1

Getting and Getting Acquainted with R

1.1 Getting started

One of the most challenging bits of getting started with R is actually getting R, installing it, and understanding how it works with your computer. Despite R's cross-platform capacity (OSX, Windows, Linux, Unix), there remain several differences in how things can look on each platform. Thankfully, a new application, RStudio, provides a way to standardize most of what you see and do with R, once it is on your computer. In this chapter, we'll walk you through the steps of getting R and RStudio, installing them on your computer, understanding what you've done, and then working through various aspects of using R and RStudio.

This introduction will make you feel comfortable using R, via RStudio. It will make you understand that R is a giant calculator that does whatever you ask it to do (within reason). It will also familiarize you with how R does things, both 'out of the box' and via additional 'add-on' packages that make R one of the most fun and widely used programs for doing statistics and visualizing data.

We will first walk you through getting and installing R and getting and installing RStudio. While for many this will be trivial, our experience suggests that many of you probably need a tiny bit of hand-holding every once and a while.

1.2 Getting R

We assume you don't yet have R on your computer. It will run on Macintosh, Windows, Linux, and Unix operating systems. R has a homepage, `r-project.org`, but the software itself is located for download on the Comprehensive R Archive Network (CRAN), which you can find at `cran.r-project.org` (Figure 1.1).

Figure 1.1 The CRAN website front page, from where you can find the links to download the R application.

The top box on CRAN provides access to the three major classes of operating systems. Simply click on the link for your operating system. As we mentioned in the Preface, R remains freely available.

You'll hear our next recommendation quite a bit throughout the book: *read the instructions*. The instructions will take you through the processes of downloading R and installing it on your computer. It might also make sense to examine some of the Frequently Asked Questions found at the bottom of the web page. R has been around quite a long time now, and these FAQs reflect more than a decade of beginners like you asking questions about how R works, etc. Go on . . . have a look!

1.2.1 LINUX/UNIX

Moving along now, the Linux link takes you to several folders for flavours of Linux and Unix. Within each of those is a set of instructions. We'll assume that if you know enough to have a Linux or Unix machine under your fine fingertips, you can follow these instructions and take advantage of the various tools.

1.2.2 WINDOWS

The Windows link takes you to a page with three more links. The link you want to focus on is 'base'. You will also notice that there is a link to the aforementioned R FAQs and an additional R for Windows FAQs. Go on . . . have a look! There is a tonne of good stuff in there about the various ways R works on Windows NT, Vista, 8, 10, etc. The base link moves you further on to instructions and the installer, as shown in Figure 1.2.

1.2.3 MACINTOSH

The (Mac) OS X link takes you to a page with several links as well (Figure 1.3). Unless you are on a super-old machine, the first link is the one on which you want to focus. It will download the latest version of R for several recent distributions of OS X and offer, via a `.dmg` installer, to put everything where it needs to be. Note that while not required for 'getting started', getting the XQuartz X11 windowing system is a good idea;

a link is provided just below the paragraph describing the installer (see Figure 1.3). As with Windows, the R FAQs and an additional R for OS X FAQs are provided . . . they are *good things*.

1.3 Getting RStudio

So, at this stage, you should have downloaded and installed R. Well done! However, we are not going to use R directly. Our experience suggests that you will enjoy your R-life a lot more if you interact with R via a different program, also freely available: the software application *RStudio*. RStudio is a lovely, cross-platform application that makes interacting with R quite a bit easier and more pleasurable. Among other things, it makes importing data a breeze, has a standardized look and feel on all platforms, and has several tools that make it much easier to keep track of the instructions you have to give R to make the magic happen.

Figure 1.4 The RStudio website front page, from where you can find the links to download the RStudio application. (Note: you must (as you have done) also download the R application from the CRAN website.)

We highly recommend you use RStudio to get started (we use it in teaching and in our research; Figure 1.4). You can read all about it here: `https://www.rstudio.com`. You can download RStudio here: `https://www.rstudio.com/products/rstudio/download/`.

At this point, you should have downloaded and installed R, and downloaded and installed RStudio.

1.4 Let's play

You are now ready to start interacting with R. RStudio is the application we will use. In the process of installing RStudio, it went around your hard drive to find the R installation. It *knows* where R is. All we need to do now is fire up RStudio.

Start RStudio. Of course, *you* need to know where it is, but we assume you know how to find applications via a 'Start' menu, or in the Applications folder or via an icon on the desktop or in a dock . . . however you do this, navigate to RStudio, and start it up. You are clever. You know how to start an application on your computer!

When we teach, several people end up opening R rather than RStudio. The RStudio icon looks different from the R icon. Make sure you are starting RStudio (Figure 1.5). If all has gone to plan, the RStudio application

Figure 1.5 The R and RStudio icons are different. You want to be using the RStudio application.

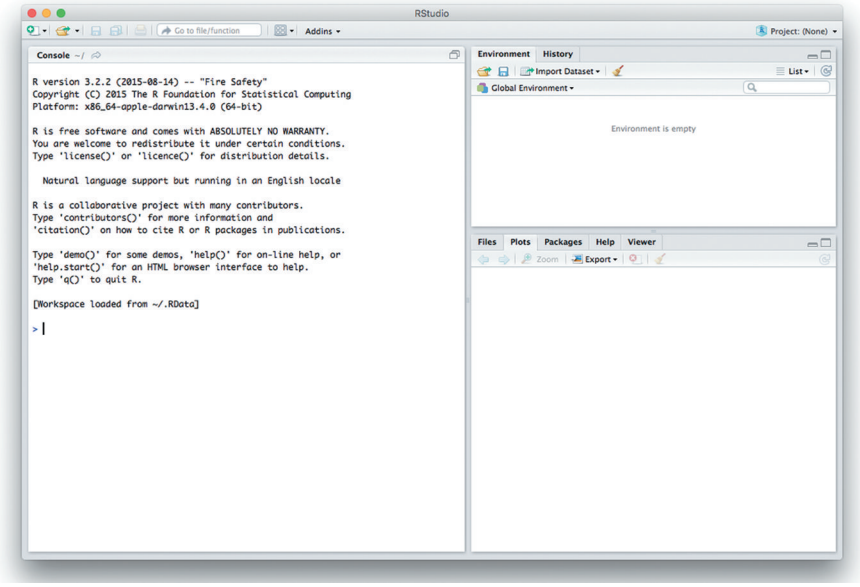

Figure 1.6 The RStudio application initiates the first time you open it with three *panes*. Left is the *Console*; top right is a pane with two tabs, *Environment* and *History*; and bottom right is a pane with five tabs, *Files*, *Plots*, *Packages*, *Help*, and *Viewer*. These are explained in detail in the main text.

will, at its very first start, give you three *panes* (Figure 1.6).[1] Let's walk through them.

On the left is the *Console* pane. This is the window that looks into the engine of R. This is the window where you can give instructions to R, they are worked on by the little people inside R that are really smart and talk in a language of 1s and 0s, and then the answer appears back in the Console pane. It is the brain. The mind. The engine.

The top right is a pane with two tabs: *Environment* and *History*. The Environment pane shows the things that R has in its head. This could be

[1] If you have used RStudio already, it might be showing four panes; don't worry. In what follows, work with the lower left pane, the *console*.

datasets, models, etc. It will probably be empty at the moment, but you will soon start to fill it up. The Environment pane also contains a very important button: *Import Dataset*. We will cover the use of this extensively in Chapter 3. The History pane contains the instructions R has run.

Bottom right is a pane with five tabs: *Files*, *Plots*, *Packages*, *Help*, and *Viewer*. They are rather self-explanatory, and as we begin to use RStudio as our interface for R, feel free to watch what shows up in each.

When you start RStudio, the Console gives some useful information about R, its open source status, etc. But, for new users, the most important is at the bottom, where you will see the symbol > with a cursor flashing after it. This is known as the *prompt*.

You can only type instructions in one place in the Console, and that is at the prompt. Try giving R your first instruction. Click in the Console and type 1 + 1 and then press enter/return. You should see something like this (though there will not be two # characters at the start of the answer line):

```
1 + 1
```

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

The instruction we gave to R was a question: 'Please, can you give us the answer to what is one plus one?' and R has given back the answer

```
[1] 2
```

You can read this as R telling you that the first (and only, in this case) part of the answer is 2. The fact that it's the first part (indicated by the one in square brackets) is redundant here, since your question only has a one-part answer. Some answers have more than one part.

After the answer is a new line with the prompt. R is again ready for an instruction/question.

1.5 Using R as a *giant* calculator (the size of your computer)

What else can R do, other than add one and one? It is a giant calculator, the size of your computer. As befits a statistical programming language, it can divide, multiply, add, and subtract; it conforms to this basic order

too (DMAS). It can also raise to powers, log numbers, do trigonometry, solve systems of differential equations . . . and lots of other maths. Here are some simple examples. Let's go ahead and type each of these into the Console, pressing enter/return after each to see the answer:

```
2 * 4
## [1] 8
3/8
## [1] 0.375
11.75 - 4.813
## [1] 6.937
10^2
## [1] 100
log(10)
## [1] 2.302585
log10(10)
## [1] 1
sin(2 * pi)
## [1] -2.449294e-16
7 < 10
## [1] TRUE
```

* In these blocks of R in this book the ## lines are answers from R. Don't type them in.

Pretty nice. There are a few things worth noting here—some 'default' behaviours hard-wired into R. These are important, because not all statistical software or spreadsheet software like Excel handles things the same way:

- If you understand and use logarithms, you might be puzzled by the result of log(10), which gives 2.30. In R, **log**(x) gives the natural log of x, and not the log to base 10. This is different from other software, which often uses `ln()` to give the natural log. In R, to make a log to base 10, use **log10**(x). See, in the example, **log10**(10) = 2. You can use **log2**() for the log to base 2.
- The trigonometric function **sin**() works in radians (not degrees) in R. So a full circle is $2 \times \pi$ radians (not 360 degrees).
- Some mathematical constants, such as π, are built into R.
- The answer to **sin**(2*pi) should be zero, but R tells us it is very close to zero but not zero. This is computer mumbo-jumbo. The people that

built R understand how computers work, so they made a function called **sinpi**() that does the 'multiply by π' bit for you—**sinpi**(2) does equal zero.

- We sometimes didn't include any spaces in the instructions (e.g. there were no spaces around pi). It would not have mattered if we had, however. R ignores such white space. It ignores all white space (spaces, new lines, and tabs).
- The last question is 'is 7 less than 10?' R gets this right, with the answer 'TRUE'. The 'less than' sign is known as a 'logical operator'. Others include == (are two things equal?), != (are two things not equal?), > (is the thing on the left greater than the thing on the right?), <= (less than or equal to), >= (greater than or equal to), | (the vertical bar symbol, not a big i or a little L; is one or the other thing true?), and & (are two things both true?).

If you were watching carefully too, you will have noticed that RStudio is very nice to you, pre-placing closing brackets/parentheses where necessary. Super-nice.

We've also just introduced you to a new concept: functions like **log10**(), **log**(), and **sin**(). Box 1.1 explains more about what functions are. Dip into it at your pleasure!

1.5.1 FROM THE SIMPLE TO THE SLIGHTLY MORE COMPLEX

All of the maths above gave just one answer. It's the kind of maths you're probably used to. But R can answer several questions at once. For example, we can ask R 'Would you please give us the integers from 1 to 10, inclusive?' We can do this two ways. Let's start with the easy way:

```
1:10
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

The answer to our question has ten elements. But hang on, there is only [1] in front of everything. Fret not, as R is being brief here. It has

limited the square brackets to the left-hand side of the screen and not told us that 2 is the second answer (`[2]`), 3 is the third (`[3]`), and so on. R assumes you're clever! You can even try extending the sequence to 50, and see how R inserts the addresses `[ ]` only on the left-hand side of the Console.

Box 1.1: But what are functions?

Hopefully you're starting to feel that R isn't so difficult after all. But you'll almost certainly have questions. One might be *what are functions*? The functions used so far include **log**(), **log10**(), **sin**(), **rm**(), **ls**(), and **seq**() (make sure you can find these in the text above). Asking R to do things usually requires using functions. R uses functions to do all kinds of things and return information to you. All functions in R are a word, or a combination of words containing no spaces, followed by an opening bracket '(' and a closing bracket ')'. Inside these brackets goes the information that we give to the function. These bits of information are called 'arguments'—yes, sometimes it feels like arguing.

Arguments are separated by commas. Remember when we used the **seq**() function to make a series of numbers:

```
seq(from = 0, to = 10, by = 1)
```

The function is **seq**(), and inside the function brackets are three arguments separated by two commas (necessarily). The first argument is the value to use at the start of the sequence, the second is the value to use at the end of the sequence, and the third is the step size.

To clear R's brain, we use a function inside another function: **ls**() inside **rm**(). The **rm**() stands for remove, and **ls**() stands for list. We combine them in the following way to make R 'clear' its brain:

```
rm(list = ls())
```

It is best to read this from the inside out. The **ls**() requests all of the objects in R's brain. The **rm**() asks R to remove all of these objects in the list. The `list=` is new, and is us telling R exactly what information (i.e. which argument) we are giving to the **rm**() function. You will have noticed that you now know a function that gives a list of all of the objects in R's brain: **ls**(). This is a handy function.

We will introduce and explain many more functions, as they're the workhorses of R. We'll also repeat some of this information about functions (because it's so important), and also explain some things in more detail (e.g. why sometimes we explicitly tell a function the information we are giving it, and why sometimes we can get away without doing so).

The : in 1:10 tells R to make a sequence of whole numbers that goes up in steps of one. We can also generate this sequence using a function that R has in its toolbox. It is called . . . wait for it . . . **seq**(). Wow! *Rocket Science*!

1.5.2 FUNCTIONS TAKE ARGUMENTS

seq() is a function, and in R, functions do clever things for us to make life easier. But we have to give functions things called arguments to control what they do. This isn't complicated. Let's look closely at how we use **seq**(). We have to provide three arguments to **seq**(): the first value of the sequence, the last value of the sequence, and the step size between numbers (the difference in value between numbers in the sequence). For example:

```
seq(from = 1, to = 10, by = 1)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

This reads 'Please give us the sequence of numbers that begins at 1, ends at 10, and has a 1 unit difference between the numbers.' Formally, the arguments are called *from*, *to*, and *by*. We suggest you *do* write the names of these arguments as you use functions. They are not required, but without naming your arguments, you risk getting strange answers if you put things in the wrong place, or, worse, a dreaded *red* error message.

And the answer is what we would expect. Note that we have included some spaces in the instruction; specifically, we have put one space at each comma. R doesn't care; it would be just as happy with no spaces. We used the spaces so the instruction is easier for you to read. More generally, you should attempt to write instructions that will be easier for the two most important readers of your instructions: you and other people. It's easy to focus on writing instructions that R can read. We really should also focus on writing instructions that are easy for humans to read, and for us to read in six months' time when we have to revise our amazing manuscript.

Let's now modify the our use of **seq**() to provide a sequence from 1 to 10 in steps of 0.5:

```
seq(from = 1, to = 10, by = 0.5)
```

```
##  [1]  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0  5.5  6.0
## [12]  6.5  7.0  7.5  8.0  8.5  9.0  9.5 10.0
```

The answers are those we'd expect. Note that we see above that we now get an address (`[12]`) on the second line of answers. This is R being helpful . . . it's giving us a clue about which answer we've got to by the second line, i.e. 6.5 is the 12th answer. Here's what happens if we make the R Console narrower:

```
seq(from = 1, to = 10, by = 0.5)
```

```
##  [1]  1.0  1.5  2.0  2.5  3.0  3.5  4.0
##  [8]  4.5  5.0  5.5  6.0  6.5  7.0  7.5
## [15]  8.0  8.5  9.0  9.5 10.0
```

The question and answer are the same. But we have made the answer go over three lines, and R tells us on each new line the number or address of the answer it's reporting at the beginning of each line.

1.5.3 NOW FOR SOMETHING REALLY IMPORTANT

So far, R has printed the answer to questions in the Console. R hasn't kept the answer in its head or saved it anywhere. As a result, we can't do anything else with the answer. Effectively, it is gone from R's head. It might even be gone from your head!

Often we will want to use the answer to one question in a subsequent question. In this case, it is convenient, if not essential, to have R keep the answer. To have R do this, we *assign* the answer of a question to an *object*. Like this, for example:

```
x <- seq(from = 1, to = 10, by = 0.5)
```

A few things to note:

- We do the assignment by using the *assignment arrow*, which is a less than sign followed (without a space) by a minus sign: `<-`. The arrow points from right to left, so the assignment goes from right to left.

- We assign the answer to something called x. We could have used more or less anything (any continuous text string that starts with a letter, more precisely) that we wanted here. We encourage you to be sensible: we will probably have to type this object name again, so don't make it too long. (But) make it informative if possible.
- After we press enter/return, a new prompt appears in the next line. We don't see the answer to our question, because R has assigned the values produced on the right-hand side to the thing on the left-hand side. That is, if R is able to interpret our instruction, it does what it is told and then just says it's ready for the next instruction. R doesn't congratulate us when we are successful; it only criticizes us when we make a mistake. Get used to this!

To see the answer to the question is easy: just type x in the Console and press return:

```
x
```

```
##  [1]  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0  5.5  6.0
## [12]  6.5  7.0  7.5  8.0  8.5  9.0  9.5 10.0
```

1.5.4 HOW DOES R DO STUFF WITH VECTORS?

Now, try asking R to give you the numbers from 101 to 110 in steps of 0.5 and assign these to an object called y. Then, add together the objects x and y. It *is* as easy as you think . . . but before you do that, think about what you expect the answer will be.

Here's what you could/should have done:

```
y <- seq(from = 101, to = 110, by = 0.5)
x + y
```

```
##  [1] 102 103 104 105 106 107 108 109 110 111 112 113
```

```
## [13] 114 115 116 117 118 119 120
```

Great, you just added together two vectors (these two vectors are both a collection of numbers) and thus experienced a hint of the power of R! But

wait . . . there is a bit more. We think it is pretty cool to note just how R did the maths here. If you look and think carefully, you will realize that R added the vectors element by element. This is very cool. All those sums, all at once.

Now, before you discover more of this awesome power, we think it's time you learn to *never again* type instructions into the Console. Well, almost never.

1.6 Your first script

Up to know, you have been typing instructions into the Console, pressing enter, and watching R give you the answer (or assigned the answer to an object if this is what you asked). We *very* strongly advise you not to regularly do this. In fact, you will have real trouble if you do this regularly. Perhaps the only consistent use of typing in the Console should be asking for help (see below).

The alternative is to type your instructions in a separate place, and then send them to the Console (on a magic carpet) when you want R to work on them. As we noted in the Preface, R allows you to write down and save the instructions that R then uses to complete your analysis. Trust us, this is a desirable feature of R. As a result of making a *script*, you, as a researcher, end up with a *permanent*, *repeatable*, *annotated*, *shareable*, *cross-platform* archive of your analysis. Your entire analysis, from your raw, transferred data from lab book to making figures and performing analyses, is all in one, *secure*, *repeatable*, *annotated* place. We think you can see the value. We continue to benefit from this every time we get referees' comments back from a journal . . .

Because this is such a good way of working with R, both the basic version of R for Windows and Mac, and RStudio for all platforms have built-in *text editors*. These are a bit like a word-processing pane that you write stuff in, but these editors tend to have at least one major feature that word processors don't—there is a keystroke combination that, when pressed

together, magically sends information from the script to the Console window. This is very convenient; much more so than copying and pasting.

1.6.1 THE SCRIPT PANE

Let's first find the script pane in RStudio, and then explore this and two other features we think make it amazing. To see the script pane, you need to look in the upper right corner of the Console pane that is dominating your screen. Up there, on the right, you should see two squares overlapping each other. Go on. Press them. All of a sudden, like magic (we all need magic), there should be two panes on the left. The top pane is your *script* pane (Figure 1.7). From now on, when you open RStudio, there will be four panes, not three.

The script editor in RStudio has some nice features *not* found in several other versions of the R application. To see some of these in action, let's get started with a new script. To open a new script, go to the *File* menu, then *New File*, and click on *R Script*. Or you can click on the top left button

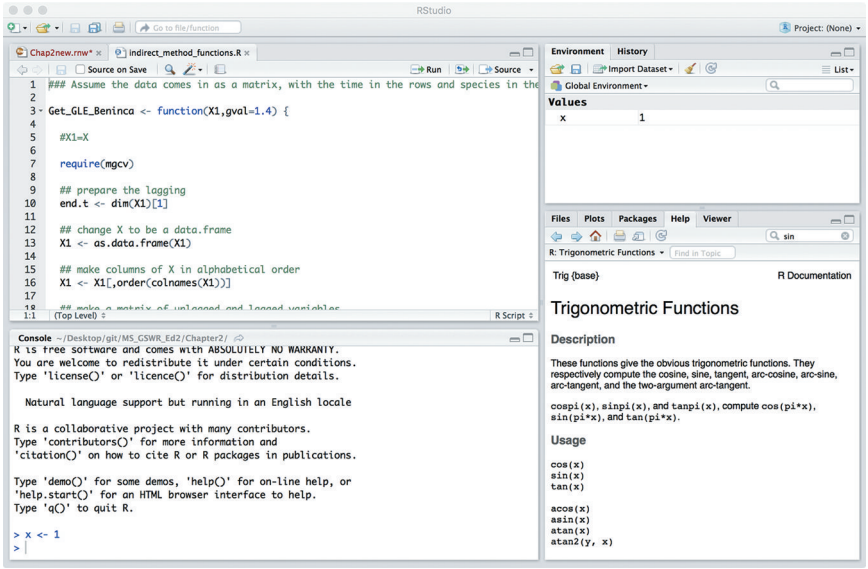

Figure 1.7 How the RStudio application looks with four panes . . . the new one is the script pane.

in the script toolbar, which looks like a document with a green and white plus, and then click on *R Script* in the drop-down menu.

You now have somewhere to write your instructions (and save them). Let's start writing in the script with a very special symbol, the # symbol. # is a very special character. It is important to find this character on your keyboard. It may not be obvious on European keyboards, especially European Macintosh keyboards. If it is not visible, it is often associated, somehow, with the number 3 (for example, alt+3 on many Macintosh keyboards). The 3 often has the £ symbol on it as well. The # symbol is sometimes called the 'pound sign', like in parts of the USA.

The # is a symbol that declares that anything after it on that line is a comment, i.e. an annotation. These are words for you, not for R. We recommend starting your scripts with some information about the script, the date, and the author . . . For example,

```
# Amazing R. User (your name) 12 January, 2021 This script is
# for the analysis of coffee consumption and burger eating
```

One thing you might notice is that this text is shown in a different colour from black (probably in green). This is good. Hold onto whatever you are thinking . . . Next in our script, we recommend adding two lines as follows:

```
# clear R's brain
rm(list = ls())
```

This bit of R magic is very important, and you should have it at the beginning of virtually every script you write. It clears R's brain. It's always good to start a script with nothing accidentally left over in R from a previous session. Why? Well, imagine that you're analysing the body size of your special species in two different experiments. And, not surprisingly, you've called it body_size in both datasets. You finish one analysis, and

then get started on the next. By clearing R's brain before you start the second analysis, you are ensuring that you don't use the wrong body size data for an analysis.

After this bit of R code, you can now add a bit more annotation/commentary, and some interesting maths operations for R to do, like this:

```
# Amazing R. User (your name)
# 12 January, 2021
# This script is for the analysis of coffee consumption and
# burger eating

# Clear R's brain
rm(list = ls())

# Some interesting maths in R
1 + 1
2 * 4
3 / 8
11.75 - 4.813
10^2
log(10) # remember that log is natural in R!
log10(10)
sin(2*pi)
x <- seq(1, 10, 0.5)
y <- seq(101, 110, 0.5)
x + y
```

Here we highlight some things to note about this script:

- The first four lines begin with a hash (sometimes called the pound sign), '#'. This is the symbol that tells R to ignore everything that follows on the line. So R doesn't even read the 'Amazing R. User (your name)' text, which is good, because it wouldn't have the first idea what this means. The # means what follows is for humans, not for R.
- The script contains only the instructions for R. It does not contain the answers. We still have to get R to calculate these for us.
- The seventh line (including the empty one) is very important, and you should have it at the beginning of virtually every script you write. Did we mention that already?

- Note that there are at least four colours being used in your script. This is called syntax highlighting. This is a good thing. It separates comments, R function, numbers, and other things.
- If you were watching carefully as you typed, you will have noticed that RStudio was completing your brackets/parentheses for you . . . every time you opened one up, RStudio provided the closing one. *Very handy*.
- It has white space. Between the first annotation and the Brain Clearing (line 7), there was a line with nothing. After the brain clearing, there was a blank line. We recommend this kind of white space. It creates *chunks* of code and annotation. It makes your script easy to read, and easy to follow; it is good practice.

Now, have a look up at the colour of the word *'untitled1'* in the tab of the script pane. It is probably *red*. That's bad. Warning colour. *Danger*. Your work is not saved. Now it is time to save this script to your analysis folder. You can either choose File -> Save or use keystrokes like ctrl+S (Windows) or cmd+S (Macintosh). Provide an informative name for the script. Save it. Don't lose all the hard work.

OK. Danger has passed. Breathe easy. The next bit is fun.

1.6.2 HOW DO I MAKE R DO STUFF, YOU ASK?

RStudio makes it easy to get these (or any) instructions from the script editor (source) to the Console. The hard, slow, and boring way to do this is to select/highlight with your mouse or keyboard or trackpad the text you want to put in the Console, copy it, then click in the Console, paste it, and then press enter to run the script. Phew . . .

But there is magic in these computer things. The super-easy, quick, and exciting method is to click anywhere in the line you want to submit (you don't have to highlight the whole line) and press the correct keyboard shortcut. On a Mac press cmd+enter OR ctrl+enter, and on Windows ctrl+enter. You will then see the line of code magically appear in the Console. Nice!

If you insist on using the mouse, in RStudio you can also press the `Run` button, found on the upper right of the script tab (it actually says 'Run') to send script lines from the editor to the Console. We think it is faster, however, to use keyboard shortcuts. Most of these shortcuts are actually documented in the `Code` and `View` menu items in RStudio (Figure 1.8) and on the RStudio website (google for `RStudio keyboard shortcuts`; probably the first link is highly relevant).

Furthermore, if you highlight several lines of code in the script editor and press the shortcut, all of them are delivered/submitted to the Console in one go. And to highlight and submit all of the instructions in a file of script, press cmd+a (Mac) or ctrl+a (Windows) to select everything in the script, then use the shortcut to submit those instructions.

Please, please, please use these shortcuts for getting instructions from the script editor to the Console. If you ever find yourself copy and pasting from the script editor to the Console, feel bad. Very bad. You have forsaken the magic.

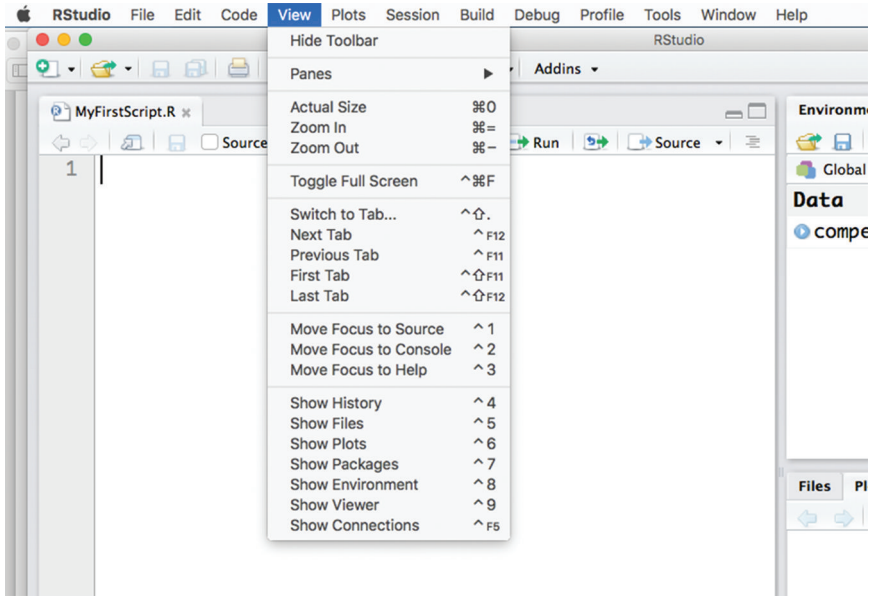

Figure 1.8 The RStudio View menu shows you many, many shortcuts via the keyboard. Check out the Code menu for more.

Let's see two more bits of handy Rstudio magic. First, click in the Console, so that the cursor is flashing at the prompt. Now, press crtl+1. Can you find the cursor? Has it moved up to the Source pane and your script? Now, try crtl+2. Has it moved back? Brilliant. Another quick way to navigate between the two, if needed. Check out the View menu (Figure 1.8) for how to use crtl+1:9!

Second, let's look at something in the Tools menu. Choose Tools -> Global Options. In the box that comes up, select Appearance. Oooh, this looks fun. Not only can you change the font, but you can also alter the way things are coloured. Recall we noted above that the script had at least four colours. Go ahead and play around with the 'editor theme'. One of us is a particular fan of Tomorrow Night 80's. Go figure. Don't forget to press 'Apply' and then 'OK' to make these changes. Just one way to take control . . .

1.7 Intermezzo remarks

At this point, you have the tools to start engaging with R. You should mess around with RStudio and make it your own. Don't let it control you . . . be in charge. You are also aware now of how powerful the use of a script can be. We want to emphasize that it offers something extraordinary. As a result of making a script, you, as a researcher, end up with a *permanent*, *repeatable*, *annotated*, *shareable*, *cross-platform* archive of your analysis. Your entire analysis, from your raw, transferred data from lab book to making figures and performing analyses, is all organized in one, *secure*, *repeatable*, *annotated* place.

1.8 Important functionality: packages

R's functionality is distributed among many *packages*. Each has a certain focus; for example, the ***stats*** package contains functions that apply common

statistical methods, and the ***graphics*** package has functions concerning plotting. When you download R, you automatically get a set of *base* packages. These are *mature* packages that contain widely used statistical and plotting functionality. These base R packages represent a small subset of all the packages you can use with R. In fact, at the time of writing, there are more than 8000. These other packages we call *add-on packages*, because you have to add them to R, from CRAN, yourself.

R packages can be installed in a number of different ways. But, as we might expect, RStudio gives you a nice way of doing this. The *Packages* tab in the bottom right pane has an Install button at the top left. Clicking on this brings up a small window with three main fields: *Install from*, *Packages*, and *Install to Library*. You only need to really worry about the *Packages* field; the other two can almost always be left at their defaults.

When you start typing in the first few letters of a package name (e.g. ***dplyr***), RStudio will provide a list of available packages that match this. It is totally possible to install more than one package, placing a comma or a space between the different ones you ask for. After we find them, all we need to do is click the *Install* button and let RStudio do its magic.

Let's do this now. This book is going to use a great deal of two add-on packages: ***dplyr*** and ***ggplot2***. We'd like you to get them and install them on your computer. Follow the instructions above. It should be painless. Once you're done, take a quick look at the Console. If it all worked, you will see that all RStudio did was send the R function **install.packages**() to the Console to install the packages for you.

1.8.1 USING THE NEW FUNCTIONS THAT COME IN A PACKAGE

Once we've installed a package onto our computer, we still have to load it into R's brain. A good way of thinking about all of this is by analogy with your phone (we assume you have some kind of smartphone; apologies if you don't). When you download an app, it's grabbed from the app store

and placed on your phone. What you did above was to download to your computer and install in R the *apps* ***dplyr*** and ***ggplot2***. The R app store is CRAN.

However, just like with your phone, these apps don't *just start*. You need to put a finger on the icon to make it start working on your phone. In R, the way we *press the icon* is to use a function called **library**(). Now that you've installed ***dplyr*** and ***ggplot2***, let's add info to your script so that when you run these lines, the packages are *activated* and ready to use!

```
# Amazing R. User (your name)
# 12 January, 2021
# This script is for the analysis of coffee consumption and
# burger eating

# make these packages and their associated functions
# available to use in this script
library(dplyr)
library(ggplot2)

# clear R's brain
rm(list = ls())

# Some interesting maths in R
1+1
2*4
3/8
11.75 - 4.813
10^2
log(10)
log10(10)
sin(2*pi)
x <- seq(1, 10, 0.5)
y <- seq(101, 110, 0.5)
x+y
```

- *Top Tip 1*. Put the **library**() commands at the top of your script, all in one place. This will help you see what you, or someone else, need to have installed in order for the script to run successfully.
- *Top Tip 2*. Just like on your phone, you do not need to *install* the packages every time you start a new R session. Once you have a copy of a package on your hard drive it will remain there for you to use, unless you delete it or you install a new version of R.

- *Top Tip 3.* Note that **rm**(**list** = **ls**()) does not remove packages (we have placed this code after the two **library**() commands). The brain clearing only removes objects you made. That's why it can come after the **library**() commands.

1.9 Getting help

You are nearly ready to be let loose with R and RStudio. But what about getting help from R?

The classic way is to type in the Console '?the-function-name'. This will open a window containing R's help information about the function. For example, ?**read.csv**() will give you the help file for the **read.csv**() function. Box 1.2 provides insight into how to read the **seq**() help file. Not easy, eh! These help files take no prisoners, but after some practice are very useful. Take the time to go through them.

There are lots of other ways of getting help. Google is a great friend, as usual. Search on Google for something and add the letter R, and you will likely get some very useful results. For example, try searching for 'how to make a scatterplot R'. Probably you will have trouble deciding which of the probably very useful first few results to use.

For more controllable searching of R resources, consider using the R channel on Stack Overflow (`http://stackoverflow.com/tags/r/info`) or RSeek (`rseek.org`), whose search results are from Google, but can be easily filtered by categories such as Support, Books, Articles, Packages, and For Beginners.

Another very nice resource is cheat sheets. These are concise and dense compilations of many common and useful functions/tasks. Excellent examples are the RStudio cheat sheets, which you can access from `https://www.RStudio.com/resources/cheatsheets/` or via the Help menu in RStudio. You will find the *Data Wrangling* and *Data Visualization* cheat sheets particularly useful. Not to mention the one for RStudio itself. We've printed these, laminated them, and have them close

to hand at all times. Our families think this a bit weird; we don't care. We know we are a bit weird.

Box 1.2: Getting help from an R help file

There are lots of ways to get help, from asking the person sitting next to you, to consulting Google and to reading books. However, the R help files are very important and very useful for R. They have a consistent formal structure: they always start with the name of the function and the package to which it belongs and a short description. They always end with some examples. In the middle, they always contain sections called *Usage*, *Arguments*, *Details*, *Value*, *Authors*, *References*, *See Also*, and *Examples*.

We can see that **seq**() (Figure 1.9) is a function in the base package of R (upper left). We can use it for 'Sequence Generation'. The description is followed by Usage and Arguments. Note that there is specific usage for a data.frame. There are five main arguments for the method. For example, the from argument is the number we'd like

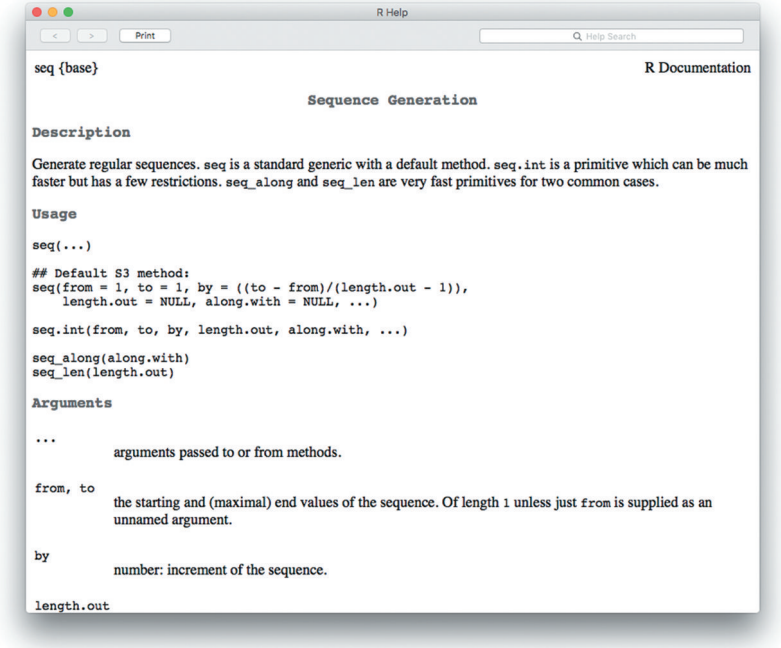

Figure 1.9 The seq help file you can access with ?seq.

R to start the sequence at. We have used from, to, and by. We draw your attention to length.out, which can be shortened to *length*. It is a valuable alternative to by, returning a fixed number of values between the start and finish numbers.

If you scroll down to the bottom of the help file, you will see quite a lot of information in the *Details* section. Then there is information on the *Value* returned by **seq**(), some information on the *Authors*, and information on the *References* that influenced the development of the function. Then comes a section *See Also*, in which related functions are listed (quite useful if the function you're looking at seems not to do exactly what you'd like, and you suspect there might be a more useful function). Finally, and importantly, there are also *Examples*. Note the often prolific use of annotation in the examples. Note as well that you can copy any of the examples and paste them into your Console and they will run. This allows you to dissect the use and anatomy of this function.

The R help files are not always so easy to understand, but often contain answers. So stick with them, with some determination to figure them out; the effort will be well spent.

1.10 A mini-practical—some in-depth play

OK. Let's see what you can do. To get a little more comfortable with R, using the script in RStudio, reading and using help files, and to challenge you, we offer you the chance to try a few exercises (the solutions are in Appendix 1a at the end of this chapter):

- Plot a graph with x^2 on the y-axis and x on the x-axis.
- Plot a graph with the sine of x on the y-axis and x on the x-axis.
- Plot a histogram of 1000 random normal deviates.

The answers are in Figures 1.10–1.12, so you can see what we're aiming for. We'll be using two new functions to solve these exercises: **qplot**() and **rnorm**(). **qplot**() makes a plot, and **rnorm**() gives us random numbers from a normal distribution! Don't forget the '?' to look at help files!

Please, please, please take at least half an hour to attempt these exercises. This is really important. It doesn't matter if you can't get exactly the graphs in Figures 1.10–1.12, but it's really important that you try. In trying, you

will experiment with R, get frustrated, get elated (hopefully), experience errors, and have questions. Experiencing all this now will pay dividends later. So it's in your very best interests to stop reading now and have a go. We provide a version of the solutions in Appendix 1a.

Figure 1.10 The solution to the first problem.

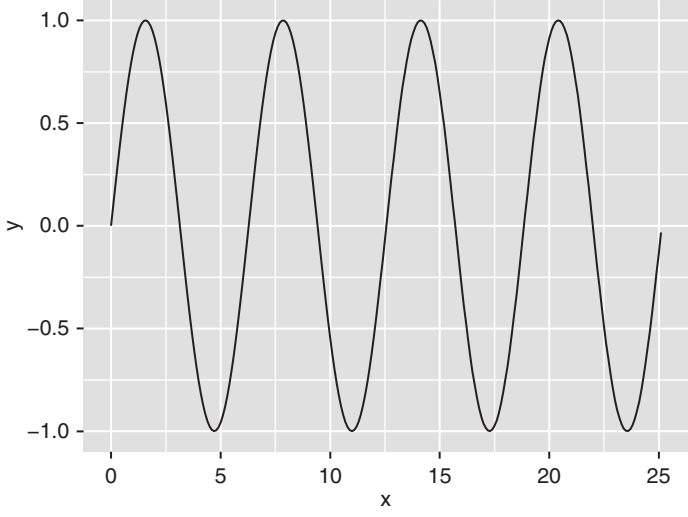

Figure 1.11 The solution to the second problem.

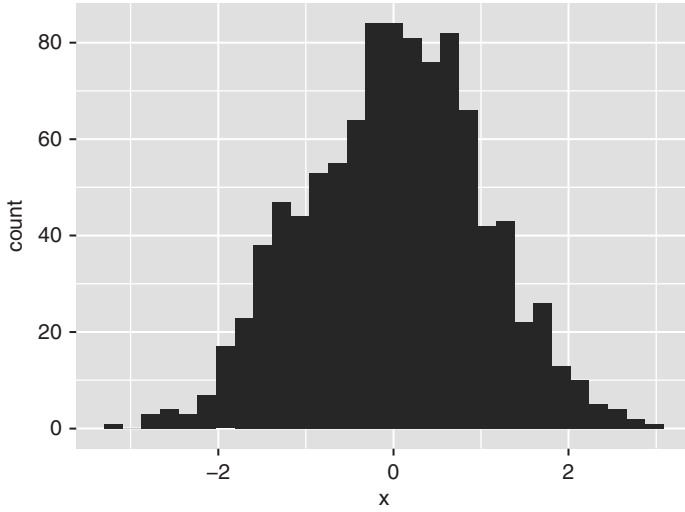

Figure 1.12 The solution to the third problem.

1.11 Some more top tips and hints for a successful first (and more) R experience

1.11.1 SAVING AND THE WORKSPACE OPTION

Save your script and quit RStudio. If you do this, you'll probably be asked if you'd like to save the workspace. Because you saved your script, and because your raw data are safe, there is no need to save the workspace. So we never save it. Well, there are always exceptions, like if you had some really involved analyses that took R much more than a few minutes to complete. Then you could save the workspace. But a better option in these cases is to use the **save**() function. Just to be sure, the two most important things in your R-life are your raw data and your script. Keep those safe and you're sorted.

1.11.2 SOME NICE THINGS ABOUT RSTUDIO

As well as doing the basics very well, RStudio has some very useful, quite advanced features; by using it you are somewhat future-proofing yourself. Hence, from here on, we assume you are using RStudio. As well as organizing the Script Editor, Console, and other panes/windows nicely,

RStudio has other features that can make R-life easier than otherwise. We cover some of these in later chapters, but it's worth exploring the help/cheat sheets for RStudio to see what is on offer. The RStudio team of developers is awesome. RStudio has the following features:

- It works very similarly on Windows PC, Mac, and Linux.
- It allows you to comment or uncomment regions of code.
- It automatically indents code.
- It offers suggestions for completion of code.
- Function help appears while scripting.
- It provides convenient handling of add-on packages.
- It has advanced debugging tools.
- You can easily create documents (reports, presentations) directly from R, using RMarkdown or Sweave.
- It includes advanced version control integration.
- It includes advanced package-building tools.

Appendix 1a Mini-tutorial solutions

Here's one way to solve the first problem. Before we touch R, let's think through what we need to do. Well, we need some x values and y values to plot against each other. The x values need to go from negative to positive, for example from –10 to 10, and in small enough steps to make a smooth curve. Then we'll need to make the y values, which we'll make equal to x^2. Then we'll make a line plot of y against x. The first two steps we nearly did already in the previous section on playing with R. The plotting is a bit new (with the result shown in Figure 1.10). In an R script, this will be:

```
# Exercise 1
# Plot a graph with x^2 on the y-axis and x on the x-axis.
rm(list=ls())
library(ggplot2)

x <- seq(-10, 10, 0.1)
y <- x^2
qplot(x, y, geom="line")
```

You probably noticed that we're using the plotting function **qplot**(). To use this function, you need to get and load into R the add-on package ***ggplot2***. We use a plotting function called **qplot**() here, rather than **ggplot**(), which we will introduce to you to soon, because it's quick. The 'q' stands for 'quick'.

The solution to the second problem is quite similar. We make an x variable, this time from 0 to say 8π, make the y variable equal to the sine of the x variable, and then use the same plotting command (with the result shown in Figure 1.11):

```
# Exercise 2
# Plot a graph with sine of x on the y-axis and x on the x-axis.
rm(list=ls())
library(ggplot2)

x <- seq(0, 8*pi, 0.1)
y <- sin(x)
qplot(x, y, geom="line")
```

The solution to the third problem is a little different. Here's one way to solve it. First we ask R to make the 1000 random normal deviates and assign them to an object, and then we use the **qplot**() function to plot the histogram (with the result shown in Figure 1.12).

```
# Exercise 3
# Plot a histogram of 1000 random normal deviates.
rm(list=ls())
library(ggplot2)

x <- rnorm(1000)
qplot(x)
```

If you were wondering how the **qplot**() function knows to make a histogram, great. Actually, it guesses, based on it receiving from us only a single variable. That is, **qplot**() thinks, 'I have only one numeric variable to work with, so what is most likely useful? Oh, I know, it's a histogram.'

Appendix 1b File extensions and operating systems

A note about the three-letter combinations on the end of filenames . . . otherwise known as filename extensions (such as .exe, .csv, .txt). These tell

your operating system what application to open a file in if you double-click the file.

When you save a script file, it will get saved with the name you choose followed by the 'filename extension' '.r' or '.R'. Ideally this will mean that when you double-click on a script file, it will automatically open in RStudio. If double-clicked R files are not opening in R, it's rather annoying, and the solution follows . . .

On a Mac and in Windows the default is for hidden file extensions (you don't see them). However, it can be quite useful to see them, especially when they get mangled, by whatever means. We usually ask to see the file extension, since it's not too confusing and is in fact quite useful. If, however, you prefer to not see file extensions, you can still see them if you right-click on the file and select 'Get info' (Mac) or 'Properties' (Windows).

MACINTOSH

To ensure that OSX shows you file extensions, you need to adjust the Finder preferences (Figure 1.13): Finder -> Preferences -> Advanced; tick 'Show all filename extensions'.

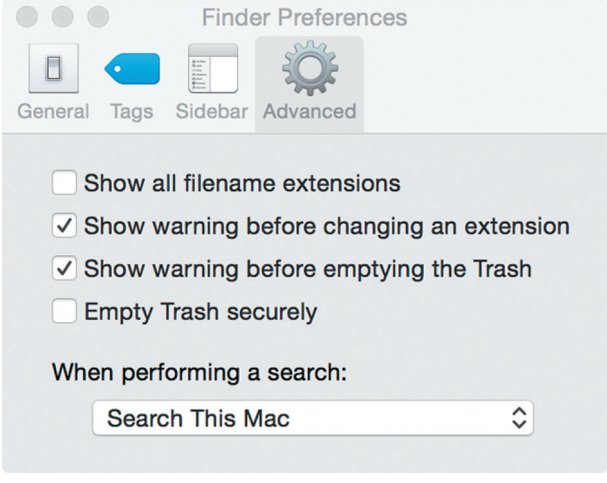

Figure 1.13 The advanced dialogue box of the Finder preferences. Here you can ask OS X to show all file extensions, which you might find useful, or not!

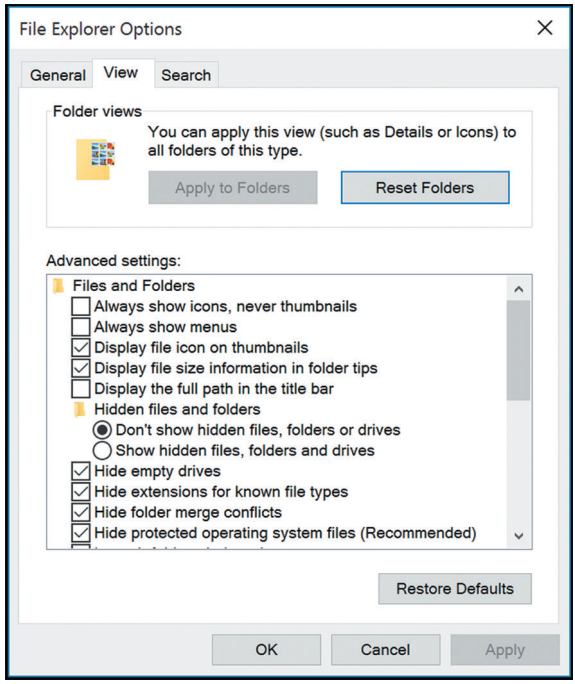

Figure 1.14 The advanced settings tab for File Explorer options in Windows. Here you can ask Windows to show all file extensions, which you might find useful, or not!

WINDOWS

To ensure that Windows presents file extensions, navigate in an Explorer window via the Tools menu to Folder Options: Tools -> Folder Options -> View Tab (Figure 1.14). Ten tick boxes/circles down, you should see an option stating 'Hide extensions for known file types'. Deselect this if it is selected, and Windows should show all file extensions.

FORCING ITEMS TO OPEN WITH RSTUDIO

If double-clicking on a script file doesn't result in it automatically being opened in RStudio, check that your script file has the `.r` or `.R` extension (though don't forget that, if you didn't ask, you probably don't see the file

extension by default). If it doesn't open, and it doesn't have the correct file extension, add `.r` or `.R` to the end of the filename. Your computer may complain about this: do you really want to change the file type/file

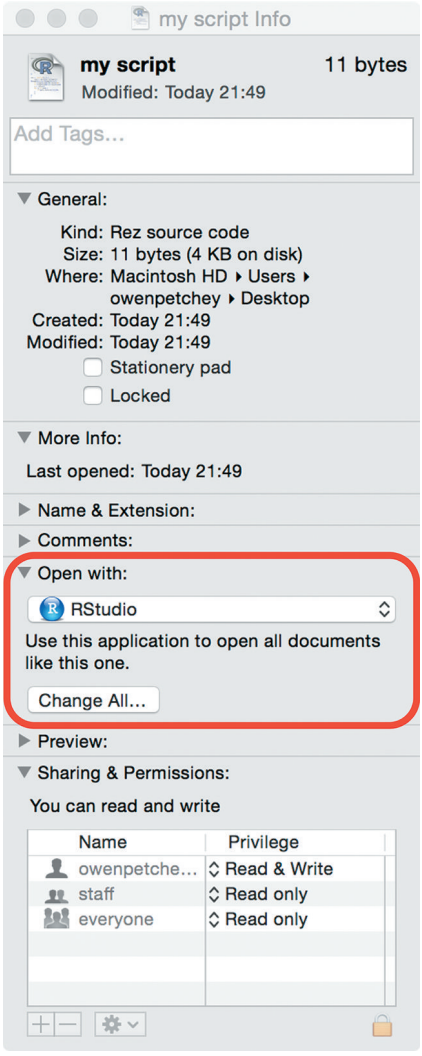

Figure 1.15 The window you get if, on a Mac, you ctrl+click on a file and select *Get Info*. Here you can change the application that opens this file, or even all files of this type (the bits inside the red box).

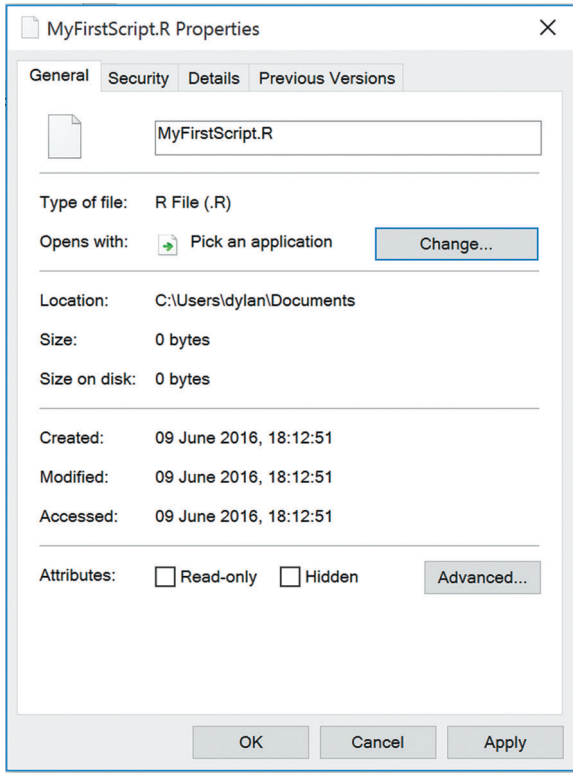

Figure 1.16 The window you get if, in Windows, you right-click on a file and select *Properties*. Here you can change the application that opens this file.

extension? Actually, this is a great check, since you can really upset your computer by changing filename extensions!

If you script still doesn't automatically open in RStudio when double-clicked, right-click on it, go to Get info (Mac, Figure 1.15) or Properties (Windows, Figure 1.16) and look to see which application if any is selected to *open* this file type. Change or set this to RStudio.