring the expertise necessary to create, maintain, and interact with data stored in a database.

Databases tend to be used for large data sets because, for most DBMS, there is no limit on the size of a database. However, even when a data set is not enormous, there are advantages to using a database because the organisation of the data can improve accuracy and efficiency. In particular, databases allow the data to be organised in a variety of ways so that, for example, data with a hierarchical structure can be stored in an efficient and natural way. These issues will be discussed further in Section 7.7.6.

Databases are also advantageous because most DBMS provide advanced features that are far beyond what is provided by the software that is used to work with data in other formats (e.g., text editors and spreadsheet programs). These features include the ability to allow multiple people to access and even modify the data at once and advanced control over who has access to the data and who is able to modify the data.

7.7.5 Database notation

In the examples so far, we have only been seeing the *contents* of a database table. In the next section, on Database Design, it will be more important to describe the *structure* of a database table—the table **schema**. For this purpose, the contents of each row are not important; instead we are interested in the names of tables, the names of columns, which columns are primary keys, and which columns are foreign keys. The notation we will use is a simple text description, with primary keys and foreign keys indicated in square brackets. For example, these are the schema for the `publisher_table` and the `book_table` in the book database:

```
publisher_table  (ID [PK], name, city, country)

book_table  (ISBN [PK], title, author,
             pub [FK publisher_table.ID])
```

For a foreign key, the name of the table and the name of the column that the foreign key references are also described.

7.7.6 Database design

In this section, we will look at some issues relating to the design of databases. Although designing a database is not a very common task for a scientist, having an understanding of the concepts and tasks involved can be useful for several reasons:

Getting data out of a database:
When extracting information from a database (Chapter 9), it is necessary to have an appreciation of why a database contains more than one table, how the tables are structured, and how multiple tables are related to each other.

It's good for you:
The concepts of database design are useful for thinking in general about how to store data (see, in particular, Section 7.7.10). The ideas in this section can have a positive influence on how we store information in other formats, even plain text files.

This section provides neither an exhaustive discussion nor a completely rigorous discussion of database design. The importance of this section is to provide a basic introduction to some useful ideas and ways to think about data.

Data modelling

Data modelling is a more general task than designing a database. The aim is to produce a model of a system (a **conceptual data model**), which can then be converted to a database representation, but could alternatively be converted into an object-oriented program, or something else (all **logical** or **physical data models**).

A conceptual data model is usually constructed using some sort of pictorial or diagram language, such as Entity-Relationship Diagrams (ERDs) or the Unified Modeling Language (UML). These languages are beyond the scope of this book, but we will use some of the building blocks of these conceptual models to help guide us in our brief discussion of database design.

Entities, attributes, and relationships

One way to approach database design is to think in terms of entities and the relationships between them.

An **entity** is most easily thought of as a person, place, or physical object (e.g., a book), an event, or a concept. An **attribute** is a piece of information about the entity. For example, the title, author, and ISBN are all attributes of a book entity.

The simple rule for starting to design a database for storing a data set is that there should be a table for each entity in the data set and a column in that table for each attribute of the entity.

Rather than storing a data set as one big table of information, this rule suggests that we should use several tables, with information about different entities in separate tables. In the book example, there is information about at least two entities, books and publishers, so we have a separate table for each of these.

A **relationship** is an association between entities. For example, a publisher *publishes* books and a book *is published by* a publisher. Relationships are represented in a database by foreign key-primary key pairs, but the details depend on the **cardinality** of the relationship—whether the relationship is one-to-one (1:1), many-to-one (M:1), or many-to-many (M:N).

A book is published by exactly one publisher,[16] but a publisher publishes many books, so the relationship between books and publishers is many-to-one. This sort of relationship can be represented by placing a foreign key

[16]Every edition or variation of a book gets its own ISBN, so the same book contents may be published by several different publishers, but they will have different ISBNs.

in the table for books (the "many" side), which refers to the primary key in the table for publishers (the "one" side).

One-to-one relationships can be handled similarly to many-to-one relationships (it does not matter which table gets the foreign key), but many-to-many relationships are more complex.

In our book database example, we can identify another sort of entity: authors. This suggests that there should be another table for author information. For now, the table only contains the author's name, but other information, such as the author's age and nationality could be added.

```
author_table  (ID [PK], name)
```

What is the relationship between books and authors? An author can write several books and a book can have more than one author, so this is an example of a many-to-many relationship.

A many-to-many relationship can only be represented by creating a new table. For example, we can create a table that contains the relationship between authors and books. This table contains a foreign key that refers to the author table and a foreign key that refers to the book table. The representaion of book entities, author entities, and the relationship between them now consists of three tables like this:

```
author_table  (ID [PK], name)

book_table  (ISBN [PK], title,
             pub [FK publisher_table.ID])

book_author_table (ID [PK],
                   book [FK book_table.ISBN],
                   author [FK author_table.ID])
```

The contents of these tables for several books are shown below. The author table just lists the authors for whom we have information:

```
ID  name
--  ----------------
2   Lynley Dodd
5   Eve Sutton
```

The book table just lists the books that are in the database:

```
ISBN        title
----------  ------------------------------------
0908606664  Slinky Malinki
1908606206  Hairy Maclary from Donaldson's Dairy
0908606273  My Cat Likes to Hide in Boxes
```

The association between books and authors is stored in the `book_author_table`:

```
ID  book        author
--  ----------  ------
2   0908606664  2
3   1908606206  2
6   0908606273  2
7   0908606273  5
```

Notice that author `2` (Lynley Dodd) has written more than one book and book `0908606273` has more than one author (rows `6` and `7`).

Data integrity

Another reason for creating an additional table in a database is for the purpose of constraining the set of possible values for an attribute. For example, if the table of authors records the nationality of the author, it can be useful to have a separate table that contains the possible nationalities. The column in the author table then becomes a foreign key referring to the nationality table and, because a foreign key must match the value of the corresponding primary key (or be `NULL`), we have a check on the validity of the nationality in the author table.

The redesigned author table now looks like this:

```
author_table  (ID [PK], name,
               nationality [FK nationality_table.ID])

nationality_table (ID [PK], nationality)
```

Normalisation

Normalisation is a formal process of ensuring that a database satisfies a set of rules called **normal forms**. There are several of these rules, but we will only mention the first three. The proper definition of normalisation

depends on more advanced relational database concepts that are beyond the scope of this book, so the descriptions below are just to give a feel for how the process works.

First normal form:

All columns should be **atomic**, there should be no **duplicative columns** and every table must have a primary key.

The first part of this rule says that a column in a database table must only contain a single value. As an example, consider the following table for storing information about books:

```
title                         authors
----------------------------  ----------------------
Slinky Malinki                Lynley Dodd
My Cat Likes to Hide in Boxes Eve Sutton, Lynley Dodd
```

The first column of this table is acceptable because it just contains one piece of information: the title of the book. However, the second column is not atomic because it contains a *list* of authors for each book. For example, the book on the second row has two authors.

The second part of the rule says that a table cannot have two columns containing the same information. For example, in the following table, the columns `author1` and `author2` are duplicative columns.

```
title                         author1      author2
----------------------------  -----------  -----------
Slinky Malinki                Lynley Dodd  NULL
My Cat Likes to Hide in Boxes Eve Sutton   Lynley Dodd
```

The final part of the rule says that there must be a column in the table that has a unique value in every row (or it must be possible to combine several columns to obtain a unique value for every row). In other words, every table must have a primary key.

Second normal form:

All tables must be in first normal form *and* all columns in a table must relate to the *entire* primary key.

This rule formalises the idea that there should be a table for each entity in the data set. Consider the following table for storing information about books:

ISBN	author	price
0908606664	Lynley Dodd	30.00
1908606206	Lynley Dodd	30.00
0908606273	Lynley Dodd	25.00
0908606273	Eve Sutton	25.00

The primary key for this table is a combination of `ISBN` and `author` (each row of the table carries information about one author of a book). The `price` column relates to the `ISBN`; this is the price of the book. However, the `price` column does not relate to the `author`; this is not the price of the author!

The table needs to be split into two tables, one with the information about books and one with the information about authors.

When a new table is created for author information, it is vital that the new table has a link to the book table via some sort of foreign key (see the earlier discussion of the *relationships* between entities).

Third normal form:

All tables must be in second normal form *and* all columns in a table must relate *only* to the primary key (not to each other).

This rule further emphasizes the idea that there should be a separate table for each entity in the data set. Consider the following table for storing information about books:

ISBN	title	publisher	country
0395193958	The Hobbit	Houghton Mifflin	USA
0836827848	Slinky Malinki	Mallinson Rendel	NZ
0908783116	Mechanical Harry	Mallinson Rendel	NZ

The primary key of this table is the ISBN, which uniquely identifies a book. The `title` column relates to the book; this is the title of the book. The `publisher` column also relates to the book; this is the publisher of the book. However, the `country` column does not relate *directly* to the book; this is the country of the *publisher*. That obviously is information about the book—it is the country of the publisher of the book—but the relationship is indirect, through the publisher.

There is a simple heuristic that makes is easy to spot this sort of problem in a database table. Notice that the information in the `publisher` and `country` columns is identical for the books published by Mallinson Rendel. When two or more columns repeat the same information over and over, it is a sure sign that either second or third normal form is not being met.

In this case, the analysis of the table suggests that there should be a separate table for information about the publisher.

Again, it is important to link to the new table with a foreign key.

Normalizing a database is an effective and formalized way of achieving the design goals that we outlined previously: memory efficiency, improved accuracy (data integrity), and ease of maintenance.

Applying the rules of normalisation usually results in the creation of more tables in a database. The previous discussion of relationships should be consulted for making sure that any new tables are linked to at least one other table in the database.

7.7.7 Flashback: The DRY Principle

A well designed database will have the feature that each piece of information is stored only once. Less repetition of data values means that a well-designed database will usually require less memory than storing an entire data set in a naive single-table format. Less repetition also means that a well-designed database is easier to maintain and update, because a change only needs to be made in one location. Furthermore, there is less chance of errors creeping into the data set. If there are multiple copies of information, then it is possible for the copies to disagree, but with only one copy there can be no disagreements.

These ideas are an expression of the DRY principle from Section 2.5. A well-designed database is the ultimate embodiment of the DRY principle for data storage.

7.7.8 Case Study: The Data Expo (continued)

The Data Expo data set consists of seven atmospheric variables recorded at 576 locations for 72 time points (every month for 6 years), plus elevation data for each location (see Section 7.3.6).

The data were originally stored as 505 plain text files, where each file contains the data for one variable for one month. Figure 7.16 shows the first few lines from one of the plain text files.

As we have discussed earlier in this chapter, this simple format makes the data very accessible. However, this is an example where a plain text format is quite inefficent, because many values are repeated. For example, the

```
             VARIABLE : Mean TS from clear sky composite (kelvin)
             FILENAME : ISCCPMonthly_avg.nc
             FILEPATH : /usr/local/fer_dsets/data/
             SUBSET   : 24 by 24 points (LONGITUDE-LATITUDE)
             TIME     : 16-JAN-1995 00:00
              113.8W 111.2W 108.8W 106.2W 103.8W 101.2W 98.8W  ...
               27     28     29     30     31     32     33    ...
 36.2N / 51:  272.7  270.9  270.9  269.7  273.2  275.6  277.3  ...
 33.8N / 50:  279.5  279.5  275.0  275.6  277.3  279.5  281.6  ...
 31.2N / 49:  284.7  284.7  281.6  281.6  280.5  282.2  284.7  ...
 28.8N / 48:  289.3  286.8  286.8  283.7  284.2  286.8  287.8  ...
 26.2N / 47:  292.2  293.2  287.8  287.8  285.8  288.8  291.7  ...
 23.8N / 46:  294.1  295.0  296.5  286.8  286.8  285.2  289.8  ...
 ...
```

Figure 7.16: One of the plain text files from the original format of the Data Expo data set, which contains data for one variable for one month. The file contains information on latitude and longitude, which is repeated in every other plain text file in the original format (for each variable and for each month; in total, over 500 times).

longitude and latitude information for each location in the data set is stored in every single file, which means that that information is repeated over 500 times! That not only takes up more storage space than is necessary, but it also violates the DRY principle, with all of the negative consequences that follow.

In this section, we will consider how the Data Expo data set could be stored as a relational database.

To start with, we will consider what entities there are in the data set. In this case, the different entities that are being measured are relatively easy to identify. There are measurements on the *atmosphere*, and the measurements are taken at different *locations* and at different *times*. We have information about each time point (i.e., a date), we have information about each location (longitude and latitude and elevation), and we have several measurements on the atmosphere. This suggests that we should have three tables: one for locations, one for time points, and one for atmospheric measures.

We could also look at the data set from a normalisation perspective. We start with a single table containing all columns (only 7 rows shown):

date	lon	lat	elv	chi	cmid	clo	ozone	press	stemp	temp
1995-01-16	-56.25	36.25	0.0	25.5	17.5	38.5	298.0	1000.0	289.8	288.8
1995-01-16	-56.25	33.75	0.0	23.5	17.5	36.5	290.0	1000.0	290.7	289.8
1995-01-16	-56.25	31.25	0.0	20.5	17.0	36.5	286.0	1000.0	291.7	290.7
1995-01-16	-56.25	28.75	0.0	12.5	17.5	37.5	280.0	1000.0	293.6	292.2
1995-01-16	-56.25	26.25	0.0	10.0	14.0	35.0	272.0	1000.0	296.0	294.1
1995-01-16	-56.25	23.75	0.0	12.5	11.0	32.0	270.0	1000.0	297.4	295.0
1995-01-16	-56.25	21.25	0.0	7.0	10.0	31.0	260.0	1000.0	297.8	296.5

In terms of first normal form, all columns are atomic and there are no duplicative columns, and we can, with a little effort, find a (composite) primary key: we need a combination of `date`, `lon` (longitude), and `lat` (latitude) to get a unique value for all rows.

Moving on to second normal form, the column `elv` (elevation) immediately fails. The elevation at a particular location clearly relates to the longitude and latitude of the location, but it has very little to do with the date. We need a new table to hold the longitude, latitude, and elevation data.

The new table looks like this (only 7 rows shown):

lon	lat	elv
-56.25	36.25	0.0
-56.25	33.75	0.0
-56.25	31.25	0.0
-56.25	28.75	0.0
-56.25	26.25	0.0
-56.25	23.75	0.0
-56.25	21.25	0.0

This "location" is in third normal form. It has a primary key (a combination of longitude and latitude), and the `elv` column relates directly to that primary key.

Going back to the original table, the remaining columns of atmospheric measurements are all related to the primary key; the data in these columns represents an observation at a particular location at a particular time point.

Having split the data set into separate tables, we must make sure that the tables are linked to each other (at least indirectly), and in order to achieve this, we need to determine the relationships between the tables.

We have two tables, one representing atmospheric measurements, at various locations and times, and one representing information about the locations. What is the relationship between these tables? Each location (each row of

the location table) corresponds to several measurements, but each individual measurement (each row of the measurement table) corresponds to only one location, so the relationship is many-to-one.

This means that the table of measurements should have a foreign key that references the primary key in the location table. The design could be expressed like this:

```
location_table ( longitude [PK],
                 latitude [PK],
                 elevation )

measure_table ( date [PK],
                longitude [PK] [FK location_table.longitude],
                latitude [PK] [FK location_table.latitude],
                cloudhigh, cloudlow, cloudmid, ozone,
                pressure, surftemp, temperature )
```

Both tables have composite primary keys, the `measure_table` also has a composite foreign key (to match the composite primary key), and the `longitude` and `latitude` columns of the `measure_table` have roles in both the primary key and the foreign key.

This design is a reasonable one, but we will go a bit further because the `date` column deserves a little more consideration.

As mentioned elsewhere, dates can be tricky to work with. The dates have been entered into the database as text. They are in the ISO 8601 format, so that alphabetical order is chronological order. This makes it easy to sort or extract information from a contiguous set of dates (e.g., all dates after December 1998). However, it would be difficult to extract non-contiguous subsets of the data (e.g., all data from December for all years). This sort of task would be much easier if we had separate columns of month and year information. If we add these columns to the data set, we get a table like this (only 7 rows shown; not all atmospheric variables shown):

```
date        lon     lat    month    year  chi   cmid  clo   ozone
----------  ------  -----  -------  ----  ----  ----  ----  -----
1995-01-16  -56.25  36.25  January  1995  25.5  17.5  38.5  298.0
1995-01-16  -56.25  33.75  January  1995  23.5  17.5  36.5  290.0
1995-01-16  -56.25  31.25  January  1995  20.5  17.0  36.5  286.0
1995-01-16  -56.25  28.75  January  1995  12.5  17.5  37.5  280.0
1995-01-16  -56.25  26.25  January  1995  10.0  14.0  35.0  272.0
1995-01-16  -56.25  23.75  January  1995  12.5  11.0  32.0  270.0
1995-01-16  -56.25  21.25  January  1995  7.0   10.0  31.0  260.0
```

With these extra columns added, the table violates second normal form again. The `month` and `year` columns relate to the `date`, but have nothing to do with longitude and latitude. We must create a new table for the date information.

This new table consists of `date`, `month`, and `year`, and we can use `date` as the primary key. The relationship between this table and the original table is many-to-one (each date corresponds to many measurements, but each individual measurement was taken on a single date), so another foreign key is added to the original table to link the tables together. The new date table looks like this (only 7 rows shown):

```
date        month    year
----------  -------  ----
1995-01-16  January  1995
1995-02-16  Februar  1995
1995-03-16  March    1995
1995-04-16  April    1995
1995-05-16  May      1995
1995-06-16  June     1995
1995-07-16  July     1995
```

One possible final adjustment to the database design is to consider a surrogate auto-increment key as the primary key for the location table, because the natural primary key is quite large and cumbersome. This leads to a final design that can be expressed like this:

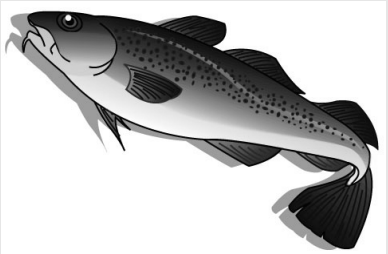
```
date_table ( date [PK],
             month, year )

location_table ( ID [PK],
                 longitude, latitude, elevation )

measure_table ( date [PK] [FK date_table.date],
                location [PK] [FK location_table.ID],
                cloudhigh, cloudlow, cloudmid, ozone,
                pressure, surftemp, temperature )
```

The final database, stored as an SQLite file, is a little over 2 MB in size, compared to 4 MB for the original plain text files.

7.7.9 Case study: Cod stomachs



Overfishing of Northwest Atlantic Cod lead to a collapse of the fishery over a four-year period in the early 1990s (the population dropped by 99.5%). Cod fishing is now completely banned in the region.[17]

One of the research projects conducted by Pêches et Océans Canada, the Canadian Department of Fisheries and Oceans (DFO), involves collecting data on the diet of Atlantic cod (Gadus morhua) in the Gulf of St.Lawrence, Eastern Canada.

Large quantities of cod are collected by a combination of research vessels and contracted fishing vessels and the contents of the cod stomachs are analysed to determine which species the cod have eaten.

Fish are collected when one or more ships set out on a fishing "trip". On a single trip, each ship performs several "sets", where each set consists of either a hooked line or a net being placed in the water and then subsequently being recovered (and checked for fish).

The primary experimental unit is a lump of something (a "prey item") found in a cod stomach. For each lump, the following variables are measured:

`prey_mass` The weight, in grams, of the prey item.

`prey_type` The species of the prey item. There are 24 different identified species, plus a special category `"Empty"` which is used when there are no prey items within the cod stomach, plus a general category `"Other"` into which all other species have been gathered.

`fish_id` A number identifying each fish *within a particular set.* In most cases, there are several records for a single fish because the fish has consumed more than one type of prey. A `fish_id` of `1` just means that the fish was the first fish measured from a particular set, so it is possible for different fish to share the same `fish_id`.

`fish_length` The length of a fish, in millimetres.

[17] Source: Carole Walsh Computer Graphics & Design
`http://carolewalsh.com/`
Used and redistributed with permission.

```
region ship_type ship_id trip set fish_id fish_length prey_mass prey_type
"SGSL"       "2"      NA "95"   3      30         530     27.06   "Other"
"SGSL"       "2"      NA "95"   3      30         530      1.47   "Other"
"SGSL"       "2"      NA "95"   3      30         530      4.77   "Other"
"SGSL"       "2"      NA "95"   3      31         490     34.11   "Other"
"SGSL"       "2"      NA "95"   3      31         490      0.17   "Other"
"SGSL"       "2"      NA "95"   3      31         490      2.27   "Other"
"SGSL"       "2"      NA "95"   3      32         470      0.52   "Other"
"SGSL"       "2"      NA "95"   3      32         470      0.21   "Other"
"SGSL"       "2"      NA "95"   3      32         470      1.7    "Other"
"SGSL"       "2"      NA "95"   3      33         480      1.97   "Other"
...
```

Figure 7.17: The first few lines of the Cod data set as a plain text file (there are 10,000 lines in total).

`set_num` A simple set label that is only unique to a particular ship on a particular trip. A `set` of `1` just means that the set was the first set by a particular ship on a particular trip.

`trip` A simple trip label that is only unique to a particulare trip region (see `region` below). Unfortunately, there is no guarantee that two ships with the same trip label were on the same trip.

`ship.type` For research vessels, this provides a unique identifier. For contract vessels, this just represents the type of fishing gear in use (`90` or type `99`), so different contract ships share the same ship type; these ships can be distinguished from one another by the `ship_id` (see below).

`ship_id` A unique identifier for ships owned by fisherman who were contracted to catch cod. The combination of `ship_type` and `ship_id` is unique for all ships.

`region` The region where the trip occurred: either Northern (`"NGSL"`) or Southern (`"SGSL"`) Gulf of St Lawrence.

We can start off by identifying a few simple entities within this data set. For example, thinking about the physical objects involved, there is clearly information about individual ships and information about individual fish, so we will have a `fish_table` and a `ship_table`.

It is abundantly clear that there are going to be several tables in the database; from a normalisation point of view, it is clear that we could not have a single table because there would be columns that do not relate to the primary key, or relate not only to the primary key, but also to each other;

not to mention that we would have trouble finding a primary key for one big table in the first place.

The ship table

For each ship we have one or two identification numbers. We need both numbers in order to uniquely identify each ship (different commercial vessels share the same `ship_type`), so we cannot use either variable on its own as a primary key. Furthermore, the `ship_id` for research vessels is missing (in database terms, the value will be `NULL`), which means that the `ship_id` variable cannot be used as part of the primary key. We will use an artificial, auto-increment key for this table.

```
ship_table  (ID [PK], ship_type, ship_id)
```

The fish table

For each fish, we have a numeric label and the length of the fish; we also have lots of information about what was in the stomach of the fish, but we will leave that until later. The `fish_id` label is not unique for each fish, so again we will use an auto-increment primary key.

```
fish_table  (ID [PK], fish_id, fish_length)
```

The prey item table

Another important physical entity in the data set is a prey item (a lump found in a fish stomach). We could have a `lump_table` where, for each lump, we have information about the species of the prey item and the weight of the lump. We will add an auto-increment variable to provide a unique identifier for each lump in the entire data set.

```
lump_table  (ID [PK], prey_mass, prey)
```

We will develop this table more a little later.

The prey type table

The `prey_type` variable is a good example where we might like to create a new table for validating the species entered in the `lump_table`. Another

reason for considering this approach is the possibility that we might be interested in a species that does not occur in the data set we have (but could conceivably occur in future studies). We could also use a `prey_table` to provide a mapping between the generic `Other` species category and individual species which have been grouped therein. We could use the species name itself as the primary key for the `prey_table`, but in terms of efficiency of storage, having a single integer identifier for each species requires less storage in the (large) `lump_table` than storing the full species label.

```
prey_table  (ID [PK], prey_type)
```

Relating lumps to prey and fish

The relationship between the `lump_table` and the `prey_table` is many-to-one, so we place a `prey` foreign key in the `lump_table`.

Lumps are also fairly obviously related to fish; each lump comes from exactly one fish and each fish can have several lumps. We also place a `fish` foreign key in the `lump_table`.

```
lump_table  (ID [PK], prey_mass,
             prey [FK prey_table.ID],
             fish [FK fish_table.ID]))
```

It is worth pointing out that, through the `lump_table`, we have resolved the many-to-many relationship between fish and prey.

Relating fish and ships

Now we come to the more complicated part of modelling the cod data set. How are fish and ships related to each other? And how do we bring in the other information in the data set (region, trip, and set)? At this point, thinking about physical entities does not help us much; we need to think in terms of the events involved in the data collection instead.

Initially, the situation does not look too bad; each fish was caught by exactly one ship (and each ship caught many fish). However, the process was not that simple. Each fish was caught in exactly one set (one check of the net or hooks) and each set occurred on exactly one trip. However, some trips involved several ships and some ships conducted more than one trip. There is another many-to-many relationship lurking within the data. To resolve this, we will focus on the fishing sets.

The set table

For each set we have a label, `set_num`. The set occurred on exactly one trip, so we can include the label for the trip.[18] The set was performed by exactly one ship, so we can include a foreign key to the ship table. This resolves the many-to-many relationship between ships and trips. Finally, we include information about the region. This is expanded into a separate table, which allows us to provide a more expansive description of each region. The original region code makes a nice primary key and because it is fairly compact, will not result in too much inefficiency in terms of space. An auto-increment variable provides a unique identifier for each set.

```
region_table  (region [PK], name)

set_table  (ID [PK], set_num, trip,
            fish [FK fish_table.ID],
            region [FK region_table.region])
```

7.7.10 Flashback: Database design and XML design

In Section 7.4.4 we discussed some basic ideas for deciding how to represent a data set in an XML format. The ideas of normalisation express very similar ideas, just in a more formal manner. In fact, there is often a simple correspondence between database designs and XML designs for the same data set.

As a rough guideline, a database table corresponds to a set of XML elements of the same type. Each row of the table will correspond to a single XML element, with each column of values recorded as a separate attribute within the element. The caveats about when attributes cannot be used still apply (see page 135).

Simple one-to-one or many-to-one relationships can be represented in XML by nesting several elements (the many) within another element (the one). More complex relationships cannot be solved by nesting, but with the correspondence of `ID` attributes to primary keys and `IDREF` attributes to foreign keys, it is possible to emulate relationships between entities via XML elements that are not nested.

[18] If we had more information on trips, such as a date, we might split the trip information into a separate table.

7.7.11 Case study: The Data Expo (continued)

The Data Expo data set consists of several atmospheric measurements taken at many different locations and at several time points. A database design that we developed for storing these data consisted of three tables: one for the location data, one for the time data, and one for the atmospheric measurements (see Section 7.7.8). The database schema is reproduced below for easy reference.

```
date_table ( date [PK],
             month, year )

location_table ( ID [PK],
                 longitude, latitude, elevation )

measure_table ( date [PK] [FK date_table.date],
                location [PK] [FK location_table.ID],
                cloudhigh, cloudlow, cloudmid, ozone,
                pressure, surftemp, temperature )
```

We can translate this database design into an XML document design very simply, by creating a set of elements for each table, with attributes for each column of data. For example, the fact that there is a table for location information implies that we should have `location` elements, with an attribute for each column in the database table. The data for the first few locations is represented like this in a database table:

```
lon     lat    elv
------  -----  ---
-56.25  36.25  0.0
-56.25  33.75  0.0
-56.25  31.25  0.0
```

The same data could be represented in XML like this:

```
    <location longitude="-56.25" latitude="36.25"
              elevation="0.0" />
    <location longitude="-56.25" latitude="33.75"
              elevation="0.0" />
    <location longitude="-56.25" latitude="31.25"
              elevation="0.0" />
```

7.7.12 Database software

Every different database software product has its own format for storing the database tables on disk, which means that data stored in a database is only accessible via one specific piece of software.

This means that, if we are given data stored in a particular database format, we are forced to use the corresponding software. Something that slightly alleviates this problem is the existence of a standard language for querying databases. We will meet this language, SQL, in the next chapter.

If we are in the position of storing information in a database ourselves, there are a number of fully-featured open source database management systems to choose from. PostgreSQL and MySQL are very popular options, though they require some investment in resources and expertise to set up because they have separate client and server software components. SQLite is much simpler to set up and use, especially for a database that only requires access by a single person on a single computer.

PostgreSQL
`http://www.postgresql.org/`

MySQL
`http://www.mysql.com/`

SQLite
`http://www.sqlite.org/`

The major proprietary database systems include Oracle, Microsoft SQL Server, and Microsoft Access. The default user interface for these software products is based on menus and dialogs so they are beyond the scope and interest of this book. Nevertheless, in all of these, as with the default interfaces for the open source database software, it is possible to write computer code to access the data. Writing these **data queries** is the topic of the next chapter.

7.8 Further reading

"Modern Database Management"
by Jeffrey A. Hoffer, Mary Prescott, and Fred McFadden
7th edition (2004) Prentice Hall.

Comprehensive text book treatment of databases and associated technologies. Has more of a business focus. Includes advanced topics way beyond the scope of this book.

Summary

Simple text data is stored using 1 byte per character. Integers are stored using 2 or 4 bytes and real values typically use 4 or 8 bytes.

There is a limit to the size of numbers that can be stored digitally and *for real values there is a limit on the precision with which values can be stored.*

Plain text files are the simplest data storage solution, with the advantage that they are simple to use, work across different computer platforms, and work with virtually any software. The main disadvantage to plain text files is their lack of standard structure, which means that software requires human input to determine where data values reside within the file. Plain text files are also generally slower and larger than other data storage options.

CSV (commas-separated values) files offer the most standardised plain text format.

XML is a language that can be used for marking up data. XML files are plain text, but provide structure that allows software to automatically determine the location of data values within the file (XML files are self-describing).

Binary file formats tend to provide smaller files and faster access speeds. The disadvantage is that data stored in a binary format can only be accessed using specific software.

Spreadsheets are ubiquitous, flexible, and easy to use. However, they lack structure so should be used with caution.

Databases are sophisticated, but relatively complex. They are useful for storing very large or very complex data sets, but require specific software and much greater expertise.

8
XML Reference

XML (the eXtensible Markup Language) is a data description language that can be used for storing data. It is particularly useful as a format for sharing information between different software systems.

8.1 XML syntax

The first line of an XML document should be a declaration that this is an XML document, including the version of XML being used.

```
<?xml version="1.0"?>
```

An XML document consists entirely of XML **elements**. An element usually consists of a start **tag**, an end tag, with plain text content or other XML elements in between.

A start tag is of the form <*elementName*> and an end tag has the form </*elementName*>.

The start tag may include **attributes** of the form *attrName*="*attrValue*". The attribute value must be enclosed within double-quotes.

The names of XML elements and XML attributes are case-sensitive.

It is also possible to have an empty element, which consists of a single tag, with attributes. In this case, the tag has the form <*elementName* />.

The following code shows examples of XML elements. The second example is an empty element with two attributes.

```
<filename>ISCCPMonthly_avg.nc</filename>

<case date="16-JAN-1994" temperature="278.9" />
```

XML elements may have other XML elements as their content. An XML element must have a single **root element**, which contains all other XML elements in the document.

A comment in XML is anything between the delimiters `<!--` and `-->`.

For the benefit of human readers, the contents of an XML element are usually indented. However, white space is preserved within XML so this is not always possible when including plain text content.

In XML code, certain characters, such as the greater-than and less-than signs, have special meanings. Escape sequences, such as `&gt;` and `&lt;`, must be used to obtain the corresponding literal character within plain text content. A special syntax is provided for escaping an entire section of plain text content for the case where many such special characters are included. Any text between the delimiters `<![CDATA[` and `]]>` is treated as literal.

8.2 Document Type Definitions

An XML document that obeys the rules of the previous section is described as **well-formed**.

It is also possible to specify additional rules for the structure and content of an XML document, via a **schema** for the document. If the document is well-formed and also obeys the rules given in a schema, then the document is described as **valid**.

The Document Type Definition language (DTD) is a language for describing the schema for an XML document. DTD code consists of **element declarations** and **attribute declarations**.

8.2.1 Element declarations

An element declaration should be included for every different type of element that will occur in an XML document. Each declaration describes what content is allowed inside a particular element. An element declaration is of the form:

```
<!ELEMENT elementName elementContents>
```

The *elementContents* can be one of the following:

`EMPTY`
: The element is empty.

`(#PCDATA)`
: The element may contain plain text.

`ANY`
: The element may contain anything (other elements, plain text, or both).

(*childName*)
: The element must contain exactly one *childName* element.

(*childName*`*`)
: The element may contain zero or more *childName* elements.

(*childName*`+`)
: The element must contain one or more *childName* elements.

(*childName*`?`)
: The element must contain zero or one *childName* elements.

(*childName*`?`)
: The element must contain zero or one *childName* elements.

(*childA*`,`*childB*)
: The element must contain exactly one *childA* element and exactly one *childB* element.

(*childA*`|`*childB*)
: The element must contain either exactly one *childA* element or exactly one *childB* element.

(`#PCDATA|`*childA*`|`*childB*)`*`
: The element may contain zero or more occurences of plain text, *childA* elements and *childB* elements.

8.2.2 Attribute declarations

An attribute declaration should be included for every different type of element that can have attributes. The declaration describes which attributes an element may have, what sort of values the attribute may take, and whether the attribute is optional. An attribute declaration is of the form:

```
<!ATTLIST elementName
    attrName attrType attrDefault
    ...
>
```

The *attrType* controls what value the attribute can have. It can have one of the following forms:

`CDATA`
: The attribute can take any value.

`ID`
: The value of this attribute must be unique for all elements of this type in the document (i.e., a unique identifier). This is similar to a primary key in a database table.

`IDREF`
: The value of this attribute must be the value of some other element's `ID` attribute. This is similar to a foreign key in a database table.

(*option1*|*option2*)
: This provides a list of the possible values for the attribute. This is a good way to limit an attribute to only valid values (e.g., only `"male"` or `"female"` for a `gender` attribute).

The *attrDefault* either provides a default value for the attribute or states whether the attribute is optional or required (i.e., must be specified). It can have one of the following forms:

value
: This is the default value for the attribute.

`#IMPLIED`
: The attribute is optional. It is valid for elements of this type to contain this attribute, but it is not required.

`#REQUIRED`
: The attribute is required so it must appear in all elements of this type.

8.2.3 Including a DTD

A DTD can be included directly within an XML document or the DTD can be located within a separate file and just referred to from the XML document.

The DTD information is included within a `DOCTYPE` definition following the XML declaration. An inline DTD has the form:

```
<!DOCTYPE rootElementName [
    ... DTDcode ...
]>
```

An external DTD stored in a file called `file.dtd` would be referred to as follows:

```
<!DOCTYPE rootElementName SYSTEM "file.dtd">
```

8.3 Further reading

The W3C XML 1.0 Specification
`http://www.w3.org/TR/2006/REC-xml-20060816/`
The formal and official definition of XML. Quite technical.

The w3schools XML Tutorial
`http://www.w3schools.com/xml/`
Quick, basic tutorial-based introduction to XML.

The w3schools DTD Tutorial
`http://www.w3schools.com/dtd/`
Quick, basic tutorial-based introduction to DTDs.

The w3schools XML Schema Tutorial
`http://www.w3schools.com/schema/`
Quick, basic tutorial-based introduction to XML Schema.

9
Data Queries

Having stored information in a particular data format, how do we get it back out again? How easy is it to access the data? The answer naturally depends on which data **format** we are dealing with.

For data stored in plain text files, it is very easy to find software that can read the files, although the software may have to be provided with additional information about the structure of the files—where the data values reside within the file.

For data stored in binary files, the main problem is finding software that is designed to read the specific binary format. Having done that, the software does all of the work of extracting the appropriate data values. This is an all or nothing scenario; we either have software to read the file, in which case data extraction is trivial, or we do not have the software, in which case we can do nothing. This scenario includes data stored in spreadsheets, though in that case the likelihood of having appropriate software is much higher.

Another factor that determines the level of difficulty involved in retrieving data from storage is the **structure** of the data within the data format. We also need to consider the fact that, for the purposes of analysing the data, we generally need to be able to extract the data in a rectangular arrangement, with a row per case and columns for each variable. This is the format required for most statistical software programs.

For plain text files, the data structure is typically a row of values for each case and a column of values for each variable in the data set, so the data are extracted in a desirable arrangement quite naturally.

For binary files, the structure of the stored data can be arbitrarily complex. We saw one such example, netCDF, in Section 7.5.1. Dealing with these structures requires knowledge of the relevant binary format, so must be dealt with on a case-by-case basis. In the case of netCDF, a solution is demonstrated later in Section 11.6.5.

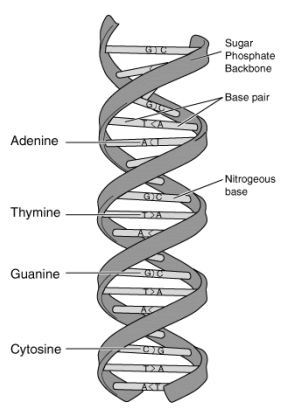
The structure of data within XML files and within relational databases can be relatively complex. Within XML files, the data may be represented as attributes within several nested elements and data that is stored in a database may be spread across several tables.

However, standard technologies exist to allow complete information to be extracted from within XML documents and relational databases, no matter how complex their structure.

The main focus of this chapter is the **Structured Query Language** (**SQL**), the language for extracting information from relational databases. We will also touch on **XPath** for XML documents.

To begin with, we will look at an example of data retrieval from a relational database. As with previous introductory examples, the focus at this point is not so much on the computer code itself as it is on the concepts involved and on what sorts of tasks we are able to achieve.

9.1 Case study: The Human Genome



Deoxyribonucleic Acid.[1]

The typical *homo sapiens* has 46 chromosomes.

The chromosomes come in pairs, with 22 pairs of *autosomes* where the pair consists of two copies of the same chromosome (one from the mother and one from the father), and one pair of sex chromosomes, where each chromosome can be either an X chromosome or a Y chromosome. This means that there are 24 distinct types of chromosome.

Each chromosome is a long strand of DNA, consisting of two long molecules that wind around each other in the famous double-helix formation.

[1]Image source: Wikimedia Commons
`http://commons.wikimedia.org/wiki/Image:ADN.jpg`
This image is in the Public Domain.

The two molecules in a DNA strand are linked together by "base pairs", like rungs on a (twisted) ladder. Each base pair is a combination of two molecules with only two possible variations: either adenine (A) connected to thymine (T), or guanine (G) connected to cytosine (C). In humans, chromosomes are quite long strands of DNA, with each chromosome containing millions of these base pairs.

Each strand of DNA can be characterised by the order of the base molecules along the strand. A chromosome can be represented by this order written as a series of letters, using the common abbreviations of the base pairs, like this: AAAGTCTGAC. This is called a DNA sequence or genetic sequence.

The Human Genome Project[2] was established in 1990 to determine the complete sequence of the entire human genome; the DNA sequence for all 24 distinct chromosomes, representing over 3 billion base pairs. This project is now complete and the human genome is publicly available for genetic researchers to explore.

The human genome data set is obviously large, not just because it contains a sequence 3 billion characters long, but also because it contains information on the genes, the important sub-sequences of the genome, plus information on many other important chemical sub-structures. All of this extra information also makes the data set complex in its structure. All of which means that this is an ideal data set for storing in a **database**.

The Ensembl Project[3] provides genomic data, including the human genome, in many different formats, including as MySQL databases. Furthermore, the Ensembl Project provides anonymous network access to their database server so that anyone can explore the genomic data.

As a simple exercise, we will attempt to extract from this database the opening sequence of the human genome—the first few characters on chromosome 1 of the typical human.

The first observation that we can make is that accessing this information is not as simple as opening a plain text file. The information is stored in a database, so we need appropriate computer tools to get access to the data. The tool that we will discuss in this chapter is the **Structure Query Language** (**SQL**), a standard language for extracting information from a database.

SQL is a standard language for interacting with database management systems. In this case, we are dealing with a MySQL DBMS, so we start a MySQL client and connect to the Ensembl server with a statement like

[2] `http://www.ornl.gov/sci/techresources/Human_Genome/home.shtml`
[3] `http://www.ensembl.org/index.html`

this:[4]

```
mysql --host=ensembldb.ensembl.org --user=anonymous
```

The second observation that we can make about these data is that people wanting to use the information in the database are unlikely to want to access the entire data set at once. It is more common to require only a subset of the data. The part of SQL that we will focus on in this chapter is the part of the language that allows querying the data in order to extract subsets.

The data on the human genome is provided as a database called `homo_sapiens_core_46_36h`.[5]

From within a MySQL session, we can select the database we are interested in as shown below:

```
mysql> use homo_sapiens_core_46_36h;
```

The human genome database is quite large and complex and contains a total of 75 tables, but we will just consider a subset of the tables that deals with the DNA sequence data. The tables, and the relationships between them are described below:

```
dna ( seq_region_id [PK] [FK seq_region.seq_region_id],
      sequence )
```

This table contains DNA sequences. Each row contains a DNA sequence for a *sequence region*, which is a small part of a chromosome. For each DNA sequence, there is the sequence data itself, and a code that uniquely identifies the sequence region that this DNA sequence is from. Only DNA sequences for small sequence regions are stored in this table. Data for large sequences, such as entire chromosomes, must be built up by combining smaller sequences together (see the `assembly` table below and Figure 9.1).

```
seq_region ( seq_region_id [PK],
             name,
             coord_system_id [FK coord_system.coord_system_id],
             length )
```

This table provides information about sequence regions. For each

[4] This is from a Linux shell on a machine with access to the internet.

[5] This was the latest version of the core DNA sequencing information at the time of writing.

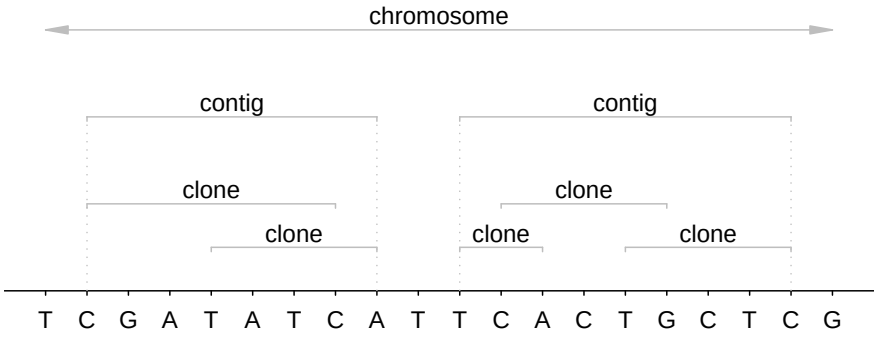


Figure 9.1: Clones are small segments of DNA, contigs are larger sequences constructed from overlapping clones, and chromosomes are constructed by combining contigs (which may also overlap). *Diagram NOT drawn to scale!*

sequence region, there is a unique identifier, a region name (for example, the sequence region corresponding to the first chromosome is called `"1"`), a *coordinate system* (see the `coord_system` table below), and a length (the number of base pairs in the DNA sequence for this sequence region). This table contains information about both large and small sequence regions. For the smaller regions, there are corresponding DNA sequences in the `dna` table, but for larger regions, there is no direct DNA sequence information. Larger sequence regions correspond to combinations of smaller sequence regions and many regions overlap with each other.

```
coord_system ( coord_system_id [PK],
               name, version, rank, attrib )
```

This table contains information about all of the possible *types* of sequence region in the data set. Larger sequence regions correspond to entire chromosomes, but smaller sequence regions represent just a piece of a chromosome (called a *clone* or a *contig*).

```
assembly ( asm_seq_region_id [PK]
                             [FK seq_region.seq_region_id],
           cmp_seq_region_id [PK]
                             [FK seq_region.seq_region_id],
           asm_start [PK],
           asm_end [PK],
           cmp_start [PK],
           cmp_end [PK],
           ori [PK] )
```

An "assembly" describes how one sequence region can be constructed from other sequence regions. For example, how a contig can be constructed from several clones, and how a chromosome can be constructed from contigs. Each row of this table describes which pieces of a larger (`asm`) sequence region correspond to which pieces of a smaller (`cmp`) sequence region. The `ori` column describes the order of the data in the sequence region (some sequence regions are read right-to-left).

We will now try to extract information from the tables in this database. We are interested in chromosome 1, and the following code extracts information on the sequence region for chromosome 1:

```
mysql> SELECT * FROM seq_region WHERE name = '1';

+---------------+------+-----------------+-----------+
| seq_region_id | name | coord_system_id | length    |
+---------------+------+-----------------+-----------+
|        226034 | 1    |              17 | 247249719 |
|       1965892 | 1    |             101 | 245522847 |
+---------------+------+-----------------+-----------+
```

This is an example of a simple query that extracts information from just one table in the database. We only require some of the rows from this table, so we use a **condition**, `name = '1'`, to specify the subset that we want.

This result tells us that there are almost 250 million base pairs on chromosome 1, but why are there are two sequence regions for chromosome 1? A quick look at the `coord_system` table shows that these are two different versions of the data:

```
mysql> SELECT coord_system_id, name, version
    ->     FROM coord_system;

+-----------------+-------------+---------+
| coord_system_id | name        | version |
+-----------------+-------------+---------+
|              17 | chromosome  | NCBI36  |
|              15 | supercontig | NULL    |
|               4 | contig      | NULL    |
|              11 | clone       | NULL    |
|             101 | chromosome  | NCBI35  |
+-----------------+-------------+---------+
```

This example again just obtains data from one of the tables in the data set. In this case we get all of the rows, but we only ask for some of the columns.

We will use the newer version of the data, `NCBI36`, which, based on the previous query, is sequence region `226034`. A better way to express that is to say that we want the sequence region that has the name `"1"` in table `seq_region` and that has the version `"NCBI36"` in the `coord_system` table.

The entire DNA sequence for chromosome 1 is not stored in one row of the `dna` table (that table only has DNA sequences for smaller sequence regions), so we need to find out which sequence regions can be combined to make up the whole chromosome. We will focus on the sequence regions that cover the start of chromosome 1.

The code below performs this task by getting information from the `seq_region` table, to get the right name, from the `coord_system` table, to get the right version, and from the `assembly` table, to get the relevant sequence regions.

```
mysql> SELECT asm_seq_region_id AS asm_id,
    ->        cmp_seq_region_id AS cmp_id,
    ->        asm_start AS asm_1, asm_end AS asm_2,
    ->        cmp_start AS cmp_1, cmp_end AS cmp_2,
    ->        ori
    -> FROM assembly INNER JOIN seq_region
    ->     ON asm_seq_region_id = seq_region_id
    ->   INNER JOIN coord_system
    ->     ON seq_region.coord_system_id =
    ->        coord_system.coord_system_id
    -> WHERE seq_region.name = '1' AND
    ->       coord_system.version = 'NCBI36' AND
    ->       asm_start = 1;

+--------+---------+-------+--------+-------+--------+-----+
| asm_id | cmp_id  | asm_1 | asm_2  | cmp_1 | cmp_2  | ori |
+--------+---------+-------+--------+-------+--------+-----+
| 226034 |  162952 |     1 |    616 | 36116 |  36731 |  -1 |
| 226034 | 1965892 |     1 | 257582 |     1 | 257582 |   1 |
| 226034 |  225782 |     1 | 167280 |     1 | 167280 |   1 |
+--------+---------+-------+--------+-------+--------+-----+
```

This task highlights a common complication when extracting data from a database. Because a properly-designed database usually consists of more than one table, anything but trivial queries on the data involve combining information from more than one table. The ability to perform this sort of **database join** is an important part of SQL.

The important result from our query is the information that the sequence region with identifier `162952` covers the first 616 base pairs on chromosome 1. This information is provided by characters 36116 to 36731 within that sequence region and these characters have to be read from right to left (the value of `ori` is `-1`).

A quick check of the `seq_region` table shows that these characters are the last 616 in this sequence region (the region only contains 36731 characters):

```
ATTGGGATTGGGATTGGGATTGGGATTGGGATTGGGATTGGGATTGGGATTGGGATTGG
ATTGGGATTGGGATTGGGATTGGGATTGGGATTGGGATTGGGATTGGGTTGGGATTGGG
TTGGGATTGGGATTGGGATTGGGATTGGGGATTGGGATTGGGATTGGGATTGGGATTGG
TTGGGATTGGGATTGGGATTGGGATTGGGATTGGGATTGGGATTGGGATTGGGGATTGG
ATTGGGATTTGGGATTTGGGATTGGGATTGGGATTGGGATTGGGATTGGGGTTGGGGTT
GGGTTGGGGTTGGGGTTGGGGTTGGGATTGGGGATTGGGATTGGGATTGGGATGGGATT
GGATTGGGATTGGGATTGGGATTGGGATTGGGGATTGGGGATTGGGATTGGGATTGGGA
TGGGATTGGGATTGGGATTGGGGATTGGGATTGGGATTGGGATTGGGAGCGCCATGGGA
TCGGCCGGGCGGGCGGGCCCAGACTGGACTCCTCTTGACACGAGGCGGAAGTCTCATGG
GGCTTTAGACACGTCTCCTGTTGCGTCGAGGCGGGAGCGCCACGAGAGGCCCAGACACG
CTCCTCTTGCGTTGAG
```

Figure 9.2: The first 616 characters of the DNA sequence on chromosome 1 of the human genome.

```
mysql> SELECT seq_region_id, name, length
    -> FROM seq_region
    -> WHERE seq_region_id = 162952;

+---------------+--------------------+--------+
| seq_region_id | name               | length |
+---------------+--------------------+--------+
|        162952 | AP006221.1.1.36731 |  36731 |
+---------------+--------------------+--------+
```

The final step is to extract the DNA sequence for this sequence region. The SQL code for the task is shown below, but because the result is quite large, the result is shown separately in Figure 9.2.

```
SELECT * FROM dna WHERE seq_region_id = 162952;
```

This example demonstrates that SQL code allows us to write expressions to extract specific rows and specific columns from one or more tables within a database. We will now look in more detail at the syntax of such SQL queries and how more complex queries can be constructed.

9.2 SQL

SQL consists of three components:

The Data Definition Language (DDL)

This is concerned with the creation of databases and the specification of the structure of tables and of constraints between tables. This part of the language is used to specify the data types of each column in each table, which column(s) make up the primary key for each table, and how foreign keys are related to primary keys. We will not discuss this part of the language in this chapter, but some mention of it is made in Section 10.3.

The Data Control Language (DCL)
This is concerned with controlling access to the database—*who* is allowed to do *what* to *which* tables. This part of the language is the domain of database administrators and need not concern us.

The Data Manipulation Language (DML)
This is concerned with getting data into and out of a database. We will focus on data extraction in this chapter, but Section 10.3 includes information about some of the other features.

In this section, we are only concerned with one particular statement within the DML part of SQL: the `SELECT` statement for extracting values from tables within a database.

9.2.1 The `SELECT` statement

Everything we do in this section will be a variation on the `SELECT` statement of SQL, which has the following basic form:

```
SELECT columns
    FROM tables
    WHERE row_condition
```

This will extract the specified *columns* from the specified *tables*, but only include the rows for which the *row_condition* is true.

9.2.2 Case study: The Data Expo (continued)

The Data Expo data set consists of seven atmospheric measurements at locations on a 24 by 24 grid averaged over each month for six years (72 time points). The elevation (height above sea level) at each location is also included in the data set (see Section 7.3.6 for more details).

The data set was originally provided as 505 plain text files, but the data can also be stored in a database with the following structure (see Section 7.7.8).

```
location_table ( ID [PK], longitude, latitude, elevation )

date_table ( date [PK], month, year )

measure_table ( date [PK] [FK date_table.date],
                location [PK] [FK location_table.ID],
                cloudhigh, cloudlow, cloudmid,
                ozone, pressure,
                surftemp, temperature )
```

The `location_table` contains all of the geographic locations at which measurements were taken, and includes the elevation at each location. The `date_table` contains all of the dates at which measurements were taken. This table also includes the text form of each month and the numeric form of the year. These have been split out to make it easier to perform queries based on months or years. The full dates are stored using the ISO 8601 format so that alphabetical ordering gives chronological order.

The `measure_table` contains all of the atmospheric measurements for all dates and locations. Locations are represented by simple ID numbers, referring to the appropriate complete information in the `location_table`.

The goal for contestants in the Data Expo was to summarize the important features of this data set. In this section, we will perform some straightforward explorations of the data in order to demonstrate a variety of simple SQL statements.

A basic first step in data exploration is just to view the univariate distribution of each measurement variable. The following code extracts all air pressure values from the database using a very simple SQL query that extracts extracting all rows from the `pressure` column of the `measure_table`.

```
SQL> SELECT pressure FROM measure_table;
```

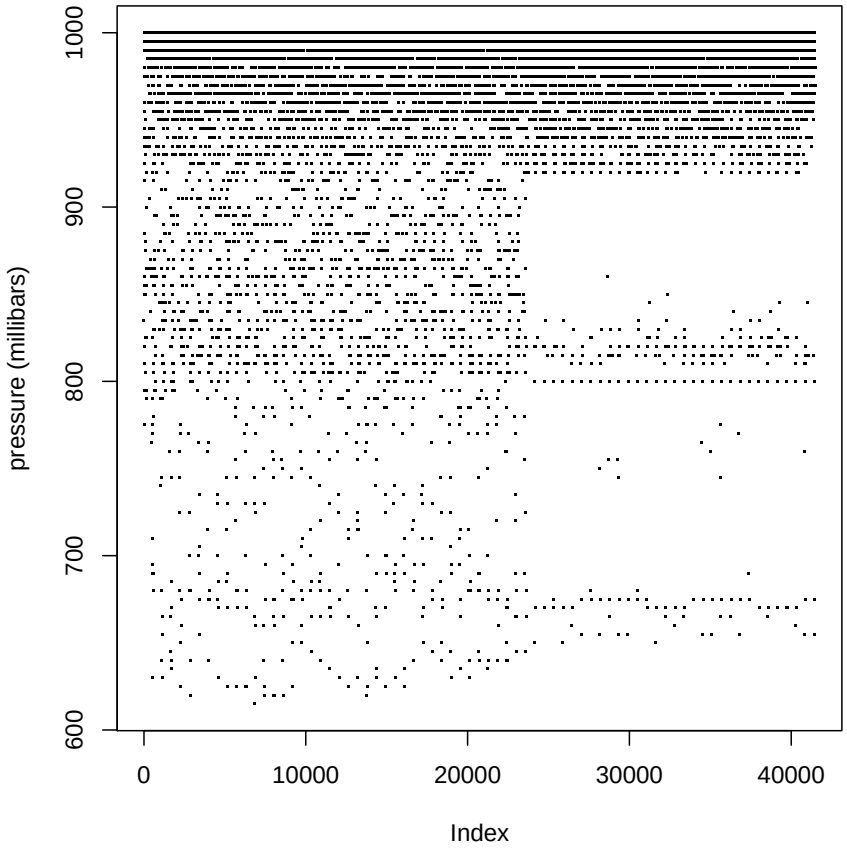


Figure 9.3: All of the air pressure measurements from the 2006 JSM Data Expo.

```
pressure
--------
835.0
810.0
810.0
775.0
795.0
915.0
...
```

The result contains 41472 ($24 \times 24 \times 72$) values, so only the first few are shown here. Figure 9.3 shows a plot of all of the pressure values.

The resolution of the data is immediately apparent; the pressure is only recorded to the nearest multiple of 5. However, the more striking feature is the change in the spread of the second half of the data. NASA have confirmed that this change is real, but unfortunately have not been able to give an explanation for why it occurred.

An entire column of data from the `measure_table` in the Data Expo database represents measurements of a single variable at *all* locations for *all* time periods. One interesting way to "slice" the Data Expo data is to look at the values for a *single* location over all time periods. For example, how does surface temperature vary over time at a particular location?

The following code shows a slight modification of the previous query to obtain a different column of values, `surftemp`, and to only return some of the rows from this column. The `WHERE` clause limits the result to rows for which the `location` column has the value `1`.

```
SQL> SELECT surftemp
         FROM measure_table
         WHERE location = 1;


surftemp
--------
272.7
282.2
282.2
289.8
293.2
301.4
...
```

Again, the result is too large to show all values, so only the first few are shown. Figure 9.4 shows a plot of all of the values.

The interesting feature here is that we can see a cyclic change in temperature, as we might expect, with the change of seasons.

The order of the rows in a database table is not guaranteed. This means that, whenever we extract information from a table, we should be explicit about the order that we want for the results. This is achieved by specifying an `ORDER BY` clause in the query. For example, the following SQL statement extends the previous one to ensure that the temperatures for location `1` are returned in chronological order.

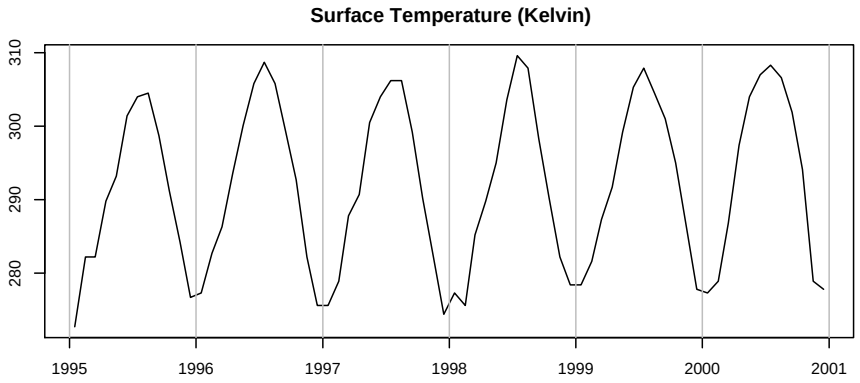


Figure 9.4: All of the surface temperature measurements from the 2006 JSM Data Expo for location `1`. Vertical grey bars mark the change of years.

```
SQL> SELECT surftemp
         FROM measure_table
         WHERE location = 1
         ORDER BY date;
```

The `WHERE` clause can use other comparison operators besides equality. As a trivial example, the following code has the same result as the previous example by specifying that we only want rows where the `location` is less than `2` (the only location value less than two is the value `1`).

```
SQL> SELECT surftemp
         FROM measure_table
         WHERE location < 2
         ORDER BY date;
```

It is also possible to combine several conditions together within the `WHERE` clause, using logical operators `AND`, to specify conditions that must *both* be true, and `OR`, to specify that we want rows where *either* of two conditions are true. As an example, the following code extracts the surface temperature for two locations. In this example, we include the `location` and `date` columns in the result to show that rows from both locations (for the same date) are being included in the result.

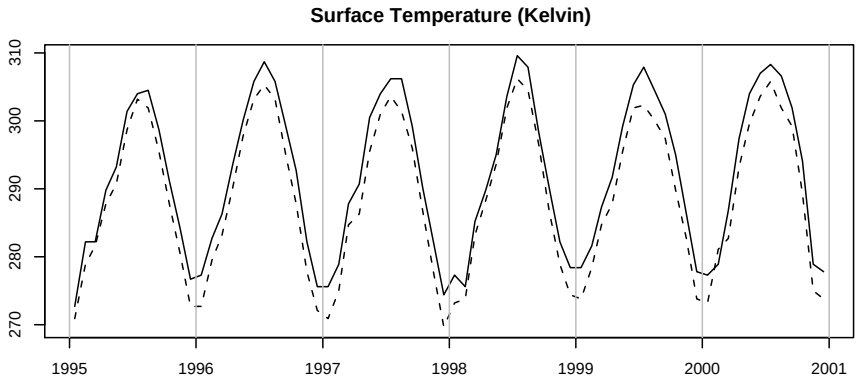


Figure 9.5: All of the surface temperature measurements from the 2006 JSM Data Expo for locations `1` (solid line) and `2` (dashed line). Vertical grey bars mark the change of years.

```
SQL> SELECT location, date, surftemp
         FROM measure_table
         WHERE location = 1 OR
               location = 2
         ORDER BY date;

location  date        surftemp
--------  ----------  --------
1         1995-01-16  272.7
2         1995-01-16  270.9
1         1995-02-16  282.2
2         1995-02-16  278.9
1         1995-03-16  282.2
2         1995-03-16  281.6
...
```

Figure 9.5 shows a plot of all of the values, which shows a clear trend of lower temperatures overall for location `2` (the dashed line).

As well as extracting raw values from a column, it is possible to calculate derived values by combining columns with simple arithmetic operators or by using a **function** to produce the sum or average of the values in a column.

As a simple example, the following code calculates the average surface temperature value across all locations and across all time points. It crudely

represents the average surface temperature of Central America for the years 1995 to 2000.

```
SQL> SELECT AVG(surftemp) avgtemp
         FROM measure_table;
```

```
avgtemp
-------
296.231
```

One extra feature to notice about this example SQL query is that it defines a **column alias**, `avgtemp`, for the column of averages. This alias can be used within the SQL query, which can make the query easier to type and easier to read. The alias is also used in the presentation of the result. Column aliases will become more important as we construct more complex queries later in the section.

An SQL function will produce a single overall value for a column of a table, but what is usually more interesting is the value of the function for subgroups within a column, so the use of functions is commonly combined with a `GROUP BY` clause, which results in a separate summary value computed for subsets of the column.

For example, instead of investigating the trend in surface temperature over time just for location `1`, we could look at the change in the surface temperature over time averaged across all locations.

The following code performs this query and Figure 9.6 shows a plot of the result. The use of `GROUP BY` clause means that we get an average surface temperature value for each different value in the `date` column.

```
SQL> SELECT date, AVG(surftemp) avgtemp
         FROM measure_table
         GROUP BY date
         ORDER BY date;
```

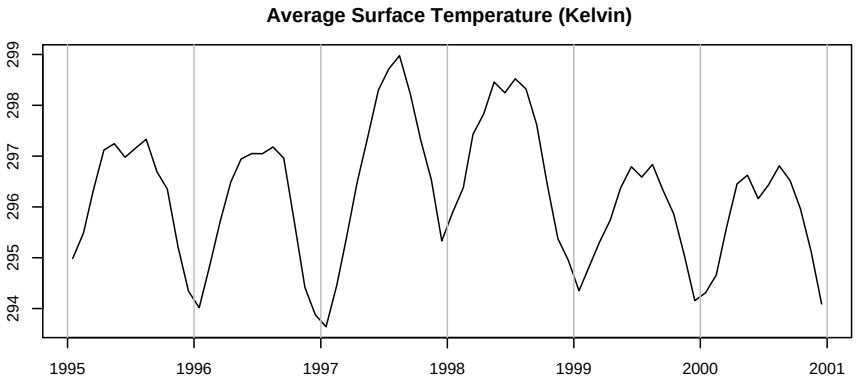


Figure 9.6: The surface temperature measurements from the 2006 JSM Data Expo averaged across all locations for each time point. Vertical grey bars mark the change of years.

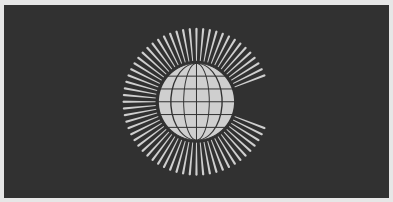
```
date        avgtemp
----------  -------
1995-01-16  294.985
1995-02-16  295.486
1995-03-16  296.315
1995-04-16  297.119
1995-05-16  297.244
1995-06-16  296.976
...
```

Overall, it appears that 1997 and 1998 were generally warmer years in Central America.

9.2.3 Querying several tables: Joins

As demonstrated in the previous section, database queries from a single table are quite straightforward. However, most databases consist of more than one table, and most interesting database queries involve extracting information from more than one table. In database terminology, most queries involve some sort of **join** between two or more tables.

9.2.4 Case study: Commonwealth swimming



The Commonwealth of Nations ("The Commonwealth") is a collection of 53 countries (most of which are former British colonies).[6]

New Zealand sent a team of 18 swimmers to the Melbourne 2006 Commonwealth Games, 10 women and 8 men. The results from their races are recorded in a database with the following structure.

`swimmer_table ( ID [PK], first, last )`

This table has one row for each swimmer and contains the first and last name of each swimmer. Each swimmer also has a unique numeric identifier.

`distance_table ( length [PK] )`

This table defines the set of valid swim distances: `50`, `100`, `200`, `400`.

`stroke_table ( ID [PK], stroke )`

This table defines the set of valid swim strokes: breaststroke (`Br`), freestyle (`Fr`), butterfly (`Bu`), backstroke (`Ba`), and individual medley (`IM`).

`gender_table ( ID [PK], gender )`

This table defines the valid genders: male (`M`) and female (`F`).

`stage_table ( stage [PK] )`

This table defines the valid types of race that can occur: heats (`heat`), semifinals (`semi`), and finals (`final`).

[6]Image source: Wikimedia Commons
`http://commons.wikimedia.org/wiki/Image:Flag_of_the_Commonwealth_of_Nations.svg`
This image is in the public domain.

```
result_table ( swimmer [PK] [FK swimmer_table.ID],
               distance [PK] [FK distance_table.length],
               stroke [PK] [FK stroke_table.ID],
               gender [PK] [FK gender_table.ID],
               stage [PK] [FK stage_table.stage],
               time, place )
```

This table contains information on the races swum by individual swimmers. Each row specifies a swimmer and the type of race (distance, stroke, gender, and stage). In addition, the swimmer's time and position in the race (`place`) are recorded.

As an example of the information stored in this database, the following code shows that the swimmer with an `ID` of `1` is called Zoe Baker.

```
SQL> SELECT * FROM swimmer_table
         WHERE ID = 1;

ID  first  last
--  -----  -----
1   Zoe    Baker
```

Notice the use of `*` in this query to denote that we want *all* columns from the table in our result.

The following code shows that Zoe Baker swam in three races, a heat, a semifinal and the final of the women's 50m breaststroke, and she came 4th in the final in a time of 31 minutes and 27 seconds.

```
SQL> SELECT * FROM result_table
         WHERE swimmer = 1;

swimmer  distance  stroke  gender  stage  time   place
-------  --------  ------  ------  -----  -----  -----
1        50        Br      F       Final  31.45  4
1        50        Br      F       Heat   31.7   4
1        50        Br      F       Semi   31.84  5
```

9.2.5 Cross joins

The most basic type of database join, upon which all other types of join are based, is a **cross join**. The result of a cross join is the cartesian product of

the rows of one table with the rows of another table. In other words, row 1 of table 1 is paired with each row of table 2, then row 2 of table 1 is paired with each row of table 2, and so on. If the first table has n_1 rows and the second table has n_2 rows, the result of a cross join is a table with $n_1 \times n_2$ rows.

The simplest way to create a cross join is simply to perform an SQL query on more than one table. As an example, the following code performs a cross join on the `distance_table` and `stroke_table` in the swimming database to generate all possible combinations of swimming stroke and event distance.

```
SQL> SELECT length, stroke
         FROM distance_table, stroke_table;
```

```
length  stroke
------  ----------------
50      Breaststroke
50      Freestyle
50      Butterfly
50      Backstroke
50      IndividualMedley
100     Breaststroke
100     Freestyle
100     Butterfly
100     Backstroke
100     IndividualMedley
200     Breaststroke
200     Freestyle
200     Butterfly
200     Backstroke
200     IndividualMedley
400     Breaststroke
400     Freestyle
400     Butterfly
400     Backstroke
400     IndividualMedley
```

A cross join can also be obtained more explicitly using the `CROSS JOIN` syntax as shown below (the result is exactly the same as for the code above).

```
SELECT length, stroke
    FROM distance_table CROSS JOIN stroke_table;
```

9.2.6 Inner joins

An **inner join** is the most common way of combining two tables. In this sort of join, only "matching" rows are extracted from two tables. Typically, a foreign key in one table is matched to the primary key in another table.

Conceptually, an inner join is a cross join, with only the desired rows retained. In practice, DBMS software analyses queries and obtains the result more directly in order to use less time and less computer memory.

9.2.7 Case study: The Data Expo (continued)

In order to demonstrate inner joins, we will return to the Data Expo database (see 9.2.2). In a previous query, we saw that the surface temperatures for location `1` were consistently higher than the surface temperatures for location `2`. Why is this? One obvious possibility is that location `1` is closer to the equator than location `2`. To test this hypothesis, we will repeat the earlier query, but add information about the latitude and longitude of the two locations.

To do this we need information from two tables. The surface temperature comes from the `measure_table` and the longitude/latitude information comes from the `location_table`.

```
SQL> SELECT longitude, latitude, location, date, surftemp
         FROM measure_table mt, location_table lt
         WHERE mt.location = lt.ID AND
               (location = 1 OR
                location = 2)
         ORDER BY date;

longitude  latitude  location  date        surftemp
---------  --------  --------  ----------  --------
-113.75    36.25     1         1995-01-16  272.7
-111.25    36.25     2         1995-01-16  270.9
-113.75    36.25     1         1995-02-16  282.2
-111.25    36.25     2         1995-02-16  278.9
-113.75    36.25     1         1995-03-16  282.2
-111.25    36.25     2         1995-03-16  281.6
...
```

The result shows that the difference between the locations is in fact that location `2` is to the East of location `1` (further inland in the US southwest)

because the `longitude` for location `2` is less negative (less westward) then the `longitude` for location `1`.

The most important feature of this code is the fact that it obtains information from two tables.

```
FROM measure_table mt, location_table lt
```

In order to merge the information from the two tables in a sensible fashion, we must specify how rows from one table are matched up with rows from the other table. In most cases, this means specifying that a foreign key from one table matches the primary key in the other table, which is precisely what has been done in this case.

```
WHERE mt.location = lt.ID
```

Another feature of this code is that it makes use of **table aliases**. For example, `mt` is defined as an alias for the `measure_table`. This makes it easier to type the code and can also make it easier to read the code.

The `WHERE` clause also demonstrates the combination of three separate conditions: the condition matching the foreign key of the `measure_table` to the primary key of the `location_table`, *plus* two conditions that limit our attention to just two values of the `location` column. The use of parentheses is important; without them, we would get rows from location `2` where the foreign key of the `measure_table` does *not* match the primary key of the `location_table`.

Another way to specify this join uses a different syntax that places all of the information about the join in the `FROM` clause of the query. The following code produces exactly the same result as before, but uses the key words `INNER JOIN` between the tables that are being joined and follows that with a specification of the columns to match `ON`. Notice how the `WHERE` clause is much simpler in this case.

```
SQL> SELECT longitude, latitude, pressure, elevation
         FROM measure_table mt INNER JOIN location_table lt
             ON mt.location = lt.ID
         WHERE location = 1 OR
               location = 2
         ORDER BY date;
```

We will now consider a major summary of temperature values: what is the average temperature *per year*, across all locations *on land* (above sea level)?

In order to answer this question, we need to know the temperatures from the `measure_table`, the elevation from the `location_table`, and the years from the `date_table`. In other words, we need to combine all three tables together.

This situation is one reason for using the `INNER JOIN` syntax shown above, because it naturally extends to joining more than two tables, and provides a way for us to control the order in which the tables are joined. The following code performs the desired query (see Figure 9.7).

```
SQL> SELECT year, AVG(surftemp) avgtemp
         FROM measure_table mt
              INNER JOIN location_table lt
                  ON mt.location = lt.ID
              INNER JOIN date_table dt
                  ON mt.date = dt.date
         WHERE elevation > 0
         GROUP BY year;

year  avgtemp
----  -------
1995  295.380
1996  295.006
1997  295.383
1998  296.416
1999  295.258
2000  295.315
```

This result shows only 1998 as warmer than other years; the higher temperatures for 1997 that we saw in Figure 9.6 must be due to higher temperatures over water.

9.2.8 Sub-queries

It is possible to use an SQL query within another SQL query, in which case the nested query is called a **sub-query**.

As a simple example, consider the problem of extracting the date at which the maximum surface temperature occurred. It is simple enough to determine the maximum surface temperature.

```
SQL> SELECT MAX(surftemp) max FROM measure_table;
```

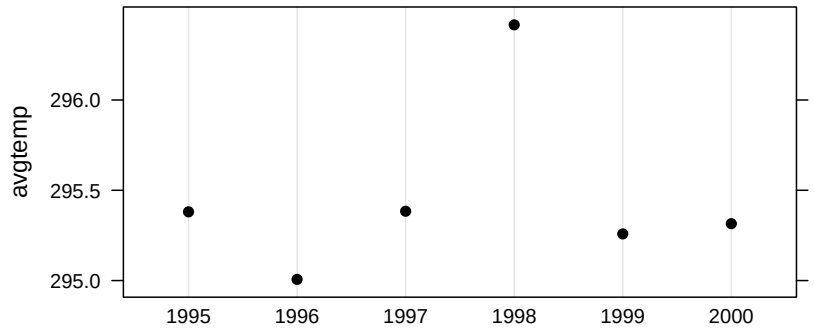


Figure 9.7: The average temperature over land per year. (Data from the the 2006 JSM Data Expo.)

```
max
-----
314.9
```

However, it is not so easy to report the date along with the maximum temperature because it is not valid to mix aggregated columns with non-aggregated columns. For example, the following SQL code will either trigger an error message or produce an incorrect result.

```
SELECT date, MAX(surftemp) max FROM measure_table;
```

The column `date` returns 41472 values, but the column `MAX(surftemp)` only returns 1 value.

The solution is to use a subquery as shown below.

```
SQL> SELECT date, surftemp temp
         FROM measure_table
         WHERE surftemp = ( SELECT MAX(surftemp)
                                FROM measure_table );

date        temp
----------  -----
1998-07-16  314.9
1998-07-16  314.9
```

The query that calculates the maximum surface temperature is inserted within brackets as a subquery within the `WHERE` clause. The outer query returns only the rows of the `measure_table` where the surface temperature is equal to the maximum.

The maximum temperature occured in July 1998 at two different locations.

9.2.9 Outer Joins

Another type of table join is the **outer join**, which differs from an inner join by including additional rows in the result.

9.2.10 Case study: Commonwealth swimming (continued)

The results of New Zealand's swimmers at the 2006 Commonwealth Games in Melbourne are stored in a database consisting of six tables: a table of information about each swimmer, separate tables for the distance of a swim event, the type of swim stroke, the gender of the swimmers in an event, and the stage of the event (heat, semifinal, or final), plus a table of results for each swimmer in different events.

In Section 9.2.5 we saw how to generate all possible combinations of distance and stroke in the swimming database using a cross join between the `distance_table` and the `stroke_table`. There are four possible distances and five different strokes, so the cross join produces 20 different combinations.

We will now take that cross join and combine it with the table of race results using an inner join. Our goal is to summarize the result of all races for a particular distance/stroke combination by calculating the average time from such races. The following code performs this inner join, with the results ordered from fastest event on average to slowest event on average.

```
SQL> SELECT length, st.stroke style, AVG(time) avg
         FROM distance_table dt
             CROSS JOIN stroke_table st
             INNER JOIN result_table rt
                 ON dt.length = rt.distance AND
                    st.ID = rt.stroke
         GROUP BY length, st.stroke
         ORDER BY avg;
```

```
length  style             avg
------  ----------------  -----
50      Freestyle         26.16
50      Butterfly         26.40
50      Backstroke        28.04
50      Breaststroke      31.29
100     Butterfly         56.65
100     Freestyle         57.10
100     Backstroke        60.55
100     Breaststroke      66.07
200     Freestyle         118.6
200     Butterfly         119.0
200     IndividualMedley  129.5
200     Backstroke        129.7
400     IndividualMedley  275.2
```

The result suggests that freestyle and butterfly events tend to be faster on average than breaststroke and backstroke events, but the feature of the result that we need to focus on for the current purpose is that this result has only 13 rows.

What has happened to the remaining 7 combinations of distance and stroke? The answer is that, for inner joins, a row is not included in the result if either of the two columns being matched has the value `NULL`. In this case, some rows from the cross join, which produced all possible combinations of distance and stroke, have been dropped from the result because some of these combinations do not appear in the `result_table`. For example, no New Zealand swimmer competed in the 400m Freestyle.[7]

This feature of inner joins is not always desirable and can produce misleading results, which is why an **outer join** is sometimes necessary. The idea of an outer join is to retain in the final result rows where one or other of the columns being compared has a `NULL` value.

The following code repeats the previous query, but instead of using `INNER JOIN` it uses `LEFT JOIN` to perform a **left outer join** so that all distance/stroke combinations are reported, even though there is no average time information available for some combinations. The result now includes all possible combinations of distance and stroke, with a `NULL` value where there is no matching `avg` value from the `result_table`.

[7]Some other events are missing because they simply do not exist. For example, there is no 50m Individual Medley at the Commonwealth Games!

```
SQL> SELECT length, st.stroke style, AVG(time) avg
         FROM distance_table dt
             CROSS JOIN stroke_table st
             LEFT JOIN result_table rt
                 ON dt.length = rt.distance AND
                    st.ID = rt.stroke
         GROUP BY length, st.stroke;


length  style             avg
------  ----------------  -----
50      Backstroke        28.04
50      Breaststroke      31.29
50      Butterfly         26.40
50      Freestyle         26.16
50      IndividualMedley  NULL
100     Backstroke        60.55
100     Breaststroke      66.07
100     Butterfly         56.65
100     Freestyle         57.10
100     IndividualMedley  NULL
200     Backstroke        129.7
200     Breaststroke      NULL
200     Butterfly         119.0
200     Freestyle         118.6
200     IndividualMedley  129.5
400     Backstroke        NULL
400     Breaststroke      NULL
400     Butterfly         NULL
400     Freestyle         NULL
400     IndividualMedley  275.2
```

The use of `LEFT JOIN` in this example is significant because it means that all rows from the original cross join are retained even if there is no matching row in the `result_table`. A different result would have been obtained if we had used `RIGHT JOIN` to perform a **right outer join** instead. In that case, all rows of the `result_table` (the table on the right of the join) would have been retained.

In this case, the result of a right outer join would be the same as using `INNER JOIN` because all rows of the `result_table` have a match in the cross join. This is not surprising because it is equivalent to saying that all swimming results came from events that are a subset of all possible combinations of event stroke and event distance.

It is also possible to perform a **full outer join**, in which case all rows from tables on *both* sides of the join are retained in the final result.

9.2.11 Self joins

It is useful to remember that database joins always begin with a cartesian product of the rows of the tables being joined (conceptually at least). The different sorts of database join are all just different subsets of a cross join. This makes it possible to answer questions that, at first sight, may not appear to be database queries.

For example, it is possible to join a table with itself—a so-called **self join**. The result is all possible combinations of the rows of a table, which can be used to answer questions that require comparing a column within a table to itself or to other columns within the same table.

9.2.12 Case study: The Data Expo (continued)

Consider the following question: at what locations and dates did the surface temperature at location 1 for January 1995 *reoccur*?

This question requires a comparison of one row of the `surftemp` column in the `measure_table` with the other rows in that column. The code below performs the query using a self join.

```
SQL> SELECT mt1.surftemp temp1, mt2.surftemp temp2,
           mt2.location loc, mt2.date date
        FROM measure_table mt1, measure_table mt2
        WHERE mt1.surftemp = mt2.surftemp AND
              mt1.date = '1995-01-16' AND
              mt1.location = 1;
```

```
temp1  temp2  loc  date
-----  -----  ---  ----------
272.7  272.7  1    1995-01-16
272.7  272.7  2    1995-12-16
272.7  272.7  3    1995-12-16
272.7  272.7  2    1996-01-16
272.7  272.7  3    1996-01-16
272.7  272.7  5    1996-12-16
272.7  272.7  5    1997-02-16
272.7  272.7  27   1997-12-16
272.7  272.7  29   1997-12-16
272.7  272.7  7    2000-12-16
272.7  272.7  8    2000-12-16
272.7  272.7  13   2000-12-16
272.7  272.7  14   2000-12-16
```

The temperature occurred again in neighbouring locations in December and January of 1995/1996 and again in several other locations in later years.

9.3 Other query languages

One of the reasons we needed to learn SQL is because information that is stored in a database has a complex structure. SQL provides a language that allows us to extract information from complex structures in a logical and predictable way, no matter what DBMS software was used to store the data. The other main reason was that databases tend to be large and we often only need to extract a subset of the data.

Data sets that are stored as XML also tend to be large and can have a complex structure, so there is also a need for a language to express subsets of XML documents.

There is a powerful language called **XQuery** for extracting specific sets of elements and attributes from an XML document, including facilities to structure the result.

A full discussion of XQuery is beyond the scope of this book, plus it is harder to find software that implements the language (compared to SQL). However, we will look briefly at a language that XQuery is built on. This language is called **XPath** and it provides a very flexible way to express subsets of an XML document.

```
<?xml version="1.0"?>
<temperatures>
    <variable>Mean TS from clear sky composite (kelvin)</variable>
    <filename>ISCCPMonthly_avg.nc</filename>
    <filepath>/usr/local/fer_dsets/data/</filepath>
    <subset>93 points (TIME)</subset>
    <longitude>123.8W(-123.8)</longitude>
    <latitude>48.8S</latitude>
    <case date="16-JAN-1994" temperature="278.9" />
    <case date="16-FEB-1994" temperature="280" />
    <case date="16-MAR-1994" temperature="278.9" />
    <case date="16-APR-1994" temperature="278.9" />
    <case date="16-MAY-1994" temperature="277.8" />
    <case date="16-JUN-1994" temperature="276.1" />

    ...

</temperatures>
```

Figure 9.8: The first few lines of the surface temperature at Point Nemo in an XML format.

9.3.1 XPath

An XPath **expression** specifies a subset of elements and attributes from within an XML document. We will look at the basic structure of XPath expressions via an example.

9.3.2 Case study: Point Nemo (continued)

Figure 9.8 shows the temperature data at Point Nemo in an XML format (this is a reproduction of Figure 7.7 for convenience).

The most basic XPath expressions consist of element names separated by forwardslashes. The following XPath selects the `temperatures` element from the XML document.

```
/temperatures
```

```
<temperatures>
    <variable>Mean TS from clear sky composite (kelvin)</variable>
    <filename>ISCCPMonthly_avg.nc</filename>
    <filepath>/usr/local/fer_data/data/</filepath>
    <subset>48 points (TIME)</subset>
    <longitude>123.8W(-123.8)</longitude>
 ...
```

More specifically, it selects the *root* element `temperatures`. If we want to select elements below the root element, we need to specify a complete path to those elements, or start the expression with a double-forwardslash. The following two expressions, both select all `case` elements from the XML document. In the first case, we specify `case` elements that are directly nested within the (root) `temperatures` element:

```
/temperatures/case
```

```
<case date="16-JAN-1994" temperature="278.9"/>
<case date="16-FEB-1994" temperature="280"/>
<case date="16-MAR-1994" temperature="278.9"/>
<case date="16-APR-1994" temperature="278.9"/>
<case date="16-MAY-1994" temperature="277.8"/>
<case date="16-JUN-1994" temperature="276.1"/>
 ...
```

The second approach selects `case` elements no matter where they are within the XML document.

```
//case
```

```
<case date="16-JAN-1994" temperature="278.9"/>
<case date="16-FEB-1994" temperature="280"/>
<case date="16-MAR-1994" temperature="278.9"/>
<case date="16-APR-1994" temperature="278.9"/>
<case date="16-MAY-1994" temperature="277.8"/>
<case date="16-JUN-1994" temperature="276.1"/>
 ...
```

Attributes may also be selected by specifying the appropriate name, preceded by an `@` character. The following example selects the `temperature` attributes from the `case` elements.

```
/temperatures/case/@temperature
```

```
temperature="278.9"
temperature="280"
temperature="278.9"
temperature="278.9"
temperature="277.8"
temperature="276.1"
...
```

Several separate paths may also be specified, separated by a vertical bar. This next XPath selects both `longitude` and `latitude` elements from anywhere within the XML document.

```
//longitude | //latitude
```

```
<longitude>123.8W(-123.8)</longitude>
<latitude>48.8S</latitude>
```

It is also possible to specify **predicates**, which are conditions that must be met for an element to be selected. These are placed within square brackets. In the following example, only `case` elements where the `temperature` attribute has the value `280` are selected.

```
/temperatures/case[@temperature=280]
```

```
<case date="16-FEB-1994" temperature="280"/>
<case date="16-MAR-1995" temperature="280"/>
<case date="16-MAR-1997" temperature="280"/>
```

We will demonstrate one example of the use of XPath expressions later in Section 11.6.4.

9.4 Further reading

The w3schools XPath Tutorial
`http://www.w3schools.com/xpath/`
Quick, basic tutorial-based introduction to XPath.

Summary

10
SQL Reference

The Structured Query Language (SQL) is a language for working with information that has been stored in a database.

SQL has three parts: the Data Manipulation Language (DML) concerns adding information to a database, modifying the information, and extracting information from a database; the Data Definition Lanuage (DDL) is concerned with the structure of a database (creating and destroying tables); and the Data Control Language (DCL) is concerned with administration of a database (deciding who gets what sort of access to which parts of the database).

This chapter is mostly focused on the `SELECT` command, which is the part of the DML that is used to extract information from a database, but other useful SQL commands are also mentioned briefly in Section 10.3.

10.1 SQL syntax

Column names and table names must all start with a letter and may include letters, digits, and the underscore character, `_`. These names may be case-sensitive if the underlying operating system is case-sensitive.

Numeric values are typed as usual and string values must be contained within single quotes. The escape sequence for a single quote (apostrophe) within a string is to type two single quotes.

The SQL key words are not case-sensitive, but it is traditional to write all key words in upper case.

White space is ignored, so long queries can (and should) be split across several lines.

Every SQL query must end with a semi-colon, `;`.

10.2 SQL queries

The basic format of an SQL query is this:

```
SELECT columns
    FROM tables
    WHERE row_condition
    ORDER BY order_by_columns
```

This will select the named *columns* from the specified *tables* and return all rows matching the *row_condition*.

The order of the rows in the result is based on the values in the *order_by_columns*.

10.2.1 Selecting columns

The special character * selects all columns, otherwise only those columns named are included in the result. If more than one column name is given, the column names must be separated by commas.

```
SELECT *
    ...

SELECT colname
    ...

SELECT colname1, colname2
    ...
```

The column name may be followed by a **column alias** and that alias can be used elsewhere in the query (e.g., in the `WHERE` clause.

```
SELECT colname colalias
    ...
```

If more than one table is included in the query, and the tables share a column with the same name, a column name must be preceded by the relevant table name, with a full stop in between.

```
SELECT tablename.colname
    ...
```

Functions and operators may be used to produce results that are calculated

from the column. The set of functions that is provided varies widely between DBMS, but the normal mathematical operators for addition, subtraction, multiplication, and division, plus a set of basic aggregation functions for maximum value (`MAX`), minimum value (`MIN`), summation (`SUM`), and arithmetic mean (`AVG`) should always be available.

```
SELECT MAX(colname)
    ...

SELECT colname1 + colname2
    ...
```

A column name can also be a constant value (number or string), in which case the value is replicated for every row of the result.

10.2.2 Specifying tables: the `FROM` clause

The `FROM` clause must contain at least one table and all columns specified in the query must exist in at least one of the tables in the the `FROM` clause.

If a single table is specified, then the result is all rows of that table, subject to any filtering applied by a `WHERE` clause. A table name may be followed by a **table alias**, in which case, the alias may be used anywhere else in the query.

```
SELECT colname
    FROM tablename
    ...

SELECT talias.colname
    FROM tablename talias
    ...
```

If two tables are specified, separated only by a comma, the result is all possible combinations of the rows of the two tables (a cartesian product). This is known as a **cross join**.

```
SELECT ...
    FROM table1, table2
    ...
```

An inner join is created from a cross join by specifying a condition so that only rows that have matching values are returned (typically using a foreign

key to match with a primary key). The condition may be specified within the `WHERE` clause, or as part of an `INNER JOIN` syntax as shown below.

```
SELECT ...
    FROM table1 INNER JOIN table2
        ON table1.primarykey = table2.foreignkey
    ...
```

An outer join extends the inner join by including in the result rows from one table that have no match in the other table. There are left outer joins (where rows are retained from the table named on the left of the join syntax), right outer joins, and full outer joins (where non-matching rows from both tables are retained).

```
SELECT ...
    FROM table1 LEFT OUTER JOIN table2
        ON table1.primarykey = table2.foreignkey
    ...
```

A self join is a join of a table with itself. This requires the use of table aliases.

```
SELECT ...
    FROM tablename alias1, tablename alias2
    ...
```

10.2.3 Selecting rows: the `WHERE` clause

By default, all rows from a table or from a combination of tables, are returned. However, if the `WHERE` clause is used to specify a condition, then only rows matching that condition will be returned.

Conditions may involve any column from any table that is included in the query. Conditions usually involve a comparison between a column and a constant value, or between two columns. Valid comparison operators include: equality (`=`), greater-than or less-than (`>`, `<`, or equal-to, `>=`, `<=`), and inequality (`!=` or `<>`).

```
SELECT ...
    FROM ...
    WHERE colname = 0;

SELECT ...
    FROM ...
    WHERE column1 > column2;
```

Complex conditions can be constructed by combining simple conditions with logical operators: `AND`, `OR`, and `NOT`. Parentheses should be used to make the order of evaluation explicit.

```
SELECT ...
    FROM ...
    WHERE column1 = 0 AND
          column2 != 0;

SELECT ...
    FROM ...
    WHERE NOT (stroke = 'IM' AND
               (distance = 50 OR
                distance = 100));
```

For the case where a column can match several possible values, the special `IN` keyword can be used to specify a range of valid values.

```
SELECT ...
    FROM ...
    WHERE column1 IN (value1, value2);
```

Comparison with string constants can be generalised to allow patterns using the special `LIKE` comparison operator. In this case, within the string constant, the underscore character, `_`, has a special meaning; it will match *any single character*. The percent character, `%`, is also special and it will match *any number* of characters *of any sort*.

```
SELECT ...
    FROM ...
    WHERE stroke LIKE '%stroke';
```

See Section 11.8.2 for more advanced pattern matching tools.

10.2.4 Sorting results: the `ORDER BY` clause

The order of the columns in the results of a query is based on the order of the column names in the query.

The order of the rows in a result is undetermined unless an `ORDER BY` clause is included in the query. The `ORDER BY` clause names the column to order results by followed by `ASC` for ascending and `DESC` for descending order.

```
SELECT ...
    FROM ...
    ORDER BY columnname ASC;
```

The results can be ordered by the values in several columns simply by specifying several column names, separated by commas. The results are ordered by the first column then, within identical values of the first column, rows are ordered by the second column, and so on.

```
SELECT ...
    FROM ...
    ORDER BY column1 ASC, column2 DESC;
```

10.2.5 Aggregating results: the `GROUP BY` clause

The aggregation functions `MAX`, `MIN`, `SUM`, and `AVG` (see Section 10.2.1) all return a single value from a column. If a `GROUP BY` clause is included in the query, aggregated values are reported for each unique value of the column specified in the `GROUP BY` clause.

```
SELECT column1, SUM(column2)
    FROM ...
    GROUP BY column1;
```

Results can be reported for combinations of unique values of several columns simply by naming several columns in the `GROUP BY` clause.

```
SELECT column1, column2, SUM(column3)
    FROM ...
    GROUP BY column1, column2;
```

The `GROUP BY` clause can include a `HAVING` sub-clause that works like the `WHERE` clause, but operates on the rows of aggregated results rather than

the original rows.

```
SELECT column1, SUM(column2) colalias
    FROM ...
    GROUP BY column1
        HAVING colalias > 0;
```

10.2.6 Sub-queries

The result of an SQL query may be used as part of a larger query. The sub-query is placed within parentheses, but otherwise follows the same syntax as a normal query.

Sub-queries are used in place of table names within the `FROM` clause and to provide comparison values within a `WHERE` clause.

```
SELECT column1
    FROM table1
    WHERE column1 IN
        ( SELECT column2
              FROM table2
                  ... );
```


10.3 Other SQL commands

This section deals with SQL statements that perform other common useful actions on a database.

We start with entering the data into a database.

Creating a table proceeds in two steps: first we must define the schema or structure of the table and then we can load rows of values into the table.

10.3.1 Defining tables

A table schema is defined using the `CREATE` statement.

Table 10.1: Common SQL data types.

Type	Description
`CHAR(`*n*`)`	Fixed-length text (*n* characters)
`VARCHAR(`*n*`)`	Variable-length text (maximum *n* characters)
`INTEGER`	Whole number
`REAL`	Real number
`DATE`	Calendar date

```
CREATE TABLE table
    (col1name col1type,
     col2name col2type)
    column_constraints;
```

This statement specifies the name of the table, the name of each column, and the data type to be stored in each column. A common variation is to add `NOT NULL` after the column data type to indicate that the value of the column can never be `NULL`. This must usually be specified for primary key columns.

The possible data types available depends on the DBMS being used, but some standard options are shown in Table 10.1.

The *column_constraints* are used to specify primary and foreign keys for the table.

```
CREATE TABLE table1
    (col1name col1type NOT NULL,
     col2name col2type)
    CONSTRAINT constraint1
        PRIMARY KEY (col1name)
    CONSTRAINT constraint2
        FOREIGN KEY (col2name)
        REFERENCES table2 (table2col);
```

The primary key constraint specifies which column or columns make up the primary key for the table. The foreign key constraint specifies which column or columns act as a foreign key *and* which (primary-key) column in which other table that foreign key refers to.

As concrete examples, code in Figure 10.1 shows the SQL code that was used to create the database tables `location_table` and `measurements_table` for

```
CREATE TABLE location_table
    (ID    INTEGER NOT NULL,
     longitude   REAL,
     latitude    REAL,
     elevation   REAL,
     CONSTRAINT location_table_pk PRIMARY KEY (ID));

CREATE TABLE measure_table
    (location    INTEGER NOT NULL,
     date        INTEGER NOT NULL,
     cloudhigh   REAL,
     cloudmid    REAL,
     cloudlow    REAL,
     ozone       REAL,
     pressure    REAL,
     surftemp    REAL,
     temperature REAL,
     CONSTRAINT measure_table_pk
         PRIMARY KEY (location, date),
     CONSTRAINT measure_location_table_fk
         FOREIGN KEY (location)
         REFERENCES location_table(ID),
     CONSTRAINT measure_date_table_fk
         FOREIGN KEY (date)
         REFERENCES date_table(date));
```

Figure 10.1: The SQL code used to define the table schema for storing the Data Expo data set in a relational database.

the Data Expo case study in Section 7.3.6.

The primary key of the `location_table` is the `ID` column. The (composite) primary key of the `measure_table` is a combination of the `location` and `date` columns and the `location` column also acts as a foreign key, referring to the `ID` column of the `location_table`.

10.3.2 Populating tables

Having generated the table schema, values are entered into the table using the `INSERT` statement.

```
INSERT INTO table VALUES
    (value1, value2);
```

Most DBMS also provide their own syntax for reading data values into a table from an external (text) file.

10.3.3 Modifying data

10.3.4 Deleting data

The `DELETE` statement can be used to remove specific rows from a table.

```
DELETE FROM table
    WHERE row_condition;
```

The `DROP` statement can be used to completely remove, not only the contents of a table, but the entire table schema so that the table no longer exists within the database.

```
DROP TABLE table;
```

In some DBMS, it is even possible to "drop" an entire database (and all of its tables).

```
DROP DATABASE database;
```

These statements should obviously be used with extreme caution.

10.4 Further reading

SQL: The Complete Reference
by James R. Groff and Paul N. Weinberg
2nd Edition (2002), McGraw-Hill.
Like the title says, a *complete* reference to SQL.

Using SQL
by Rafe Colburn
Special Edition (1999), Que.
Still a thorough treatment, but an easier read (more of a learning resource than a reference).

11
Data Crunching

In previous chapters, we have encountered a number of different computer languages for specific applications: HTML and CSS for web publishing and electronic forms, XML for data storage, and SQL for working with relational databases. In this chapter, we will look at a general purpose computer language; it is not designed for one specific task, but has facilities for conquering many different sorts of tasks.

As we might expect, a general purpose language will let us do a lot more than the specific languages can do, but this will come at a cost; we will need to learn a few more complex concepts and the general purpose language will not always do a job as well as the specific-purpose languages.

Many general-purpose languages exist, such as Perl, Python, and Ruby. The R language is used as the primary general-purpose language in this chapter because it is particularly well-suited to working with data.

11.1 Case study: The Population Clock

The Doomsday Clock symbolizes how close the world is to complete disaster. It currently stands at 5 minutes to midnight ...[1]

As early as the 1970s, Isaac Asimov was attempting to draw the eye of the general public to the looming problem of the overpopulation of Planet Earth. In several books and speeches[2] he dramatically pointed out that

[1] Image source: The Open Clip Art Library
`http://openclipart.org/people/ernes/ernes_orologio_clock.svg`
This image is in the public domain.

[2] For example, *The Future of Humanity: a Lecture by Isaac Asimov*, given at Newark College of Engineering November 8, 1974.
`http://www.asimovonline.com/oldsite/future_of_humanity.html`

the world population at the time stood at around 4 billion, but with the increasing rate of growth, it would be around 7 billion by the turn of the millenium. Asimov also pointed out that, one way or another, the growth of the human population *would* slow, it was just a matter of how messy the deceleration was. Nature and the limits of natural resources would do the job if necessary, but the least messy solution, he suggested, was voluntary birth control.

Asimov did not live to see the turn of the millenium, but he may have been pleased to see that his prediction was slightly pessimistic, not because his calculations were wrong, but because population growth had begun to slow, and because it was slowing for non-messy reasons.

Overall estimates of the population of the world can be obtained from the U.S. Census Bureau.[3] Figure 11.1 shows estimates dating from 1900 and extending until 2050. This clearly shows the upward curve during the first three quarters of the 20^{th} Century, but already a straightening and tailing off beginning as we pass the turn of the millenium.

This slowing in the growth of the world's population is mainly thanks to lower fertility rates (the non-messy solution), which is due to cultural changes such as people marrying later, and the greater availability and use of contraceptives. Even longer term projections by the United Nations suggest that, assuming trends in lower fertility continue, the world's population may actually stabilize at around 9 or 10 billion before 2500. Asimov, or at least his descendants, have some reason to hope.

11.1.1 Estimating population growth

Another population-related service offered by the U.S. Census Bureau is the World Population Clock (see Figure 11.2).

This web site provides an up-to-the-minute snapshot of the current estimate of the world's population, based on estimates by the U.S. Census Bureau. It is updated every few seconds.

What we are going to do in this case study is use this clock to generate a rough estimate of the current *rate of growth* of the world's population.

We will do this by looking at the steps involved, how we might perform this task "by hand", and how we might use the computer to do the work instead.

[3] `http://www.census.gov/ipc/www/worldhis.html`
`http://www.census.gov/ipc/www/idb/worldpop.html`.

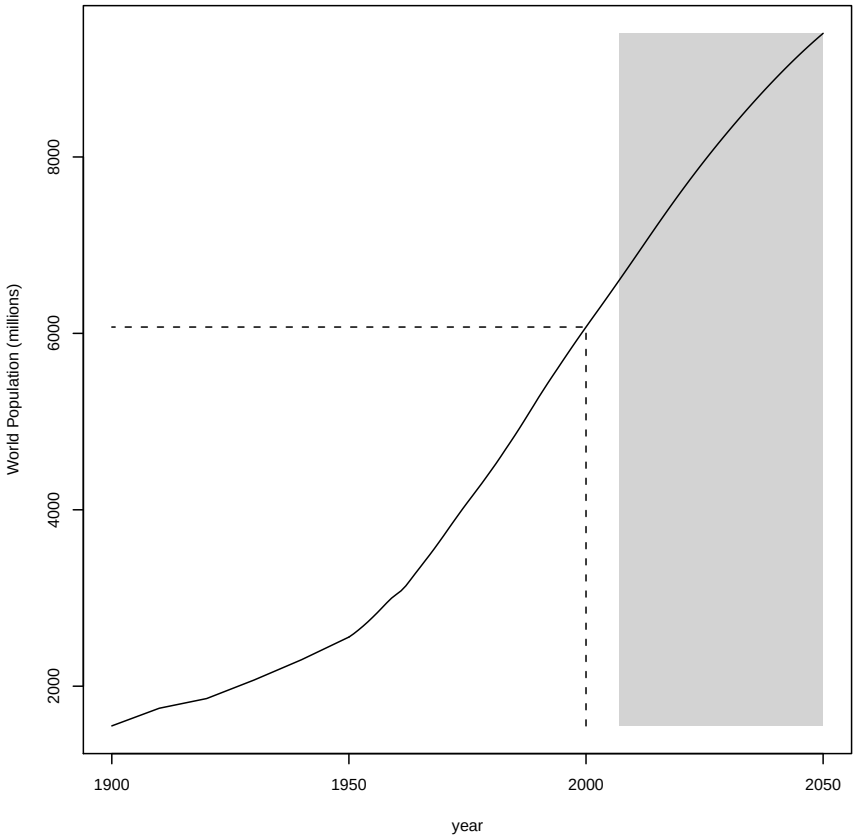


Figure 11.1: The population of the world, based on estimates by the U.S. Census Bureau. The shaded area to the right indicates projected estimates of world population.

Figure 11.2: The World Population Clock shows an up-to-the-minute snapshot of the current estimate of the world's population (based on estimates by the U.S. Census Bureau).

Copy the current value of the population clock.

The first step is to capture a snapshot of the world population from the U.S. Census Bureau web site.

This is very easy to do by simply navigating a web browser to the population clock web page and typing out or cutting-and-pasting the current population value.

What about getting the computer to do the work?

Navigating to a web page and downloading the information is not actually very difficult. The following R code will do this **data import** task:[4]

```
R> clockHTML <-
    readLines("http://www.census.gov/ipc/www/popclockworld.html")
```

Getting the population estimate from the downloaded information is a bit more difficult, but not much.

The above code says "read the HTML code from the given URL." The first thing to realise is that the result is not a nice picture of the web page like we see in a browser. Instead, we have the raw HTML

[4]We will not focus on understanding the details of the R code in this section—that is the purpose of the the remainder of this chapter. The code is just provided as concrete evidence that the task can be done and as a simple visual indication of the level of effort and complexity involved.

code that describes the web page (see Figure 11.3). This is actually a good thing because it would be incredibly difficult for the computer to extract the population information from a picture.

The HTML code is better than a picture because there is structure to this information and we can use that structure to get the computer to extract the relevant population value for us.

The current population value on the web page is contained within an HTML `div` tag with a unique `id` attribute (line 41 in Figure 11.3). This makes it very easy to find the line that contains the population estimate. This **text search** task can be performed using the following code:

```
R> popLine <- grep('id="worldnumber"', clockHTML)
R> popLine

[1] 41
```

This code says "tell me which line of HTML code contains the text `id="worldnumber"`."

It is easy to extract the population estimate from this line by deleting all of the bits of the line that we do not want. This is a **text search-and-replace** task and can be performed using the following code:

```
R> popString <- gsub('^.+id="worldnumber">', "",
                     gsub("</div>.*$", "",
                          clockHTML[popLine]))
R> popString

[1] "6,617,746,521"
```

This code says "extract the relevant line of HTML code and delete everything from the `</div>` to the end of the line, then delete everything from the start of the line up to the text `id="worldnumber">`.

The final thing we need to do is turn the text of the population estimate into a number so that we can later carry out mathematical operations. This is called **data coercion** and appropriate code is shown below (notice that we have to remove the commas that are so useful for human viewers, but a complete distraction for computers):

```
R> pop <- as.numeric(gsub(",", "", popString))
R> pop

[1] 6617746521
```

This example provides a classic demonstration of the difference between performing a task by hand and writing code to get a computer

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xml:lang="en" lang="en">
<head>
    <title>World POPClock Projection</title>
    <link rel="stylesheet"
          href="popclockworld%20Files/style.css"
          type="text/css">
    <meta name="author" content="Population Division">
    <meta http-equiv="Content-Type"
          content="text/html; charset=iso-8859-1">
    <meta name="keywords" content="world, population">
    <meta name="description"
          content="current world population estimate">
    <style type="text/css">
        #worldnumber {
            text-align: center;
            font-weight: bold;
            font-size: 400%;
            color: #ff0000;
        }
    </style>
</head>
<body>
    <div id="cb_header">
    <a href="http://www.census.gov/">
    <img src="popclockworld%20Files/cb_head.gif"
         alt="U.S. Census Bureau"
         border="0" height="25" width="639">
    </a>
    </div>

    <h1>World POPClock Projection</h1>

    <p></p>
    According to the <a href="http://www.census.gov/ipc/www/">
    International Programs Center</a>, U.S. Census Bureau,
    the total population of the World, projected to 09/12/07
    at 07:05 GMT (EST+5) is<br><br>
    <div id="worldnumber">6,617,746,521</div>
    <p></p>
    <hr>

    ...
```

Figure 11.3: HTML code for the World Population Clock (see 11.2). The line numbers (in grey) are just for reference.

to do the work. The manual method is simple, requires no new skills, and takes very little time. On the other hand, the computer code approach requires learning new information (it will take substantial chunks of this chapter to explain just the code we have used so far), so it is slower and harder. However, the computer code approach *will* pay off in the long run, as we are about to see.

Wait ten minutes.

The second step involves letting time pass so that we can obtain a second snapshot of the world population after a fixed time interval.

Doing nothing is about as simple as it gets for a do-it-yourself task. However, it highlights two of the major advantages of automating tasks by computer. First, computers will perform boring tasks without complaining or falling asleep and, second, their accuracy will not degrade as a function of the boredom of the task.

The following code will make the computer wait for 10 minutes (600 seconds):

```
R> Sys.sleep(600)
```

Copy the new value of the population clock.

The third step is to take another snapshot of the world population from the U.S. Census Bureay web site.

This is the same as the first task. If we do it by hand, it is just as easy as it was before, though the boredom issue quite quickly comes into play. What about doing it by computer code? Here we see the third major benefit of writing computer code: once code has been written to perform a task, repetitions of the task become essentially free. All of the pain of writing the code in the first place starts to pay off very rapidly once a task has to be repeated. Almost exactly the same code as before will produce the new population clock estimate.

```
R> clockHTML2 <-
       readLines("http://www.census.gov/ipc/www/popclockworld.html")
R> popLine2 <- grep('id="worldnumber"', clockHTML2)
R> popString2 <- gsub('^.+id="worldnumber">', "",
                      gsub("</div>.+", "",
                           clockHTML2[popLine2]))
R> pop2 <- as.numeric(gsub(",", "", popString2))
```

Calculate the growth rate.

The fourth step is to divide the change in population by the time interval.

This is a very simple calculation that is, again, easy to do by hand. Computer code still provides an advantage because there is less chance of making an error in the calculation. There is the usual cost to pay in terms of writing the code in the first place, but in this case, that is fairly small. All we need to do is divide the change in population by the elapsed time (10 minutes):

```
R> rateEstimate <- (pop2 - pop)/10
```

```
[1] 146.6
```

Repeat several times to get a decent sample

Because we are unaware of the process going on behind the scenes at the population clock web site, it would be unwise to trust a single point estimate of the population growth rate using this technique. A safer approach would be to generate a sample of several estimates and that means that we should repeat the whole process.

As mentioned previously, computers are world champions when it comes to mindlessly repeating tasks, so the computer code approach will now pay off handsomely.

The computer code that will generate 10 population growth rate estimates is shown in Figure 11.4. As mentioned previously, the details of how this code works are not important at this stage. However, there are several important features that we should highlight.

The core task in this example involves downloading the World Population Clock and processing the information to extract a time and a population estimate. For each estimate of the population growth rate, this core task must be performed twice. A naive approach would suggest writing out two copies of the code to perform the task. However, that would violate the DRY principle (see Section 2.5) because it would create two copies of an important piece of information; the information in this case being computer code to perform a certain task. As can be seen from Figure 11.4, the code can be written so that only one copy, written as a **function**, is required (lines 2 to 9) and that single copy can be referred to, via **functions calls**, from elsewhere in the code (lines 15 and 17).

At a slightly higher level, the task of calculating an estimate of the population growth is also repeated, in this case, 10 times. Again, rather than having 10 copies of the code to calculate an estimate, there is only one copy (lines 15 to 18), with other code, a **for loop**, to express the fact that the this sub-task needs to be repeated 10 times (lines 14 and 19).

```
checkTheClock <- function() {
    clockHTML <-
     readLines("http://www.census.gov/ipc/www/popclockworld.html")
    popLine <- grep('id="worldnumber"', clockHTML)
    popString <- gsub('^.+id="worldnumber">', "",
                      gsub("</div>.*", "",
                           clockHTML[popLine]))
    pop <- as.numeric(gsub(",", "", popString))
    return(list(popString=popString, pop=pop))
}

rateEstimates <- rep(0, 10)

for (i in 1:10) {
    clock1 <- checkTheClock()
    Sys.sleep(600)
    clock2 <- checkTheClock()
    rateEstimates[i] <- (clock2$pop - clock1$pop) / 10
}

writeLines(as.character(rateEstimates),
           paste("popGrowthEstimates",
                 as.Date(Sys.time()), sep=""))
```

Figure 11.4: R code for estimating world population growth by downloading the World Population Clock web site and processing it at 10 minute intervals. The line numbers (in grey) are just for reference.

```
146.6
146.5
146.6
148.1
134.181818181818
146.8
146.3
146.6
146.8
147.9
```

Figure 11.5: Ten estimates of the rate of growth (people per ten minutes) of the World's population, based on the U.S. Census Bureau's World Population Clock.

These ideas of encapsulating chunks of code as **functions** and repeating chunks of code within **loops** are examples of the extra concepts that will be introduced in this chapter that will allow us the flexibility to perform a wide variety of tasks.

Calculate a final growth rate estimate

Now that we have 10 estimates of the population growth rate, we can generate an overall estimate of the growth rate by averaging these values.

This sort of arithmetic is equally simple whether done by hand or by writing computer code, but code has all of the advantages already mentioned.

```
R> mean(rateEstimates)

[1] 145.6382
```

At the time of writing, we estimate that the world population was growing at the rate of about 146 people every ten minutes.

Write the answer down

The final step in this exercise is to record the results of all of our work. This will be useful if, for example, we want to compare the current population growth rate with the rate next month, or next year. This is the purpose of lines 21 to 23 in Figure 11.4. This code creates a plain text file containing our estimates and includes the current date in the name of the file so that we know when it was generated.

Figure 11.5 shows the full set of 10 population growth estimates.

To reiterate, that all may seem like quite a lot of work to go through to perform a relatively simple task, but the effort is worth it. By writing code

so that the computer performs the task, we can improve our accuracy and efficiency, and we can repeat the task whenever we like for no additional cost.

This chapter is concerned with writing code like this, using the R language, to conduct and automate general data handling tasks: importing and exporting data, manipulating the shape of the data, and processing data into new forms.

11.2 The R language

The R language is a popular general-purpose language for working with and analysing data.

R code consists of one or more **expressions**. Each expression describes an action to take and the expressions are carried out, or **evaluated** one at a time, in the order they appear.

The following code consists of two expressions. The first expression reads HTML code from a web location and the second expression searches within that HTML code for a particular pattern. Notice that a single expression can flow across several lines.

```
clockHTML <-
  readLines("http://www.census.gov/ipc/www/popclockworld.html")
popLine <- grep('id="worldnumber"', clockHTML)
```

R is used to describe both the language *and* the software package that is used to run code written in the language. In this chapter, we will focus just on writing code in the R language. Chapter 12 contains a brief description of the R software that is used to run R code.

Having said that, most R code examples in this chapter will be presented in a format resembling the R software command-line interface, with R expressions shown to the right of an `R` "prompt" and the results of the expressions displayed below.

In the example below, we type a very simple piece of R code consisting of the number `1` and R prints the result of running that code, which is displayed as `[1] 1` (i.e., a single value, 1).

```
R> 1

[1] 1
```

11.2.1 Constant values

The simplest sort of R expression is just a constant value—a piece of text (a string) or a number. The value of such expressions is just the constant itself.

For example, if we need to specify the name of a file that we want to read data from, we specify the name as a string.

```
R> "http://www.census.gov/ipc/www/popclockworld.html"

[1] "http://www.census.gov/ipc/www/popclockworld.html"
```

If we need to specify a number of seconds corresponding to 10 minutes, we specify a number.

```
R> 600

[1] 600
```

11.2.2 Arithmetic

Numeric values can be combined in simple arithmetic expressions: addition, subtraction, multiplication, division, and exponentiation.

For example, the following code shows the arithmeric calculation that was performed in Section 11.1 to obtain the rate of growth of the world's population—the change in population divided by the elapsed time. Note the use of the forward-slash character, `/`, to represent division and the use of brackets to control the order of evaluation.[5]

```
R> (6617747987 - 6617746521) / 10

[1] 146.6
```

11.2.3 Function calls

The most common and useful type of R expression is a **function call**.

Function calls are very important because they are how we use R to perform any non-trivial task.

[5] R obeys the normal BODMAS rules of precedence for arithmetic operators, but brackets are a useful way of avoiding any ambiguity, especially for the human audience.

A function call consists of the function name followed by, within parentheses and separated from each other by commas, expressions called **arguments** that provide necessary information for the function to perform its task.

The following code gives an example of a function call.

```
Sys.sleep(600)
```

The name of the function is `Sys.sleep` (this function makes the computer wait, or "sleep", for a number of seconds). There is one argument to the function, the number of of seconds to wait, and in this case the value supplied for this argument is `600` (10 minutes).

Because function calls are so common and important, it is worth looking at a few more examples to show some of the variations in their format.

The `grep()` function is used to search for a pattern in text, so it has two arguments: a pattern to search for and the text to search within. The following expression shows a call to `grep()`, demonstrating that a comma must be placed between arguments in a function call. The first argument is the text to search for `'id="worldnumber"'` and the second argument is the text to search within, which in this case has been stored in an object called `clockHTML` (see the discussion of symbols and assignment in Section 11.2.4 below).

```
R> grep('id="worldnumber"', clockHTML)

[1] 41
```

This example also demonstrates that many functions have a **return value**. The result of calling the `grep()` function is a number indicating in which line(s) of text the pattern was found. In this case, the pattern occurs on line 41.

The next example shows a call to the `readLines()` function, which is used to read text from a file. The following expression reads the first six lines from the world population clock web page.

```
R> readLines("http://www.census.gov/ipc/www/popclockworld.html",
             n=6)

[1] "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.0 Transitional//EN\""
[2] "\t\"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd\">"
[3] "<html xmlns=\"http://www.w3.org/1999/xhtml\" xml:lang=\"en\" lang=\"en\">"
[4] "<head>"
[5] "<title>World POPClock Projection</title>"
[6] ""
```

The interesting thing about this example is that the `readLines()` function actually has *five* arguments. For example, there is an argument that controls whether this function call will fail (produce an error) if we ask for more lines than exist in the file.

We have only specified two arguments in the function call, which demonstrates the fact that some function arguments have default values that will be used if no value is specified for the argument. The default behaviour of the function is *not* to generate an error.

The other thing that this example shows is that function arguments have names and these names can be used in the function call. In the case of the `readLines()` function, the argument with the name `n` controls how many lines of text are read from the file. We specify the value `6` specifically for this argument by typing `n=6`.

The argument that controls whether an error will occur has the name `ok` (the meaning of some function and argument names in R are not immediately transparent). If we want the function to generate an error if we ask for more lines than there are in a file, we would specify `ok=FALSE`.

11.2.4 Symbols and assignment

Code written in a general-purpose language can be compared to a cooking recipe; there is a series of steps to perform and the results of the initial steps are used later on. For example, eggs and milk may be mixed together in one bowl and flour and salt mixed in another bowl, then the two sets of ingredients may be combined together.

This idea of storing intermediate results is an important feature of general-purpose languages. In R, we describe this process as **assigning** values to **symbols**.

Anything that we type that starts with a letter, and which is not one of the special R keywords, is interpreted by R as a symbol.

A symbol represents a container where a value can be stored. When R encounters a symbol it returns the value that has been stored with that symbol. For example, there is a predefined symbol called `pi` and the value stored in `pi` is the mathematical constant π.

```
R> pi

[1] 3.141593
```

The result of any expression can be **assigned** to a symbol, which means that the result is stored and remembered for use later on.

For example, when we read the contents of a text file into R using the `readLines()` function, we usually want to store the contents so that we can work with the information later on. This is accomplished by assigning the result of the function to a symbol, like in the following code.

```
R> clockHTML <-
    readLines("http://www.census.gov/ipc/www/popclockworld.html")
```

We say that `clockHTML` is **assigned** the value returned by the `readLines()` function.[6]

Whenever we use the symbol `clockHTML`, R retrieves the value that we assigned to it, namely the strings containing the contents of the text file (not all lines are shown).

```
R> clockHMTL
```

```
[1] "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.0 Transitional//EN\" "
[2] "    \"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd\">"
[3] "<html xmlns=\"http://www.w3.org/1999/xhtml\" "
[4] "      xml:lang=\"en\" lang=\"en\">"
[5] "<head>"
[6] "    <title>World POPClock Projection</title>"
```

11.2.5 Control flow

Computer code in a general-purpose language typically consists of more than a single expression. When there are several expressions, they are run in the order that they appear.

For example, in the following code, the first expression determines which line in an HTML file contains the special text `id="worldnumber"`. The second expression takes that line and removes unwanted text from the line to end up with a string that contains the current population of the world.

```
popLine <- grep('id="worldnumber"', clockHTML)
popString <- gsub('^.+id="worldnumber">', "",
                  gsub("</div>.*", "",
                       clockHTML[popLine]))
```

[6] The assignment operator in R is either `=` or `<-`. For clarity, we will use the latter in all examples.

The second expression can make use of the result in the first expression because that result has been assigned to a symbol. The second expression uses the symbol `popLine` to access the result from the first expression.

General-purpose languages include features that allow some exceptions from the usual rule that expressions are run one at a time, from first to last. One such exception is the **loop**, which allows a collection of expressions to be run repeatedly.

In the example in Section 11.1, we performed the same task, calculating the rate of population growth, 10 times. This was achieved, not by typing out the relevant code 10 times, but by using a loop.

```
for (i in 1:10) {
    pop1 <- checkTheClock()
    Sys.sleep(600)
    pop2 <- checkTheClock()
    rateEstimates[i] <- (pop2 - pop1) / 10
}
```

The line first line of this code specifies that this loop will run 10 times.

```
for (i in 1:10) {
```

The expression `1:10` is a shorthand way of expressing the integer values from 1 to 10.

The expressions within the braces, the **body** of the loop, are run each time through the loop. In this case, almost exactly the same actions are taken each time the loop is run; the one thing that does change is that the symbol `i` is assigned a different value. The first time the loop runs, `i` is assigned the value `1`. The second time through the loop, `i` has the value `2`, and so on. In this example, the changing value of `i` is just used to make sure that each estimate of the population growth rate is stored in a different place, `rateEstimates[i]`, rather than overwriting the result from previous times through the loop.

In R, there is less need for loops compared to other scripting languages, because R naturally deals with entire vectors, or matrices of data at once (see Sections 11.3 and 11.7).

R also provides a **while loop**, which can be used when it is not known how many times the code will repeat (e.g., an iterative optimisation algorithm). See Section 12.2.4 for more information.

There are also other ways to control the evaluation of expressions, such as

conditional statements. These are described in Section 12.2.5.

11.2.6 Flashback: Writing for an audience

Chapter 2 introduced general principles for writing computer code. In this section, we will look at some specific issues related to writing scripts in R.

The same principles, such as commenting code and laying out code so that it is easy for a human audience to consume, still apply. In R, a comment is anything on a line after the special hash character, `#`. For example, the comment in the following line of code is useful as a reminder of why the number `600` has been chosen.

```
Sys.sleep(600) # Wait 10 minutes
```

Indenting is also very important. We need to consider indenting whenever an expression is too long and has to be broken across several lines of code. The example below shows a standard approach that ensures that arguments to a function call are left-aligned.

```
popString <- gsub('^.+id="worldnumber">', "",
                  gsub("</div>.*", "",
                       clockHTML[popLine]))
```

When using loops, the expressions within the body of the loop should be indented, as below.

```
for (i in 1:10) {
    pop1 <- checkTheClock()
    Sys.sleep(600)
    pop2 <- checkTheClock()
    rateEstimates[i] <- (pop2 - pop1) / 10
}
```

It is also important to make use of whitespace. Examples in the code above include the use of spaces around the assignment operater (`<-`), around arithmetic operators, and between arguments (after the comma) in function calls.

11.2.7 Naming variables

When writing R code, because we are constantly assigning intermediate values to symbols, we are forced to come up with lots of different symbol

names. It is important that we choose sensible symbol names for several reasons:

1. Good symbol names are a form of documentation in themselves. A name like `dateOfBirth` tells the reader a lot more about what value has been assigned to the symbol than a name like `d`, or `dob`, or even `date`.
2. Short or convenient symbol names, such as `x`, or `xx`, or `xxx` should be avoided because it too easy to create conflict by reusing them in our own code or by having other code reuse them.

Anyone with children will know how difficult it can be to come up with even one good name, let alone a constant supply, but fortunately there are several good guidelines for producing sensible variable names:

- The symbol name should fully and accurately represent the information that has been assigned to that symbol. Unlike children, symbols usually have a specific purpose, so the symbol name naturally arises from a description of that purpose.
- Use a mixture of lowercase and uppercase letters when typing the name; treat the symbol name like a sentence and start each new word with a capital letter (e.g., `dateOfBirth`). This naming mechanism is called "camelCase" (the uppercase letters form humps like the back of a camel).

  This topic involves quite a lot of personal preference and some very strong opinions. Fortunately, R, allows for a large degree of flexibility. Other naming conventions include placing a dot, `.`, or an underline, `_`, between words in the name (e.g., `date.of.birth` or `date_of_birth`). For historical reasons, both of these alternatives have potentially serious downsides so we will always use camelCase in this book.
- Use a naming scheme that reflects the structure of the data within a symbol. For example, when naming a data frame in R (see Section 11.3), one approach would be to always end the name with `.df`.
- There are some short symbol names that are acceptable because they are so consistently used for a particular purpose. Some examples are `i`, `j`, and `k` as counters within loops (see Section 11.2.5).
- Beware of using names of predefined constants. For example, R defines the symbol `pi` and redefining this value to, for example, the name of the i^{th} *p*arent in a data set would have disastrous consequences for subsequent arithmetic calculations.

11.3 Basic Data types and data structures

General-purpose languages allow us to work with numbers, text (strings), and logical values. This section briefly describes important features of how values are represented using these basic **data types**, then goes on to the larger topic of how multiple values, possibly of differing types, can be stored together in **data structures**.

11.3.1 Case study: Counting candy

The image in Figure 11.6 shows a simple counting puzzle. The task is to count how many of each different type of candy there are in the image.

Our task is to record the different shapes (round, oval, or long), the different shades (light or dark), and whether there is a pattern on the candies that we can see in the image. How can this information be entered into R?[7]

We will start by just entering the names of the possible categories of candy.

One way to enter data in R is to use the `scan()` function. This allows us to type data into R separated by spaces. Once we have entered all of the data, we enter an empty line to indicate to R that we have finished. We can use this to enter the information describing the possible candy shapes as strings:

```
R> shapeNames <- scan(what="character")
1: round oval long
4:
Read 3 items

R> shapeNames

[1] "round" "oval"  "long"
```

Another way to enter data is using the `c()` function. In the following code, we use this to enter the information on the possible patterns and shades of the candies:

[7] There are two ways to view the information that we will enter: either the number of candies is fixed and, *for each candy*, we are making three measurements (shape, shade, and pattern); or the categories are fixed and, *for each type of candy*, we are measuring how many candies there are of that type. At different times we will use whichever of these interpretations is most convenient in order to illustrate a point.

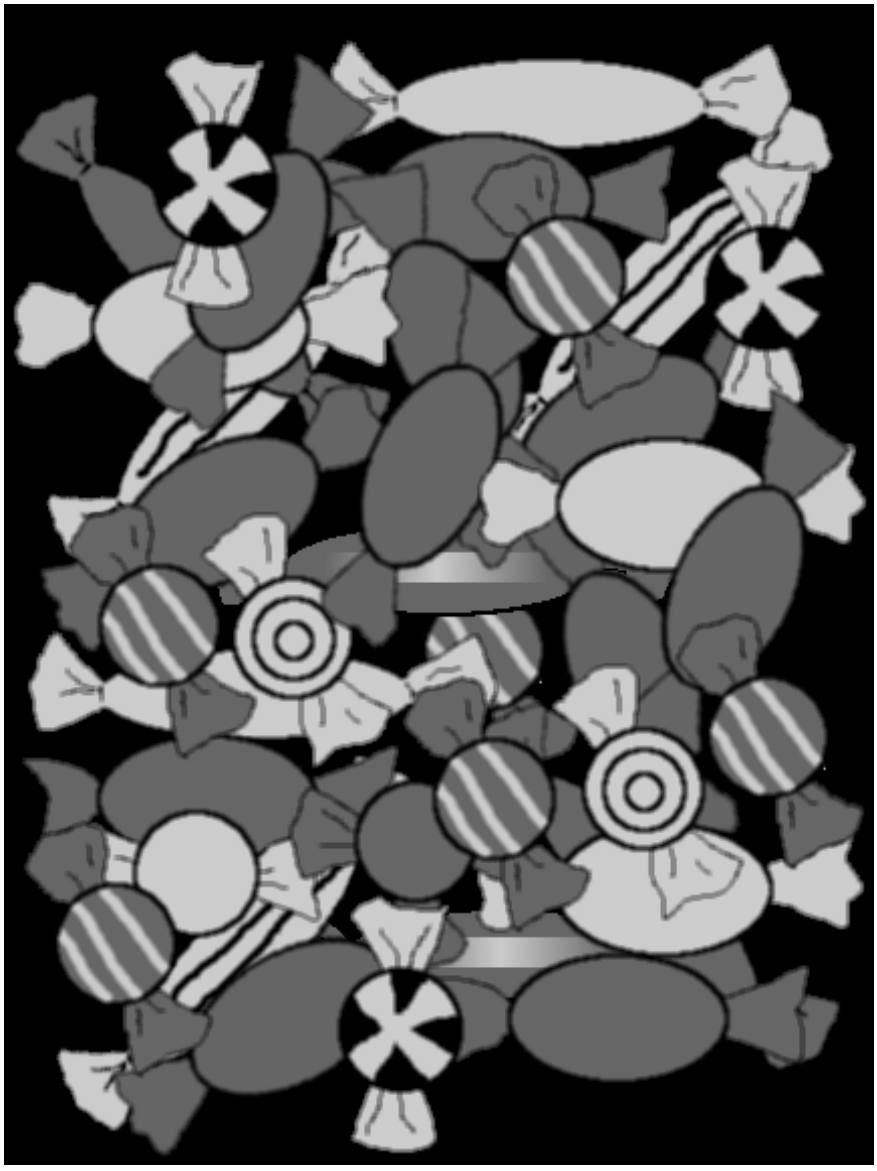


Figure 11.6: A counting puzzle. How many candies of each different shape are there? (round, oval, and long). How many candies have a pattern? How many candies are dark and how many are light?

```
R> patternNames <- c("pattern", "plain")
R> patternNames

[1] "pattern" "plain"

R> shadeNames <- c("light", "dark")
R> shadeNames

[1] "light" "dark"
```

These are examples of ways to enter text values into R. The same functions can be used to enter numbers as well, as subsequent examples will demonstrate.

At this point in our case study, we have only entered some metadata; the possible categories of candies. In the following sections, we will add counts of how many times each shape-shade-pattern combination occurs.

11.3.2 Vectors

One of the reasons that R is a good environment for working with data is because it works very naturally with **vectors** of values. Any symbol, like `shapeNames`, `patternNames`, and `shadeNames` above, can contain several values at once. For example, `shapeNames` is a character vector containing three strings.

```
R> shapeNames

[1] "round" "oval"  "long"
```

Many functions exist for creating and for manipulating vectors.

The previous section demonstrated the `c()` function for combining values together. Another example is the `rep()` function, which can be used to repeat values in a vector. We can use this function to create a symbol representing the shade for each of the candies in the image above (by my count, there are 11 light-coloured candies and 25 dark-coloured candies).

```
R> shades <- rep(shadeNames, c(11, 25))
R> shades
```

```
 [1] "light" "light" "light" "light" "light" "light" "light"
 [8] "light" "light" "light" "light" "dark"  "dark"  "dark"
[15] "dark"  "dark"  "dark"  "dark"  "dark"  "dark"  "dark"
[22] "dark"  "dark"  "dark"  "dark"  "dark"  "dark"  "dark"
[29] "dark"  "dark"  "dark"  "dark"  "dark"  "dark"  "dark"
[36] "dark"
```

This example also demonstrates that many R functions accept vectors of values for arguments. In this case, two values are provided to be repeated and a number of replicates is specified for each value (so the first `shadeNames` value is repeated 11 times and the second `shadeNames` value is repeated 25 times). The return value is also a vector (36 strings).

A vector can only contain values of the same sort, so we can have **numeric** vectors containing all numbers, **character** vectors contianing only strings, and **logical** vectors containing only true/false values.

11.3.3 The recycling rule

Because the arguments to a function, or an arithmetic operation, can involve vectors, there arises a general problem of what to do when the vectors have different lengths.

There is a general, but informal, rule in R that, in such cases, the shorter vector is recycled to become the same length as the longer vector. This is easiest to demonstrate via simple arithmetic.

In the following code, a vector of length 3 is added to a vector of length 6.

```
R> c(1, 2, 3) + c(1, 2, 3, 4, 5, 6)
```

```
[1] 2 4 6 5 7 9
```

What happens is that the first vector is recycled to make a vector of length 6, then element-wise addition can occur.

This rule is not followed in all possible situations, but it is the expected behaviour in most cases.

11.3.4 Factors

A **factor** is a basic data structure in R that is ideal for storing categorical data.

For example, consider the `shades` symbol that we created previously as just a vector of text (a character vector) recording the word "light" for each of the light-shaded candies and "dark" for each of the dark-shaded candies.

This is not the ideal way to store this information because it does not acknowledge that elements containing the same text (e.g., `"light"`) really are the same value. A text vector can contain any strings at all, so there are no data integrity constraints (see Section 7.7.3). The information would be represented better using a **factor**.

The following code creates the candy shade information as a factor:

```
R> shades.f <- factor(shades, levels=shadeNames)
R> shades.f
```

```
 [1] light light light light light light light light light
[10] light light dark  dark  dark  dark  dark  dark  dark
[19] dark  dark  dark  dark  dark  dark  dark  dark  dark
[28] dark  dark  dark  dark  dark  dark  dark  dark  dark
Levels: light dark
```

This is a better representation because every value in `shades.f` is guaranteed to be one of the valid "levels" of the factor. It is also more efficient because what is stored is only integer codes which refer to the appropriate levels.

A factor is the best way to store categorical information in R. If we need to work with the data as text (see Section 11.8), we can convert the factor back to text using the `as.character()` function (see page 261).

11.3.5 Data Frames

A vector in R contains values that are all of the same type. Vectors correspond to a single variable in a data set.

Most data sets consist of more than just one variable, so to store a complete data set we need a different data structure. In R, several variables can be stored together in an object called a **data frame**.

We will now build a data frame for the candy example, with variables in-

dicating the different combinations of shape, pattern, and shade, and a variable containing the number of candies for each combination.

The function `data.frame()` creates a data frame object. If we just consider shape and shade, the following code generates a data frame with all possible combinations of these catgories.

```
R> data.frame(shape=factor(rep(shapeNames, 2),
                           levels=shapeNames),
              shade=factor(rep(shadeNames, each=3),
                           levels=shadeNames))
```

```
  shape shade
1 round light
2  oval light
3  long light
4 round  dark
5  oval  dark
6  long  dark
```

Rather than enumerate all of the combinations ourselves, we can use the function `expand.grid()`. This function takes several factors and produces a data frame containing all possible combinations of the levels of the factors.[8]

```
R> candy <- expand.grid(shape=shapeNames,
                        pattern=patternNames,
                        shade=shadeNames)
R> candy
```

```
   shape pattern shade
1  round pattern light
2   oval pattern light
3   long pattern light
4  round   plain light
5   oval   plain light
6   long   plain light
7  round pattern  dark
8   oval pattern  dark
9   long pattern  dark
10 round   plain  dark
11  oval   plain  dark
12  long   plain  dark
```

Now we can count the number of candies for each of these combinations

[8] The `gl()` function is similar.

and enter the counts in a variable in our data frame called `count`. This demonstrates how to add new variables to an existing data frame.

```
R> candy$count <- c(2, 0, 3, 1, 3, 2, 9, 0, 2, 1, 11, 2)
R> candy
```

```
   shape pattern shade count
1  round pattern light     2
2   oval pattern light     0
3   long pattern light     3
4  round   plain light     1
5   oval   plain light     3
6   long   plain light     2
7  round pattern  dark     9
8   oval pattern  dark     0
9   long pattern  dark     2
10 round   plain  dark     1
11  oval   plain  dark    11
12  long   plain  dark     2
```

At this point, it is worth checking that our two counting efforts are consistent (I had previously counted 11 light and 25 dark candies). We can extract just one variable from a data frame using the special character `$`. The function `sum()` provides the sum of a numeric vector.

```
R> sum(candy$count)
```

```
[1] 36
```

We can also check that our counts sum to the correct amounts for the different shades using the `aggregate()` function. This calls the `sum()` function for subsets of the `candy$count` variable corresponding to the different candy shades.

```
R> aggregate(candy$count, list(shade=candy$shade), sum)
```

```
  shade  x
1 light 11
2  dark 25
```

More examples of this sort of data frame manipulation are described in Section 11.7.

11.3.6 Accessing variables in a data frame

Several previous examples have demonstrated the simple way to specify a variable within a data frame: *dataFrameName*$*variableName*.

Always typing the data frame name can become tiring, but there are two ways to avoid it. The `attach()` function adds a data frame to the list of places where R will look for variable names. For example, instead of typing `candy$count`, we can instead do the following:

```
R> attach(candy)
R> count
```



```
 [1]  2  0  3  1  3  2  9  0  2  1 11  2
```

After the call to `attach()`, R will look at the names of the variables in the `candy` data frame and will find `count` there.

If we use `attach()` like this, we should remember to call `detach()` again once we have finished with the data frame, as below.

```
R> detach(candy)
```

Another way to avoid always having to type the data frame name is to use the `with()` function. This is similar to `attach()`, but, in effect, it only temporarily adds the data frame to R's search path, and it automatically removes the data frame from the search path again. The code below shows how to check the candy counts for the different shades using the `with()` function.

```
R> with(candy,
        aggregate(count, list(shade=shade), sum))
```

```
  shade  x
1 light 11
2  dark 25
```

These approaches are convenient, but can harbour some nasty side-effects, so some care is warranted. In particualr, unexpected things can happen if we perform assignments on an attached data frame, so these usages are really only safe when we are just accessing information in a data frame (see the discussion on the help page for this function by typing `?attach`).

11.3.7 Lists

So far we have seen two sorts of data structures: vectors and data frames. A vector corresponds to a single variable; it is a set of values all of the same type. A numeric vector contains numbers, a character vector contains strings, and a factor contains categories.

A data frame corresponds to a data set, or a set of variables. It can be thought of as a two-dimensional structure with a variable in each column and a case in each row. All columns of a data frame have the same length.

These are the two data structures that are most commonly used for storing data, but, like any general purpose programming language, R also provides a variety of other data structures. We will need to know about these other data structures because different R functions produce results in a variety of different formats. For example, consider the result of the following code:

```
R> candyLevels <- lapply(candy, levels)
R> candyLevels
```

```
$shape
[1] "round" "oval"  "long"

$pattern
[1] "pattern" "plain"

$shade
[1] "light" "dark"

$count
NULL
```

The `lapply()` function is described in more detail in Section 11.7.3. For now, we just need to know that this code calls the `levels()` function for each of the variables in the `candy` data frame. The `levels()` function returns the levels of a factor (the possible categories in a categorical variable), so the result of the `lapply()` function is a set of levels for each variable in the data set.

The number of levels is different for each variable; in fact, the `count` variable is numeric so it has no levels at all. This means that the `lapply()` function has to return several vectors of information, each of which may have a different length. This cannot be done using a data frame; instead the result is returned as a data structure called a **list**.

A list is a very flexible data structure. It can have several **components**,

each of which can be any data structure of any length or size. In the current example, there are four components, three of which are character vectors of differing lengths, and one of which is `NULL` (empty).

The components of a list can have names—in this case, the names have come from the names of the variables in the data set—and the `names()` function can be used to extract these names. Individual components of a list can be extracted using the `$` operator, just like for data frames.

```
R> names(candyLevels)
[1] "shape"   "pattern" "shade"   "count"

R> candyLevels$shape
[1] "round" "oval"  "long"
```

Section 11.4 contains more information about extracting subsets of a list.

It is also possible to create a list directly using the `list()` function. For example, the following code creates a list of the levels of just the factors in the candy data frame:

```
R> list(shape=levels(candy$shape),
        pattern=levels(candy$pattern),
        shade=levels(candy$shade))
$shape
[1] "round" "oval"  "long"

$pattern
[1] "pattern" "plain"

$shade
[1] "light" "dark"
```

Everyone who has worked with a computer should be familiar with the idea of a list because a directory or folder of files has this sort of structure; a folder contains multiple files of different kinds and sizes and a folder can contain other folders, which can contain more files or even more folders, and so on. Lists have this hierarchical structure.

11.3.8 Matrices and arrays

Another sort of data structure in R, that lies in between vectors and data frames, is the **matrix**. This is a two-dimensional structure (like a data frame), but one where all values are of the same type (like a vector).

As for lists, it is useful to know how to work with matrices because many R functions either return a matrix as their result or take a matrix as an argument. The `data.matrix()` function is one example; it takes a data frame and returns a matrix by converting all factors to their underlying integer codes (see page 249).

```
R> data.matrix(candy)
```

```
      shape pattern shade count
 [1,]     1       1     1     2
 [2,]     2       1     1     0
 [3,]     3       1     1     3
 [4,]     1       2     1     1
 [5,]     2       2     1     3
 [6,]     3       2     1     2
 [7,]     1       1     2     9
 [8,]     2       1     2     0
 [9,]     3       1     2     2
[10,]     1       2     2     1
[11,]     2       2     2    11
[12,]     3       2     2     2
```

It is also possible to create a matrix directly using the `matrix()` function, as in the following code (notice that values are used column-first).

```
R> matrix(1:100, ncol=10, nrow=10)
```

```
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
 [1,]    1   11   21   31   41   51   61   71   81    91
 [2,]    2   12   22   32   42   52   62   72   82    92
 [3,]    3   13   23   33   43   53   63   73   83    93
 [4,]    4   14   24   34   44   54   64   74   84    94
 [5,]    5   15   25   35   45   55   65   75   85    95
 [6,]    6   16   26   36   46   56   66   76   86    96
 [7,]    7   17   27   37   47   57   67   77   87    97
 [8,]    8   18   28   38   48   58   68   78   88    98
 [9,]    9   19   29   39   49   59   69   79   89    99
[10,]   10   20   30   40   50   60   70   80   90   100
```

The **array** data structure extends the idea of a matrix to more than two dimensions. For example, a three-dimensional array corresponds to a data cube. The `array()` function can be used to create an array.

```
R> array(1:8, dim=c(2, 2, 2))
```

```
, , 1

     [,1] [,2]
[1,]    1    3
[2,]    2    4

, , 2

     [,1] [,2]
[1,]    5    7
[2,]    6    8
```

11.4 Subsetting

In the previous section, we saw that data can be represented in R as quite large and complex data structures.

In order to work with the data, we need to be able to extract smaller parts of these data structures, such as a single variable from a data frame.

We will refer to this sort of operation as selecting a **subset** from a data structure. It is analogous to performing a query on a database (see Chapter 9).

R has very powerful mechanisms for subsetting. In this section, we will outline the basic format of these operations and many more examples will be demonstrated as we progress through the rest of the chapter.

We have previously seen that we can get a single variable from a data frame using the `$` operator. For example, the `count` variable can be obtained from the `candy` data set using `candy$count`. Another way to do the same thing is to use the double-square-bracket subsetting operator, `[[`, and specify the variable of interest as a string.

```
R> candyCounts <- candy[["count"]]
R> candyCounts
```

```
 [1]  2  0  3  1  3  2  9  0  2  1 11  2
```

The advantage of the [[operator is that it allows a number or an expression as the index. For example, the count variable is the fourth variable, so this code also works.

```
R> candy[[4]]
```

```
 [1]  2  0  3  1  3  2  9  0  2  1 11  2
```

The following code evaluates an expression that determines which of the variables in the data frame is numeric and then selects just that variable from the data frame. The sapply() function is described in Section 11.7. The which() function returns the indices of all TRUE values in a logical vector.

```
R> sapply(candy, is.numeric)
```

```
  shape pattern   shade   count
  FALSE   FALSE   FALSE    TRUE
```

```
R> which(sapply(candy, is.numeric))
```

```
count
    4
```

```
R> candy[[which(sapply(candy, is.numeric))]]
```

```
 [1]  2  0  3  1  3  2  9  0  2  1 11  2
```

R has another subsetting operator consisting of single square brackets, [. This is similar to the [[operator, but it is more flexible because it allows several elements to be selected, rather than just one. For example, the following code produces the first three counts (the number of light-shaded candies with a pattern).

```
R> candyCounts[1:3]
```

```
[1] 2 0 3
```

The indices can be any integer sequence, they can include repetitions, and even negative numbers (to exclude specific values). The following two examples produce counts for all candies with a pattern and then all counts *except* the count for round plain dark candies.

```
R> candyCounts[c(1:3, 7:9)]
```

```
[1] 2 0 3 9 0 2
```

```
R> candyCounts[-10]
```

```
 [1]  2  0  3  1  3  2  9  0  2 11  2
```

As well as using integers for indices, we can use logical values. For example, a better way to express the idea that we want the counts for all candies with a pattern is to use an expression like this:

```
R> hasPattern <- candy$pattern == "pattern"
R> hasPattern
```

```
 [1]  TRUE  TRUE  TRUE FALSE FALSE FALSE  TRUE  TRUE  TRUE
[10] FALSE FALSE FALSE
```

This vector of logical values can be used as an index to return all of the counts where `hasPattern` is `TRUE`.

```
R> candyCounts[hasPattern]
```

```
[1] 2 0 3 9 0 2
```

Better still would be to work with the entire data frame and retain the pattern with the counts, so that we can check that we have the correct result. A data frame can be indexed using the `[` operator too, though slightly differently because we have to specify both which rows *and* which columns we want. Here are two examples which extract the `pattern` and `count` variables from the data frame for all candies with a pattern:

```
R> candy[hasPattern, c(2, 4)]
```

```
R> candy[hasPattern, c("pattern", "count")]
```

```
  pattern count
1 pattern     2
2 pattern     0
3 pattern     3
7 pattern     9
8 pattern     0
9 pattern     2
```

In both cases, the index is of the form `[<rows>, <columns>]`, but the first example uses column numbers and the second example uses column names. The result is still a data frame, just a smaller one.

The function `subset()` provides another way to perform this sort of subsetting, with a `subset` argument for specifying the rows and a `select` argument for specifying the columns.

```
R> subset(candy, subset=hasPattern,
          select=c("pattern", "count"))
```

```
  pattern count
1 pattern     2
2 pattern     0
3 pattern     3
7 pattern     9
8 pattern     0
9 pattern     2
```

It is possible to leave the row or column index completely empty, in which case all rows or columns are returned. For example, this code extracts all variables for the light-shaded candies with a pattern:

```
R> candy[1:3, ]
```

```
  shape pattern shade count
1 round pattern light     2
2  oval pattern light     0
3  long pattern light     3
```

It is also possible to extract all rows for a selection of variables in a data frame by just specifying a single index with the `[` operator.

```
R> candy["count"]
```

```
   count
1      2
2      0
3      3
4      1
5      3
6      2
7      9
8      0
9      2
10     1
11    11
12     2
```

This result, which is a data frame, is quite different to the result from using the `[[` operator, which gives a vector.

```
R> candy[["count"]]
```

```
 [1]  2  0  3  1  3  2  9  0  2  1 11  2
```

This is another difference between the subsetting operators: `[[` extracts just a single component from a larger data structure, but `[` extracts a smaller version of the larger data structure. It is like the difference between removing a single egg from an egg carton and ripping an egg carton into two pieces (and keeping the eggs intact).

11.4.1 Accessor functions

Some R functions produce complex results (e.g., the result of a linear regression returned by `lm()`). These results are often returned as list objects, which makes it tempting to obtain various subsets of the results, e.g., the model coefficients from a linear regression, using the basic subsetting syntax. However, this is usually *not* the correct approach.

Many functions that produce complex results are accompanied by a set of other functions that perform the extraction of useful subsets of the result. For example, the coefficients of a linear regression analysis should be obtained using the `coef()` function.

The reason for this is so that the people who write a function that produces complex results can change the structure of the result without breaking code that extracts subsets of the result. This idea is known as **encapsulation**.

11.4.2 Assigning to a subset

The subsetting operators can also be used to assign a value to only some components of a larger data structure. As an example, we will look at replacing the zero values in the counts of candies with a missing value, `NA`.

The zeroes occur as the second and eighth value of the vector `candyCounts`.

```
R> candyCounts
```

```
 [1]  2  0  3  1  3  2  9  0  2  1 11  2
```

The following code replaces the zeroes with `NA`s.

```
R> candyCounts[c(2, 8)] <- NA
R> candyCounts
```

```
 [1]  2 NA  3  1  3  2  9 NA  2  1 11  2
```

As with extracting subsets, the indices can be a wide variety R expressions. A better way to perform the above replacement would be to let the computer figure out which values are zeroes, as in the following code.

```
R> candyCounts[candyCounts == 0] <- NA
```

11.5 More on Data Types

11.5.1 Type coercion

We have seen several examples of functions that take an object of one type and return an object of a different type. For example, the `data.matrix()` function takes a data frame and returns a matrix.

There are general functions of the form `as.`*`type`*`()` for deliberately converting between different types of object; an operation known as **type coercion**. For example, the function for converting an object to a numeric vector is called `as.numeric()` and the function for converting to a character vector is called `as.character()`.

An interesting example is the `as.matrix()` function to convert an object from a data frame to a matrix. The following code does this for the `candy` data frame.

```
R> as.matrix(candy)
```

```
      shape   pattern   shade   count
 [1,] "round" "pattern" "light" " 2"
 [2,] "oval"  "pattern" "light" " 0"
 [3,] "long"  "pattern" "light" " 3"
 [4,] "round" "plain"   "light" " 1"
 [5,] "oval"  "plain"   "light" " 3"
 [6,] "long"  "plain"   "light" " 2"
 [7,] "round" "pattern" "dark"  " 9"
 [8,] "oval"  "pattern" "dark"  " 0"
 [9,] "long"  "pattern" "dark"  " 2"
[10,] "round" "plain"   "dark"  " 1"
[11,] "oval"  "plain"   "dark"  "11"
[12,] "long"  "plain"   "dark"  " 2"
```

This is interesting because, unlike the `data.matrix()` function, which converts all values to numbers, the `as.matrix()` function has converted all of the values in the data set to strings (because not all of the variables in the data frame are numeric). Even the factors have been converted to strings.

It is important to keep in mind that many functions will automatically perform type coercion if we give them an argument in the wrong form. For

example, the `paste()` function expects to be given strings to concatenate (see Section 11.8). If we give it objects which do not contain strings, `paste()` will automatically coerce them to strings.

```
R> paste(candy$shape, candy$count)
 [1] "round 2" "oval 0"  "long 3"  "round 1" "oval 3"
 [6] "long 2"  "round 9" "oval 0"  "long 2"  "round 1"
[11] "oval 11" "long 2"
```

11.5.2 Attributes

The value of a symbol in R can be any sort of data structure, e.g., a vector, a data frame, a matrix, or a list. In addition, a symbol may contain other information, perhaps metadata related to the data structure, in what are called **attributes**.

For example, the names of the variables in a data frame and the levels of a factor are stored as attributes.

Some very common attributes are:

`dim` The dimensions of a structure, e.g., the number of rows and columns in a data frame or matrix. The `dim()` function returns this attribute; `nrow()` and `ncol()` return just the appropriate dimension.

```
R> dim(candy)
[1] 12  4
```

`names` The labels associated with each element of a vector or list. These are usually obtained using the `names()` function. The `rownames()` and `colnames()` functions are useful for obtaining labels from two-dimensional structures. The `dimnames()` function is useful for arrays of arbitrary dimension.

```
R> colnames(candy)
[1] "shape"   "pattern" "shade"   "count"
```

Any information can be stored in the attributes of an object. The function `attributes()` lists all existing attributes of an object and the function `attr()` can be used to get or set a single attribute.

11.5.3 Classes

With all of these different data structures available in R and with different functions returning results in various formats, it is useful to be able to determine the exact nature of an R object.

Every object in R has a special attribute called its **class** that describes the object's structure; the class of an object can be obtained using the `class()` function. For example, `candy` is a data frame, `candy$shape` is a factor, and `candy$count` is a (numeric) vector.

```
R> class(candy)

[1] "data.frame"

R> class(candy$shape)

[1] "factor"

R> class(candy$count)

[1] "numeric"
```

It is possible to create new classes in R and many R packages use this facility to identify complicated R objects that are created by functions in the package. For example, the `stats` package has a `ts()` function for entering time-series data. The object returned by this function contains information about both the data values and the time points associated with the data values; the complete object has the class `"ts"`.

11.5.4 Generic functions

We saw previously that many R functions will accept a variety of data structures as arguments. For example, the `paste()` function can be given numbers as well as text and it will automatically convert the numbers into text before concatenating everything together.

```
R> paste("a", 1)

[1] "a 1"
```

This approach makes use of type coercion; the function always does the same thing (e.g., pastes strings together) and it forces the arguments into the data type that it requires.

Some other functions, known as **generic functions**, take a different approach. Instead of coercing the arguments, they look at the class of the arguments and behave differently for different classes. Instead of changing the arguments to fit the function, the function changes to fit the arguments.

An example of a generic function is the `summary()` function. This function prints a brief summary of the important contents of an object; what sort of information gets displayed depends on what sort of object is being summarised.

For example, the summary for a factor gives a table of counts, but the summary for a numeric vector gives a five-number summary (plus the mean).

```
R> summary(candy$shape)

round  oval  long
    4     4     4

R> summary(candy$count)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
      0       1       2       3       3      11
```

Notice that the table of counts is not a vector or any other data structure that we have encountered so far. It is in fact an object of class `"table"` (which behaves very like and array).

11.5.5 Exploring objects

If the class of an object is unfamiliar, the `str()` function is a useful way of seeing what information is stored in the object. This function is also useful when dealing with large objects because it only shows a sample of the values in each part of the object.

```
R> str(candy$shape)

 Factor w/ 3 levels "round","oval",..: 1 2 3 1 2 3 1 2 3 1 ...
```

Another function that is useful for inspecting a large object is the `head()` function. This just shows the first few elements of an object, so we can see the basic structure without seeing all of the values. There is also a `tail()` function for viewing the last few elements of an object.

```
R> head(candy)
```

```
  shape pattern shade count
1 round pattern light     2
2  oval pattern light     0
3  long pattern light     3
4 round   plain light     1
5  oval   plain light     3
6  long   plain light     2
```

11.5.6 Flashback: Numbers in computer memory

In Section 7.2.3, we discussed how numbers are stored in computer memory. That discussion took place in the context of storing data in a file on a persistent storage medium such as a hard disk or CD.

We are now in a different context. When we work with data in R, data values need to be represented in "resident" computer memory, in the computer's RAM (Random Access Memory). We need the data to be stored within the computer memory that is being used by R.

This means that we have the same issues as before with representing text and numbers as series of binary digits. In particular, there are limits to the precision with which numeric values can be represented in computer memory.

11.5.7 Case study: Network packets (continued)

The network packet data set described in Section 7.2.4 contains measurements of the time that a packet of information arrives at a location in a network. These measurements are the number of seconds since January 1st 1970 and are recorded to the nearest 10,000th of a second, so they are very large and very precise numbers. For example, one time measurement from 2006 was 1156748010.47817 seconds.

What happens if we try to enter numbers like these into R?

```
R> 1156748010.47817
```

```
[1] 1156748010
```

The result looks worse than it is. It appears that R has only read in the value up to the decimal point. However, this illustrates an important conceptual distinction between how R stores values and how R prints values. R has

stored the value with full precision, but it does not print numeric values to full precision by default. The number of significant digits printed by R is controlled via the `options()` function. This function can be used to view and control global settings for an R session. For example, we can ask R to print numbers with full precision as follows.

```
R> options(digits=15)
R> 1156748010.47817
```

```
[1] 1156748010.47817
```

The default is to print only seven significant digits, though the option value is only approximate and will not be obeyed exactly in all cases.

```
R> options(digits=7)
```

Section 11.8 has more information about how to display numbers with precise control.

Comparisons between real values must be performed with care because of the inaccuracy inherent in a real value that is stored in computer memory (see page 111).

In particular, the function `all.equal()` is useful for determining whether two real values are (approximately) equivalent.

11.5.8 Case study: The greatest equation ever

A protrait of Leonhard Euler by Emanuel Handmann, 1753.[9]

Euler's identity is one of the most famous and admired equations in mathematics. It holds such an exalted status because it uses the fundamental mathematical operations of addition, multiplication, and exponentiation exactly once and it relates several fundamental mathematical constants: 0, 1, π, e, and i (the square root of minus one; the imaginary unit).

$$e^{i\pi} + 1 = 0$$

Unfortunately, R does not appear to have such a high opinion of Euler's identity. In fact, R thinks that Euler is wrong!

```
R> exp(pi*1i) + 1 == 0 + 0i
```

```
[1] FALSE
```

What is going on? The problem is that it is not sensible to compare real values for equality. In this case, we are comparing complex values, but that boils down to comparing the real components and the real coefficients of the imaginary components. The problem is that the imaginary component of the left-hand side of the equation is very close to, but not quite exactly `0i`.

[9] Source: Wikimedia Commons
`http://commons.wikimedia.org/wiki/Image:Leonhard_Euler_3.jpg`
This image is in the Public Domain.

```
R> exp(pi*complex(im=1)) + 1

[1] 0+1.224606e-16i
```

This is an example where the `all.equal()` function can be used to compare real values and ignore tiny differences at the level of the precision of the computer.

```
R> all.equal(exp(pi*complex(im=1)) + 1, 0 + 0i)

[1] TRUE
```

Hooray! Euler's identity is saved!

11.6 Data import/export

Almost all of the examples to this point have used data that is typed explicitly as R expressions. In practice, data usually reside in one or more files of various formats. This section looks at R functions that can be used to read data into R from external files.

We will also look at some functions to go the other way and write data from within an R session to an external format.

11.6.1 Specifying files

The first thing we need to be able to do is specify which file we want to work with. Any function that works with a file requires a precise description of the name of the file and the location of the file.

A file name is just a string, e.g., `"pointnemotemp.txt"`, but specifying the location of a file can involve a **path**, which describes a location on a persistent storage medium, such as a hard drive.

The best way to specify a path in R is via the `file.path()` function because this avoids the differences between path descriptions on different operating systems. For example, the following code generates a path to a file within a directory:[10]

```
R> file.path("LAS", "pointnemotemp.txt")

[1] "LAS/pointnemotemp.txt"
```

[10] The result shown is appropriate for a Linux system.

The `file.choose()` function can be used to allow interactive selection of a file. This is particularly effective on Windows because it provides a familiar file selection dialog box.

11.6.2 Text files

A very common format in which to receive a data set is just as a plain text file.

R has functions for reading in the standard types of plain text formats (see Section 7.3), each of which creates a data frame from the contents of the text file:

- `read.table()` for data in a delimited format (by default, the delimiter is white space).
- `read.fwf()` for data in a fixed-width format.
- `read.csv()` for CSV files.

There is also a function `readLines()` that creates a character vector from a text file, where each line of the text file becomes a separate element of the vector. This is useful for processing the text within a file that does not have a standard format (see Section 11.8).

11.6.3 Case Study: Point Nemo (continued)

The temperature data obtained from NASA's Live Access Server for the Pacific Pole of Inaccessibility (see Section 1.1) was delivered in a plain text format (see Figure 11.7, which reproduces Figure 1.2 for convenience). How can we load this temperature information into R?

One way to view the format of the file in Figure 11.7 is that the data start on line 9 and data values are separated by whitespace. We will use the `read.table()` function to read the Point Nemo temperature information and create a data frame.

```
             VARIABLE : Mean TS from clear sky composite (kelvin)
             FILENAME : ISCCPMonthly_avg.nc
             FILEPATH : /usr/local/fer_data/data/
             SUBSET   : 48 points (TIME)
             LONGITUDE: 123.8W(-123.8)
             LATITUDE : 48.8S
                       123.8W
                        23
  16-JAN-1994 00 /  1:  278.9
  16-FEB-1994 00 /  2:  280.0
  16-MAR-1994 00 /  3:  278.9
  16-APR-1994 00 /  4:  278.9
  16-MAY-1994 00 /  5:  277.8
  16-JUN-1994 00 /  6:  276.1
  ...
```

Figure 11.7: The first few lines of output from the Live Access Server for the surface temperature at Point Nemo. This is a reproduction of Figure 1.2.

```
R> pointnemotemp <-
       read.table("pointnemotemp.txt", skip=8)
R> head(pointnemotemp)

           V1 V2 V3 V4    V5
1 16-JAN-1994  0  / 1: 278.9
2 16-FEB-1994  0  / 2: 280.0
3 16-MAR-1994  0  / 3: 278.9
4 16-APR-1994  0  / 4: 278.9
5 16-MAY-1994  0  / 5: 277.8
6 16-JUN-1994  0  / 6: 276.1
```

By default, `read.table()` assumes that the text file contains a data set with one case on each row and that each row contains multiple values, with each value separated by white space (one or more spaces or tabs). The `skip` argument is used to ignore the first few lines of a file when, for example, there is header information or metadata at the start of the file before the actual data values.

A data frame is produced with a variable for each column of values in the text file. The types of variables are determined automatically; if a column only contains numbers, the variable is numeric, otherwise, the variable is a factor.

The names of the variables in the data frame can be read from the file, or

specified explicitly in the call to `read.table()`. Otherwise, as in this case, R will generate a unique name for each column.

The result in this case is not perfect because we end up with several columns of junk that we do not want. We can use a few more arguments to `read.table()` to improve things greatly.

```
R> pointnemotemp <-
       read.table("pointnemotemp.txt", skip=8,
                  colClasses=c("character",
                    "NULL", "NULL", "NULL",
                    "numeric"),
                  col.names=c("date", "", "", "", "temp"))
R> head(pointnemotemp)

         date  temp
1 16-JAN-1994 278.9
2 16-FEB-1994 280.0
3 16-MAR-1994 278.9
4 16-APR-1994 278.9
5 16-MAY-1994 277.8
6 16-JUN-1994 276.1
```

The `colClasses` argument allows us to control the types of the variables explicitly. In this case, we have forced the first variable to be just text (these values are dates, not categories). There are five columns of values in the text file (treating white space as a column break), but we are not interested in the middle three, so we use `"NULL"` to indicate that these columns should not be included in the data frame.

It is common for the names of the variables to be included as the first line of a text file (the `header` argument can be used to read variable names from such a file). In this case, we provide the variable names explicitly, using the `col.names` argument.

The `read.table()` function is quite flexible and can be used for a variety of plain text formats. If the format is too complex for `read.table()` to handle, the `scan()` function may be able to help; this function is also useful for reading in very large text files because it is faster than `read.table()`.

Reading a fixed-width format

Another way to view the Point Nemo text file is as a fixed-width format file. For example, the date data always resides in the first 11 characters of

each line. This means that we could also read the file using `read.fwf()`.

```
R> pointnemotemp <-
       read.fwf("pointnemotemp.txt", skip=8,
                widths=c(12, -10, 6),
                colClasses=c("character", "numeric"),
                col.names=c("date", "temp"))
R> head(pointnemotemp)

          date  temp
1  16-JAN-1994 278.9
2  16-FEB-1994 280.0
3  16-MAR-1994 278.9
4  16-APR-1994 278.9
5  16-MAY-1994 277.8
6  16-JUN-1994 276.1
```

The `widths` argument specifies how wide each column of data is, with negative values used to ignore the specified number of characters.

One thing that should be disturbing about the previous examples is that they have just completely ignored all of the metadata in the head of the file. This information is also very important and we would like to have some way to access it.

The `readLines()` function can help us here, at least in terms of getting raw text into R. The following code reads the first 8 lines of the text file into a character vector.

```
R> readLines("pointnemotemp.txt", n=8)

[1] "             VARIABLE : Mean TS from clear sky composite (kelvin)"
[2] "             FILENAME : ISCCPMonthly_avg.nc"
[3] "             FILEPATH : /usr/local/fer_dsets/data/"
[4] "             SUBSET   : 93 points (TIME)"
[5] "             LONGITUDE: 123.8W(-123.8)"
[6] "             LATITUDE : 48.8S"
[7] "                       123.8W "
[8] "                        23"
```

Section 11.8 will describe some tools that could be used to extract the metadata values from this text.

```
date,temp
 16-JAN-1994,278.9
 16-FEB-1994,280
 16-MAR-1994,278.9
 16-APR-1994,278.9
 16-MAY-1994,277.8
 16-JUN-1994,276.1
 16-JUL-1994,276.1
 16-AUG-1994,275.6
...
```

Figure 11.8: The first few lines of the surface temperature at Point Nemo in a cleaned up CSV format.

Writing a CSV format

As a simple demonstration of the use of functions that can write plain text files, we can export the R data frame containing the Point Nemo values in a CSV format using `write.csv()` (see Figure 11.8).

```
R> write.csv(pointnemotemp, "pointnemoplain.csv",
             quote=FALSE, row.names=FALSE)
```

The `quote` argument controls whether quote-marks are printed around text values and the `row.names` argument controls whether an extra column of unique names is printed at the start of each line.

We will continue to use the Point Nemo data set, in various formats throughout the rest of this section.

11.6.4 XML

It is possible to import data from an XML document into R using a standard function like `readLines()` because XML is just text. However, extracting the information from the text is not trivial because it requires knowledge of XML.

Fortunately, there is an R package called `XML` that contains functions for reading and extracting data from XML files into R.

The R objects that are generated from reading XML files are complex list structures and should be worked with using the functions provided by the `XML` package.

```
<?xml version="1.0"?>
<temperatures>
    <variable>Mean TS from clear sky composite (kelvin)</variable>
    <filename>ISCCPMonthly_avg.nc</filename>
    <filepath>/usr/local/fer_dsets/data/</filepath>
    <subset>93 points (TIME)</subset>
    <longitude>123.8W(-123.8)</longitude>
    <latitude>48.8S</latitude>
    <case date="16-JAN-1994" temperature="278.9" />
    <case date="16-FEB-1994" temperature="280" />
    <case date="16-MAR-1994" temperature="278.9" />
    <case date="16-APR-1994" temperature="278.9" />
    <case date="16-MAY-1994" temperature="277.8" />
    <case date="16-JUN-1994" temperature="276.1" />

    ...

</temperatures>
```

Figure 11.9: The first few lines of the surface temperature at Point Nemo in two formats: plain text and XML. This is a reproduction of the top half of Figure 7.7.

On the other hand, these objects are excellent examples of data structures with a specific *class*, so there are also a number of familiar *generic functions* that will work appropriately with objects generated by the `XML` pacakage.

The following case study demonstrates some of the functions from the `XML` package.

The Point Nemo temperature data (see Section 1.1) were originally obtained in a plain text format (see Figure 11.7). Figure 11.9 shows one possible XML format for the same information.

Using the `xmlTreeParse()` function from the `XML` package, this file can be read into R quite easily.

```
R> library(XML)

R> nemoDoc <-
       xmlTreeParse(file.path("LAS",
                              "pointnemotemp.xml"))
```

However, the `nemoDoc` object is relatively complex so we must use special functions to extract information from it. For example, the `xmlRoot()` func-

tion extracts the "root" or top-most element from the document.

For each element, it is simple to extract the name of the element using `xmlName()`. In our example, the root element of the document is a `temperatures` element.

```
R> nemoDocRoot <- xmlRoot(nemoDoc)
R> xmlName(nemoDocRoot)

[1] "temperatures"
```

The root element is itself quite complex; in particular, it is hierarchical to reflect the nested nature of the XML elements in the original document. However, normal R subsetting operations can be used to extract child elements of the root element.

For example, the first child element is a `variable` element describing the variable that is stored in the data set.

```
R> nemoDocRoot[[1]]

<variable>Mean TS from clear sky composite (kelvin)</variable>
```

Because XML elements all have names, elements can also be extracted using text indices.

```
R> nemoDocRoot[["variable"]]

<variable>Mean TS from clear sky composite (kelvin)</variable>
```

The values of attributes or the contents of individual elements can be obtained using the functions `xmlAttrs()` and `xmlValue()`.

```
R> xmlValue(nemoDocRoot[["variable"]])

[1] "Mean TS from clear sky composite (kelvin)"
```

Where there are several elements with the same name, subsetting will return only the first element with the appropriate name. Here we extract the temperature value from the first `case` element (note that it is a text value!).

```
R> xmlGetAttr(nemoDocRoot[["case"]], "temperature")

[1] "278.9"
```

Single-square brackets, plus additional `all` argument can be used to return

all elements with a particular name. The result in this case is a list of XML elements.

```
R> head(nemoDocRoot["case", all=TRUE])

$case
<case date="16-JAN-1994" temperature="278.9"/>

$case
<case date="16-FEB-1994" temperature="280"/>

$case
<case date="16-MAR-1994" temperature="278.9"/>

$case
<case date="16-APR-1994" temperature="278.9"/>

$case
<case date="16-MAY-1994" temperature="277.8"/>

$case
<case date="16-JUN-1994" temperature="276.1"/>
```

Some of the concepts from Section 11.7 for working with list objects will be needed to become fully proficient with the `XML` package. For example, extracting all of the temperature data values requires calling the `xmlGetAttr()` function on each `case` element of the document root.

```
R> nemoDocTemps <-
       as.numeric(sapply(nemoDocRoot["case", all=TRUE],
                         xmlGetAttr, "temperature"))
R> nemoDocTemps

 [1] 278.9 280.0 278.9 278.9 277.8 276.1 276.1 275.6 275.6
[10] 277.3 276.7 278.9 281.6 281.1 280.0 278.9 277.8 276.7
[19] 277.3 276.1 276.1 276.7 278.4 277.8 281.1 283.2 281.1
[28] 279.5 278.4 276.7 276.1 275.6 275.6 276.1 277.3 278.9
[37] 280.5 281.6 280.0 278.9 278.4 276.7 275.6 275.6 277.3
[46] 276.7 278.4 279.5
```

Another way to extract elements from an XML object is to make use of the XPath language that was described in Section 9.3.1. The XML document must be read into R in a slightly different manner (to avoid wasting memory) and then the `xpathApply()` function can be used to extract elements, attribute, and their values. In the following example, we ex-

tract the `temperature` attribute values from all `case` elements. The XPath `"/temperatures/case/@temperature"` selects all of the `temperature` attributes of the `case` elements within the root `temperatures` element.

```
R> nemoInternalDoc <-
       xmlTreeParse(file.path("LAS",
                              "pointnemotemp.xml"),
                    useInternalNodes=TRUE)
R> nemoDocTemps <-
       as.numeric(
           unlist(
               xpathApply(
                   nemoInternalDoc,
                   "/temperatures/case/@temperature",
                   xmlValue)))
R> free(nemoInternalDoc)
```

Again, a complete understanding of this code may only come after a greater familiarity with list-handling functions has been achieved.

11.6.5 Binary files

As discussed in Section 7.5, it is only possible to extract data from a binary file with an appropriate piece of software.

This obstacle is less of a problem for binary formats that are publicly documented because it is then possible (at least in principle) to write software that can read the format. In some cases, such software has already been written and made available to the public. An example is the NetCDF software library.[11]

A number of R packages exist for reading particular formats. For example, the `foreign` package contains functions for reading files produced by other popular statistics software systems, such as SAS, SPSS, Systat, Minitab, and Stata. The `ncdf` package provides functions for reading NetCDF files.

We will look again at the Point Nemo temperature data (see Section 1.1) to demonstrate a simple use of the `ncdf` package. Yet another format for the Point Nemo temperature data is as a NetCDF file. Figure 11.10 shows both a structured and an unstructured view of the raw binary file.

A NetCDF file has a flexible structure, but it is self-describing. This means that we must read the file in stages. First of all, we open the file and inspect

[11] `http://www.unidata.ucar.edu/software/netcdf/`

```
  0  :  43 44 46 01 00 00  |  CDF...
  6  :  00 00 00 00 00 0a  |  ......
 12  :  00 00 00 01 00 00  |  ......
 18  :  00 04 54 69 6d 65  |  ..Time
 24  :  00 00 00 5d 00 00  |  ...]..
 30  :  00 00 00 00 00 00  |  ......
 36  :  00 00 00 0b 00 00  |  ......
 42  :  00 02 00 00 00 04  |  ......
 48  :  54 69 6d 65 00 00  |  Time..
 54  :  00 01 00 00 00 00  |  ......
 ...
```

```
=========magicNumber
  0  :  43 44 46 01        |  CDF.
=========numRecords
  4  :  00 00 00 00        |  0
=========dimensionArrayFlag
  8  :  00 00 00 0a        |  10
=========dimensionArraySize
 12  :  00 00 00 01        |  1
=========dim1NameSize
 16  :  00 00 00 04        |  4
=========dim1Name
 20  :  54 69 6d 65        |  Time
=========dim1Size
 24  :  00 00 00 5d        |  93
...
```

Figure 11.10: The first few bytes of the surface temperature at Point Nemo in a NetCDF format. The top view shows the unstructures bytes and the lower view shows a more meaningful interpretation of the first few components of the file. The latter reveals that there is a single dimension variable in the NetCDF file, which is named `Time` and has 93 values.

the contents.

```
R> library(ncdf)

R> nemonc <- open.ncdf("pointnemotemp.nc")
R> nemonc

file pointnemotemp.nc has 1 dimensions:
Time   Size: 93
------------------------
file pointnemotemp.nc has 1 variables:
float Temperature[Time]  Longname:Temperature
```

Having seen that the file contains a single variable called `Temperature`, we extract that variable from the file with a separate function call.

```
R> nemoTemps <- get.var.ncdf(nemonc, "Temperature")
R> nemoTemps

 [1] 278.9 280.0 278.9 278.9 277.8 276.1 276.1 275.6 275.6
[10] 277.3 276.7 278.9 281.6 281.1 280.0 278.9 277.8 276.7
[19] 277.3 276.1 276.1 276.7 278.4 277.8 281.1 283.2 281.1
[28] 279.5 278.4 276.7 276.1 275.6 275.6 276.1 277.3 278.9
[37] 280.5 281.6 280.0 278.9 278.4 276.7 275.6 275.6 277.3
[46] 276.7 278.4 279.5 282.2 281.6 281.6 280.0 278.9 277.3
[55] 276.7 276.1 276.1 276.1 277.8 277.3 278.4 284.2 279.5
[64] 277.3 278.4 275.0 275.6 274.4 275.6 276.7 276.1 278.4
[73] 279.5 279.5 278.9 277.8 277.8 275.6 275.6 275.0 274.4
[82] 275.6 277.3 278.4 281.1 283.2 281.1 279.5 277.3 276.7
[91] 275.6 274.4 275.0
```

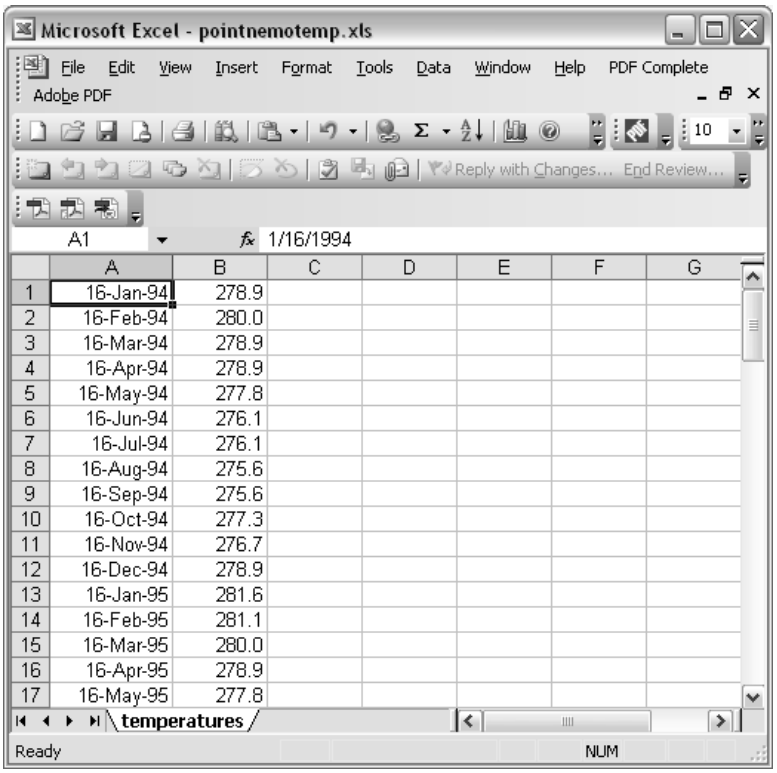
In most cases, where a function exists to read a particular binary format, there will also be a function to write data out in that format.

11.6.6 Spreadsheets

When data is stored in a spreadsheet, one common approach is to save the data in a text format as an intermediate step and then read the text file into another program.

This makes the data easy to share, because text formats are very portable, but it has the disadvantage that another copy of the data is created.

This is less efficient in terms of storage space and it creates issues if the



	A	B
1	16-Jan-94	278.9
2	16-Feb-94	280.0
3	16-Mar-94	278.9
4	16-Apr-94	278.9
5	16-May-94	277.8
6	16-Jun-94	276.1
7	16-Jul-94	276.1
8	16-Aug-94	275.6
9	16-Sep-94	275.6
10	16-Oct-94	277.3
11	16-Nov-94	276.7
12	16-Dec-94	278.9
13	16-Jan-95	281.6
14	16-Feb-95	281.1
15	16-Mar-95	280.0
16	16-Apr-95	278.9
17	16-May-95	277.8

Figure 11.11: The Point Nemo data stored as an Excel spreadsheet.

original spreadsheet is updated.

If changes are made to the original spreadsheet, at best, there is extra work to be done to update the text file as well. At worst, the text file is forgotten and the update does not get propagated to other places.

There are several packages that provide ways to directly read data from a spreadsheet into R. One example is the (Windows only) `xlsReadWrite` package, which includes the `read.xls()` function.

Figure 11.11 shows a screen shot of the Point Nemo temperature data (see Section 1.1) stored in a Microsoft Excel spreadsheet.

These data can be read into R using the following code.

```
> library(xlsReadWrite)

> read.xls("temperatures.xls", colNames=FALSE)
      V1    V2
1  34350 278.9
2  34381 280.0
3  34409 278.9
4  34440 278.9
5  34470 277.8
6  34501 276.1
...
```

Notice that the date information has come across as numbers (specifically, the number of days since the 1st of January 1900).

11.6.7 Large data sets

Very large data sets are often stored in relational database management systems. Again, a simple approach to extracting information from the database is to simply export it as text files and work with the text files. This is an even worse option for databases than it was for spreadsheets because it is more common to extract just part of a database, rather than an entire spreadsheet. This can lead to several different text files from a single database and these are even harder to maintain if the database changes.

There are packages for connecting directly to several major database management systems. Two main approaches exist, one based on the `DBI` package and one based on the `RODBC` package.

The `DBI` package defines a set of standard (generic) functions for communicating with a database and a number of other packages, e.g., `RMySQL` and `RSQLite`, build on that to provide functions specific to a particular database system. The important functions to know about with this approach are:

`dbDriver()`
: to create a "device driver" for the relevant database system.

`dbConnect()`
: to connect to the database.

`dbGetQuery()`
: to send an SQL statement to the database and receive a result, as a data frame.

`dbDisconnect()`
: to sever the connection with the database and release resources.

The `RODBC` package defines functions for communicating with any ODBC (Open Database Connectivity) compliant software. This allows connections with many different types of software, including, but not limited to, most database management systems. The important functions to know about with this approach are:

`odbcConnect()`
: to connect to the ODBC application.

`sqlQuery()`
: to send an SQL statement to the database and receive a result, as a data frame.

`odbcClose()`
: to sever the ODBC connection and release resources.

The simplest approach is provided by the `RSQLite` package because it includes the complete SQLite application, so no other software needs to be installed.

11.6.8 Case Study: The Data Expo (continued)

The Data Expo data set (see Section 7.3.6) contains several different atmospheric measurements, all measured at 72 different time periods and 576 different locations. These data have been stored in an SQLite database, with a table for location information, a table for time period information, and a table of the atmospheric measurements (see Section 9.2.2).

The following code extracts the average surface temperature across all locations and times.

```
R> library(RSQLite)

R> con <- dbConnect(dbDriver("SQLite"),
                    dbname="NASA/dataexpo")
R> result <-
       dbGetQuery(con,
                  "SELECT AVG(surftemp) avgtemp
                      FROM measure_table")
R> dbDisconnect(con)
```

```
R> result

   avgtemp
1 296.2311
```

11.6.9 Basic file manipulations

Some data management tasks do not involve the contents of files at all, but are only concerned with reorganising entire sets of files. Examples include moving files between directories and renaming files.

These are tasks that are commonly performed via a GUI that is provided by the operating system, for example, Windows Explorer on Windows or the Finder on a Macintosh. However, as with most tasks we have discussed, if a large number of files are involved it is much more efficient and much less error-prone to perform these tasks by writing a script.

On the other hand, these operations do require more caution than usual because, if a command is entered incorrectly, the effect is felt on the file system, rather than just within the current R session, so mistakes are much harder to recover from.

R provides functions for basic file manipulation, including `list.files()` for listing the files in a directory, `file.copy()` for moving files between directories, and `file.rename()` for renaming files.

11.6.10 Case study: Digital photography

Digital camera.[12]

Digital cameras have revolutionised photography.

No longer are we restricted to a mere 26 shots per film; it is now common

[12]Source: OpenClipart Library
`http://openclipart.org/clipart//computer/hardware/digital-camera_aj_ashton_01.svg`
This image is in the Public Domain.

to be able to take hundreds of photographs on a basic camera. No longer do we have to process a film in order to find out whether we have captured the perfect shot; we can preview photographs instantly and photos can be viewed in all their glory with a simple download to a computer. Printing photographs is instantaneous and sharing photographs with friends and family is almost too easy.

Unfortunately, digital cameras have also created a serious headache for many amateur snappers.

Every digital camera owner has now acquired the task of storing and maintaining thousands of computer files. Keeping track of this many files is a novel, often unpleasant, and occasionally hair-raising experience for unsophisticated computer users.

As an example, we will look at a small subset of digital camera files that have been stored in a directory called `"Photos"`. Every time photos were downloaded from the camera, they were stored in a separate directory, which was named after the date of the download. The `list.files()` function can be used to show the names of these directories:

```
R> directories <- list.files("Photos")
R> directories

 [1] "061111"  "061118"  "061119"  "061207"  "061209"
 [6] "061216"  "061219"  "061231"  "06Nov05" "06Oct05"
[11] "06Oct15" "06Oct28" "06Sep17" "070103"  "070105"
[16] "070107"  "070108"  "070113"  "070114"  "070117"
[21] "070202"  "070218"  "070223"  "070303"  "070331"
```

We can also list the individual files within each of the directories. The following code shows that 11 files were downloaded on the 11^{th} of November 2006:

```
R> list.files(file.path("Photos", "061111"),
              pattern=".jpg")

 [1] "061111 001.jpg" "061111 002.jpg" "061111 005.jpg"
 [4] "061111 009.jpg" "061111 013.jpg" "061111 016.jpg"
 [7] "061111 021.jpg" "061111 022.jpg" "061111 025.jpg"
[10] "061111 026.jpg" "061111 028.jpg"
```

Going back to the list of directories we can see that, unfortunately, the naming of these directories has been a little undisciplined. Many of the directories are named using a `YYMMDD` format where year, month, and day are all represented by two-digit integers. However, some of the earlier directories

used a slightly different format: `YYmmmDD`, where year and day are two-digit integers, but month is a three-character abbreviated name.

There are two problems here: first, the `YYMMDD` can be ambiguous because in a number of cases it is indistinguishable from `YYDDMM` (among others); second, inconsistency in the naming scheme can become a major headache when dealing with the photos later on. It would be much nicer if the directories had a consistent, unambiguous naming scheme.

The international standard for expressing dates (ISO 8601) specifies a `YYYY-MM-DD` format (four-digit year, two-digit month, and two-digit day). Our task will be to rename the directories to an ISO 8601 format.

One way to perform this task would be via a GUI, such as Windows Explorer, but it should be clear by now that writing code to perform the task instead will be faster, more accurate, and more fun.

The first part of the task is to conver the directory names to the ISO 8601 format. The following code uses the `sub()` function to replace `"Nov"` with `"11"`, `"Oct"` with `10`, and so on, in all directory names.[13]

```
R> newDirs <- sub("Nov", "11", directories)
R> newDirs <- sub("Oct", "10", newDirs)
R> newDirs <- sub("Sep", "09", newDirs)
R> newDirs

 [1] "061111" "061118" "061119" "061207" "061209" "061216"
 [7] "061219" "061231" "061105" "061005" "061015" "061028"
[13] "060917" "070103" "070105" "070107" "070108" "070113"
[19] "070114" "070117" "070202" "070218" "070223" "070303"
[25] "070331"
```

Now that the directory names are all in the same `YYMMDD` format, we can easily convert them to dates.

```
R> dateDirs <- as.Date(newDirs, format="%y%m%d")
R> dateDirs

 [1] "2006-11-11" "2006-11-18" "2006-11-19" "2006-12-07"
 [5] "2006-12-09" "2006-12-16" "2006-12-19" "2006-12-31"
 [9] "2006-11-05" "2006-10-05" "2006-10-15" "2006-10-28"
[13] "2006-09-17" "2007-01-03" "2007-01-05" "2007-01-07"
[17] "2007-01-08" "2007-01-13" "2007-01-14" "2007-01-17"
[21] "2007-02-02" "2007-02-18" "2007-02-23" "2007-03-03"
[25] "2007-03-31"
```

[13] Section 11.8 contains many more examples of this sort of text manipulation.

Now we can loop over the directory names and change the original name to one based on the date in a standard format. This is the crucial step in this task; writing R code to perform file renaming automatically, rather than doing it by hand via a GUI.

```
R> numDirs <- length(directories)
R> for (i in 1:n) {
       file.rename(file.path("Photos", directories[i]),
                   file.path("Photos", dateDirs[i]))
   }
```

The files are all now in a common, standard format.

```
R> list.files("Photos")

 [1] "2006-09-17" "2006-10-05" "2006-10-15" "2006-10-28"
 [5] "2006-11-05" "2006-11-11" "2006-11-18" "2006-11-19"
 [9] "2006-12-07" "2006-12-09" "2006-12-16" "2006-12-19"
[13] "2006-12-31" "2007-01-03" "2007-01-05" "2007-01-07"
[17] "2007-01-08" "2007-01-13" "2007-01-14" "2007-01-17"
[21] "2007-02-02" "2007-02-18" "2007-02-23" "2007-03-03"
[25] "2007-03-31"
```

One final word of warning: it is important to remember that file manipulations like this are a one-way trip. The original file names are lost. At a minimum, If we ever need to be able to go back to the original file names, for example, if we want to match the new directories with old versions of the directories from a backup, we should keep a record of how the files got their new names. One simple way to do this is to make sure that we store the R code that we used in a file and that we store that file in the `"Photos"` directory. In other words, we use the code as documentation of what we did.

11.7 Data manipulation

This section describes a number of techniques for rearranging objects in R, particularly larger data structures, such as data frames, matrices, and lists.

11.7.1 Sorting

One of the simplest manipulations that we can perform is to sort a set of values into ascending or descending order.

In R, the function `sort()` can be used to arrange a vector of values in order, but of more general use is the `order()` function, which returns the indices of the sorted values. An example will demonstrate the difference.

11.7.2 Case study: Counting Candy (continued)

The `candy` data frame (see Section 11.3.1) contains a record of the number of candies of various different types that appear in a simple counting puzzle. The full `candy` data frame is reproduced below.

```
R> candy

   shape pattern shade count
1  round pattern light     2
2   oval pattern light     0
3   long pattern light     3
4  round   plain light     1
5   oval   plain light     3
6   long   plain light     2
7  round pattern  dark     9
8   oval pattern  dark     0
9   long pattern  dark     2
10 round   plain  dark     1
11  oval   plain  dark    11
12  long   plain  dark     2
```

We can use the `sort()` function to arrange the counts of the different types of candies in ascending or descending order.

```
R> sort(candy$count)

 [1]  0  0  1  1  2  2  2  2  3  3  9 11

R> sort(candy$count, decreasing=TRUE)

 [1] 11  9  3  3  2  2  2  2  1  1  0  0
```

However, this result is of limited use because we have lost the crucial connection between the counts and the type of candy that each count corresponds to.

This is where the `order()` function may be more useful. The result below shows the order in which the values of the `count` variable need to be used so that they are arranged in ascending order.

```
R> order(candy$count)

 [1]  2  8  4 10  1  6  9 12  3  5  7 11
```

These values can be used, via subsetting, to arrange the entire `candy` data set by ascending values of the `count` variable, while still keeping the connection between types of candy and the appropriate count.

```
R> candy[order(candy$count), ]

   shape pattern shade count
2   oval pattern light     0
8   oval pattern  dark     0
4  round   plain light     1
10 round   plain  dark     1
1  round pattern light     2
6   long   plain light     2
9   long pattern  dark     2
12  long   plain  dark     2
3   long pattern light     3
5   oval   plain light     3
7  round pattern  dark     9
11  oval   plain  dark    11
```

11.7.3 The "apply" functions

The basic operation that underlies almost all of these techniques involves performing some task on each sub-component or sub-group of a larger structure. A very simple example is the task of subtracting the mean from each of a set of numbers, as shown below.

```
R> sample <- sample(1:10, 5)
R> sample

[1] 2 1 9 8 3

R> sampleMean <- mean(sample)
R> sampleMean

[1] 4.6
```

```
R> sample - sampleMean

[1] -2.6 -3.6  4.4  3.4 -1.6
```

For each value in `sample`, we subtract the value 4.6.

The traditional approach to this problem in a general-purpose language is to perform a loop (see Section 11.2.5), but, as the code above shows, this can often be avoided because R automatically performs operations element-wise and because R provides several functions that automatically perform a task once for each sub-component of a larger data structure.

The idea of the set of "apply" functions in R is to allow us to call a function for each row or column of a matrix, or for each component of a list.

We will use the `candy` data frame again to demonstrate some uses of the "apply" functions.

Given a data frame like this, we might be interested in checking what sort of variables are in the data frame. The `class()` function provides the information we need; for example, the following code shows us that the `shape` variable is a factor.

```
R> class(candy$shape)

[1] "factor"
```

What we want to do is to repeat this code for each variable in the data set. This is possible by typing a similar expression for each variable in the data, but we can do it much more simply using a function called `lapply()`.

The idea of this function is that it will call a given function for each component of a list. We have a data frame, not a list, but automatic type coercion will take care of that problem for us (see Section 11.5.1).

The following code calls the function `class()` for each variable in the data frame `candy`.

```
R> lapply(candy, class)

$shape
[1] "factor"

$pattern
[1] "factor"

$shade
[1] "factor"

$count
[1] "numeric"
```

The result, which is a list, shows us that the first three variables in the data frame are factors, and the last variable is numeric (integer).

When the results of each function call are very simple, like in this case, it is useful to know about a very similar function called `sapply()`. This function does the same thing as `lapply()`, but tries to simplify the result to a vector rather than a list if it can. In this example, the result can be more succinctly represented as a character vector.

```
R> sapply(candy, class)

    shape   pattern     shade     count
 "factor"  "factor"  "factor" "numeric"
```

We will now look at a more sophisticated use of `lapply()`.

The data frame `candy` provides a somewhat abbreviated form for the data, because it records only the number of occurrences of each possible combination of candy characteristics. Another way to represent the data is as a case per candy, with three variables giving the pattern, shade, and shape of each piece of candy (see Figure 11.12).

The following code produces what we want for the `shape` variable. We repeat each of the values in the shape variable the appropriate number of times, as given by the `count` variable.

```
R> rep(candy$shape, candy$count)

 [1] round round long  long  long  round oval  oval  oval
[10] long  long  round round round round round round round
[19] round round long  long  round oval  oval  oval  oval
[28] oval  oval  oval  oval  oval  oval  oval  long  long
Levels: round oval long
```

We could perform this operation on each variable and explicitly glue the new variables back together, as follows.

```
R> candyCases <-
       data.frame(shape=rep(candy$shape, candy$count),
                  pattern=rep(candy$pattern, candy$count),
                  shade=rep(candy$shade, candy$count))
```

This is a feasible approach for a small number of variables, but for larger data sets, and for pure elegance, we will demonstrate how to do it using `lapply()`.

```
R> candyCasesList <- lapply(candy[, 1:3], rep, candy$count)
R> candyCases <- data.frame(candyCasesList)
```

Three important details in this code are worth noting. First, we do not send the entire data frame to `lapply()`; instead, we use subsetting to send just the first three variables. Second, we have supplied *three* arguments to `lapply()`: the data frame to work with; the function to call, in this case, the `rep()` function; *and* an argument which is passed on as an argument to the calls to `rep()`. The result is the same as if we had written the following code:

```
list(shape=rep(candy[, 1], candy$count),
     pattern=rep(candy[, 2], candy$count),
     shade=rep(candy[, 3], candy$count))
```

The final detail is that the result of `lapply()` is a list, so we need to explicitly convert it back into a data frame. The end result is shown in Figure 11.12.

Before we leave this example, we will also use it to demonstrate the `apply()` function.

The `apply()` function works on matrices or arrays, rather than lists. It allows us to call a function for each column (or for each row) of a matrix. For this example, we can treat the first three variables of the candy data set

```
R> candyCases

   shape pattern shade
1  round pattern light
2  round pattern light
3   long pattern light
4   long pattern light
5   long pattern light
6  round   plain light
7   oval   plain light
8   oval   plain light
9   oval   plain light
10  long   plain light
11  long   plain light
12 round pattern  dark
13 round pattern  dark
14 round pattern  dark
15 round pattern  dark
16 round pattern  dark
17 round pattern  dark
18 round pattern  dark
19 round pattern  dark
20 round pattern  dark
21  long pattern  dark
22  long pattern  dark
23 round   plain  dark
24  oval   plain  dark
25  oval   plain  dark
26  oval   plain  dark
27  oval   plain  dark
28  oval   plain  dark
29  oval   plain  dark
30  oval   plain  dark
31  oval   plain  dark
32  oval   plain  dark
33  oval   plain  dark
34  oval   plain  dark
35  long   plain  dark
36  long   plain  dark
```

Figure 11.12: The candy data set in a case-per-candy format; each row describes the shape, pattern, and shade characteristics of a single piece of candy from the picture on page 246.

as a matrix (of strings) with three columns, and apply the `rep()` function to each column (see Figure 11.12).

```
R> candyCasesMatrix <- apply(candy[,1:3], 2, rep, candy$count)
R> candyCases <- data.frame(candyCasesMatrix, row.names=NULL)
```

The second argument to `apply()` specifies whether to call the function for each row of the matrix (`1`) or for each column of the matrix (`2`). The result of `apply()` is a matrix, so we use `data.frame()` to turn the final result back into a data frame.

A number of other apply functions exist (see Section 12.4.7), but we will only mention one other at this stage called `tapply()`.

The idea of `tapply()` is to call a function once for each of a set of categories. We supply a variable that will be sliced into separate groups, a categorical variable that indicates which group each observation belongs to, and a function to call for each group.

For example, we can use this function to sum up the total number of light and dark candies in the `candy` data set. The `count` variable provides us with the counts to sum up, the `shade` variable indicates which counts correspond to light candies and which to dark, and the `sum()` function can do the summation. The appropriate call to `tapply()` is shown below.

```
R> tapply(candy$count, candy$shade, sum)

light  dark
   11    25
```

Further examples of the use of the apply functions appear in many of the examples and case studies throughout the rest of this chapter.

11.7.4 Tables of Counts

In the previous section, we saw an example where a data set was transformed from a table of counts into a row-per-case format (see page 290). The reverse operation is also common: given one or more categorical variables, collapse the data into a simple count of the number of times each possible category occurs.

Using the candy data set in a case-per-candy format (see Figure 11.12), we will explore the standard functions for generating a table of counts.

First up is the `table()` function, which produces counts for each combina-

tion of the levels of the factors it is given.

```
R> candyTable <- table(candyCases)
R> candyTable

, , shade = dark

       pattern
shape   pattern plain
  long        2     2
  oval        0    11
  round       9     1

, , shade = light

       pattern
shape   pattern plain
  long        3     2
  oval        0     3
  round       2     1
```

The `ftable()` function produces the same result, but as a "flat" (2-dimensional) contingency table, which can be easier to read.

```
R> candyFTable <- ftable(candyCases)
R> candyFTable

              shade dark light
shape pattern
long  pattern          2     3
      plain            2     2
oval  pattern          0     0
      plain           11     3
round pattern          9     2
      plain            1     1
```

The results of these functions are arrays or matrices, but they can easily be converted back to a data frame, like the one we originally entered in Section 11.3.1.

```
R> as.data.frame(candyTable)

   shape pattern shade Freq
1   long pattern  dark    2
2   oval pattern  dark    0
3  round pattern  dark    9
4   long   plain  dark    2
5   oval   plain  dark   11
6  round   plain  dark    1
7   long pattern light    3
8   oval pattern light    0
9  round pattern light    2
10  long   plain light    2
11  oval   plain light    3
12 round   plain light    1
```

The function `xtabs()` produces the same result as `table()`, but provides a formula interface for specifying the factors to tabulate.

```
R> candyXTable <- xtabs(~ shape + pattern + shade, candyCases)
R> candyXTable

, , shade = dark

       pattern
shape   pattern plain
  long        2     2
  oval        0    11
  round       9     1

, , shade = light

       pattern
shape   pattern plain
  long        3     2
  oval        0     3
  round       2     1
```

This function can also be used to create a frequency table from data which has been entered as counts, by specifying the variable containing the counts on the left-hand side of the formula. This means that we can create the table of counts from the original `candy` data frame.

```
R> candyXTable <- xtabs(count ~ shape + pattern + shade, candy)
```

11.7.5 Aggregation

Another way to collapse groups of observations into summary values (in this case counts) is to use the `aggregate()` function.

```
R> aggregate(candy["count"],
             list(shape=candy$shape, pattern=candy$pattern),
             sum)

  shape pattern count
1 round pattern    11
2  oval pattern     0
3  long pattern     5
4 round   plain     2
5  oval   plain    14
6  long   plain     4
```

One advantage of this function is that any summary function can be used (e.g., `mean()` instead of `sum()`). Another advantage is that the result of the function is a data frame, not a contingency table like from `table()` or `xtabs()`.

The `aggregate()` function is also like `tapply()` (see Section 11.7.3) because it calls a function for each sub-group of observations within a variable. The difference again is that the format of the result, a data frame, may be more convenient.

```
R> aggregate(candy["count"], list(shade=candy$shade), sum)

  shade count
1 light    11
2  dark    25

R> tapply(candy$count, candy$shade, sum)

light  dark
   11    25
```

11.7.6 Merging data sets

The functions `cbind()` and `rbind()` can be used to append data frames together—`cbind()` adds two sets of variables on common cases (a "column" bind) and `rbind()` adds two sets of cases on common variables (a "row"

bind).

For example, the following code creates a new data frame, `moreCandy`, with a single variable, `propn`, that contains the percentage of candies of each type.

```
R> moreCandy <- data.frame(propn=round(candy$count /
                                sum(candy$count), 2))
R> moreCandy

   propn
1   0.06
2   0.00
3   0.08
4   0.03
5   0.08
6   0.06
7   0.25
8   0.00
9   0.06
10  0.03
11  0.31
12  0.06
```

This data frame has the same number of rows as the `candy` data frame, so the two can be combined using `cbind()`.

```
R> cbind(candy, moreCandy)

   shape pattern shade count propn
1  round pattern light     2  0.06
2   oval pattern light     0  0.00
3   long pattern light     3  0.08
4  round   plain light     1  0.03
5   oval   plain light     3  0.08
6   long   plain light     2  0.06
7  round pattern  dark     9  0.25
8   oval pattern  dark     0  0.00
9   long pattern  dark     2  0.06
10 round   plain  dark     1  0.03
11  oval   plain  dark    11  0.31
12  long   plain  dark     2  0.06
```

The `merge()` function can be used to perform the equivalent of a database join (see Section 9.2.3) with two data frames.

As a simple example, consider the following data frame, which contains information about the two companies that produced the candies that we have been counting.

```
R> candyCompany <- data.frame(pattern=c("pattern", "plain"),
                              company=c("SuperChoc", "Chocs-R-Us"))
R> candyCompany

  pattern    company
1 pattern  SuperChoc
2   plain Chocs-R-Us
```

All patterned candies were made by a company called `SuperChoc` and all plain candies were made by `Chocs-R-Us`.

We would like to combine this information with the information about how many of each type of candy there is. In other words, we want to add a new variable to the `candy` data frame with the value `"Superchoc"` wherever the `pattern` variable has the value `"pattern"`, and `"Chocs-R-Us"` wherever the `pattern` is `"plain"`.

This task is achieved via the following call to the `merge()` function.

```
R> merge(candy, candyCompany)

   pattern shape shade count    company
1  pattern round light     2  SuperChoc
2  pattern  oval light     0  SuperChoc
3  pattern  long light     3  SuperChoc
4  pattern  oval  dark     0  SuperChoc
5  pattern  long  dark     2  SuperChoc
6  pattern round  dark     9  SuperChoc
7    plain  long light     2 Chocs-R-Us
8    plain round light     1 Chocs-R-Us
9    plain  oval light     3 Chocs-R-Us
10   plain round  dark     1 Chocs-R-Us
11   plain  oval  dark    11 Chocs-R-Us
12   plain  long  dark     2 Chocs-R-Us
```

The `merge()` function needs a column on which to match the rows of the two data sets, but if the data frames share a variable with the same name (as in this case) the function will automatically match on that column. The following code makes the matching column explicit.

```
R> merge(candy, candyCompany, by="pattern")
```

11.7.7 Reshaping

For multivariate data sets, where a single individual is measured several times, there are two common formats for storing the data.

The so-called "wide" format uses a separate variable for each measurement. For example, the `candyCases` data frame (see the left-hand box in Figure 11.13) is in wide format because it has a row for each candy and a column for each measurement made on each candy (what shape is the candy? what shade is the candy? and does the candy have a pattern?). There are *three obervations* on each row.

The "long" format for these data has a *single observation* on each row (see the right-hand box in Figure 11.13). In this format, there are three rows for each candy, with the `variable` column indicating which measure is being recorded on each row.

There is a function called `reshape()` to perform the transformation between wide and long formats in R, but we will focus on the `reshape` package because it provides a wider range of options.

The two main functions in this package are called `melt()` and `cast()`. The `melt()` function converts a data frame into long format and the `cast()` function can then be used to reshape the data into a variety of other forms.

For example, the following code creates the long format version of the case-per-candy data set (see the right-hand box in Figure 11.13). First, we add a unique identifier for each candy to the `candyCases` data frame (see Figure 292), then we use `melt()` to create the long format for the data.

```
R> library(reshape)
R> wideCandy <- cbind(candy=1:36, candyCases)
R> longCandy <- melt(wideCandy, measure=2:4)
```

The `measure` argument to `melt()` is used to specify which variables in the data frame are measurements. All other variables are treated as "identifiers"; fixed values that characterise the context of the measurements.

There is also an `id` argument to specify identifier variables, so the following call to `melt()` would produce the same result.

```
R> melt(wideCandy, id=1)
```

The wide format of the data can be reconstructed using `cast()`, as follows:

```
R> cast(longCandy, candy ~ variable)
```

```
R> wideCandy

   candy shape pattern shade
1      1 round pattern light
2      2 round pattern light
3      3  long pattern light
4      4  long pattern light
5      5  long pattern light
6      6 round   plain light
7      7  oval   plain light
8      8  oval   plain light
9      9  oval   plain light
10    10  long   plain light
11    11  long   plain light
12    12 round pattern  dark
13    13 round pattern  dark
14    14 round pattern  dark
15    15 round pattern  dark
16    16 round pattern  dark
17    17 round pattern  dark
18    18 round pattern  dark
19    19 round pattern  dark
20    20 round pattern  dark
21    21  long pattern  dark
22    22  long pattern  dark
23    23 round   plain  dark
24    24  oval   plain  dark
25    25  oval   plain  dark
26    26  oval   plain  dark
27    27  oval   plain  dark
28    28  oval   plain  dark
29    29  oval   plain  dark
30    30  oval   plain  dark
31    31  oval   plain  dark
32    32  oval   plain  dark
33    33  oval   plain  dark
34    34  oval   plain  dark
35    35  long   plain  dark
36    36  long   plain  dark
```

```
R> head(longCandy, n=40)

   candy variable   value
1      1    shape   round
2      2    shape   round
3      3    shape    long
4      4    shape    long
5      5    shape    long
6      6    shape   round
7      7    shape    oval
8      8    shape    oval
9      9    shape    oval
10    10    shape    long
11    11    shape    long
12    12    shape   round
13    13    shape   round
14    14    shape   round
15    15    shape   round
16    16    shape   round
17    17    shape   round
18    18    shape   round
19    19    shape   round
20    20    shape   round
21    21    shape    long
22    22    shape    long
23    23    shape   round
24    24    shape    oval
25    25    shape    oval
26    26    shape    oval
27    27    shape    oval
28    28    shape    oval
29    29    shape    oval
30    30    shape    oval
31    31    shape    oval
32    32    shape    oval
33    33    shape    oval
34    34    shape    oval
35    35    shape    long
36    36    shape    long
37     1  pattern pattern
38     2  pattern pattern
39     3  pattern pattern
40     4  pattern pattern
```

Figure 11.13: The case-per-candy data set in a wide format (left) and a long format (right). Only the first 40 rows of the long format are shown.

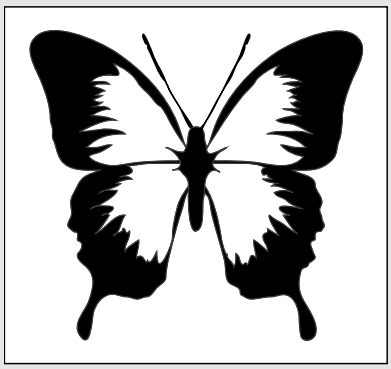
The second argument to `cast()` is a formula. Variables on the left-hand side of this formula are used to form the rows of the result and variables on the right-hand side are used to form columns. In the above example, there is a row for each candy and a column for each different measurement.

The following code uses a different formula to demonstrate the power that `cast()` provides for reshaping data. In this case, the data set is arranged in a format so that each candy is represented by a single column and each row of the data frame gives the measurements for a single variable (for reasons of space, only the first 6 candies are shown).

```
R> cast(longCandy, variable ~ candy)
```

```
  variable       1       2       3       4       5     6 ...
1    shape   round   round    long    long    long round ...
2  pattern pattern pattern pattern pattern pattern plain ...
3    shade   light   light   light   light   light light ...
```

11.7.8 Case study: Rothamsted moths



Despite the clear distinction that exists in popular culture, there is still controversy amongst taxonomists over how best to distinguish between butterfly species and moth species. The image to the left is "Farfallo contorno" (butterfly silhouette) by Architetto Francesco Rollandin.[14]

Rothamsted is a very large and very old agricultural research centre situated in Hertfordshire in the United Kingdom. One of its long term research projects is the Rothamsted Insect Survey, part of which involves the daily monitoring of nearly 100 "light traps" in order to count the number of moths of different species at various locations around the UK. The oldest light traps date back to 1933.

[14]Image source: Open Clip Art Library
`http://openclipart.org/clipart//animals/bugs/farfalla_contorno_archit_01.svg`
This image is in the Public Domain.

This example deals with a subset of the Rothamsted moth data in the form of two CSV format text files. One file contains 14 years worth of total yearly counts for a limited range of moth species. The first row of the file contains the species label and each subsequent row contains the counts of each species for a single year:

```
sp117,sp120,sp121,sp125,sp126,sp139,sp145,sp148,sp154, ...
2,1,1,0,0,1,0,3,0,1,3,4,4,0,0,9,2,6,0,0,0,16,31,2,40, ...
3,1,1,0,0,1,0,1,0,2,11,4,8,0,0,19,2,1,0,0,0,20,11,1, ...
2,0,3,6,0,1,0,1,0,1,9,1,18,0,0,15,15,0,1,0,0,14,12,5, ...
5,0,2,2,0,2,1,2,1,0,3,2,17,0,0,17,23,1,0,2,1,14,22,2, ...
...
```

Another file contains the species labels for the full set of 263 moth species for which counts exist, with one label per row. Notice that there are species labels in this file that do not appear in the file of counts (e.g., `sp108`, `sp109`, `sp110`, and `sp111`).

```
"x"
"sp108"
"sp109"
"sp110"
"sp111"
"sp117"
"sp120"
...
```

We can read in the counts using `read.csv()` to produce a data frame and we can read in the names using `scan()` to produce a simple character vector.

```
R> mothCounts <- read.csv(file.path("Rothamsted",
                                    "mothcounts.csv"))
R> mothCounts

  sp117 sp120 sp121 sp125 sp126 sp139 sp145 sp148 sp154 ...
1     2     1     1     0     0     1     0     3     0 ...
2     3     1     1     0     0     1     0     1     0 ...
3     2     0     3     6     0     1     0     1     0 ...
4     5     0     2     2     0     2     1     2     1 ...
...

R> mothNames <- scan(file.path("Rothamsted",
                               "mothnames.csv"),
                     what="character", skip=1)
R> mothNames
```

```
[1] "sp108" "sp109" "sp110" "sp111" "sp117" "sp120" ...
```

Our goal in this data manipulation example is to produce a single data set containing the count data from the file `mothCounts.csv` *plus* empty columns for all of the species for which we do not have counts. The end result will be a data frame as shown below, with columns of `NA`s where a species label occurs in `mothNames`, but not in `mothCounts`, and all other columns filled with the appropriate counts from `mothCounts`:

```
  sp108 sp109 sp110 sp111 sp117 sp120 sp121 sp125 sp126 ...
1    NA    NA    NA    NA     2     1     1     0     0 ...
2    NA    NA    NA    NA     3     1     1     0     0 ...
3    NA    NA    NA    NA     2     0     3     6     0 ...
4    NA    NA    NA    NA     5     0     2     2     0 ...
...
```

We will look at solving this problem in two ways. The first approach involves building up a large empty data set and then inserting the counts we know.

The first step is to create an empty data frame of the appropriate size. The `matrix()` function can create a matrix of the appropriate size and the `as.data.frame()` function converts it to a data frame. The function `colnames()` is used to give the columns of the data frame the appropriate names.

```
R> allMoths <- as.data.frame(matrix(NA,
                                    ncol=length(mothNames),
                                    nrow=14))
R> colnames(allMoths) <- mothNames
R> allmoths

  sp108 sp109 sp110 sp111 sp117 sp120 sp121 sp125 sp126 ...
1    NA    NA    NA    NA    NA    NA    NA    NA    NA ...
2    NA    NA    NA    NA    NA    NA    NA    NA    NA ...
3    NA    NA    NA    NA    NA    NA    NA    NA    NA ...
4    NA    NA    NA    NA    NA    NA    NA    NA    NA ...
...
```

Now we just fill in the columns where we know the moth counts. This is done simply using indexing; each column in the `mothCounts` data frame is assigned to the column with the same name in the `allMoths` data frame.

```
R> allMoths[, colnames(mothCounts)] <- mothCounts
R> allMoths
```

```
  sp108 sp109 sp110 sp111 sp117 sp120 sp121 sp125 sp126 ...
1    NA    NA    NA    NA     2     1     1     0     0 ...
2    NA    NA    NA    NA     3     1     1     0     0 ...
3    NA    NA    NA    NA     2     0     3     6     0 ...
4    NA    NA    NA    NA     5     0     2     2     0 ...
...
```

An alternative approach to the problem is to treat it like a database join. This time we start with a data frame that has a variable for each moth species, but no rows. The `list()` function creates a list with a zero-length vector, and the `rep()` function replicates that an appropriate number of times. The `data.frame()` function turns the list into a data frame (with zero rows).

```
R> emptyMoths <- data.frame(rep(list(numeric(0)),
                                length(mothNames)))
R> colnames(emptyMoths) <- mothNames
R> emptyMoths

[1] sp108 sp109 sp110 sp111 sp117 sp120 sp121 sp125 sp126 ...
<0 rows> (or 0-length row.names) ...
```

Now we get the final data frame by joining this data frame with the data frame containing moth counts; we perform an outer join to retain rows where there is no match between the data frames (in this case, all rows in `mothCounts`). The `merge()` function does the join, automatically detecting the columns to match on by the columns with common names in the two data frames. There are no matching rows between the data frames (`emptyMoths` has no counts), but the outer join retains all rows of `mothCounts` and adds missing values in columns which are in `emptyMoths`, but not in `mothCounts` (`all.x=TRUE`). In this code, we have been careful to retain the original order of the rows (`sort=FALSE`) and the original order of the columns (`[,mothNames]`).

```
R> mergeMoths <- merge(mothCounts, emptyMoths,
                       all.x=TRUE, sort=FALSE)[,mothNames]
R> allmoths

  sp108 sp109 sp110 sp111 sp117 sp120 sp121 sp125 sp126 ...
1    NA    NA    NA    NA     2     1     1     0     0 ...
2    NA    NA    NA    NA     3     1     1     0     0 ...
3    NA    NA    NA    NA     2     0     3     6     0 ...
4    NA    NA    NA    NA     5     0     2     2     0 ...
...
```

11.7.9 Case study: Utilities

A compact fluorescent light bulb.[15] This sort of light bulb lasts up to 16 times longer and consumes about one quarter of the power of a comparable incandescent light bulb.

A resident of Baltimore, Maryland in the United States collected data from his residential gas and electricity power bills over 8 years. The data are in a text file called `baltimore.txt` and include the start date for the bill, the number of therms of gas used and the amount charged, the number of kilowatt hours of electricity used and the amount charged, the average daily outdoor temperature (as reported on the bill), and the number of days in the billing period.

Several events of interest occurred in the household over this time period and the aim of the analysis was to determine whether any of these events had any effect on the energy consumption of the household. The events were:

- An additional resident moved in on July 31st 1999.
- Two storm windows were replaced on April 22nd 2004.
- Four storm windows were replaced on September 1st 2004.
- An additional resident moved in on December 18th 2005.

Figure 11.14 shows the first few lines of the data file. This sort of text file can be read conveniently using the `read.table()` function, with the `header=TRUE` argument specified to use the variable names on the first line of the file. We also use `as.is=TRUE` to keep the dates as strings for now.

[15] Source: Wikimedia Commons
`http://commons.wikimedia.org/wiki/Image:Spiralformige_Energiesparlampe_quadr.png`
This image is in the Public Domain.

```
start   therms  gas KWHs        elect   temp    days
10-Jun-98       9       16.84   613     63.80   75      40
20-Jul-98       6       15.29   721     74.21   76      29
18-Aug-98       7       15.73   597     62.22   76      29
16-Sep-98       42      35.81   460     43.98   70      33
19-Oct-98       105     77.28   314     31.45   57      29
17-Nov-98       106     77.01   342     33.86   48      30
17-Dec-98       200     136.66  298     30.08   40      33
19-Jan-99       144     107.28  278     28.37   37      30
18-Feb-99       179     122.80  253     26.21   39      29
 ...
```

Figure 11.14: The first few lines of the utilities data set.

```
R> utilities <- read.table("baltimore.txt",
                           header=TRUE, as.is=TRUE)
R> head(utilities)

      start therms   gas KWHs elect temp days
1 10-Jun-98      9 16.84  613 63.80   75   40
2 20-Jul-98      6 15.29  721 74.21   76   29
3 18-Aug-98      7 15.73  597 62.22   76   29
4 16-Sep-98     42 35.81  460 43.98   70   33
5 19-Oct-98    105 77.28  314 31.45   57   29
6 17-Nov-98    106 77.01  342 33.86   48   30
```

The first thing we want to do is to convert the first variable into actual dates. This will allow us to perform calculations and comparisons on the date values.

```
R> utilities$start <- as.Date(utilities$start,
                              "%d-%b-%y")
R> head(utilities)

       start therms   gas KWHs elect temp days
1 1998-06-10      9 16.84  613 63.80   75   40
2 1998-07-20      6 15.29  721 74.21   76   29
3 1998-08-18      7 15.73  597 62.22   76   29
4 1998-09-16     42 35.81  460 43.98   70   33
5 1998-10-19    105 77.28  314 31.45   57   29
6 1998-11-17    106 77.01  342 33.86   48   30
```

The five significant events can be used to break the data set into five different time periods; we will be interested in the average daily charges for each of

these periods.

The first problem is that none of these events coincide with the start of a billing period, so we need to split some of the bills into two separate pieces in order to obtain data for each of the time periods we are interested in.

We could do this by hand, but by now it should be clear that there are good reasons to write code to perform the task instead. This will reduce the chance of silly errors and make it a trivial task to repeat the calculations if, for example, we discover that one of the event dates needs to be corrected, or a new event is discovered. The code will also serve as documentation of the steps we have taken in preparing the data set for analysis.

What we need to do is to discover which period each event occurred in and how far through the relevant period the event occurred. We can then split the single billing period into two billing periods.[16]

How do we determine which period an event occurred in? This provides a nice example of working with dates in R. The code below creates a vector of dates for the events.

```
R> events <- as.Date(c("1999-07-31", "2004-04-22",
                       "2004-09-01", "2005-12-18"))
```

One thing we can do with dates is subtract one from another; the result in this case is a number of days between the dates. We can do the subtraction for all events at once using the `outer()` function. This will subtract each event date from each billing period start date and produce a matrix of the differences, where column i of the matrix shows the differences between the event i and the start of each billing period. A subset of the rows of this matrix are shown below. For example, the first column gives the number of days between the first event and the start of each billing period; negative values indicate that the billing period began before the first event.

[16] The result will not be ideal because a proportional split assumes that the event has no effect on energy usage. However, we will pursue this as a first approximation anyway.

```
R> dayDiffs <- outer(utilities$start, events, "-")
R> dayDiffs[10:15,]

Time differences in days
     [,1]  [,2]  [,3]  [,4]
[1,] -106 -1833 -1965 -2438
[2,]  -74 -1801 -1933 -2406
[3,]  -44 -1771 -1903 -2376
[4,]  -11 -1738 -1870 -2343
[5,]   18 -1709 -1841 -2314
[6,]   48 -1679 -1811 -2284
```

For each event, we want the billing period where the difference is closest to zero and non-positive. This is the billing period in which the event occurred. First, we can set all of the positive differences to large negative ones, then the problem simply becomes finding the maximum of the differences. Rather than overwrite the original data, we will work with a new copy. This is generally a safer technique, if there is enough computer memory to allow it.

```
R> nonNegDayDiffs <- dayDiffs
R> nonNegDayDiffs[dayDiffs > 0] <- -9999
R> nonNegDayDiffs[10:15,]

Time differences in days
      [,1]  [,2]  [,3]  [,4]
[1,]  -106 -1833 -1965 -2438
[2,]   -74 -1801 -1933 -2406
[3,]   -44 -1771 -1903 -2376
[4,]   -11 -1738 -1870 -2343
[5,] -9999 -1709 -1841 -2314
[6,] -9999 -1679 -1811 -2284
```

The function `which.max()` returns the index of the maximum value in a vector. We can use the `apply()` function to calculate this index for each event (i.e., for each column of the matrix of differences).

```
R> eventPeriods <- apply(nonNegDayDiffs, 2, which.max)
R> eventPeriods

[1] 13 68 72 79
```

The `eventPeriods` variable now contains the row numbers for the billing periods that we want to split. These billing periods are shown below.

```
R> utilities[eventPeriods, ]

        start therms    gas KWHs elect temp days
13 1999-07-20      6  15.88  723 74.41   77   29
68 2004-04-16     36  45.58  327 29.99   65   31
72 2004-08-18      7  19.13  402 43.00   73   31
79 2005-12-15    234 377.98  514 43.55   39   33
```

This may seem a little too much work for something which on first sight appears much easier to do "by eye", because the data appear to be in date order. The problem is that it is very easy to be seduced into thinking that the task is simple, whereas a code solution like we have developed is harder to fool. In this case, a closer inspection reveals that the data are not as orderly as we first thought. Shown below are the last 10 rows of the data set, which reveal a change in the ordering of the dates during 2005 (look at the `start` variable), so it was just as well that we used a code solution.[17]

```
R> tail(utilities, n=10)

        start therms    gas KWHs  elect temp days
87 2005-04-18     65  80.85  220  23.15   58   29
88 2005-03-16    126 141.82  289  27.93   49   30
89 2005-02-15    226 251.89  257  25.71   36   29
90 2006-02-14    183 257.23  332  30.93   41   30
91 2006-03-16    115 146.31  298  28.56   51   33
92 2006-04-18     36  54.55  291  28.06   59   28
93 2006-05-17     22  37.98  374  40.55   68   34
94 2006-06-19      7  20.68  614  83.19   78   29
95 2006-07-18      6  19.94  746 115.47   80   30
96 2006-08-17     11  26.08  534  84.89   72   33
```

Now that we have identified which billing periods the events occurred in, the next step is to split each billing period that contains an event into two new periods, with energy consumptions and charges divided proportionally between them. The proportion will depend on where the event occurred within the billing period and that information is nicely represented by the differences in the matrix `dayDiffs`, which was created on page 307. The differences we want are in the rows corresponding to the billing periods containing events, as shown below. The difference on row 1 of column 1 is the number of days from the start of billing period 13 to the first event. The difference on row 2 of column 2 is the number of days between the start of billing period 68 and the second event.

[17] If you are like me, it will still take you a little while to see the problem, which just reinforces why it is best to leave this sort of thing to the computer!

```
R> dayDiffs[eventPeriods,]

Time differences in days
     [,1]  [,2]  [,3]  [,4]
[1,]  -11 -1738 -1870 -2343
[2,] 1721    -6  -138  -611
[3,] 1845   118   -14  -487
[4,] 2329   602   470    -3
```

The differences we want lie on the diagonal of this sub-matrix and these can be extracted using the `diag()` function.

```
R> eventDiffs <- diag(dayDiffs[eventPeriods,])
R> eventDiffs

[1] -11  -6 -14  -3
```

We want to convert the differences in days into a proportion of the billing period. The proportions are these differences divided by the number of days in the billing period, which is stored in the `days` variable.

```
R> proportions <- -eventDiffs/utilities$days[eventPeriods]
R> proportions

[1] 0.3793103 0.1935484 0.4516129 0.0909091
```

The modified periods are the original billing periods multiplied by these proportions. We want to mutiply all values in the data set *except* the `start` variable (hence the `-1` below).

```
R> modifiedPeriods <- utilities[eventPeriods, -1]*proportions
R> modifiedPeriods$start <- utilities$start[eventPeriods]
R> modifiedPeriods


      therms       gas      KWHs     elect      temp days      start
13  2.275862  6.023448 274.24138 28.224483 29.206897   11 1999-07-20
68  6.967742  8.821935  63.29032  5.804516 12.580645    6 2004-04-16
72  3.161290  8.639355 181.54839 19.419355 32.967742   14 2004-08-18
79 21.272727 34.361818  46.72727  3.959091  3.545455    3 2005-12-15
```

We also need new periods representing the remainder of the original billing periods. The start dates for these new periods are the dates on which the events occurred.

```
R> newPeriods <- utilities[eventPeriods, -1]*(1 - proportions)
R> newPeriods$start <- events
R> newPeriods
```

```
       therms        gas     KWHs    elect     temp days      start
13   3.724138   9.856552 448.7586 46.18552 47.79310   18 1999-07-31
68  29.032258  36.758065 263.7097 24.18548 52.41935   25 2004-04-22
72   3.838710  10.490645 220.4516 23.58065 40.03226   17 2004-09-01
79 212.727273 343.618182 467.2727 39.59091 35.45455   30 2005-12-18
```

Finally, we combine the unchanged billing periods with the periods that we have split in two. When the data frames being combined have exactly the same set of variables, the `rbind()` function can be used to combine them.[18]

```
R> periods <- rbind(utilities[-eventPeriods, ],
                    modifiedPeriods,
                    newPeriods)
```

We should check that our new data frame has the same basic properties as the original data frame. The code below simply calculates the sum of each variable in the data set (other than the start dates).

```
R> apply(utilities[, -1], 2, sum)

  therms      gas     KWHs    elect     temp     days
 8899.00  9342.40 41141.00  4058.77  5417.00  2929.00

R> apply(periods[, -1], 2, sum)

  therms      gas     KWHs    elect     temp     days
 8899.00  9342.40 41141.00  4058.77  5417.00  2929.00
```

Now that we have the data in a format where the significant events occur at the start of a billing period, the last step is to calculate average daily usage and costs in the time periods between the events. To do this, we need a new variable `phase` that will identify the "time phase" for each billing period (between which events did the billing period occur).

This will demonstrate a use of the `cut()` function.

[18] Section 11.7.8 provides an example of combining data frames for the case where only some variables are in common.

```
R> periods$phase <- cut(periods$start,
                        c(as.Date("1900-01-01"),
                          events,
                          as.Date("2008-01-01")),
                        labels=FALSE)
R> periods$phase

  [1] 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 [28] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 [55] 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 4 4 4 4 4 5 4 4 4 4 4
 [82] 4 4 4 4 5 5 5 5 5 5 5 1 2 3 4 2 3 4 5
```

Now we can sum usage and costs for each phase, then divide by the number of days to obtain averages that can be compared between phases.

```
R> phases <- aggregate(periods[c("therms", "gas", "KWHs",
                                 "elect", "days")],
                       list(phase=periods$phase), sum)
R> phases

  phase     therms       gas      KWHs     elect days
1     1  871.27586  693.4134  5434.241  556.6545  388
2     2 5656.69188 5337.3285 24358.049 2291.1400 1664
3     3   53.19355  103.1574  1648.258  170.8748  132
4     4 1528.11144 2017.5025  5625.179  551.8997  470
5     5  789.72727 1190.9982  4075.273  488.2009  275

R> phaseAvgs <-
       phases[c("therms", "gas", "KWHs", "elect")]/phases$days
```

For inspecting these values, it helps if we round the values to two significant figures.

```
R> signif(phaseAvgs, 2)

  therms  gas KWHs elect
1    2.2 1.80   14   1.4
2    3.4 3.20   15   1.4
3    0.4 0.78   12   1.3
4    3.3 4.30   12   1.2
5    2.9 4.30   15   1.8
```

The last step above works because when a data frame is divided by a vector, each variable in the data frame gets divided by the vector. This is not necessarily obvious from the code; a more explicit way to perform the operation

is to use the `sweep()` function, which forces us to explicitly state that, for each row of the data frame (`MARGIN=1`), we are dividing (`FUN="/"`) by the corresponding value from a vector (`STAT=phase$days`).

```
R> phaseSweep <- sweep(phases[c("therms", "gas",
                                "KWHs", "elect")],
                       MARGIN=1, STAT=phases$days, FUN="/")
```

Looking at the average daily energy values for each phase, the values that stand out are the gas usage and cost during phase 3 (after the first two storm windows were replaced, but before the second set of four storm windows were replaced). The naive interpretation is that the first two storm windows were incredibly effective, but the second four storm windows actually made things worse again!

At first sight this appears strange, but it is easily explained by the fact that phase 3 coincided with summer months, as shown below using the `table()` function.

```
R> table(months(periods$start), periods$phase)[month.name, ]
```

```
          1 2 3 4 5
January   1 5 0 1 1
February  1 5 0 1 1
March     0 5 0 1 1
April     1 5 1 1 1
May       1 3 1 1 1
June      2 4 1 1 1
July      2 5 1 1 1
August    1 5 1 1 1
September 1 5 0 3 0
October   1 5 0 2 0
November  1 4 0 2 0
December  1 5 0 2 1
```

The function `months()` extracts the names of the months from the dates in the `start` variable. Subsetting the resulting table by `month.name` just rearranges the order of the rows of the table; this ensures that the months are reported in calendar order rather than alphabetical order.

We would expect the gas usage (for heating) to be a lot lower during summer. This is confirmed by calculating the daily average usages and costs for each month. Again we must take care to get the answer in calendar order; this time we do that by aggregating over a factor that is based on the extracting the months from the `start` variable, with the order of the levels of the factor

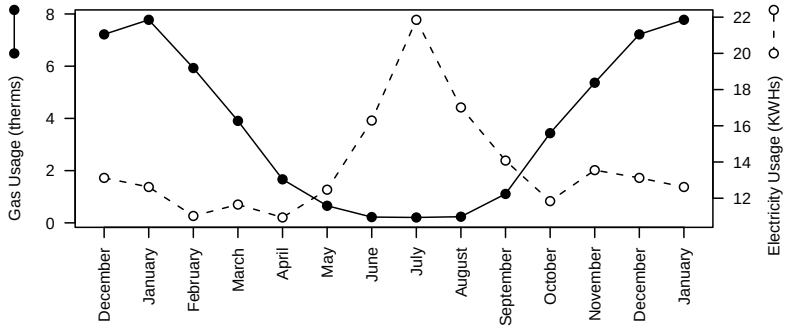


Figure 11.15: The average daily energy usage and costs for each month from the Utilities data set.

explicitly set to be `month.name`. These results are presented graphically in Figure 11.15.

```
R> months <- aggregate(periods[c("therms", "gas", "KWHs",
                                 "elect", "days")],
                       list(month=factor(months(periods$start),
                                         levels=month.name)),
                       sum)
R> cbind(month=months$month,
         signif(months[c("therms", "gas",
                         "KWHs", "elect")]/months$days,
                2))
```

```
       month therms  gas KWHs elect
1    January   7.80 7.50   13   1.2
2   February   5.90 5.90   11   1.0
3      March   3.90 4.00   12   1.1
4      April   1.70 1.90   11   1.0
5        May   0.65 0.99   12   1.3
6       June   0.22 0.57   16   1.8
7       July   0.21 0.59   22   2.3
8     August   0.24 0.60   17   1.9
9  September   1.10 1.30   14   1.3
10   October   3.40 3.50   12   1.1
11  November   5.40 5.50   14   1.2
12  December   7.20 7.10   13   1.2
```

This example has demonstrated a number of data manipulation techniques

in a more realistic setting. The simple averages that we have calculated serve to show that any attempt to determine whether the significant events lead to a significant change in energy consumption and cost for this household is clearly going to require careful analysis. Fortunately, having prepared the data, our work is done, and we will leave the difficult part to those who follow.

11.8 Text processing

Plain text is a very common format for storing information, so it is very useful to be able to manipulate text. It may be necessary to convert a data set from one text format to another. It is also common to search for and extract important keywords or specific patterns of characters from within a large set of text.

This section describes some important functions for working with text in R.

11.8.1 Case study: The longest placename

One of the longest placenames in the world, with a total of 85 characters, is the Maori name for a hill in the Hawkes Bay region, on the east coast of the North Island of New Zealand.

One of the longest place names in the world is attributed to a hill in the Hawke's Bay region of New Zealand. The name (in Maori) is:

Taumatawhakatangihangakoauauotamateaturipukakapikimaungahoronukupokaiwhenuakitanatahu

which means "The hilltop where Tamatea with big knees, conqueror of mountains, eater of land, traveller over land and sea, played his koauau [flute] to his beloved."

Children at an Auckland primary school were set a homework assignment that included counting the number of letters in this name. This task of counting the number of characters in a string is a simple example of what we will call **text processing** and is the sort of task that often comes up

when working with data that has been stored in a text format.

Counting the number of characters in a string is something that any general purpose language will do. Assuming that the name has been saved into a text file called `placename.txt`, here is how to use the `scan()` function to read the name into R, as a character vector of length 1.

```
R> placename <- scan(file.path("Placename", "placename.txt"),
                     "character")
```

We can now use the `nchar()` function to count the number of characters in this text.

```
R> nchar(placename)

[1] 85
```

Counting characters is a very simple text processing task, though even with something that simple, performing the task using a computer is much more likely to get the right answer. We will now look at some more complex text processing tasks.

The homework assignment went on to say that, in Maori, the combinations 'ng' and 'wh' can be treated as a single letter. Given this, how many letters are in the place name? There are two possible approaches: convert every 'ng' and 'wh' to a single letter, or count the number of 'ng's and 'wh's and subtract that from the total number of characters. We will consider both approaches because they illustrate two different text processing tasks.

For the first approach, we could try counting all of the 'ng's and 'wh's as single letters by searching through the string and converting all of the 'ng's and 'wh's into single characters and then redoing the count. In R, we can perform this search-and-replace task this task using the `gsub()` function,[19] which takes three arguments: a pattern to search for, a replacement value, and the string to search within. The result is a string with the pattern replaced. Because we are only counting letters, it does not matter what letter we choose as a replacement. First, we replace occurrences of 'ng' with a full stop.

```
R> replacengs <- gsub("ng", ".", placename)
R> replacengs

[1] "Taumatawhakata.iha.akoauauotamateaturipukakapikimau.ahoronukupokaiwhenuakitanatahu"
```

[19] The `sub()` function is similar, but only replaces the *first* match in the string. There is also a function `chartr()` for converting a single letter to another single letter, and `tolower()` and `toupper()` for converting between cases, see for example, page 319.

Next, we replace the occurrences of 'wh' with a full stop.

```
R> replacewhs <- gsub("wh", ".", replacengs)
R> replacewhs
```

```
[1] "Taumata.akata.iha.akoauauotamateaturipukakapikimau.ahoronukupokai.enuakitanatahu"
```

Finally, we count the number of letters in the resulting string.

```
R> nchar(replacewhs)
```

```
[1] 80
```

The alternative approach involves just finding out how many 'ng's and 'wh's are in the string and subtracting that number from the original count. This simple step of searching within a string for a pattern is yet another common text processing task. There are several R functions that perform variations on this task[20], but for this example we need the function `gregexpr()` because it returns *all* of the matches within a string. This function takes two arguments: a pattern to search for and the string to search within. The return value gives a vector of the starting positions of the pattern within the string plus an attribute that gives the lengths of each match.

```
R> ngmatches <- gregexpr("ng", placename)[[1]]
R> ngmatches
```

```
[1] 15 20 54
attr(,"match.length")
[1] 2 2 2
```

This shows that the pattern 'ng' occurs three times in the place name, starting at character positions 15, 20, and 54, respectively, and that the length of the match is 2 characters in each case. Here is the result of searching for occurrences of 'wh':

```
R> whmatches <- gregexpr("wh", placename)[[1]]
R> whmatches
```

```
[1]  8 70
attr(,"match.length")
[1] 2 2
```

The return value of `gregexpr()` is a list to allow for more than one string

[20] `charmatch()`, `match()`, `pmatch()`.

to be searched at once. In this case, we are only searching a single string, so we just need the first component of the result.

We can use the `length()` function to count how many matches there were in the string.

```
R> length(ngmatches)

[1] 3

R> length(whmatches)

[1] 2
```

The final answer is simple arithmetic.

```
R> nchar(placename) -
     (length(ngmatches) + length(whmatches))

[1] 80
```

For the final question in the homework assignment, the students had to count how many times each letter appeared in the place name (treating 'wh' and 'ng' each as two separate letters each again).

One way to do this in R is by breaking the place name into individual characters and creating a table of counts. Once again, we have a standard text processing task: breaking a single string into multiple pieces. The `strsplit()` function performs this task in R. It takes two arguments: the string to break up and a pattern which is used to decide where to split the string. If we give a zero-length pattern, the string is split at each character.

```
R> nameLetters <- strsplit(placename, NULL)[[1]]
R> nameLetters

 [1] "T" "a" "u" "m" "a" "t" "a" "w" "h" "a" "k" "a" "t" "a"
[15] "n" "g" "i" "h" "a" "n" "g" "a" "k" "o" "a" "u" "a" "u"
[29] "o" "t" "a" "m" "a" "t" "e" "a" "t" "u" "r" "i" "p" "u"
[43] "k" "a" "k" "a" "p" "i" "k" "i" "m" "a" "u" "n" "g" "a"
[57] "h" "o" "r" "o" "n" "u" "k" "u" "p" "o" "k" "a" "i" "w"
[71] "h" "e" "n" "u" "a" "k" "i" "t" "a" "n" "a" "t" "a" "h"
[85] "u"
```

Again, the result is a list to allow for breaking up multiple strings at once. In this case, bceause we only have one string, we are only interested in the first component of the list. One minor complication is that we want the

uppercase 'T' to be counted as a lowercase 't'. The function `tolower()` performs this task.

```
R> lowerNameLetters <- tolower(nameLetters)
R> lowerNameLetters

 [1] "t" "a" "u" "m" "a" "t" "a" "w" "h" "a" "k" "a" "t" "a"
[15] "n" "g" "i" "h" "a" "n" "g" "a" "k" "o" "a" "u" "a" "u"
[29] "o" "t" "a" "m" "a" "t" "e" "a" "t" "u" "r" "i" "p" "u"
[43] "k" "a" "k" "a" "p" "i" "k" "i" "m" "a" "u" "n" "g" "a"
[57] "h" "o" "r" "o" "n" "u" "k" "u" "p" "o" "k" "a" "i" "w"
[71] "h" "e" "n" "u" "a" "k" "i" "t" "a" "n" "a" "t" "a" "h"
[85] "u"
```

Now it is a simple matter of calling the `table` function to produce a table of counts of the letters.

```
R> letterCounts <- table(lowerNameLetters)
R> letterCounts

lowerNameLetters
 a  e  g  h  i  k  m  n  o  p  r  t  u  w
22  2  3  5  6  8  3  6  5  3  2  8 10  2
```

As well as pulling strings apart into smaller pieces as we have done so far, we also need to be able to put strings together to make larger strings. In R, this can done with the `paste()` function. For example, in the following code we will build a text statement about the most commonly occurring letter in the placename.

First of all, we need to identify which letter occurred most often. The `which.max()` function is useful here.

```
R> which.max(letterCounts)

a
1
```

Now the letter itself is determined with the following code.

```
R> mostCommon <- names(letterCounts)[which.max(letterCounts)]
R> mostCommon

[1] "a"
```

The complete text is constructed by concatenating different strings together

using `paste()`. Notice the implicit coercion of the numeric value to a string and the use of the `sep` argument to specify that no extra characters need to be placed between the strings that we are concatenating.

```
R> paste("The letter ", mostCommon, " occurs ",
         max(letterCounts), " times.",
         sep="")

[1] "The letter a occurs 22 times."
```

This section has introduced a number of functions for counting letters in text, transforming strings, breaking strings apart, and putting them back together again. More examples of the use of these functions are given in the next section and in case studies later on. Section 12.4.11 also provides more information on these functions.

11.8.2 Regular expressions

Two of the tasks we looked at when working with the long Maori place name involved treating both 'ng' and 'wh' as if they were a single letter, either counting the number of occurrences of these character pairs, or replacing them both with full stops. In each case, we performed the task in two steps, one for 'ng' and one for 'wh'. For example, when converting to full stops, we performed the following two steps: convert all occurrences of 'ng' to a full stop; convert all occurrences of 'wh' to a full stop. Conceptually, it would be simpler, and more efficient, to perform the task in a single step: convert all occurrences of 'ng' *or* 'wh' to a full stop. Regular expressions allow us to do this.

With the place name in the variable called `placename`, converting both 'ng' and 'wh' to full stops in a single step is achieved as follows:

```
R> gsub("ng|wh", ".", placename)

[1] "Taumata.akata.iha.akoauauotamateaturipukakapikimau.ahoronukupokai.enuakitanatahu"
```

A similar approach allows us to count the number of occurrences of either 'ng' or 'wh' in the place name in a single step.

```
R> gregexpr("ng|wh", placename)[[1]]

[1]  8 15 20 54 70
attr(,"match.length")
[1] 2 2 2 2 2
```

The regular expression we are using, `ng|wh`, describes a **pattern**: the character '`n`' followed by the character '`g`' *or* the character '`w`' followed by the character '`h`'. The vertical bar, `|`, is a **metacharacter**. It does not have its normal meaning, but instead denotes an optional pattern; a match will occur if the string contains either the pattern to the left of the vertical bar or the pattern to the right of the vertical bar. The characters '`n`', '`g`', '`w`', and '`h`' are all **literals**; they have their normal meaning.

Regular expressions provide a very powerful way to describe general patterns in text. The next case study looks at some more complex uses and provides some more examples. Chapter 13 describes several other important metacharacters that can be used to build more complex regular expressions.

11.8.3 Case study: Rusty wheat

Cereal crops account for almost half of global food production. Maize, rice, and wheat make up almost 90% of that production, with barley (pictured) fourth on the list.[21]

As part of a series of field trials conducted by the Institut du Végétal in France,[22] data were gathered on the effect of the disease Septoria tritici on wheat. The amount of disease on individual plants was recorded using data collection forms that were filled in by hand by researchers in the field.

In 2007, due to unusual climatic conditions, two other diseases, Puccinia recondita ("brown rust") and Puccinia striiformis ("yellow rust") were also observed to be quite prevalent. The data collection forms had no specific field for recording the amount of rust on each wheat plant, so data were recorded ad hoc in a general area for "diverse observations". These data were transcribed verbatim into a plain text file (see Figure 11.16).

Unsurprisingly, the rust data that was recorded was relatively untidy. It

[21] Image source: Wikimedia Commons
`http://commons.wikimedia.org/wiki/Image:Usda_rachis_barley.jpg`
This image is in the Public Domain.

[22] Thanks to David Gouache, Arvalis - Institut du Végétal

```
lema, rb 2%
rb 2%
rb 3%
rb 4%
rb 3%
rb 2%,mineuse
rb
rb
rb 12
rb
rj 30%
rb
rb
rb 25%
rb
rb
rb
rj 10, rb 4
```

Figure 11.16: Data recording the occurrence of brown or yellow rust diseases on wheat plants. Each line represents one wheat plant.

included other comments unrelated to rust and there were variations in how the rust data was expressed by different researchers. Fortunately, for the purposes of recovering these results, some basic features of the data were consistent.

Each line of data represents one wheat plant. If brown rust was present, the line contains the letters `rb`, followed by a space, followed by a number indicating the percentage of the plant affected by the rust (possibly with a percentage sign). If the plant was afflicted by yellow rust, the same pattern applies except that the letters `rj` are used.[23] It is possible for both diseases to be present on the same plant (see the last line of data in Figure 11.16).

The task of recovering the rust data from these recordings will provide us with several more examples of the use of regular expressions.

The first step is to get the data into R, which is just a call to `readLines()`.

[23]The abbreviations `rb` and `rj` were used because the French common names for the diseases are *rouille brune* and *rouille jaune*.

```
R> wheat <- readLines(file.path("Wheat", "wheat.txt"))
R> wheat

 [1] "lema, rb 2%"   "rb 2%"         "rb 3%"
 [4] "rb 4%"         "rb 3%"         "rb 2%,mineuse"
 [7] "rb"            "rb"            "rb 12"
[10] "rb"            "rj 30%"        "rb"
[13] "rb"            "rb 25%"        "rb"
[16] "rb"            "rb"            "rj 10, rb 4"
```

What we want to end up with are two variables, one recording the amount of brown rust on each plant and one recording the amount of yellow rust.

Starting with brown rust, the first thing we could do is find out which plants have any brown rust on them. The following code does this using the `grep()` function. The result is a vector of indices that tells us which lines contain the pattern we are searching for.

```
R> rbLines <- grep("rb [0-9]+", wheat)
R> rbLines

[1]  1  2  3  4  5  6  9 14 18
```

The regular expression in this call demonstrates two more important metacharacters. The brackets, `[` and `]`, are used to describe a **character set** that will be matched. Within the brackets we can specify individual characters or, as in this case, ranges of characters; `0-9` means any character between 0 and 9.

The `+` is also a metacharacter, known as a **modifier**. It says that whatever immediately precedes the `+` in the regular expression can repeat several times. In this case, `[0-9]+` will match one or more digits.

The letters `r`, `b`, and the space are all literal, so the entire regular expression will match the letters `rb`, followed by a space, followed by one or more digits. In other words, this will only match rows on which brown rust has been observed on the wheat plant.

Having found which lines contain information about brown rust, we want to extract the information from those lines. The indices from the call to `grep()` can be used to subset out just the relevant lines of data.

```
R> wheat[rbLines]

[1] "lema, rb 2%"   "rb 2%"         "rb 3%"
[4] "rb 4%"         "rb 3%"         "rb 2%,mineuse"
[7] "rb 12"         "rb 25%"        "rj 10, rb 4"
```

We will extract just the brown rust information from these lines in two steps, partly so that we can explore more about regular expressions, and partly because we have to in order to cater for plants that have been afflicted by both brown and yellow rust.

The first step is to reduce the line down to just the information about brown rust. In other words, we want to discard everything except the pattern we are looking for. The following code performs this step.

```
R> rbOnly <- gsub("^.*(rb [0-9]+).*$", "\\1",
                  wheat[rbLines])
R> rbOnly

[1] "rb 2"  "rb 2"  "rb 3"  "rb 4"  "rb 3"  "rb 2"  "rb 12"
[8] "rb 25" "rb 4"
```

Again, we have some new metacharacters to explain. First up is the "hat" character, `^`, which matches the start of the line (or the start of the string). Next is the full stop, `.`. This will match any single character, no matter what it is. The `*` character is similar to the `+`; it modifies the immediately-preceding part of the expression and allows for *zero* or more occurrences. An expression like `^.*` allows for any number of characters at the start of the string (including zero characters, or an empty string).

The parentheses, `(` and `)`, are used to create **sub-patterns** within a regular expression. In this case, we are isolating the pattern `rb [0-9]+`, which matches the brown rust information that we are looking for. Parentheses are useful if we want a modifier, like `+` or `*`, to effect a whole sub-pattern rather than a single character *and* they can be useful when specifying the replacement text in a search-and-replace operation, as we will see below.

After the parenthesized sub-pattern, we have another `.*` expression to allow for any number of additional characters then, finally, a dollar sign, `$`. The latter is the counterpart to `^`; it matches the end of a line (or the end of a string).

So the complete regular expression explicitly matches an entire string that contains information on brown rust. Why do we want to do this? Because we are going to replace the entire string with only the piece that we want to

keep. That is the purpose of the funny-looking **replacement text** `"\\1"`.

The text used to replace a matched pattern in `gsub()` is mostly just literal text. The one exception is that we can refer to sub-patterns within the regular expression that was used to find a match. By specifying `"\\1"`, we are saying reuse whatever matched the sub-pattern within the first set of parentheses in the regular expression. If there were a second set of parentheses, we could refer to that sub-pattern as `"\\2"`.

The overall meaning of the `gsub()` call is therefore to replace the entire string with just the part of the string that contains the information about brown rust.

The final step we have to perform is to extract just the numeric data from the brown rust information. We will do this in three ways in order to demonstrate several different techniques.

One approach is to take the strings that contain just the brown rust information and throw away everything except the numbers. The following code does this using a regular expression.

```
R> gsub("[^0-9]", "", rbOnly)

[1] "2"  "2"  "3"  "4"  "3"  "2"  "12" "25" "4"
```

The point about this regular expression is that it uses `^` as the first character within the square brackets. This has the effect of *negating* the set of characters within the brackets, so `[^0-9]` means any character that is *not* a digit. The effect of the complete `gsub()` call is to replace anything that is not a digit with the empty string, so only the digits remain.

An alternative approach is to recognize that the strings we are dealing with have a very regular structure. In fact, all we need to do is drop the first three characters from each string. The following code does this with a simple call to `substring()`. The second argument to the function says which character to start with; in this case, the first character we want is character 4. There is an optional third argument that specifies which character to stop at, but if, as in this example, the third argument is not specified, then we keep going to the end of the string.

```
R> substring(rbOnly, 4)

[1] "2"  "2"  "3"  "4"  "3"  "2"  "12" "25" "4"
```

The final approach that we will consider works with the entire original string, `wheat[rbLines]`, and uses a regular expression containing an extra

set of parentheses to isolate just the numeric content of the brown rust information as a sub-pattern of its own. The replacement text just refers to this second sub-pattern to perform the extraction in a single step.

```
R> gsub("^.*(rb ([0-9]+)).*$", "\\2", wheat[rbLines])

[1] "2"  "2"  "3"  "4"  "3"  "2"  "12" "25" "4"
```

We are not quite finished because we want to produce a variable that contains the brown rust information for all plants. We will just use `NA` for plants that were not afflicted.

A simple way to do this is to create a vector of `NA`s and then fill in the rows for which we have brown rust information. The other important detail in the following code is the conversion of the textual information into numeric values using `as.numeric()`.

```
R> rb <- rep(NA, length(wheat))
R> rb[rbLines] <- as.numeric(gsub("^.*(rb ([0-9]+)).*$",
                                  "\\2", wheat[rbLines]))
R> rb

 [1]  2  2  3  4  3  2 NA NA 12 NA NA NA NA 25 NA NA NA  4
```

To complete the exercise, we need to repeat the process for yellow rust. Rather than repeat the approach used for brown rust, we will investigate a different solution, which will again allow us to demonstrate more text processing techniques.

This time, we will use `regexpr()` rather than `grep()` to find the lines that we want, which are now the lines containing *yellow* rust data.

```
R> rjData <- regexpr("rj [0-9]+", wheat)
R> rjData

 [1] -1 -1 -1 -1 -1 -1 -1 -1 -1 -1  1 -1 -1 -1 -1 -1 -1  1
attr(,"match.length")
 [1] -1 -1 -1 -1 -1 -1 -1 -1 -1 -1  5 -1 -1 -1 -1 -1 -1  5
```

The result is a numeric vector with a positive number for lines that contain yellow rust data and `-1` otherwise. The number indicates the character where the data start. There are only two lines containing data and, in both cases, the data starts at the first character.

The result also has an attribute called `match.length` which contains the number of characters that produced the match with the regular expression

that we were searching for. In both cases, the pattern matched a total of 5 characters, the letters r and j, followed by a space, followed by two digits. This length information is particularly useful because it will allow us to extract the yellow rust data immediately using `substring()`. This time we specify both a start and an end character for the substring.

```
R> rjText <- substring(wheat, rjData,
                       attr(rjData, "match.length"))
R> rjText

 [1] ""      ""      ""      ""      ""      ""      ""
 [8] ""      ""      ""      "rj 30" ""      ""      ""
[15] ""      ""      ""      "rj 10"
```

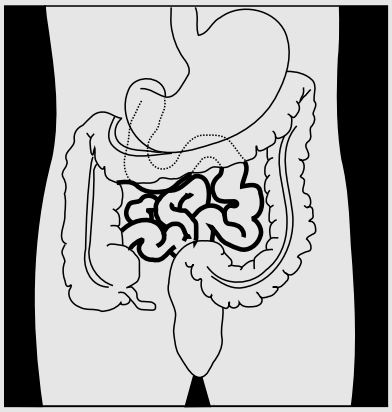
Obtaining the actual numeric data can be carried out using any of the techniques we described above for the brown rust case.

The following code produces the final result, including both brown and yellow rust as a data frame.

```
R> rj <- as.numeric(substring(rjText, 4))
R> data.frame(rb=rb, rj=rj)

   rb rj
1   2 NA
2   2 NA
3   3 NA
4   4 NA
5   3 NA
6   2 NA
7  NA NA
8  NA NA
9  12 NA
10 NA NA
11 NA 30
12 NA NA
13 NA NA
14 25 NA
15 NA NA
16 NA NA
17 NA NA
18  4 10
```

11.8.4 Case study: Crohn's disease



Crohn's disease is an inflammation of the small intestine of the digestive tract. (the thicker lines in the diagram to the left).[24]

Genetic data consists of (usually large amounts of) information on the genotypes of individuals—which alleles do people have at particular loci on their chromosomes. Genetics is a very fast-moving field, with many new methods being developed for collecting genetic data and a number of specialized software systems for performing analyses. One of the problems that genetics researchers face is the difficulty of dealing with many different data formats. The various methods for collecting genetic data produce a variety of raw formats and some of the analysis software requires the data to be in a very specific format for processing.

We will consider an example of this problem, using data from a study of Crohn's disease (an inflammatory bowel disease).[25] The data were originally obtained in a format appropriate for analysis using the software package called LINKAGE,[26] but the analysis was performed using software called PHASE,[27] which requires an entirely difference format for the data.

The original LINKAGE format looks like this:

[24] Image source: Wikimedia Commons
`http://commons.wikimedia.org/wiki/Image:Stomach_colon_rectum_diagram.svg`
This image is in the Public Domain.

[25]

[26] `http://linkage.rockefeller.edu/soft/linkage/`

[27] `http://www.stat.washington.edu/stephens/software.html`

```
PED054  430  0  0  1  0  1  3  3  1  4  1  4  2  2 ...
PED054  412  430  431  2  2  1  3  1  3  4  1  4   ...
PED054  431  0  0  2  0  3  3  3  3  1  1  2  2  1 ...
PED058  438  0  0  1  0  3  3  3  3  1  1  2  2  1 ...
PED058  470  438  444  2  2  3  3  3  3  1  1  2   ...
PED058  444  0  0  2  0  3  3  3  3  1  1  2  2  1 ...
...
```

Each line in the file represents one individual. On each row, the first value is a pedigree label (all individuals who are related to each other are grouped into a single pedigree), the second value is the individual's unique identifier, and the third and fourth values identify the individual's genetic parents (if they exist within the data set). The fifth value on each row indicates gender (`1` is male, `2` is female) and the sixth value indicates whether the individual has Crohn's disease (`1` is no disease, `2` is disease, `0` is unknown). From the first three lines of the data file we can see that individual `412` is the child of individuals `430` (the father) and `431` (the mother), she is female, and she has Crohn's disease. We do not know whether either of her parents have the disease.

The remainder of each line, after the sixth value, consists of pairs of values, where each pair gives the alleles for the individual at a particular locus. For example, individual `412` has alleles `1` and `3` at locus 1, `1` and `3` at locus 2, and `4` and `1` at locus 3.

We want to convert the data to the following format:

```
430
1 3 4 4 2 3 2 3 3 4 4 2 2 3 2 2 3 3 1 3 1 2 3 1 2  ...
3 1 1 2 1 1 4 2 3 2 2 1 1 1 2 2 3 2 1 3 1 2 3 1 2  ...
412
1 1 4 4 2 3 4 3 3 2 2 2 1 1 2 2 3 ? 1 3 1 2 3 1 2  ...
3 3 1 2 1 1 2 2 3 4 4 1 2 3 2 2 3 ? 1 3 1 2 3 1 2  ...
431
3 3 1 2 1 1 2 2 3 4 4 ? 2 3 2 2 3 3 1 3 1 2 3 1 2  ...
3 3 1 2 1 1 2 2 3 4 4 ? 2 3 2 2 3 3 1 3 1 2 3 1 2  ...
...
```

In this format, the information for each individual is stored on three lines. The first line gives the individual's unique identifier, the second line gives the first allele at each locus, and the third line gives the second allele at each locus. Instead of alleles being in pairs of columns, they are in pairs of rows. Furthermore, any zeroes in the original allele information, which indicate missing values, must be encoded as question marks (e.g., individual

412 has missing values at the 18th locus).

There are many ways that we could perform this transformation, but we will use an approach that involves a number of the file handling, data manipulation and text processing tools that we have discussed.

The first step is to read the original file into R. We keep all values as strings so that we can work with the data as one large matrix. The `read.table()` function conveniently splits the data into separate values for us. We also calculate the number of individuals in the data set (there are 387).

```
R> crohn <- as.matrix(read.table(file.path("Crohns",
                                           "Dalydata.txt"),
                                 colClasses="character"))
R> ncase <- nrow(crohn)
R> crohn
```

```
     V1       V2    V3    V4    V5  V6  V7  V8  V9  V10 ...
[1,] "PED054" "430" "0"   "0"   "1" "0" "1" "3" "3" "1" ...
[2,] "PED054" "412" "430" "431" "2" "2" "1" "3" "1" "3" ...
[3,] "PED054" "431" "0"   "0"   "2" "0" "3" "3" "3" "3" ...
[4,] "PED058" "438" "0"   "0"   "1" "0" "3" "3" "3" "3" ...
[5,] "PED058" "470" "438" "444" "2" "2" "3" "3" "3" "3" ...
[6,] "PED058" "444" "0"   "0"   "2" "0" "3" "3" "3" "3" ...
...
```

It is a simple matter to extract the unique identifiers for the individuals from this matrix. These are just the second column of the matrix.

```
R> ids <- crohn[, 2]
R> ids
```

```
[1] "430" "412" "431" "438" "470" "444" "543" "516" "513" ...
```

These identifiers represent the first, fourth, seventh, etc line of the final format. We can generate an empty object with the apropriate number of lines and start to fill in the lines that we know.

```
R> crohnPHASE <- vector("character", 3*ncase)
R> crohnPHASE[seq(by=3, length.out=ncase)] <- ids
R> crohnPHASE
```

```
[1] "430" ""    ""    "412" ""    ""    "431" ""    ""    ...
```

The genotype information (the pairs of alleles) requires considerable rearrangement. To make it easy to see what we are doing, we will just extract that part of the data set and take a note of how many genotypes we have (there are 103).

```
R> genotypes <- crohn[, -(1:6)]
R> ngenotype <- ncol(genotypes)/2
R> genotypes
```

```
     V7  V8  V9  V10 V11 V12 V13 V14 V15 V16 ...
[1,] "1" "3" "3" "1" "4" "1" "4" "2" "2" "1" ...
[2,] "1" "3" "1" "3" "4" "1" "4" "2" "2" "1" ...
[3,] "3" "3" "3" "3" "1" "1" "2" "2" "1" "1" ...
...
```

What we want to do is take the odd alleles for an individual and put them together in a single row. We can extract the odd alleles using simple indexing:

```
R> allele1 <- genotypes[, seq(by=2, length.out=ngenotype)]
R> allele1
```

```
     V7  V9  V11 V13 V15 V17 V19 V21 V23 V25 ...
[1,] "1" "3" "4" "4" "2" "3" "2" "3" "3" "4" ...
[2,] "1" "1" "4" "4" "2" "3" "4" "3" "3" "2" ...
[3,] "3" "3" "1" "2" "1" "1" "2" "2" "3" "4" ...
...
```

Each row of this matrix contains the information we need for one row of the final format. We can combine all of the strings on each row of the matrix into a single string by using `apply()` to call the `paste()` function on each row of the matrix.

```
R> alleleLine1 <- apply(allele1, 1, paste, collapse=" ")
R> alleleLine1
```

```
[1] "1 3 4 4 2 3 2 3 3 4 4 2 2 3 2 2 3 3 1 3 1 2 3 ...
[2] "1 1 4 4 2 3 4 3 3 2 2 2 1 1 2 2 3 0 1 3 1 2 3 ...
[3] "3 3 1 2 1 1 2 2 3 4 4 0 2 3 2 2 3 3 1 3 1 2 3 ...
...
```

These strings now represent the second, fifth, eighth, etc rows of the final format, so we can fill in more of the `crohnPHASE` object. At this point, we also do the conversion of `0` values to `?` symbols.

```
R> crohnPHASE[seq(2, by=3, length.out=ncase)] <-
       gsub("0", "?", alleleLine1)
R> crohnPHASE
```

```
[1] "430"
[2] "1 3 4 4 2 3 2 3 3 4 4 2 2 3 2 2 3 3 1 3 1 2 3 ...
[3] ""
[4] "412"
[5] "1 1 4 4 2 3 4 3 3 2 2 2 1 1 2 2 3 ? 1 3 1 2 3 ...
[6] ""
...
```

The same series of steps can be carried out for the even allele values to generate the third, sixth, ninth, etc lines of the final format.

```
R> allele2 <- genotypes[, seq(2, by=2, length.out=ngenotype)]
R> alleleLine2 <- apply(allele2, 1, paste, collapse=" ")
R> crohnPHASE[seq(3, by=3, length.out=ncase)] <-
       gsub("0", "?", alleleLine2)
R> crohnPHASE
```

```
[1] "430"
[2] "1 3 4 4 2 3 2 3 3 4 4 2 2 3 2 2 3 3 1 3 1 2 3 ...
[3] "3 1 1 2 1 1 4 2 3 2 2 1 1 1 2 2 3 2 1 3 1 2 3 ...
[4] "412"
[5] "1 1 4 4 2 3 4 3 3 2 2 2 1 1 2 2 3 ? 1 3 1 2 3 ...
[6] "3 3 1 2 1 1 2 2 3 4 4 1 2 3 2 2 3 ? 1 3 1 2 3 ...
...
```

The final step is to write the new format to a file. Because we have the data in a character vector, with each string representing one line of the new file format, this is just a matter of calling the `writeLines()` function.

```
R> writeLines(crohnPHASE, "DalydataPHASE.txt")
```

```
             VARIABLE : Mean Near-surface air temperature (kelvin)
             FILENAME : ISCCPMonthly_avg.nc
             FILEPATH : /usr/local/fer_data/data/
             SUBSET   : 24 by 24 points (LONGITUDE-LATITUDE)
             TIME     : 16-JAN-1995 00:00
              113.8W 111.2W 108.8W 106.2W 103.8W 101.2W 98.8W  ...
               27     28     29     30     31     32     33    ...
 36.2N / 51:  272.1  270.3  270.3  270.9  271.5  275.6  278.4  ...
 33.8N / 50:  282.2  282.2  272.7  272.7  271.5  280.0  281.6  ...
 31.2N / 49:  285.2  285.2  276.1  275.0  278.9  281.6  283.7  ...
 28.8N / 48:  290.7  286.8  286.8  276.7  277.3  283.2  287.3  ...
 26.2N / 47:  292.7  293.6  284.2  284.2  279.5  281.1  289.3  ...
 23.8N / 46:  293.6  295.0  295.5  282.7  282.7  281.6  285.2  ...
 ...
```

Figure 11.17: The first few lines of output from the Live Access Server for the near-surface air temperature of the Earth for January 1995, over a coarse 24 by 24 grid of locations covering central America.

11.8.5 Flashback: Regular expressions in HTML Forms

11.8.6 Flashback: Regular expressions in SQL

11.9 Writing Functions

It is quite straightforward to create new functions in R.

In this section, we will look at why it is useful and sometimes necessary to write our own functions.

11.9.1 Case Study: The Data Expo (continued)

The data for the 2006 JSM Data Expo (Section 7.3.6) were obtained from NASA's Live Access Server as a set of 505 text files (see Section 1.1).

Seventy-two of those files contain near-surface air temperature measurements, with one file for each month of recordings. Each file contains average temperatures for the relevant month at 576 different locations. Figure 11.17 shows the first few lines of the temperature file for the first month, January 1995.

The complete set of 72 file names for the files containing temperature record-

ings can be obtained by the following code (only the first fifteen file names are shown):

```
R> nasaAirTempFiles <- list.files(file.path("NASA", "Files"),
                                  pattern="^temperature",
                                  full.names=TRUE)
R> head(nasaAirTempFiles, n=15)

 [1] "NASA/Files/temperature10.txt"
 [2] "NASA/Files/temperature11.txt"
 [3] "NASA/Files/temperature12.txt"
 [4] "NASA/Files/temperature13.txt"
 [5] "NASA/Files/temperature14.txt"
 [6] "NASA/Files/temperature15.txt"
 [7] "NASA/Files/temperature16.txt"
 [8] "NASA/Files/temperature17.txt"
 [9] "NASA/Files/temperature18.txt"
[10] "NASA/Files/temperature19.txt"
[11] "NASA/Files/temperature1.txt"
[12] "NASA/Files/temperature20.txt"
[13] "NASA/Files/temperature21.txt"
[14] "NASA/Files/temperature22.txt"
[15] "NASA/Files/temperature23.txt"
```

This code assumes that the files are stored in a local directory `NASA/Files`. Notice also that we are using a regular expression to specify the pattern of the filenames that we want; we have selected all file names that start with `temperature`.[28]

The file names are in an awkward order because of the default alphabetical ordering of the names;[29] the data for the first month are in the file `temperature1.txt`, but this is the eleventh file name. We can ignore this problem for now, but we will come back to it later.

We will conduct a simple task with these data: calculating the near-surface air temperature for each month, averaged over all locations. In other words, we will calculate a single average temperature from each file. The result will be a vector of 72 monthly averages.

[28] For Linux users who are used to using file name globs with the `ls` shell command, this use of regular expressions for file name patterns can cause confusion. Such users may find the `glob2rx()` function helpful.

[29] The default ordering is dependent on the operating system and the locale, so the result may differ if this code is run on a non-Linux machine and/or in a non-english locale.

The following code reads in the data using the first file name, `temperature10.txt`, and averages all of the temperatures in the file.

```
R> mean(as.matrix(read.fwf(nasaAirTempFiles[1],
                           skip=7,
                           widths=c(-12, rep(7, 24)))))

[1] 298.0564
```

The call to `read.fwf()` ignores the first 7 lines of the file and the first 12 characters on the remaining lines of the file. This just leaves the temperature values, which are converted into a matrix so that the `mean()` function can calculate the average across all of the values.

We want to perform this calculation for each of the air temperature files. Conceptually, we want to perform a loop, once for each file name. Indeed, we can write code to perform the task with an explicit loop.

```
R> avgTemp <- numeric(72)
R> for (i in 1:72) {
       avgTemp[i] <-
           mean(as.matrix(read.fwf(nasaAirTempFiles[i],
                                   skip=7,
                                   widths=c(-12,
                                     rep(7, 24)))))
   }
R> avgTemp

 [1] 298.0564 296.7019 295.9568 295.3915 296.1486 296.1087
 [7] 297.1007 298.3694 298.1970 298.4031 295.1849 298.0682
[13] 298.3148 297.3823 296.1304 295.5917 295.5562 295.6438
[19] 296.8922 297.0823 298.4793 295.3175 299.3575 299.7984
[25] 299.7314 299.6090 298.4970 297.9872 296.8453 296.9569
[31] 296.9354 297.0240 296.3335 298.0668 299.1821 300.7290
[37] 300.6998 300.3715 300.1036 299.2269 297.8642 297.2729
[43] 296.8823 296.9587 297.4288 297.5762 298.2859 299.1076
[49] 299.1938 299.0599 299.5424 298.9135 298.2849 297.0981
[55] 297.7286 296.2639 296.1943 296.5868 297.5510 298.6106
[61] 299.7425 299.5219 299.7422 300.3411 299.5781 298.5809
[67] 298.6965 297.0830 296.3813 299.1863 299.0660 298.4634
```

That is the vector of 72 monthly averages that we require.

However, as we saw in Section 11.7.3, it is more natural in R to use a function like `sapply()` to perform this sort of task, rather than using an

explicit loop.

What we want to do is use `sapply()` to call a function once for each of the file names in the vector `nasaAirTempFiles`.

There is a small problem because `nasaAirTempFiles` is a vector amd `sapply()` works with a list. However, it is not difficult to coerce the vector of file names into a list; in fact, `sapply()` will do that coercion automatically for us.

A larger problem in this case is that there is no existing function to perform the task that we need to perform for each file name. That is, we have no ready-made function that will read in the file, discard all but the temperature values, generate a matrix of the values, and calculate an overall mean.

Fortunately, it is very easy to define such a function ourselves. Here is code that defines a function called `monthAvg()` to perform this task.

```
R> monthAvg <- function(filename) {
       mean(as.matrix(read.fwf(filename,
                               skip=7,
                               widths=c(-12, rep(7, 24)))))
   }
```

The object `monthAvg` is a **function** object that can be used like any other R function. This function has a single argument called `filename`. When the `monthAvg()` function is called, the `filename` argument *must* be specified and, within the function, the symbol `filename` contains the value specified as the first argument in the function call.

The value returned by a function is the value of the last expression within the function. In this case, the return value is the result of the call to the `mean()` function.

As a simple example, the following two pieces of code produce exactly the same result; they both calculate the overall average temperature from the contents of the file `temperature10.txt`.

```
R> mean(as.matrix(read.fwf(nasaAirTempFiles[1],
                           skip=7,
                           widths=c(-12, rep(7, 24)))))

R> monthAvg(nasaAirTempFiles[1])

[1] 298.0564
```

With this function defined, it is now possible to calculate the monthly av-

erages from all of the temperature files using `sapply()`. We supply the vector of file names and `sapply()` automatically converts this to a list of file names. We also supply our new `monthAvg()` function and `sapply()` calls that function for each file name. We specify `USE.NAMES=FALSE` in this call because otherwise each element of the result would have the corresponding filename as a label, which looks messy.

```
R> sapply(nasaAirTempFiles, monthAvg, USE.NAMES=FALSE)

 [1] 298.0564 296.7019 295.9568 295.3915 296.1486 296.1087
 [7] 297.1007 298.3694 298.1970 298.4031 295.1849 298.0682
[13] 298.3148 297.3823 296.1304 295.5917 295.5562 295.6438
[19] 296.8922 297.0823 298.4793 295.3175 299.3575 299.7984
[25] 299.7314 299.6090 298.4970 297.9872 296.8453 296.9569
[31] 296.9354 297.0240 296.3335 298.0668 299.1821 300.7290
[37] 300.6998 300.3715 300.1036 299.2269 297.8642 297.2729
[43] 296.8823 296.9587 297.4288 297.5762 298.2859 299.1076
[49] 299.1938 299.0599 299.5424 298.9135 298.2849 297.0981
[55] 297.7286 296.2639 296.1943 296.5868 297.5510 298.6106
[61] 299.7425 299.5219 299.7422 300.3411 299.5781 298.5809
[67] 298.6965 297.0830 296.3813 299.1863 299.0660 298.4634
```

This is the same result as we got from an explicit loop (see page 335), but it uses only a single call to `sapply()`.

In order to show another example of writing our own functions, we will try to solve an outstanding issue with this result: the fact that these results are not actually in chronological order.

Recall that the file names are ordered alphabetically, so the answer for the first month is actually the eleventh average in the result above. We will now develop a different function so that we can get the averages in the right order.

The idea of this new function is that, instead of taking a file name as an argument, it takes an integer as its argument and it calculates a file name based on that integer. This will allow us to control the order of the file names by controlling the order of the integers. The new function is called `ithAvg()`.

```
R> ithAvg <- function(i=1) {
       monthAvg(file.path("NASA", "Files",
                          paste("temperature", i,
                                ".txt", sep="")))
   }
```

This function has a single argument called `i`. The value of this argument is combined with the path to the file to produce a complete file name and the file name is then passed to the `monthAvg()` function.

One difference between this function and the previous `monthAvg()` function is that, when this function is called, the first argument, `i`, is **optional**. The function definition provides a default value, `1`, for the argument `i`. This means that if the function is called with no arguments, `i` will have the value `1` and the function will calculate the average temperature for the file `temperature1.txt`.

```
R> ithAvg()

[1] 295.1849
```

With this function defined, we can calculate the monthly average temperatures in a chronological order.

```
R> sapply(1:72, ithAvg, USE.NAMES=FALSE)

 [1] 295.1849 295.3175 296.3335 296.9587 297.7286 298.5809
 [7] 299.1863 299.0660 298.4634 298.0564 296.7019 295.9568
[13] 295.3915 296.1486 296.1087 297.1007 298.3694 298.1970
[19] 298.4031 298.0682 298.3148 297.3823 296.1304 295.5917
[25] 295.5562 295.6438 296.8922 297.0823 298.4793 299.3575
[31] 299.7984 299.7314 299.6090 298.4970 297.9872 296.8453
[37] 296.9569 296.9354 297.0240 298.0668 299.1821 300.7290
[43] 300.6998 300.3715 300.1036 299.2269 297.8642 297.2729
[49] 296.8823 297.4288 297.5762 298.2859 299.1076 299.1938
[55] 299.0599 299.5424 298.9135 298.2849 297.0981 296.2639
[61] 296.1943 296.5868 297.5510 298.6106 299.7425 299.5219
[67] 299.7422 300.3411 299.5781 298.6965 297.0830 296.3813
```

These are the same values as before, just in a different order.

11.9.2 Flashback: Writing functions and the DRY Principle

The previous example demonstrates that it is useful to be able to define our own functions for use with functions like `apply()`, `lapply()`, and `sapply()`. However, there are many other good reasons for being able to write functions. In particular, functions are useful for organising code, simplifying code, and for making it easier to maintain code.

For example, we could also use the `monthAvg()` and `ithAvg()` functions that we defined, to make the explicit for loop solution to our task (see page 335) much simpler and easier to read.

```
R> avgTemp <- numeric(72)
R> for (i in 1:72) {
       avgTemp[i] <- ithAvg(i)
   }
```

This is an example of just making our code tidier, which is just an extension of the ideas of laying out and documenting code for the benefit of human readers. A further advantage that we obtain from writing functions is the ability to safely and efficiently **reuse** our code.

The `ithAvg()` function demonstrates the idea of code reuse. Here is the function definition again:

```
R> ithAvg <- function(i=1) {
       monthAvg(file.path("NASA", "Files",
                          paste("temperature", i,
                                ".txt", sep="")))
   }
```

The important feature of this function is that it calls our other function `monthAvg()`. By way of contrast, consider the following alternative way that we could define `ithAvg()`.

```
R> ithAvgBad <- function(i=1) {
       mean(
         as.matrix(
           read.fwf(file.path("NASA", "Files",
                              paste("temperature", i,
                                    ".txt", sep="")),
                    skip=7,
                    widths=c(-12, rep(7, 24)))))
   }
```

What is wrong with this function? The problem is that it repeats almost all of the code that is already in `monthAvg()` and this leads to a number of familiar issues: there is obvious inefficiency because it is wasteful to type all of that code again; what is worse, the code is harder to maintain because there are two copies of the code—if any changes need to be made, they must now be made in two places; worse still, we are now vulnerable to making mistakes because we can change one copy of the code without changing the other copy and thereby end up with two functions that behave differently even though we think they are the same.

The merits of reusing our `monthAvg()` function in the `ithAvg()` function should now be clear. The existence of the `monthAvg()` function means that we only have one copy of the code used to read data from the Data Expo files and that leads to greater efficiency and better accuracy.

11.10 Debugging

It is very easy to make a blanket statement that we should always use a computer to perform menial and repetitive tasks because the computer will make fewer stupid mistakes. However, this conveniently ignores the fact that the computer has to be told, by a person, to do the right thing.

Writing a script to tell a computer how to do a task is just another opportunity to make a mistake. Also, while we only have to write one script, rather than perform a menial task a thousand times, writing a script is a much more complex task and so the chance of making a mistake is higher. Even worse, the consequences of getting it wrong are greater. If our script contains a mistake, we could massacre our entire data set.

The silver lining is that, if we get the script wrong, as long as we notice the mistake and can fix it, it is trivial to repeat the data processing to fix it up.

The important thing is that we must acknowledge that there is a chance that a script will contain mistakes. We must not assume that our script will work; we should always check the results of a script (**testing**); and we must be capable of determining the source of any errors (**debugging**).

11.11 Other software

There are two major disadvantages to working with data using R: R is an interpreted language (as opposed to compiled languages such as C), which means it can be relatively slow; and R holds all data in memory, so it cannot perform tasks on very large data sets.

11.11.1 Perl

11.11.2 Calling other software from R

The `system()` function can be used to run other programs from R.

```
             VARIABLE : Mean TS from clear sky composite (kelvin)
             FILENAME : ISCCPMonthly_avg.nc
             FILEPATH : /usr/local/fer_dsets/data/
             SUBSET   : 24 by 24 points (LONGITUDE-LATITUDE)
             TIME     : 16-JAN-1995 00:00
              113.8W 111.2W 108.8W 106.2W 103.8W 101.2W 98.8W  ...
               27     28     29     30     31     32     33    ...
 36.2N / 51:  272.7  270.9  270.9  269.7  273.2  275.6  277.3  ...
 33.8N / 50:  279.5  279.5  275.0  275.6  277.3  279.5  281.6  ...
 31.2N / 49:  284.7  284.7  281.6  281.6  280.5  282.2  284.7  ...
 28.8N / 48:  289.3  286.8  286.8  283.7  284.2  286.8  287.8  ...
 26.2N / 47:  292.2  293.2  287.8  287.8  285.8  288.8  291.7  ...
 23.8N / 46:  294.1  295.0  296.5  286.8  286.8  285.2  289.8  ...
 ...
```

Figure 11.18: The first few lines of output from the Live Access Server for the surface temperature of the Earth on January 16th 1995 over a coarse 24 by 24 grid of locations covering central America.

11.11.3 Case Study: The Data Expo (continued)

The data for the 2006 JSM Data Expo (Section 7.3.6) were obtained from NASA's Live Access Server (see Section 1.1).

There were 505 files to download so, rather than use the web interface, the data were downloaded using a command-line interface to the Live Access Server. An example of a command used to download a file is shown below and the resulting file is shown in Figure 11.18.

```
lasget.pl -x -115:-55 -y -22:37 -t 1995-Jan-16 \
          -o surftemp.txt -f txt \
          http://mynasadata.larc.nasa.gov/las-bin/LASserver.pl \
          ISCCPMonthly_avg_nc ts
```

The data were downloaded with one file per month of observations, which made for 504 files in total, so it was most efficient to write a script to perform the downloads within two loops. The basic algorithm is this:

```
1 for each variable
2     for each month
3         download a file
```

The actual download can be performed from within R using the `system()` function. For example, the one-off download shown above (to produce the file shown in Figure 7.6) can be performed from R with the following code.

```
R> system("lasget.pl -x -115:-55 -y -22:37 -t 1995-Jan-16 \
          -o surftemp.txt -f txt \
          http://mynasadata.larc.nasa.gov/las-bin/LASserver.pl \
          ISCCPMonthly_avg_nc ts")
```

More generally, we could write a function to perform the download for a given `variable` and `date` and store the output in a file called `filename`.

```
R> lasget <- function(variable, date, filename) {
     command <-
       paste(
         "lasget.pl -x -115:-55 -y -22:37 -t ",
         date,
         " -o ", filename, " -f txt ",
         "http://mynasadata.larc.nasa.gov/las-bin/LASserver.pl ",
         "ISCCPMonthly_avg_nc ", variable,
         sep="")
     system(command)
   }
```

Now it is a simple matter to add a loop over the variables we want to download and a loop over the months that we want to download.

```
R> variables <- list(c("ts", "surftemp"),
                     c("tsa_tovs", "temperature"),
                     c("ps_tovs", "pressure"),
                     c("o3_tovs", "ozone"),
                     c("ca_low", "cloudlow"),
                     c("ca_mid", "cloudmid"),
                     c("ca_high", "cloudhigh"))
R> dates <- seq(as.Date("1995/1/16"), by="month", length.out=72)
R> for (variable in variables) {
       for (date in as.character(dates)) {
           lasget(variable[1], date,
                  file.path("lasfiles", variable[2]))
       }
   }
```

I have chosen to enter the variables and filenames in a list because this makes a strong connection between related variables and filenames and makes maintaining the lists of variable names and file names more con-

venient and accurate. This means that, for example, it is very unlikely that I could accidentally associate the wrong filename with a variable and it is very unlikely that I could accidentally remove one of the variables without also removing the corresponding filename.

It is also worth mentioning that the download is creating files in a separate directory, rather than cluttering up the current directory. This keeps things orderly and makes it easy to clean up if things go haywire. The final file name is generated using `file.path()` to make sure that the code will run on any operating system.

The curious reader may be wondering about the double for loop in the above code. Like all of the other examples, we can do this task without loops, although we have to rearrange the data a little in order to do so.

First of all, we need to convert the `variables` list into a matrix. This will allow us to address the information by column.

```
R> variableMatrix <- matrix(unlist(variables),
                            byrow=TRUE, ncol=2)
R> variableMatrix

     [,1]       [,2]
[1,] "ts"       "surftemp"
[2,] "tsa_tovs" "temperature"
[3,] "ps_tovs"  "pressure"
[4,] "o3_tovs"  "ozone"
[5,] "ca_low"   "cloudlow"
[6,] "ca_mid"   "cloudmid"
[7,] "ca_high"  "cloudhigh"
```

Next, we need to produce all possible combinations of variables and dates.

```
R> datesAndVariables <-
       expand.grid(variable=variableMatrix[, 1],
                   month=dates)
R> head(datesAndVariables, n=10)

   variable      month
1        ts 1995-01-16
2  tsa_tovs 1995-01-16
3   ps_tovs 1995-01-16
4   o3_tovs 1995-01-16
5    ca_low 1995-01-16
6    ca_mid 1995-01-16
7   ca_high 1995-01-16
8        ts 1995-02-16
9  tsa_tovs 1995-02-16
10  ps_tovs 1995-02-16
```

The full variable information needs to be merged back together.

```
R> allCombinations <- merge(datesAndVariables, variableMatrix,
                            by.x="variable", by.y=1)
R> head(allCombinations[order(allCombinations$month), ], n=10)

    variable      month          V2
59   ca_high 1995-01-16   cloudhigh
74    ca_low 1995-01-16    cloudlow
153   ca_mid 1995-01-16    cloudmid
246  o3_tovs 1995-01-16       ozone
293  ps_tovs 1995-01-16    pressure
361       ts 1995-01-16    surftemp
483 tsa_tovs 1995-01-16 temperature
66   ca_high 1995-02-16   cloudhigh
73    ca_low 1995-02-16    cloudlow
160   ca_mid 1995-02-16    cloudmid
```

Now we can use the `mapply()` function to call our `lasget()` function on each of these combinations:

```
R> mapply(lasget,
          allCombinations[, 1],
          allCombinations[, 2],
          file.path("lasfiles", allCombinations[, 3]))
```

Another way to solve the problem makes use of the `outer()` function. To do this, we need to write a function that takes an integer, representing the

index of the variable that we want to download, and a date.

```
R> lasgeti <- function(i, date, variables) {
       lasget(variables[[i]][1], date,
              file.path("lasfiles", variables[[i]][2]))
   }
```

Now we can call this function for all combinations of `i` and `dates` in a call to `outer()`.

```
R> outer(1:7, dates, lasgeti, variables)
```

11.12 Flashback: HTML forms and R

11.13 Literate data analysis

Summary

12
R Reference

The R language and environment for statistical computing and graphics is an open source software project that runs on all major operating systems. It is available as a standard Windows self-installer, as a universal binary disk image for Mac OS X, and as an `rpm` or similar for the major Linux distributions. Because it is open source, it is also possible to build R from the source code on any platform.

R is used to describe both the language *and* the software package that is used to run code written in the language.

The R language is a general-purpose computer language with excellent support for manipulating data sets. The R software provides an interactive command-line interface plus, depending on the platform, various features to support the development of R code.

The next section provides a brief introduction on how to use the R software and subsequent sections provide details of the R language.

12.1 Using R

There are a number of graphical user interfaces for R, including the basic Windows default, the more sophisticated Mac OS X GUI, and several independently-developed GUIs,[1] but the canonical interface is an interactive **command line**.

12.1.1 The command line

The R command line interface consists of a **prompt**, usually the > character.[2] The user can type commands or **expressions**, R echoes what is typed

[1] For example, John Fox's R Commander, Simon Urbanek's JGR, and Phillipe Grosjean's SciViews-R.

[2] The R prompt is shown as R> in this section to distinguish it from the command-line prompt of other software, especially the SQL code examples in Chapter 9.

and, at the end of each expression, R prints out a result. A very simple interaction with R looks like this:

```
R> "hello R"

[1] "hello R"
```

A simple piece of text has been typed and the value of this sort of simple expression is just the text itself.

Section 12.2 explains how to construct more complex R expressions.

12.1.2 Managing R Code

One way to write R code is simply to enter it interactively at the command line as shown above. This interactivity is beneficial for experimenting with a new function or expression, or for exploring a data in a casual manner. For example, in order to find out the result of division-by-zero in R, the user can quickly find out by trying it.

```
R> 1/0

[1] Inf
```

However, interactively typing code at the R command line is a very bad approach from the perspective of recording and documenting code because the code is lost when R is shut down. A superior approach in general is to write R code in a file and get R to read the code from the file. This can be performed in an ad-hoc way by simply cutting and pasting code from a text editor into R. Alternatively, some editors can be associated with an R session and allow submission of code chunks via a single key-stroke (the Windows GUI provides a script editor with this facility). Another option is to read an entire file of R code into R using the `source()` function (see Section 12.4.10).

12.1.3 The working directory

Any files created during an R session are created in the current **working directory** of the session, unless an explicit path or folder or directory is specified. Similarly, when files are read into an R session they are read from the current working directory.

On Linux, the working directory is the directory that the R session was

started in. This means that the standard way to work on Linux is to create a directory for a particular project, put any relevant files in that directory, change into that directory, then start an R session.

On Windows, it is typical to start R by double-clicking a shortcut or by selecting from the list of programs in the 'Start' menu. This approach will, by default, set the working directory to one of the directories where R was installed. This is a bad place to work, so it is a good idea to set the Start in field on the properties dialog of the short cut or menu item. It may also be necessary to use the `setwd()` function or the Change dir option on the File menu to explicitly change the working directory to something appropriate.

12.1.4 Finding the exit

One of the most important things to learn when immersing oneself in a new software environment is how to get out. In R, the expression `q()` quits the session. R will ask whether the user wants to save the "workspace image", which means the results of any code that has been run during the session. As mentioned already, it is better to keep a record of the R code that is used in a session in a separate file, so it is safe to say "no" to this question. Section 12.3.1 contains a broader discussion of this issue.

12.2 R syntax

A string value must be typed within double quotes, for example, `"pointnemotemp.txt"`. A number is anything made up of digits, plus possibly a decimal point, and scientific notation is also accepted (e.g., `6e2` for `600`).

12.2.1 Mathematical operators

R has all of the standard mathematical operators such as addition (`+`), subtraction (`-`), division (`/`), multiplication (`*`), and exponentation (`^`). R also has operators for integer division (`%/%`) and remainder on integer division (`%%`; also known as modulo arithmetic).

12.2.2 Logical operators

The comparison operators `<`, `>`, `<=`, `>=`, and `==` are used to determine whether one value is larger or smaller or equal to another. The result of these

operatores is a logical value, `TRUE` or `FALSE`.

The logical operators `||` (or) and `&&` (and) can be used to combine logical values and produce another logical value as the result. These allow complex conditions to be constructed.

12.2.3 Symbols and assignment

Anything not starting with a digit, that is not a special keyword, is treated as a symbol. Values may be assigned to symbols using the `<-` operator, otherwise any expression involving a symbol will produce the value that has been assigned.

```
R> x <- 1:10

R> x

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

12.2.4 Loops

The general format of the for loop is shown below:

```
for (symbol in sequence) {
    expressions
}
```

The *expressions* are run once for each element in the *sequence*, with the relevant element of the *sequence* assigned to the *symbol*.

The while loop has the following general form:

```
while (condition) {
    expressions
}
```

The `while` loop repeats until the `condition` is `FALSE`. The `condition` is an expression that should produce a single logical value.

12.2.5 Conditional statements

A conditional statement in R has the following form:

```
if (condition) {
    expressions
}
```

The `condition` is an expression that should produce a single logical value.

The curly braces are not necessary, but it is good practice to always include them; if the braces are omitted, only the first complete expression following the `condition` is treated as the `trueBody`.

It is also possible to have an `else` clause.

```
if (condition) {
    trueExpressions
} else {
    falseExpressions
}
```

12.3 Data types and data structures

Inidividual values are either strings, numbers, or logical values (R also supports complex values with an imaginary component).

There is a distinction between integers and real values, but integer values tend to be coerced to real values if anything is done to them. If an integer is required it is best to ensure it by explicitly using a function that generates integer values.

Chapter 7 discussed the amount of memory required to store various types of values. For the specific case of R (on a 32-bit operating system), (ASCII) text uses 1 byte per character. An integer uses 4 bytes, as does a logical value, and a real number uses 8 bytes. The function `object.size()` returns the approximate number of bytes used by an R object in memory.

```
R> object.size(1:1000)

[1] 4024
```

```
R> object.size(as.numeric(1:1000))

[1] 8024
```

The simplest data structure in R is a vector. Most operators and many functions accept vector arguments and return a vector result. All elements of a vector must have the same basic type.

Matrices and arrays are multidimensional analogues of the vector. All elements must have the same type.

Data frames are collections of vectors where each vector must have the same length, but different vectors can have different types. This data structure is the standard way to represent a data set in R.

Lists are like vectors that can have different types of object in each component. In the simplest case, each component of a list may be vector of values. Like the data frame, each component can be a vector of a different basic type, but for lists there is no requirement that each component has the same size. More generally, the components of a list can be more complex objects, such as matrices, data frames, or even other lists. Lists can be used to efficiently represent hierarchical data in R.

12.3.1 The workspace

When quitting R, the option is given to save the current workspace. The workspace consists of all symbols that have been assigned a value during the session.

The `ls()` function displays the names of all symbols that have been created in the current session.

It is possible to save the value of only specific symbols using the `save()` command. The `load()` function can be used to load objects from disk that were created using `save()`. For very large objects, the `save()` function has a `compress` argument.

It is possible to have R exit without asking whether to save the workspace by supplying the argument `--no-save` when starting R.[3]

The workspace is saved as a file called `.Rdata`. When R starts up, it checks for such a file in the current working directory and loads it automatically.

[3] On Linux, this means typing something like `R --no-save` from a shell. On Windows, one way to do it is to create a shortcut to the `Rgui.exe` and modify the properties of that shortcut to add the `--no-save` to the shortcut "Target".

Saving the R workspace is not the recommended approach. It is better to save the original data set and R code, rather than saving intermediate calculations, in order to avoid having multiple copies of the data set to manage. In addition, the workspace, or any object stored using `save()` produces a binary file, with all of the associated disadvantages (see Section 7.5). In particular, if a workspace is corrupted for some reason, it may be impossible to recover the lost information.

12.4 Functions

A function call is an expression of the form:

```
functionName(arg1, arg2)
```

A function can have any number of arguments, including zero. Every argument has a name. Arguments can be specified by position or by name (name overrides position). Arguments may have a default value, which they will take if no value is supplied for the argument in the function call.

All of the following function calls give the same result:

```
seq(1, 10)            # positional arguments
seq(from=1, to=10)    # named arguments
seq(to=10, from=1)    # names trump position
seq(1, 10, by=1)      # 'by' argument has default
```

This section provides a list of some of the functions that are useful for working with data in R. The descriptions of these functions is very brief and only some of the arguments to each function are mentioned. For a complete description of the function and its arguments, the relevant function help page should be consulted.

12.4.1 Generating vectors

`c(...)`
: Concatenate or combine values (or vectors of values) to make a vector. All values must be of the same type (or they will be coerced to the same type). This function can be used to concatenate lists.

`seq(from, to, by, length.out)`
: Generate a sequence of values from `from` to (not greater than) `to` in steps of `by` for a total of `length.out` values.

`rep(x, times)`

`rep(x, each)`

`rep(x, length.out)`
: Repeat all values in a vector `times` times, or each value in the vector `each` times, or all values in the vector until the total number of values is `length.out`.

12.4.2 Numeric functions

`sum(..., na.rm=FALSE)`
: Sum the value of all arguments. Arguments should be vectors, but, for example, matrices will be accepted. If `NA` values are included, the result is `NA` (unless `na.rm=TRUE`). This function is generic.

`max(..., na.rm=FALSE)`
`min(..., na.rm=FALSE)`
`range(..., na.rm=FALSE)`
: Calculate the minimum, maximum, or range of all values in all arguments.

`floor(x)`
`ceiling(x)`
`round(x, digits)`
: Round a numeric value to a number of digits or to an integer value. `floor()` returns largest integer not greater than `x` and `ceiling()` returns smallest integer not less than `x`.

12.4.3 Comparisons

`identical(x, y)`
: Tests whether two objects are equivalent down to the binary storage level.

`all.equal(target, current, tolerance)`
: Tests whether two numeric values are effectively equal (i.e., only differ by a tiny amount, as specified by `tolerance`).

12.4.4 Subsetting

Subsetting is generally performed via the `[` operator (e.g., `candyCounts[1:4]`). In general, the result is of the same class as the original object that is being subsetted. The subset may be numerical indices, string names, or a logical vector (the same length as the original object).

When subsetting objects with more than one dimension, e.g., data frames, matrices or arrays, the subset may be several vectors, separated by commas (e.g., `candy[1:4, 4]`).

The `[[` operator selects only one component of an object. This is typically used to extract a component from a list.

`subset(x, subset, select)`
: Extract the rows of the data frame `x` that satisfy the condition in `subset` and the columns that are named in `select`. The advantage of this over the normal subset syntax is that column names are searched for within the data frame (i.e., you can use just `count`; no need for `candy$count`).

12.4.5 Merging

`rbind(...)`
: Create a new data frame by combining two or more data frames that have the same columns. The result is the union of the rows of the original data frames. This function also works for matrices.

`cbind(...)`
: Create a new data frame by combining two or more data frames that have the same number of rows. The result is the union of the columns of the original data frames. This function also works for matrices.

`merge(x, y)`
: Create a new data frame by combining two data frames in a database-join operation. The two data frames will usually have different columns, though they will typically share at least one column, which is used to match the rows. Additional arguments allow the matching column to be specified explicitly.

  The default join is a natural join. Additional arguments allow for the equivalent of inner joins and outer joins.

`ifelse(test, yes, no)`
: Creates a vector consisting of the values in the vector `yes` wherever `test` is `TRUE` and the values in `no` where `test` is `FALSE`.

12.4.6 Summarizing and collapsing

`aggregate(x, by, FUN)`
: Call the function `FUN` for each subset of `x` defined by the grouping factors in the list `by`. It is possible to apply the function to multiple variables (`x` can be a data frame) and it is possible to group by multiple factors (the list `by` can have more than one component). The result is a data frame. The names used in the `by` list are used for the relevant columns in the result. If `x` is a data frame, then the names of the variables in the data frame are used for the relevant columns in the result.

`sweep(x, MARGIN, STATS, FUN)`
: Take an array and add or subtract (more generally, apply the function `FUN`) the `STATS` values from the rows or columns (depending on value of `MARGIN`). For example, remove column means from all columns.

`table(...)`
: Generate table of counts for one or more factors. The result is a `"table"` object, with as many dimensions as there were arguments.

`xtabs(formula, data)`
: Similar to `table()` except factors to cross-tabulate are expressed in a formula. Symbols in the formula will be searched for in the data frame given by the `data` argument.

`ftable(...)`
: Similar to `table()` except that the result is always a two-dimensional `"ftable"` object, no matter how many factors are cross-tabulated. This makes for a more readable display.

12.4.7 The "apply" functions

`apply(X, MARGIN, FUN, ...)`
: Call a function on each row or each column of a data frame or matrix. The function `FUN` is called for each row of the matrix `X` (if `MARGIN` equals `1`; if `MARGIN` is `2`, the function is called for each column of `X`). All other arguments are passed as arguments to `FUN`.

  The data structure that is returned depends on the value returned by `FUN`. In the simplest case, where `FUN` returns a single value, the result is a vector with one value per row (or column) of the original matrix `X`.

`tapply(X, INDEX, FUN, ...)`
: Call a function once each subset of the vector `X`, where the subsets correspond to unique values of the factor `INDEX`. The `INDEX` argument can be a list of factors, in which case the subsets are unique combinations of the levels of the factors.

  The result depends on how many factors are given in `INDEX`. For the simple case, where there is only one factor, and `FUN` returns a single value, the result is a vector.

`lapply(X, FUN, ...)`
: Call the function `FUN` once for each component of the list `X`. The result is a list. Additional arguments are passed on to each call to `FUN`.

`sapply(X, FUN, ...)`
: Similar to `lapply()`, but will simplify the result to a vector if possible (e.g., if all components of `X` are vectors and `FUN` returns a single value).

`mapply(FUN, ..., MoreArgs)`
: A "multivariate" apply. Similar to `lapply()`, but will call the function `FUN` on the first element of each of the supplied arguments, then on the second element of each argument, and so on. `MoreArgs` is a list of arguments to pass to each call to `FUN`.

`rapply(object, f)`
: A "recursive" apply. Calls the function `f` on each component of the list `object`, *but* if a component is itself a list, then `f` is called on each component of that list, and so on.

12.4.8 Reshaping

Functions from the `reshape` package.

```
melt(data, id.var, measure.var)
```

```
cast(data, formula)
```

12.4.9 Sorting

`sort(x)`
: Put a vector in order. For sorting by more than one factor, see `order()`.

`order(...)`
: Calculate an ordering of one or more vectors (all the same length). The result is a numeric vector, which can be used, via subsetting, to reorder another vector.

`with(data, expr)`
: Run the code in `expr` and search within the variables of the data frame specified by `data` for any symbols used in `expr`.

12.4.10 Data import/export

`file.path(...)`
: Given the names of nested directories, combine them together using an appropriate separator to form a path.

`file.choose()`
: Interactively select a file (on Windows, using a dialog box interface).

`readLines(con)`
: Read the text file specified by the file name and/or path in `con`. The file can also be a URL. The result is a string vector with one element for each line in the file.

`read.table(file, header, skip, sep)`
: Read the text file specified by the string value in `file`, treating each line of text as a case in a data set that contains values for each variable in the data set, with values separated by the string value in `sep`. Ignore the first `skip` lines in the file. If `header` is `TRUE`, treat the first line of the file as variable names.

  The result is a data frame.

`read.fwf(file, widths)`
: Read a text file in fixed-width format. The name of the file is specified by `file` and `widths` is a numeric vector specifying the width of each column of values. The result is a data frame.

`read.csv(file)`
: A front end for `read.table()` with default argument settings designed for reading a text file in CSV format. The result is a data frame.

`scan(file, what)`
: Read data from a text file and produce a vector of values. The type of the value provided for the argument `what` determines how the values in the text file are interpreted. If this argument is a list, then the

result is a list of vectors, each of a type corresponding to the relevant component of `what`.

This function is faster than `read.table()` and its kin.

`save(..., file)`
: Save the symbols named in `...` (and their values), in an R-secific format, to the specified `file`.

`load(file)`
: Load R symbols (and their values) from the specified `file` (which has been created by a previous call to `save()`).

`source(file)`
: Read a file containing R code and evaluate the R code.

12.4.11 Text processing

`grep(pattern, x)`
: Search for the regular expression `pattern` in the string vector `x` and return a vector of numbers, where each number is the index to a string in `x` that matches `pattern`. If there are no matches, the result has length zero.

`gsub(pattern, replacement, x)`
: Search for the regular expression `pattern` in the character vector `x` and replace all matches with the string value in `replacement`. The result is a vector containing the modified strings.

`substr(x, start, stop)`
: For each string in `x`, return a substring consisting of the characters at positions `start` through `stop` inclusive. The first character is at position `1`.

`strsplit(x, split)`
: For each string in `x`, break the string into separate strings, using `split` as the delimiter. The result is a *list*, with one component for each string in the original vector `x`.

`paste(..., sep, collapse)`
: Combine strings together, placing the string `sep` in between. The result is a string vector the same length as the *longest* of the arguments, so shorter arguments are recycled. If the `collapse` argument is not `NULL`, the result vector is collapsed to a single string, with the string `collapse` placed in between each element of the result.

```
Sys.sleep                package:base            R Documentation

Suspend Execution for a Time Interval

Description:

     Suspend execution of R expressions for a given number of
     seconds

Usage:

     Sys.sleep(time)

Arguments:

    time: The time interval to suspend execution for, in seconds.

Details:

     Using this function allows R to be given very low priority
     and hence not to interfere with more important foreground
     tasks. A typical use is to allow a process launched from R
     to set itself up and read its input files before R execution
     is resumed.
```

Figure 12.1: The help page for the function `Sys.sleep()` as displayed in a Linux system. This help page is displayed by the expression `help(Sys.sleep)`.

12.4.12 Getting help

The `help()` function is special in that it provides information about other functions. This function displays a **help page**, which is online documentation that describes what a function does. This includes an explanation of all of the arguments to the function and a description of the return value for the function. Figure 12.1 shows the beginning of the help page for the `Sys.sleep()` function, which is obtained by typing `help(Sys.sleep)`.

A special shorthand using the question mark character, `?`, is provided for getting the help page for a function. Instead of typing `help(Sys.sleep)` it is also possible to simply type `?Sys.sleep`.

Many help pages also have a set of examples to demonstrate the proper use of the function and these examples can be run using the `example()` function.

12.4.13 Packages

There are many thousand R functions in existence. They are organised into collections of functions called **packages**. A number of packages are installed with R by default and several packages are loaded automatically in every R session. The `search()` function shows which packages are currently available, as shown below:

```
R> search()

[1] ".GlobalEnv"        "package:stats"     "package:graphics"
[4] "package:grDevices" "package:utils"     "package:datasets"
[7] "package:methods"   "Autoloads"         "package:base"
```

The top line of the help page for a function shows which package the function comes from. For example, `Sys.sleep()` comes from the `base` package (see Figure 12.1).

Other packages may be loaded using the `library()` function. For example, the `foreign` package provides functions for reading in data sets that have been stored in the native format of a different statistical software system. In order to use the `read.spss()` function from this package, the `foreign` package must be loaded as follows:

```
R> library(foreign)
```

The `search()` function confirms that the `foreign` package is now loaded and all of the functions from that package are now available.

```
R> search()

 [1] ".GlobalEnv"        "package:foreign"   "package:stats"
 [4] "package:graphics"  "package:grDevices" "package:utils"
 [7] "package:datasets"  "package:methods"   "Autoloads"
[10] "package:base"
```

There are usually 25 packages distributed with R. Over a thousand other packages are available for download from the web via the Comprehensive R Archive Network (CRAN).[4] These packages must first be *installed* before they can be loaded. A new package can be installed using the `install.packages()` function.

[4]The main package repository is the Comprehensive R Archive Network (CRAN) `http://cran.r-project.org`.

12.4.14 Searching for functions

Given the name of a function, it is not difficult to find out what that function does and how to use the function by reading the function's help page. A more difficult job is to find the name of a function that will perform a particular task.

The `help.search()` function can be used to search for functions relating to a keyword within the current R installation and the `RSiteSearch()` function performs a more powerful and comprehensive web-based search of functions in almost all known R packages, R mailing list archives, and the main R manuals.[5] There is also a Google customised search available[6] that provides a convenient categorisation of the search results.

Another problem that arises is that, while information on a single function is easy to obtain, it can be harder to discover how several related functions work together. One way to get a broader overview of functions in a package is to read a package **vignette** (see the `vignette()` function). There are also overviews of certain areas of research or application provided by CRAN Task Views (see `http://cran.r-project.org`) and there is a growing list of books on R.

12.5 Further reading

[5] This is based on Jonathan Baron's search site `http://finzi.psych.upenn.edu/search.html`

[6] `http://www.rseek.org` which was set up and is maintained by Sasha Goodman.

13
Regular Expressions Reference

A regular expression consists of **literal characters**, which have their normal meaning, and **metacharacters** that have a special meaning. The combination describes a **pattern** that can be used to find matches amongst text values.

13.1 Metacharacters

^
: The "hat" character matches the start of a string or the start of a line of text.

$
: The dollar character matches the end of a string or the end of a line of text.

.
: The full stop character matches any single character of any sort.

(and)
: Parentheses can be used to define a subpattern within a regular expression. This is useful for applying a modifier to more than a single character (see Section 13.1.2). This is also useful for retaining original portions of a string when peforming a search-and-replace operation (see Section 13.2).

  In some implementations of regular expressions, parentheses are literal and must be escaped in order to have their special meaning.

|
: The vertical bar character subdivides a regular expression into alternative subpatterns. A match is made if either the pattern to the left of the vertical bar or the pattern to the right of the vertical bar is found.

Table 13.1: Some of the POSIX regular expression character classes.

`[:alpha:]`	Alphabetic (only letters)
`[:digit:]`	Digits
`[:alnum:]`	Alphanumeric (letters and digits)
`[:space:]`	White space
`[:punct:]`	Punctuation

Pattern alternatives can be made a subpattern within a large regular expression by enclosing the vertical bar and the alternatives within parentheses.

13.1.1 Ranges

`[` and `]`

Square brackets in a regular expression are used to indicate a character range. A character range will match any character in the range.

Within square brackets, common ranges may be specified by start and end characters, with a dash in between (e.g., `0-9`).

If a hat character appears as the first character within square brackets, the range is inverted so that a match occurs if any character other than the range specified within the square brackets is found.

Within square brackets, most metacharacters revert to their literal meaning. For example, `[.]` means a literal full stop.

In POSIX regular expressions, common character ranges can be specified using special character sequences of the form `[:`*keyword*`:]` (see Table 13.1). The advantage of this approach is that the regular expression will work in different languages. For example, `[a-z]` will not capture all characters in languages that include accented characters, but `[[:alpha:]]` will.

13.1.2 Modifiers

Modifiers specify how many times a subpattern can occur at once. The modifier relates to the subpattern that immediately precedes it in the regular expression. By default, this is just the previous character, but if the preceding character is a closing parenthesis then the modifier relates to the entire subpattern within the parentheses.

`?`
: The question mark means that the subpattern can be missing or it can occur exactly once.

`*`
: The asterisk character means that the subpattern can occur *zero* or more times.

`+`
: The plus character means that the subpattern can occur *one* or more times.

13.2 Replacement text

When performing a search-and-replace operation, the text that is used to replace a matched pattern is usually just literal text. However, it is also possible to use a special escape sequence within the replacement text that represents part of the matched pattern.

When parentheses are used in a pattern to delimit sub-patterns, each sub-pattern may be referred to in the replacement text. The special escape sequence `\1` refers to the first sub-pattern (reading from the left); `\2` refers to the second sub-pattern, and so on. These are referred to as **backreferences**.

Within an R expression, the backslash character must be escaped as usual, so the replacement text referring to the first sub-pattern would have to written like this: `"\\1"`.

Some examples of the use of backreferences are given in Section 11.8.3.

13.3 Further reading

14
Glossary

keyword
: A word that has a special meaning within a computer language. The main point is that we cannot use such words ourselves when choosing names within our code. For example, in R code, the word `for` is a keyword, so this cannot be used as a variable name; in SQL, `CREATE` is a keyword, so this cannot be used for the name of a column or the name of a table.

attribute
: A piece of information about an object, e.g., a measurement made on a person. In some uses, the term roughly corresponds to what statisticians call a variable in a data set. The precise meaning depends on the context.

  In HTML, an attribute is additional information about an element. In terms of syntax, the attribute occurs within the start tag of an element. For example, in the element `<img src="picture.jpg">`, the attribute part is `src="picture.jpg"`. This is an attribute called `img`, with the value `"picture.jpg"`.

  Attributes in XML are identical in syntax to HTML attributes, and are often used to store the value of a single variable for a single case from a data set.

  In database terminology, the term **attribute** is used to describe information about an **entity** and typically corresponds to a column within a database table. Again, there is the correspondence to a variable from a data set.

  In R, an attribute is additional information about an object, supplementary to the fundamental information stored in the object. For example, an 3×2 matrix object contains 6 data values and also has an attribute called `dim` that contains the number of rows and columns in the matrix.