18.1: Scripts

Computer programs come in quite a few different forms: the kind of program that we're mostly interested in from the perspective of everyday data analysis using R is known as a ***script***. The idea behind a script is that, instead of typing your commands into the R console one at a time, instead you write them all in a text file. Then, once you've finished writing them and saved the text file, you can get R to execute all the commands in your file by using the `source()` function. In a moment I'll show you exactly how this is done, but first I'd better explain why you should care.

18.1.1 scripts?

Before discussing scripting and programming concepts in any more detail, it's worth stopping to ask why you should bother. After all, if you look at the R commands that I've used everywhere else this book, you'll notice that they're all formatted as if I were typing them at the command line. Outside this chapter you won't actually see any scripts. Do not be fooled by this. The reason that I've done it that way is purely for pedagogical reasons. My goal in this book is to teach statistics and to teach R. To that end, what *I've* needed to do is chop everything up into tiny little slices: each section tends to focus on one kind of statistical concept, and only a smallish number of R functions. As much as possible, I want you to see what each function does in isolation, one command at a time. By forcing myself to write everything as if it were being typed at the command line, it imposes a kind of discipline on me: it *prevents* me from piecing together lots of commands into one big script. From a teaching (and learning) perspective I think that's the right thing to do… but from a *data analysis* perspective, it is not. When you start analysing real world data sets, you will rapidly find yourself needing to write scripts.

To understand why scripts are so very useful, it may be helpful to consider the drawbacks to typing commands directly at the command prompt. The approach that we've been adopting so far, in which you type commands one at a time, and R sits there patiently in between commands, is referred to as the ***interactive*** style. Doing your data analysis this way is rather like having a conversation … a very annoying conversation between you and your data set, in which you and the data aren't directly speaking to each other, and so you have to rely on R to pass messages back and forth. This approach makes a lot of sense when you're just trying out a few ideas: maybe you're trying to figure out what analyses are sensible for your data, or maybe just you're trying to remember how the various R functions work, so you're just typing in a few commands until you get the one you want. In other words, the interactive style is very useful as a tool for exploring your data. However, it has a number of drawbacks:

- *It's hard to save your work effectively*. You can save the workspace, so that later on you can load any variables you created. You can save your plots as images. And you can even save the history or copy the contents of the R console to a file. Taken together, all these things let you create a reasonably decent record of what you did. But it does leave a lot to be desired. It seems like you ought to be able to save a single file that R could use (in conjunction with your raw data files) and reproduce everything (or at least, everything interesting) that you did during your data analysis.
- *It's annoying to have to go back to the beginning when you make a mistake.* Suppose you've just spent the last two hours typing in commands. Over the course of this time you've created lots of new variables and run lots of analyses. Then suddenly you realise that there was a nasty typo in the first command you typed, so all of your later numbers are wrong. Now you have to fix that first command, and then spend another hour or so combing through the R history to try and recreate what you did.
- *You can't leave notes for yourself*. Sure, you can scribble down some notes on a piece of paper, or even save a Word document that summarises what you did. But what you really want to be able to do is write down an English translation of your R commands, preferably right "next to" the commands themselves. That way, you can look back at what you've done and actually remember what you were doing. In the simple exercises we've engaged in so far, it hasn't been all that hard to remember what you were doing or why you were doing it, but only because everything we've done could be done using only a few commands, and you've never been asked to reproduce your analysis six months after you originally did it! When your data analysis starts involving hundreds of variables, and requires quite complicated commands to work, then you really, really need to leave yourself some notes to explain your analysis to, well, yourself.
- *It's nearly impossible to reuse your analyses later, or adapt them to similar problems*. Suppose that, sometime in January, you are handed a difficult data analysis problem. After working on it for ages, you figure out some really clever tricks that can be used to solve it. Then, in September, you get handed a really similar problem. You can sort of remember what you did, but not very well. You'd like to have a clean record of what you did last time, how you did it, and why you did it the way you did. Something like that would really help you solve this new problem.
- *It's hard to do anything except the basics.* There's a nasty side effect of these problems. Typos are inevitable. Even the best data analyst in the world makes a lot of mistakes. So the chance that you'll be able to string together dozens of correct R commands

in a row are very small. So unless you have some way around this problem, you'll never really be able to do anything other than simple analyses.

- *It's difficult to share your work other people.* Because you don't have this nice clean record of what R commands were involved in your analysis, it's not easy to share your work with other people. Sure, you can send them all the data files you've saved, and your history and console logs, and even the little notes you wrote to yourself, but odds are pretty good that no-one else will really understand what's going on (trust me on this: I've been handed lots of random bits of output from people who've been analysing their data, and it makes very little sense unless you've got the original person who did the work sitting right next to you explaining what you're looking at)

Ideally, what you'd like to be able to do is something like this… Suppose you start out with a data set `myrawdata.csv`. What you want is a single document – let's call it `mydataanalysis.R` – that stores all of the commands that you've used in order to do your data analysis. Kind of similar to the R history but much more focused. It would only include the commands that you want to keep for later. Then, later on, instead of typing in all those commands again, you'd just tell R to run all of the commands that are stored in `mydataanalysis.R`. Also, in order to help you make sense of all those commands, what you'd want is the ability to add some notes or *comments* within the file, so that anyone reading the document for themselves would be able to understand what each of the commands actually does. But these comments wouldn't get in the way: when you try to get R to run `mydataanalysis.R` it would be smart enough would recognise that these comments are for the benefit of humans, and so it would ignore them. Later on you could tweak a few of the commands inside the file (maybe in a new file called `mynewdatanalaysis.R`) so that you can adapt an old analysis to be able to handle a new problem. And you could email your friends and colleagues a copy of this file so that they can reproduce your analysis themselves.

In other words, what you want is a *script*.

18.1.2 first script

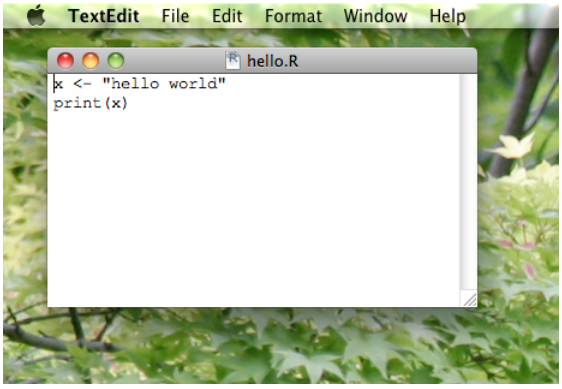

Figure 8.1: A screenshot showing the `hello.R` script if you open in using the default text editor (TextEdit) on a Mac. Using a simple text editor like TextEdit on a Mac or Notepad on Windows isn't actually the best way to write your scripts, but it is the simplest. More to the point, it highlights the fact that a script really is just an ordinary text file.

Okay then. Since scripts are so terribly awesome, let's write one. To do this, open up a simple text editing program, like TextEdit (on a Mac) or Notebook (on Windows). Don't use a fancy word processing program like Microsoft Word or OpenOffice: use the simplest program you can find. Open a new text document, and type some R commands, hitting enter after each command. Let's try using `x <- "hello world"` and `print(x)` as our commands. Then save the document as `hello.R`, and remember to save it as a plain text file: don't save it as a word document or a rich text file. Just a boring old plain text file. Also, when it asks you *where* to save the file, save it to whatever folder you're using as your working directory in R. At this point, you should be looking at something like Figure 8.1. And if so, you have now successfully written your first R program. Because I don't want to take screenshots for every single script, I'm going to present scripts using extracts formatted as follows:

```
## --- hello.R
x <- "hello world"
print(x)
```

The line at the top is the filename, and not part of the script itself. Below that, you can see the two R commands that make up the script itself. Next to each command I've included the line numbers. You don't actually type these into your script, but a lot of text

editors (including the one built into Rstudio that I'll show you in a moment) will show line numbers, since it's a very useful convention that allows you to say things like "line 1 of the script creates a new variable, and line 2 prints it out".

So how do we run the script? Assuming that the `hello.R` file has been saved to your working directory, then you can run the script using the following command:

```
source( "hello.R" )
```

If the script file is saved in a different directory, then you need to specify the path to the file, in exactly the same way that you would have to when loading a data file using `load()`. In any case, when you type this command, R opens up the script file: it then reads each command in the file in the same order that they appear in the file, and executes those commands in that order. The simple script that I've shown above contains two commands. The first one creates a variable `x` and the second one prints it on screen. So, when we run the script, this is what we see on screen:

```
source("./rbook-master/scripts/hello.R")
```

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

If we inspect the workspace using a command like `who()` or `objects()`, we discover that R has created the new variable `x` within the workspace, and not surprisingly `x` is a character string containing the text `"hello world"`. And just like that, you've written your first program R. It really is that simple.

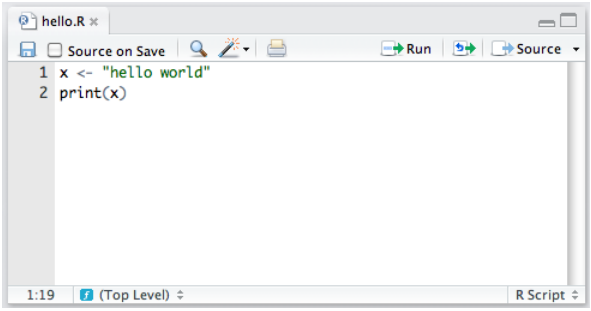

Figure 8.2: A screenshot showing the `hello.R` script open in Rstudio. Assuming that you're looking at this document in colour, you'll notice that the "hello world" text is shown in green. This isn't something that you do yourself: that's Rstudio being helpful. Because the text editor in Rstudio "knows" something about how R commands work, it will highlight different parts of your script in different colours. This is useful, but it's not actually part of the script itself.

18.1.3 Using Rstudio to write scripts

In the example above I assumed that you were writing your scripts using a simple text editor. However, it's usually more convenient to use a text editor that is specifically designed to help you write scripts. There's a lot of these out there, and experienced programmers will all have their own personal favourites. For our purposes, however, we can just use the one built into Rstudio. To create new script file in R studio, go to the "File" menu, select the "New" option, and then click on "R script". This will open a new window within the "source" panel. Then you can type the commands you want (or ***code*** as it is generally called when you're typing the commands into a script file) and save it when you're done. The nice thing about using Rstudio to do this is that it automatically changes the colour of the text to indicate which parts of the code are comments and which are parts are actual R commands (these colours are called ***syntax highlighting***, but they're not actually part of the file – it's just Rstudio trying to be helpful. To see an example of this, let's open up our `hello.R` script in Rstudio. To do this, go to the "File" menu again, and select "Open…". Once you've opened the file, you should be looking at something like Figure 8.2. As you can see (if you're looking at this book in colour) the character string "hello world" is highlighted in green.

Using Rstudio for your text editor is convenient for other reasons too. Notice in the top right hand corner of Figure 8.2 there's a little button that reads "Source"? If you click on that, Rstudio will construct the relevant `source()` command for you, and send it straight to the R console. So you don't even have to type in the `source()` command, which actually I think is a great thing, because it really bugs me having to type all those extra keystrokes every time I want to run my script. Anyway, Rstudio provide several other convenient little tools to help make scripting easier, but I won't discuss them here.[134]

18.1.4 Commenting your script

When writing up your data analysis as a script, one thing that is generally a good idea is to include a lot of comments in the code. That way, if someone else tries to read it (or if you come back to it several days, weeks, months or years later) they can figure out what's going on. As a beginner, I think it's especially useful to comment thoroughly, partly because it gets you into the habit of commenting the code, and partly because the simple act of typing in an explanation of what the code does will help you keep it clear in your own mind what you're trying to achieve. To illustrate this idea, consider the following script:

```
## --- itngscript.R
# A script to analyse nightgarden.Rdata_
#  author: Dan Navarro_
#  date: 22/11/2011_

# Load the data, and tell the user that this is what we're
# doing.
cat( "loading data from nightgarden.Rdata...\n" )
load( "./rbook-master/data/nightgarden.Rdata" )

# Create a cross tabulation and print it out:
cat( "tabulating data...\n" )
itng.table <- table( speaker, utterance )
print( itng.table )
```

You'll notice that I've gone a bit overboard with my commenting: at the top of the script I've explained the purpose of the script, who wrote it, and when it was written. Then, throughout the script file itself I've added a lot of comments explaining what each section of the code actually does. In real life people don't tend to comment this thoroughly, but the basic idea is a very good one: you really do want your script to explain itself. Nevertheless, as you'd expect R completely ignores all of the commented parts. When we run this script, this is what we see on screen:

```
## --- itngscript.R
# A script to analyse nightgarden.Rdata
#  author: Dan Navarro
#  date: 22/11/2011

# Load the data, and tell the user that this is what we're
# doing.
cat( "loading data from nightgarden.Rdata...\n" )
```

```
## loading data from nightgarden.Rdata...
```

```
load( "./rbook-master/data/nightgarden.Rdata" )

# Create a cross tabulation and print it out:
cat( "tabulating data...\n" )
```

```
## tabulating data...
```

```
itng.table <- table( speaker, utterance )
print( itng.table )
```

```
##             utterance
## speaker       ee onk oo pip
##   makka-pakka  0   2  0   2
##   tombliboo    1   0  1   0
##   upsy-daisy   0   2  0   2
```

Even here, notice that the script announces its behaviour. The first two lines of the output tell us a lot about what the script is actually doing behind the scenes (the code do to this corresponds to the two `cat()` commands on lines 8 and 12 of the script). It's usually a pretty good idea to do this, since it helps ensure that the output makes sense when the script is executed.

18.1.5 Differences between scripts and the command line

For the most part, commands that you insert into a script behave in exactly the same way as they would if you typed the same thing in at the command line. The one major exception to this is that if you want a variable to be printed on screen, you need to explicitly tell R to print it. You can't just type the name of the variable. For example, our original `hello.R` script produced visible output. The following script does not:

```
## --- silenthello.R
x <- "hello world"
x
```

It *does* still create the variable `x` when you `source()` the script, but it won't print anything on screen.

However, apart from the fact that scripts don't use "auto-printing" as it's called, there aren't a lot of differences in the underlying mechanics. There are a few stylistic differences though. For instance, if you want to load a package at the command line, you would generally use the `library()` function. If you want do to it from a script, it's conventional to use `require()` instead. The two commands are basically identical, the only difference being that if the package doesn't exist, `require()` produces a warning whereas `library()` gives you an error. Stylistically, what this means is that if the `require()` command fails in your script, R will boldly continue on and try to execute the rest of the script. Often that's what you'd like to see happen, so it's better to use `require()` . Clearly, however, you can get by just fine using the `library()` command for everyday usage.

18.1.6 Done!

At this point, you've learned the basics of scripting. You are now officially allowed to say that you can program in R, though you probably shouldn't say it too loudly. There's a *lot* more to learn, but nevertheless, if you can write scripts like these then what you are doing is in fact basic programming. The rest of this chapter is devoted to introducing some of the key commands that you need in order to make your programs more powerful; and to help you get used to thinking in terms of scripts, for the rest of this chapter I'll write up most of my extracts as scripts.
