Farr Institute

Data Management and Statistics

Day 1: Data Management

# Introduction

## Why be interested in data management?

There’s a joke[[1]](#footnote-1) I remember from the 2014 Royal Statistical Society annual conference:

**Question**: How can a statistician double their salary?

**Answer**: Call themselves a ‘data scientist’.

The truth behind the joke is that there’s a lot of demand for data scientists, both within public and private organisations, and on first impressions they look a lot like statisticians. However, there are some important differences between data scientists and statisticians. In essence, data scientists are generalists, concerned with the complete data-to-knowledge value chain:

1. The initial generation of quantitative data records;
2. Cleaning, standardising and tidying the data records;
3. Statistical analysis;
4. Evidence-based decision making.

By contrast, statisticians tend to be, or at least to start off as, specialists focused on stage (3) of the above, adept with understanding and applying statistical theories and concepts to particular types of data, prepared as datasets which have been constructed in particular ways. The datasets that most statistics courses provide to students are ‘tidy’ (a term I’ll define more clearly later on), and typically what is taught in such courses is how to analyse the data in this format.

However, routine and administrative data seldom emerges in a tidy data format, ready to be loaded up and analysed in a statistical package. Instead, the data needs to be prepared and processed in a large number of ways. For example:

* characters may need to be removed from fields;
* rows and columns may need to be combined;
* tables may need to be joined;
* derived variables may need to be generated;
* typos need to be identified and fixed;
* information (‘metadata’) about the types of variables (logical, categorical, ordinal, or cardinal) may need to be passed to formally specified in particular software
* et cetera, et cetera, et cetera

Although researchers using quantitative data are generally motivated to use such data by stage (4), the production of knowledge and making good evidence-based decisions, a great deal of the time spent doing quantitative can be spent at stage (2). Often, stage (2) does not just take up ‘much’ of the time, but **most** of the time. When the ‘base metal’ is routinely collected administrative data, the production of tidy data often takes much longer than the statistical analysis.

The purpose of this first day is to provide a series of tools, both conceptual and practical, which make stage (2), the management and tidying of administrative data, much quicker and easier to do. The reason for going into more depth about the concepts and practice of data management is, paradoxically, because data management is not interesting. The more of your time you spend on data management issues, the less time you have to analyse the data, and to make informed decisions about the data. Conversely, if you have a series of tools and concepts at hand for managing data efficiently, you can pass through this stage more quickly, and spend more of your time at stages (3) and (4).

## A motivation for surviving Day 1

A lot of material, and a lot of new ideas, will be presented in the next few hours. There is a lot to cover, as there are a lot of potential data management challenges that can be encountered, each requiring different tools and approaches to solve efficiently. You will typically need to apply many different tools and techniques even when working with just a single data source, and to know how to use these techniques seamlessly and in combination with each other.

If all goes well, the last hour of Day 1 will be spent on a practical session which will require using many of the tools introduced just a few hours earlier. This extended practical will involve downloading, tidying, and performing some basic analyses of data available from the Center for Disease Control (CDC) Wonder database:

<http://wonder.cdc.gov/>

This database was used in a recent high profile US public health paper, published in the journal PNAS in late 2015:

<http://www.pnas.org/content/112/49/15078.abstract>

As you can see from the associated ‘Altmetrics’ webpage, this paper’s findings and claims generated an exceptionally high level of mainstream media attention:

<http://www.pnas.org/content/112/49/15078.abstract?tab=metrics>

Your motivation, in the short-term, for persevering with the next few hours, is to be able to recreate some of the analyses presented in the paper, then hopefully to go beyond some of these analyses.

# Getting Started

## R and RStudio

R is not a statistical package, but a statistical programming language. For researchers used to standard statistical packages like SPSS and Stata, this distinction can be a major stumbling block when first learning R. R has a higher learning curve than a statistical package, requiring a large investment of time, self-esteem and possibly even sanity at the outset. However, for those who persevere with R, there are great advantages in terms of much greater power and flexibility. R is highly adaptable, and unlike most statistical packages can handle almost all stages in the data-to-value knowledge chain sketched above. This, combined with its script-based rather than point-and-click-based interface, means there can be great efficiency gains to be made in the entire data-to-knowledge generation process. Functions, code and methods, learned once, can be re-used and re-applied, potentially reducing the marginal costs of additional analyses from hours to seconds. The flexibility of a programming language means there is no need to be constrained to pre-prepared menu of statistical models or processes. R can be made to be whatever you need it to be.

R, though not free in terms of time, is open source and free in terms of money. One of its strengths but also its challenges is that is the product of many minds.

## Installing R

Though it has already been installed on the machines you will be using today, on home and other machines it will have to be installed. You can get R from the Comprehensive R Archive Network, or CRAN:

<https://cran.r-project.org/>

You will then need to download the right version of R for your operating system: Windows, OS X, or Linux.

Once installed, R can be opened up, and will look something like this in Windows:

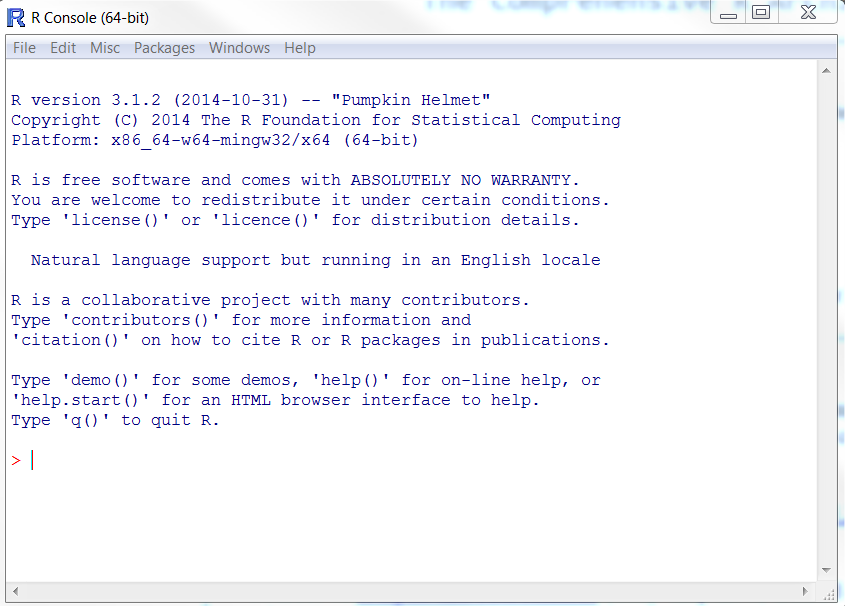


Figure Example of the basic R Graphical User Interface (GUI) in Windows

Within day 1, however, we will use another free program, called RStudio, to work with R, as it makes the process of working with R and R data projects easier in a number of ways.

## Installing RStudio

RStudio is known as an Integrated Design Environment (IDE), a term more familiar with programmers than statisticians. IDEs ‘sit on top’ of a programming language, making it easier to do things like explore the contents and components of a program, find help and information about functions, manage large numbers of files that need to work together towards a common end (a ‘project’), automatically check for and correct syntactical errors and bugs and code, and in general make the process of performing complex series of tasks easier. RStudio should already be installed on the machines in this workshop, but for home and other office use can be downloaded from the following links:

<https://www.rstudio.com/products/RStudio/#Desktop>

**Note 1**: Only install RStudio *after* installing R.

**Note 2**: Select the free Rstudio Open Source Desktop Edition rather than the $995/year commercial edition!

## Getting Started with RStudio

When you first load up RStudio you will probably get something as follows:

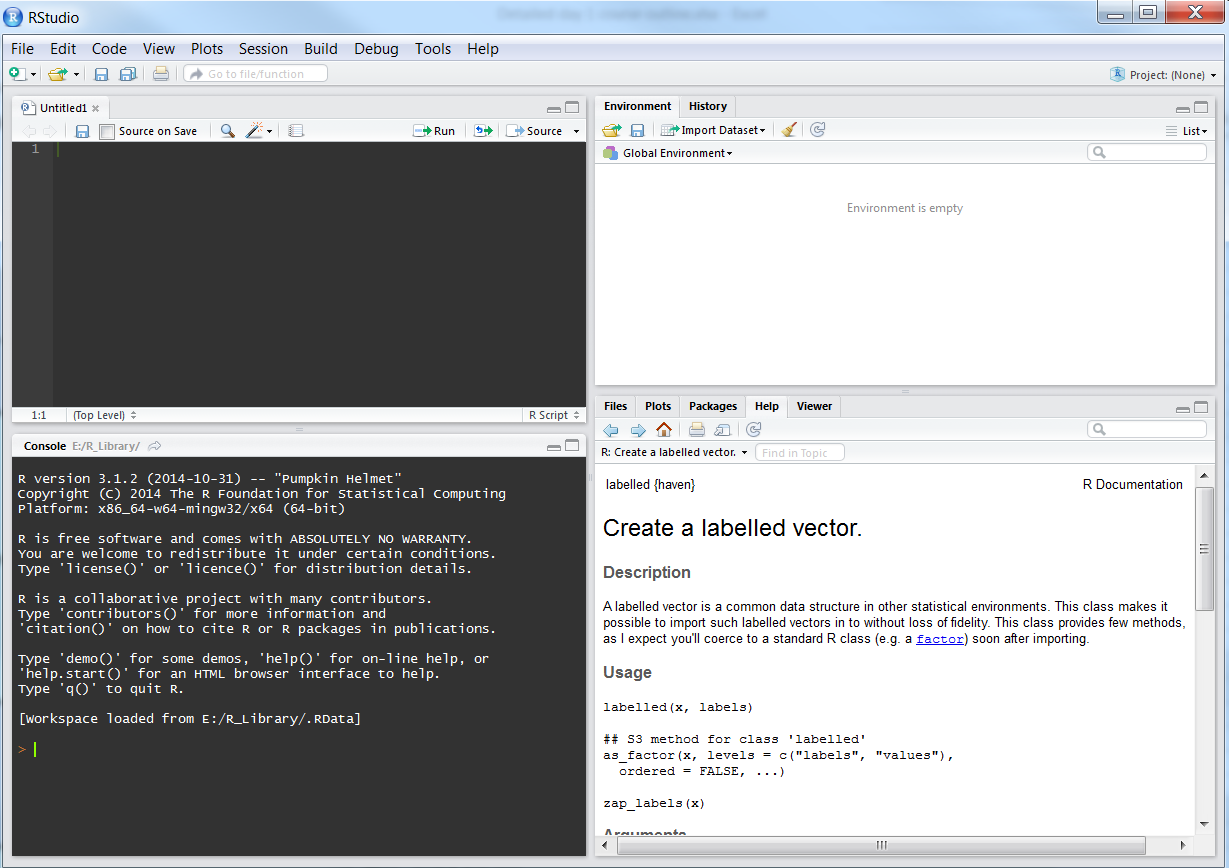


Figure Illustration of Rstudio and its four panes. Top left: Script pane; Bottom left: I/O pane; Bottom right: figures, help and documentation; Top right: history and environment

RStudio basically divides the display into four panes. With the exception of the bottom left pane, each pane contains a number of tabs, allowing you to switch between the types of information displayed. The relative size of each pane can be adjusted, and you can minimise and maximise each pane too. The contents and purpose of each of the four panes is as follows:

* **Bottom right**: the Console or I/O (Input/Output) pane. This is basically what you saw when you opened up R directly. You can use this to work interactively with R, giving it one instruction (input) at a time, and getting results (outputs) as a result of these instructions.[[2]](#footnote-2)
* **Top left**: The script window and data viewer. This allows you to access, toggle between, and edit a number of script files, which are text files containing lots of instructions for passing to the R Console. For any serious data management project, you will work with and generate scripts, passing script chunks to the console, exploring the effects they have once processed by R, and then modifying the scripts based on whether or not R did what you wanted it to do. Code in the script window looks a bit different to that in the console itself, in that it is colour coded according to the types of instructions RStudio things it contains, and in general is better formatted to be easier for people to understand. By using the function View() or similar, you can also create and view datasets in this pane, with an SPSS or Excel style look about them.
* **Top Right**: History and Environment: This pane provides access to two tabs: A History tab which contains a list of the log of instructions which you have passed to R during its current session; and an environment tab showing the objects that exist within the current R session. The environment tab allows you to get some additional information about some of these R objects using something other than the R console.
* **Bottom Left**: Figures and Help. These pane contains a number of tabs, the most important of which are the ‘Plots’ tab, which show figures and other images which have been created by R; and ‘Help’, which provides information on particular R functions and how to use them. It also contains ‘Files’ (self explanatory), Packages, which allows downloading of and information about the packages of functions which have currently been installed and loaded, and Viewer (which I’ve never used).

Although the two panes on the right hand side of the RStudio IDE offer a number of ways of indirectly creating and passing instructions to the R Console – for example: accessing data objects; installing and loading R packages and libraries – it is important to be able to know how to construct these instructions without depending on these interfaces. Instead, if something is required to complete a data management process, you should get into the habit of including those instructions in the scripts.

## RStudio Features

RStudio offers a number of subtle but important features compared with accessing R through its native GUI. In particular, the features available in the script window offer a number of important advantages over writing the scripts in a simple text editor like notepad or the script editor included with R. To start to learn both about some of these features, and about R as a programming language, let’s look at the following figure:



Figure Example of an R script in the RStudio console

One of the first things to note is that different parts of the text are coloured in different ways. These colours aren’t put in manually by the user, by generated automatically by RStudio, as it recognises different parts of the text as specifying different types of information for the R Console. Some examples:

* **Light orange** (lines 1, 2, 4, 17, 25): These are comments, which R ignores, and written by the user to help them and other users understand the code better. Comments are distinguished from commands with the # (hash) symbol. The R console knows to ignore any text to the right of this symbols. (This means you can put a comment on the same line as an instruction, with the instruction on the left, the comment on the right, and the # symbol separating the two.)
* **Dark orange** (most lines 9-27): These are some of the functions contained in ‘Base R’, the functions that are an integral part of the R language. The function shown here is the ‘require’ function, which loads R packages, containing additional functions, into the current R environment.
* **Green** (lines 30 onwards): These are ‘string’ objects. Strings are technically vectors of character objects, but more intuitively ‘chunks’ of text. The distinction between string and other objects is subtle but important.
* **Blue** (throughout): These include symbols like parentheses (), the assignment operator <-, the pipe operator %>%, and simple (‘unpaired’) values (10, 50, 90, etc). Apart from the simple values, these symbols can be best thought of as being like conjunctions (‘and’, ‘with’, ‘and then’ etc) in sentences, with the roles of joining and connecting statements together.
* **White** (throughout): These pieces of the text are objects and non-base functions. They are the equivalent to nouns and verbs in sentences: ‘things’ and ‘stuff that gets done to things’. Just as in English, where nouns can get turned to verbs (e.g. ‘crawfished’, to use a Bushism), and verbs can get turned into nouns (e.g. ‘decide’ to ‘decision’), there’s a similar kind of mutability in the R language, hence the same colour being applied to both parts of the text.

## Autocomplete features

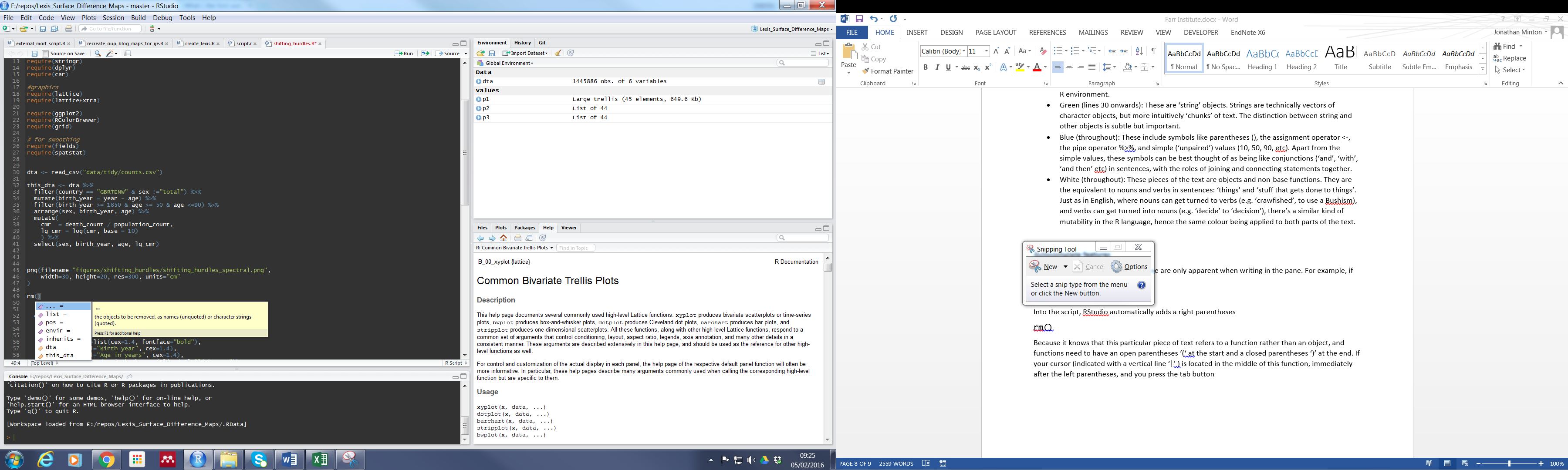
Most of the features in the script pane are only apparent when writing in the pane. For example, if you type

rm(

Into the script, RStudio automatically adds a right parentheses

rm()

Because it knows that this particular piece of text refers to a function rather than an object, and functions need to have an open parentheses ‘(‘ at the start and a closed parentheses ‘)’ at the end. If your cursor (indicated with a vertical line ‘|’ ) is located in the middle of this function, immediately after the left parentheses, and you press the tab button, the following drop-down list opens up



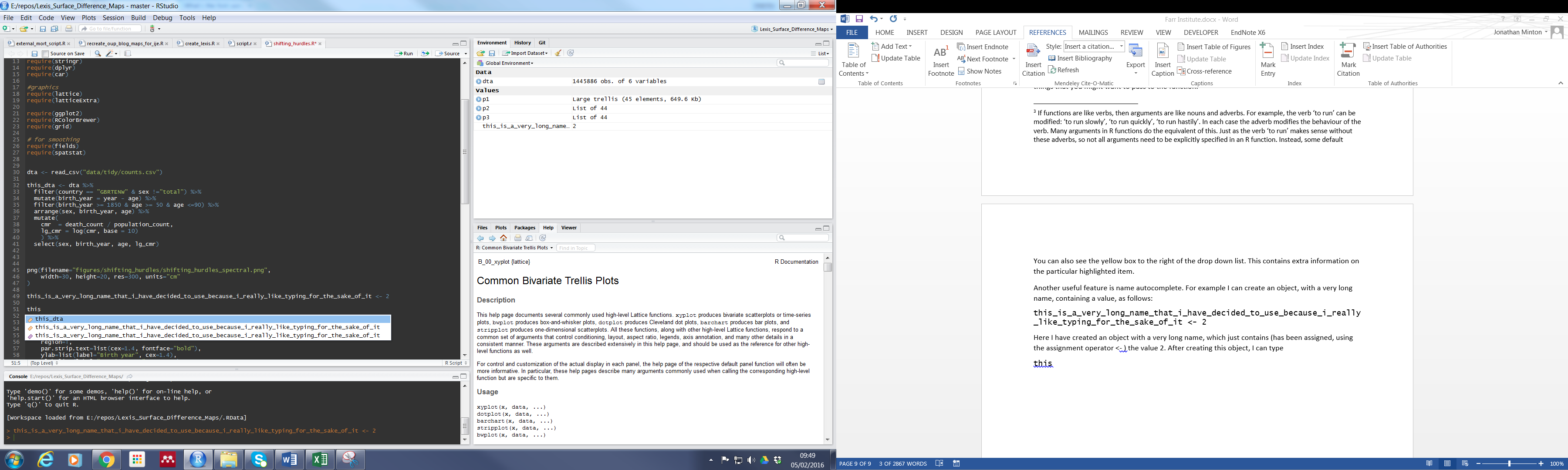
The purple items in the list are known as the **arguments** to the function, which modify the function’s behaviour.[[3]](#footnote-3) The yellow items in the list are objects found in the current R environment. These are things that you might want to pass to the function.

You can also see the yellow box to the right of the drop down list. This contains extra information on the particular highlighted item.

Another useful feature is name autocomplete. For example I can create an object, with a very long name, containing a value, as follows:

this\_is\_a\_very\_long\_name\_that\_i\_have\_decided\_to\_use\_because\_i\_really\_like\_typing\_for\_the\_sake\_of\_it <- 2

Here I have created an object with a very long name, which just contains (has been assigned, using the assignment operator <- ) the value 2. After creating this object, I can type the first bit of the name, then press ‘tab’, and I get a drop-down menu:



Rstudio knows that there are two possible objects in the workspace that I might be referring to, this\_data and the object with the very long name I’ve just created. I can use the arrow keys to select between these objects, then press tab, and the rest of the object name is automatically filled in. This little feature can be very important for two reasons: firstly it allows you to use long object names, which can be more helpful for helping to describe the code, without being penalised as much in terms of time spent typing. Secondly, it greatly reduces the risk of typos. If you mis-type something in R, it will think you are referring to a completely different object, either creating an object by accident (leading to errors occurring later in the code), or not finding the object you are referring to (leading to an error at this point in the code).

It is very important to note that R is CASE SENSITIVE, meaning that, for example, this\_object and This\_object are thought to be completely different objects. Similarly, this\_object and this.object refer to distinct objects. Consistent object naming conventions are therefore important for minimising the risks of code not working because of these distinctions. Hadley Wickham (much discussed later) recommends sticking to a convention of only using lower case characters, and separating words within object names using the underscore \_ symbol rather than anything else. Finally, it is important to be aware that, with one exception,[[4]](#footnote-4) the objects should contain no spaces: this\_object is recognised as one object, but this object is thought to be two objects, this and object, and will produce an error as you have not specified how this and object should relate to each other.

# Setting up R projects

## Creating an R Project

An important feature of RStudio is that it allows the creation of R Projects. When you create an R Project, you designate a particular directory on your complete as the base location from which R should search for files. As long as you can keep all the data, scripts and other material you need for a particular task within this R project directory, any code within that refers to files – either reading from or writing to files – should still work even as you move the project between locations on a single machine, or from one machine to another. You can think about R Projects as like a well organised briefcase, with all the material you need to work on something in the pockets and files you expect it to be, no matter where that briefcase is located. To set up an R project:

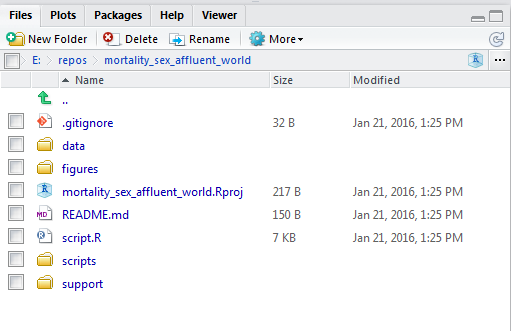
First, create a new directory on you completer, giving it the name of the project you want to use. It is best to use the coding conventions as within an R session, so no uppercase symbols and no spaces between words. To start with, create a directory called my\_first\_project, and remember its location on the file system.

1. Click on the little downwards facing chevron on top right icon on RStudio, initially saying ‘Project: None’ next to it (This means you are not currently in project mode).  This will open up a drop-down menu with a range of options, including one saying ‘new project’. Select this.
2. Select the option ‘Existing Directory (Associate a project with an existing working directory)’ and choose the my\_first\_project directory you created previously.
3. If you now look at the contents of the my\_first\_project directory on a file explorer, you will now see that RStudio has put some additional files into it. In particular, you should notice the file my\_first\_project.Rproj has been added. (A hidden directory .RProj.user, will also have been added, but this should only be visible if you have selected to view hidden files)
4. When you are in project mode, the top right icon which previously said ‘Project: None’ should now have the project name listed.

When you have multiple projects, you can use the project icon on the top right to open, close, and switch between projects. Whenever you switch to a new project, there will be a different R session, with different objects and contents, and a different base directory from which R will search for and write files.

## Directory Conventions within an R Project

Unless you are working with very large datasets, where it is sometimes not feasible to do so, it is a good idea to have everything a project needs within the base project directory you have created. It is also a good idea to have a similar layout within each project. It is partly a matter of personal preference and judgement, but an example of a within-project directory structure I use is as follows:



* **A ‘data’ directory**, containing the initial data I am working with. Whenever I read or write data files, I do so to and from this subdirectory. If a lot of data management is required, I might have two sub-directories within the data directory, ‘messy’ and ‘tidy’, with the ‘data/messy’ containing the data I initially am given or download, and ‘data/tidy’ containing derived data I have created having cleaned and tidied the data. I try to approach the creation of the tidy data as a separate activity to analysing the data, and contain the data tidying code in a separate script file to the analysis code. The analysis code will read in and work with the data in the ‘data/tidy’ directory rather than the ‘data/messy’ directory.
* **A ‘figures’ directory**, containing images and figures produced within the R project. I might have subdirectories within this if there are distinct groups of figures (for example, figures for different subgroups or outcomes).
* **A ‘scripts’ directory**, which contains various (hopefully) informatively named scripts which each perform distinct tasks. For example, I might have one script within here called ‘tidy\_messy\_data.R’ which works with the contents of data/messy and creates the contents of data/tidy, and another script called ‘analyse\_tidy\_data.R’ which takes the contents of data/tidy as its input and performs analyses. It is important to note that though these script files are in a subdirectory of the project, the base file directory they work from will still be the base directory of the project (e.g. ‘E:/projects/my\_first\_project’) rather than the location of the scripts subdirectory (‘E:/projects/my\_first\_project/scripts/’).
* **A ‘support’ directory**, containing information useful to me when working on a particular project, but in general not used by R. Examples of this include: Word files containing the paper I am writing up about the project or notes; PDFs and other documents about the source (‘messy’) data I am using; Excel spreadsheets containing selected and formatted outputs from R sessions; particular R sessions containing pertinent results saved as a text file; email correspondents about the project; and so on.
* (Sometimes) **a ‘tables’ directory** containing tables created within R.

## Script conventions

Within the base directory, I usually have a file ‘script.R’. This usually contains the selection of code I am actively working on at the moment. For example, the contents of ‘tidy\_messy\_data.R’ may have started within this file, and only once I have created, successfully run, and appropriately commented on the code do I copy the code which performs this task into a distinct and separate script. This is to help keep the tasks within the project modular.

I try to ensure that the ‘script.R’ file has the same structure:

1. (Optional but preferably): A series of comments describing the project, the current sub-project, the aims, and perhaps things like current and completed tasks. I might also include information like the dates and revision history here. This is mainly to make it easier to get an overview of the project, especially if has been a while since I last worked on it.
2. The command rm(list = ls()). This removes all objects from the current R session, ‘clears the desk’. This is useful if you want to avoid a script depending on the contents and results of a previous R session.
3. A series of lines beginning require() which load in the particular R packages I want to use. I usually sort the R packages according to the stage they fit within the data-to-knowledge value chain, beginning with packages which read and write data, then those which perform initial data tidying, followed by those which do more complex data management and tidying, then finally those required for data visualisation and analysis. In some cases, it is very important that libraries are loaded in a particular order. This is because two packages can contain functions with the same name, and only the most recently loaded function is immediately accessible to the coder.
4. Script for loading in the dataset I am working on. I currently tend to use read\_csv for this.
5. Script for data tidying and re-arranging prior to analysis and visualisation.
6. Script for performing analyses and visualisation of the data prepared above
7. Script for saving any derived data outputs.

Each of the above usually constitutes a distinct stage of the data management and analysis process, and it can be useful to signal in the script the breaks between each of these types of task. Within R studio you can do this by adding ‘sections’ to the code. You can do this by pressing ctrl + shift + R. This opens up a dialogue box where you type the section name. This then adds a large chunk of commented text to the script file, containing this name, which Rstudio recognises as distinct from normal comments. You can then jump to different sections within a single script by searching for and selecting different section names.

## Libraries

Libraries, also known as packages, are collections of functions that have been created by members of the R community. Only a small number of libraries are included with R when you first install it, and all other packages first have to be installed on your computer, then loaded into a particular R session, before you can use it.

Installing a library means downloading it from the internet, usually from CRAN, but sometimes from other locations. The standard way to install packages is to use the install.packages function. This requires one argument, which is the name of the package you want to install. It is important that this argument is a string, enclosed in the “ character on either side. Hopefully, all required packages will have been installed on your machine prior to this workshop, but if not the process for installing packages is usually straightforward.[[5]](#footnote-5)

Assuming the appropriate packages have been installed on your machine, you can load them into a current session using either the library or the require function. Both functions are very similar, the only difference being that the require function first checks whether the package has already been loaded before trying to load it. It is for this reason that I tend to use require, not least because sometimes I want to run an entire script at a time, and not have to check whether a particular library has already been loaded.

## Libraries to turn Base R into ‘Wickhamese R’

One of the main ways I hope this day of the course will be distinct from many introduction to R courses is that I will focus on using a series of packages that help to change the ‘grammar’ or ‘sentence construction’ of R code. These packages were all developed by Hadley Wickham, Chief Engineer at RStudio.[[6]](#footnote-6) The packages he has developed reflect an evolving but consistent, data science focused, design philosophy. They are focused not on pushing the theoretical limits of what R can do (such as coding a new Bayesian model only recently described in a scientific paper), but on making the existing capabilities of R much easier to access, and on building an syntactical interface onto Base R which makes it much more intuitive for humans to understand. Older packages, reflecting a slightly older design philosophy, include reshape2 and plyr. More recent packages include readr, tidyr and dplyr. One of the main design departures these more recent packages make is in the use of ‘piping’, a simple addition to the range of R operators, taken from programming languages like unix, and introduced to R only within the last couple of years through the maggritr package.

# An introduction to ‘piped R’

To start with, consider these two lines of code:

output <- function\_4(function\_3(function\_2(function\_1(input, arg\_a), arg\_c, arg\_d), arg\_b), arg\_f)

input %>% function\_1(arg\_a) %>% function\_2(arg\_c, arg\_d) %>% function\_3(arg\_b) %>% function\_4(arg\_f) -> output

The second line of code is the ‘piped’ version of the first line of code. The pipe symbol is %>%. Even with very limited knowledge of R or programming, it is easier to get a sense of what the piped version of the code is doing, as we can read the second example from left to right, starting with the word ‘input:

“The input is passed to function\_1 with added argument a, and then to function\_2 with arguments c and d, and then to function\_3 with argument b, and then to function\_4 with argument f. The output of this process is then passed to an object called output.”

In the first sentence above, the phrase ‘and then’ is equivalent to the %>% operator. The %>% operator links each of these stages in a clear sequence, running from left to right. The assignment operator -> at the end passes the outcome of all of these processes to the object output.

Graphically, the above process can be represented as follows:

a

c d

b

f

Like the English sentence, the command and sequence runs from left to right. By contrast, the Base R version of the same code has to be parsed ‘inside out’. Graphically, the Base R version of the code looks something like this:

f

B b

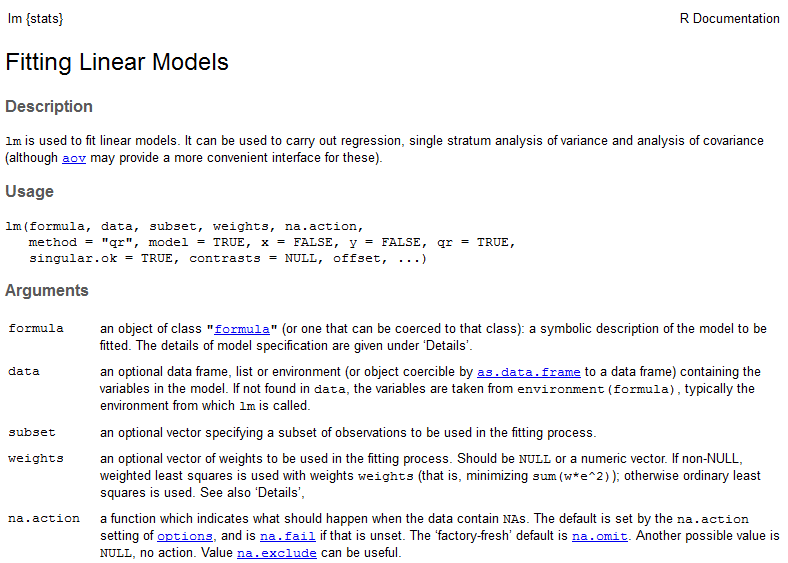
C c, d

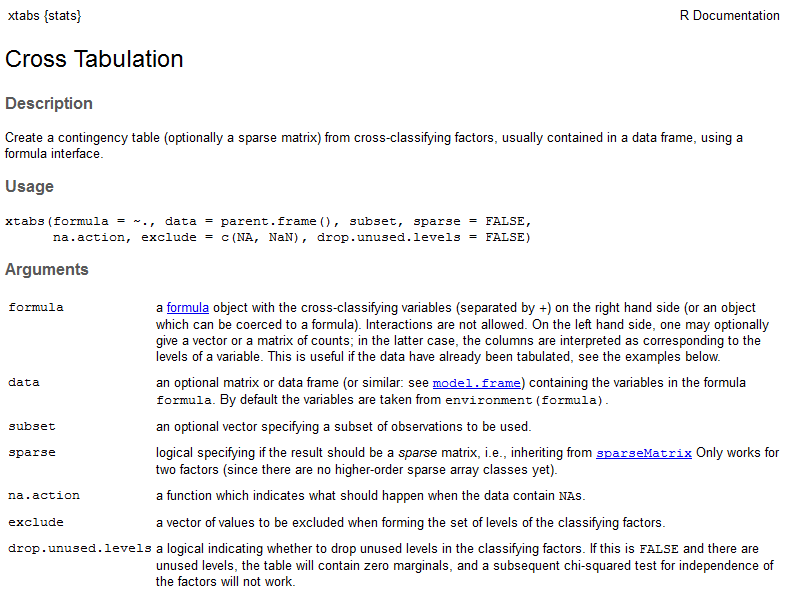
a

To work effectively in R, you will need to learn how to use Base R, and to know how to ‘translate’ between piped and non-piped versions of the same code. However, if you are starting off learning and using R for the first time, I would recommend you use code piping unless you find the Base R form preferable.

By default, the pipe operator passes the contents of the pipe to the first argument slot in the receiving function. All other arguments passed to the receiving function therefore pass to the second argument slot onwards. All functions within dplyr and tidyr are built around this behaviour, with the first argument slot being reserved for the data, and later slots for additional arguments. However many other functions are not built around this paradigm, and expect their main data inputs in second or subsequent argument slots. Two very important examples of this are the Base R functions lm, which generates linear models, and xtabs, which performs cross tabulations; in both cases the first argument slot is reserved for a formula and the second argument slot, named ‘data’, reserved for the input data.

By typing ? followed (without spaces) by the name of a function, you get help about that function, including a list of its argument slots. Here is what is displayed if you type ?lm and ?xtabs





Initially, help files in R can appear anything but helpful. However, for now what’s important to note, for both the lm and the xtabs function, is the text immediately below ‘Usage’. This shows the names and positions of the argument slots for these two functions. You can see that the first argument slot for both functions is called ‘formula’, and the second argument slot is called ‘data’. To use piping with these functions you will need to use the . (full stop) operator, which is a short placeholder symbol for indicating to the pipe operator where the contents of the pipe should feed into. For example, if the input dta is a dataframe (discussed later) which contains the variables (columns) height (in cm), weight (in kg), age (in years), obese (binary indicator indicating ‘1’ for obese and ‘0’ otherwise, and gender (categorical), then you could using piping to specify a cross-tab of gender against obesity status using

dta %>% xtabs( ~ obese + gender, data = .)

The comma separates the argument slots in the function xtabs, with the first argument being the formula used in the crosstab. The . operator is used to pass the dta object to the data slot of xtabs.

The base R equivalent way of specifying this would be

xtabs(~ obese + gender, data = dta)

You can see in this example that the base R version of the code is slightly shorter, and no harder to understand. The suggestion to use piping is ultimately just that, a suggestion, rather than a requirement.

Similarly, imagine you want to run a regression of BMI against age and gender. Here you would need to first derive bmi from two of the variables, height, and weight. Given BMI is defined as the weight in kg divided by the square of the height in metres, something like the following would be required

dta %>%

mutate(bmi = weight / (height / 100) ^ 2) %>%

lm(bmi ~ age + gender, data = .)

Note here firstly that the code now runs over more than a single line; this is fine as long as the pipe operator is at the end of a line that describes only an intermediate instruction rather than the final output. Note here also the use of the mutate command, which generates the bmi variable given the weight and height variables contained in the dta object. The mutate function is part of dplyr, which we will cover in much more depth soon.

There are a number of ways using base R of achieving the same outcome, for example:

dta$bmi <- dta$weight / (dta$height / 100) ^ 2

lm(bmi ~ age + gender, data = dta)

However, as before, the Base R expression is arguably harder to interpret. In this example the outcome is also slightly different, in that the first of the two lines permanently alters the contents of the dta object, adding a bmi variable to it. In the piped R example, the bmi variable is created ‘on the fly’, just for passing to the lm function, and the original contents of dta are unchanged.

1. Note: Statisticians aren’t comedians. [↑](#footnote-ref-1)
2. Technically, the outputs which appear in the Console after giving R an instruction are known as ‘outputs’, but these instructions are not the only possible effects of the instructions. Each time you work with R, you create and modify ‘R Sessions’. These R Sessions include ‘environments’, containing various objects which R works with and generates, and a log describing the series and order of instructions R has been given in that session. Giving instructions to R also changes these environments and logs, as well as creating outputs which appear in the Console. Additionally, instructions in R can operate graphical devices, producing things like graphs and figures, and create various types of files, such as text files, image files, and so on. Though the distinction is often not important, all these other types of changes that passing instructions to R can cause are known as ‘side effects’. When writing files or creating images, for example, the main purpose of the instruction is therefore to create a ‘side effect’ rather than an output from the console. (This is a bit like buying a chocolate bar from a vending machine: from the machine’s perspective the output resulting from inputting the correct change is to rotate a spiralled bar for a few seconds; but the user is interested in the ‘side effect’ of this motion, which is to push the bar off the edge of a ledge, causing it to fall and so making it accessible to the user and their mouth.) [↑](#footnote-ref-2)
3. If functions are like verbs, then arguments are like nouns and adverbs. For example, the verb ‘to run’ can be modified: ‘to run slowly’, ‘to run quickly’, ‘to run hastily’. In each case the adverb modifies the behaviour of the verb. Many arguments in R functions do the equivalent of this. Just as the verb ‘to run’ makes sense without these adverbs, so not all arguments need to be explicitly specified in an R function. Instead, some default behaviours and characteristics are assumed. For example, running is assumed to be faster than walking. ‘To run runningly’ is superfluous! However, just as many verbs do not make sense in a sentence unless they are linked to a noun which they act on in some way, some function arguments are needed, i.e. must be specified, in order for the function to work. In the case of ‘rm’, which removes specific objects from the R environment, you must at the very least specify which objects you want to be removed. Otherwise you get an error message. [↑](#footnote-ref-3)
4. The exception is if you contain an object name within the ` symbol (located at the top left of most keyboards). `this object`is considered a valid object name in R. There are some circumstances where you might want to use this symbol, but in general the use of spaces in object names is best avoided. [↑](#footnote-ref-4)
5. The main exception being when working with ‘managed desktops’, used by many organisations, which restrict the access rights of users unless you have administrator privileges, sometimes stopping packages from being installed. If this affects you, you will need to discuss and negotiate access rights with your computer/network administrator. Good luck! [↑](#footnote-ref-5)
6. <http://www.geirfreysson.com/2015/08/hadley-wickham-the-statistics-celebrity-and-r-programmer/> Accessed 5 Feb 2016 [↑](#footnote-ref-6)