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:

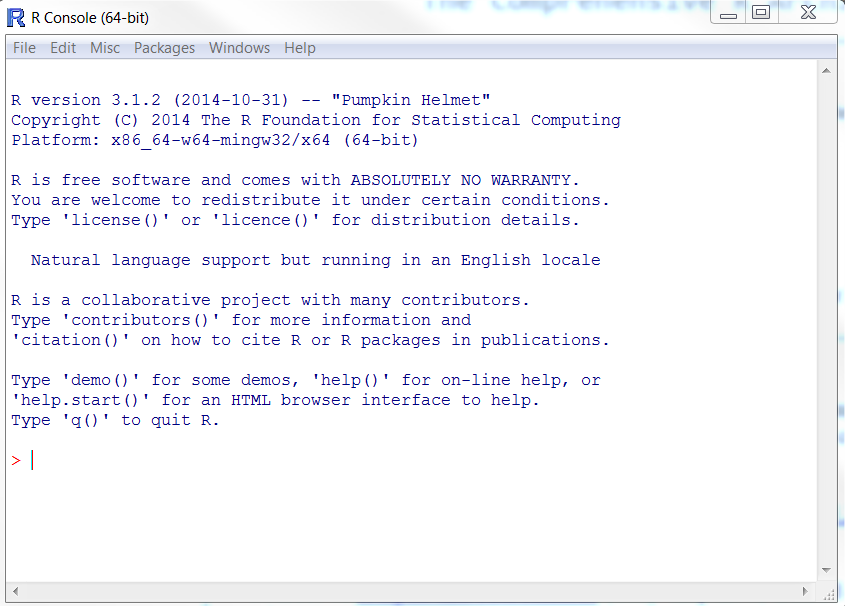


Figure Example of the basic R Graphical User Interface (GUI) in Windows

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:

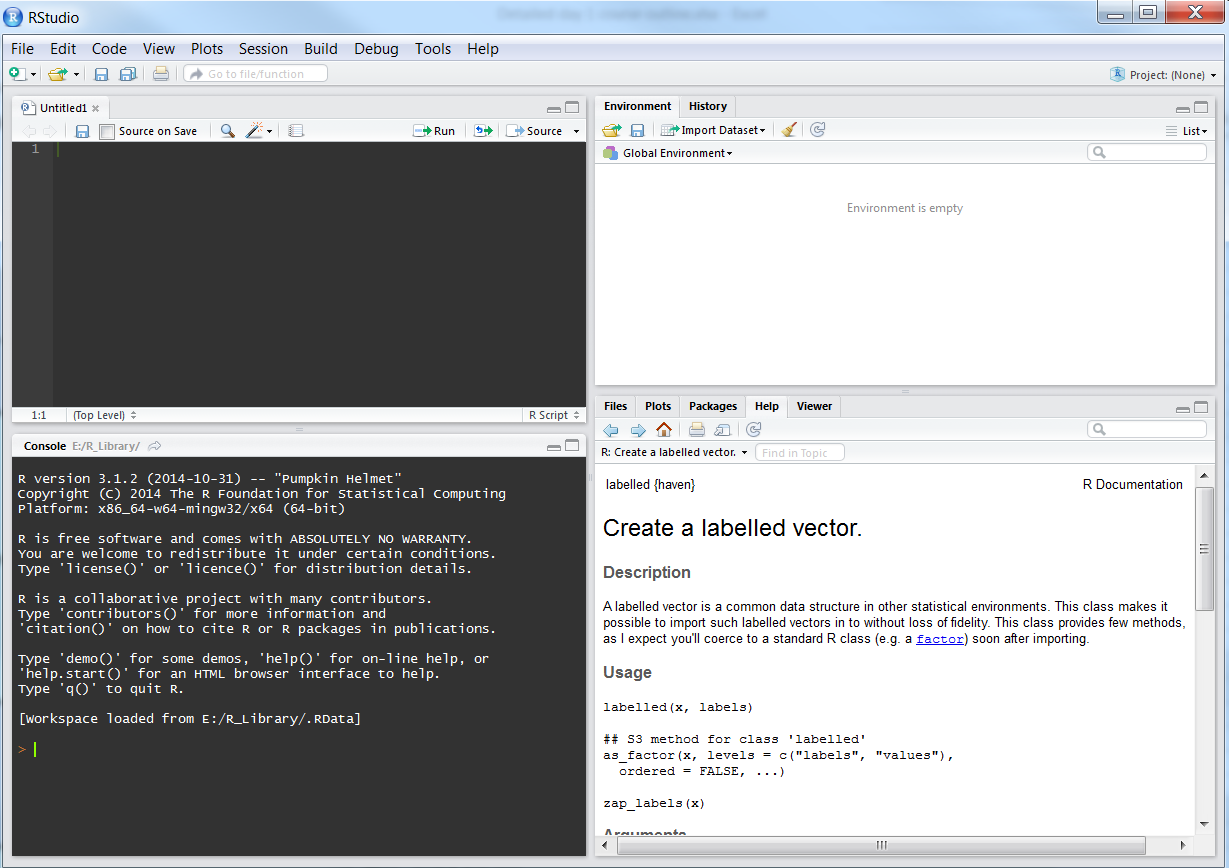


Figure Illustration of Rstudio and its four panes. Top left: Script pane; Bottom left: I/O pane; Bottom right: figures, help and documentation; Top right: history and environment

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

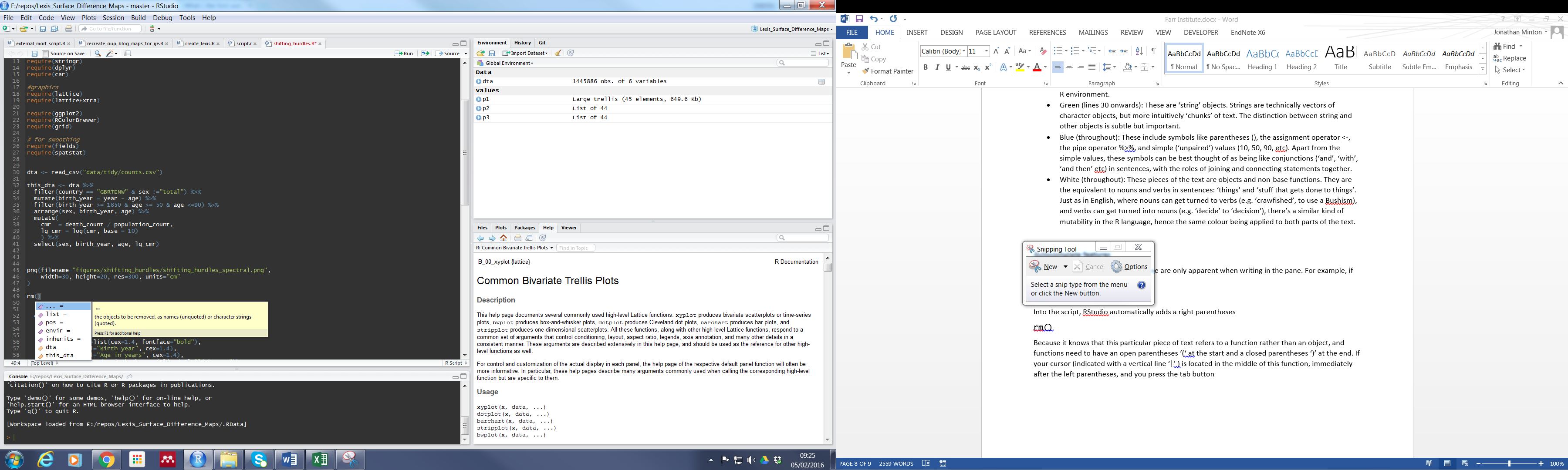
Most of the features in the script pane are only apparent when writing in the pane. For example, if you type

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



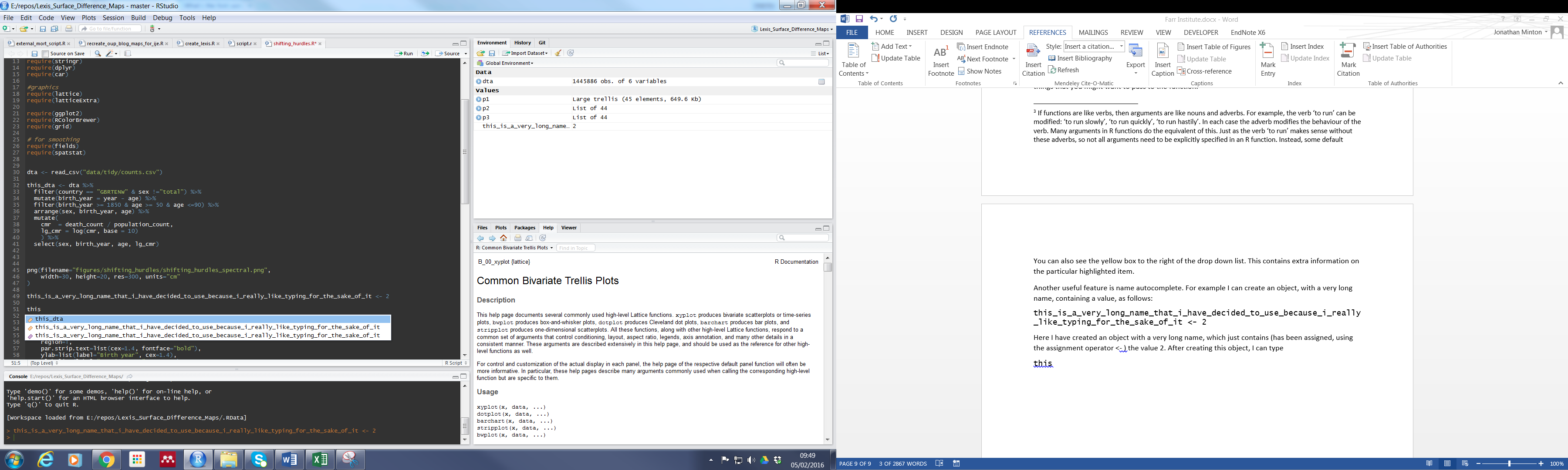
The purple items in the list are known as the **arguments** to the function, which modify the function’s behaviour.[[3]](#footnote-3) The yellow items in the list are objects found in the current R environment. These are things that you might want to pass to the function.

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:

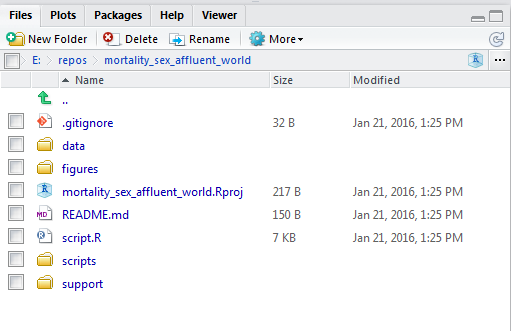
First, create a new directory on you completer, giving it the name of the project you want to use. It is best to use the coding conventions as within an R session, so no uppercase symbols and no spaces between words. To start with, create a directory called my\_first\_project, and remember its location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

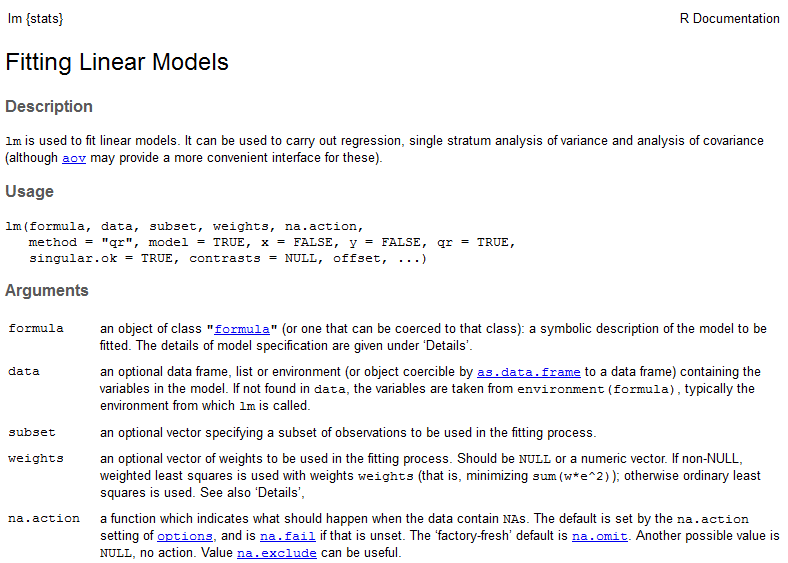
C c, d

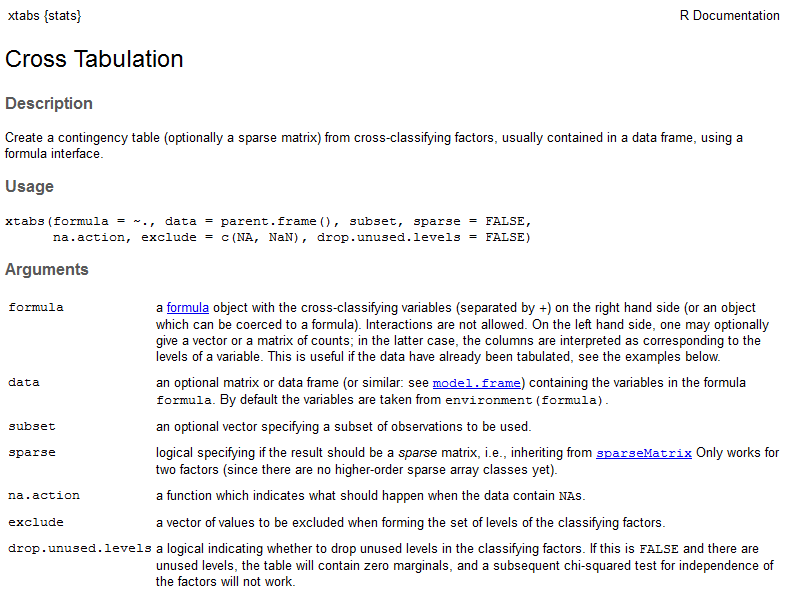
a

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.

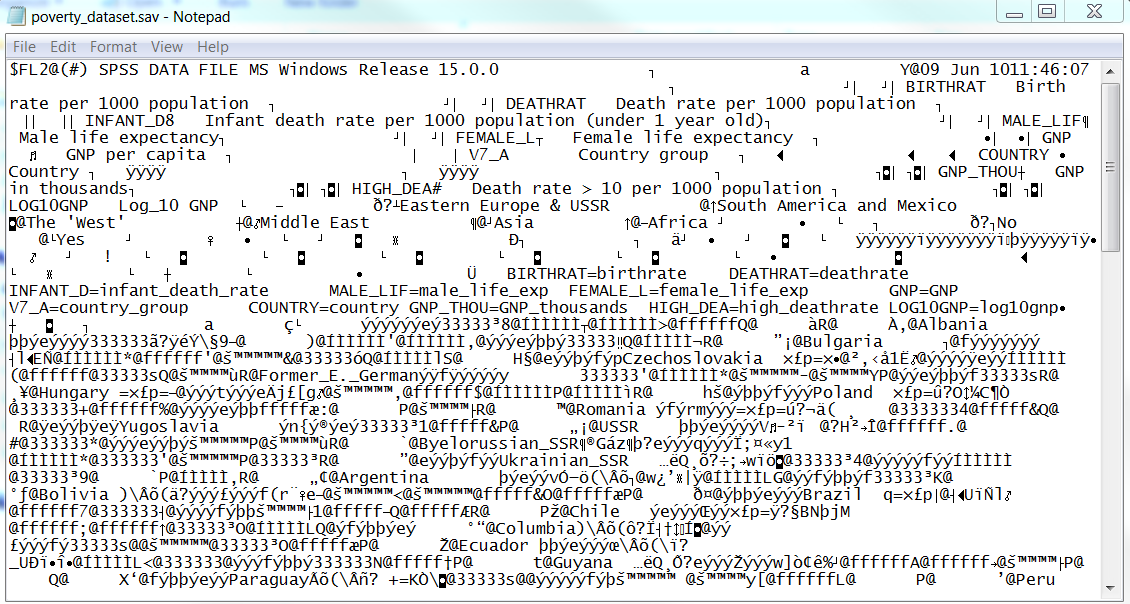
# Loading and saving data

## Introduction

The purpose of this session is to learn how to load and save data in a range of data types. There are a number of Base R functions for loading and saving files, but to be consistent with the ‘Wickhamese’ approach used elsewhere, this session will focus on using a series of newer packages created to work seamlessly alongside packages like tidyr and dplyr.

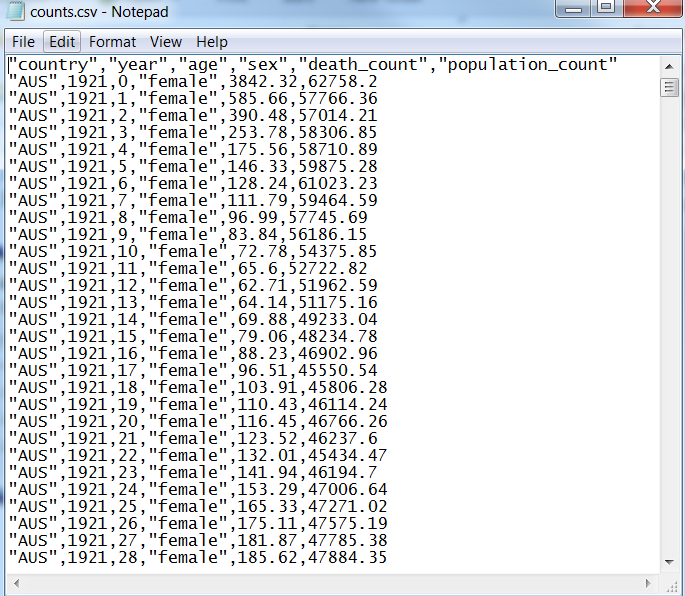
## File types

As everyone knows, computers store data as very long streams of ‘ones and zeros’. However computer scientists usually make a high level distinction between two types of file: text and binary. Although even text files are really binary, the term ‘text file’ is used to indicate files that, when opened, are human readable on most machines using a simple text editor. For example, I have a file named ‘poverty\_dataset.sav’, 9kb in size, for use in this training. The file extension indicates that this is a .sav file, a type commonly used by SPSS. If I select to open this file in Microsoft notepad, which has shipped virtually unchanged with every version of Windows ever, the file looks as follows:

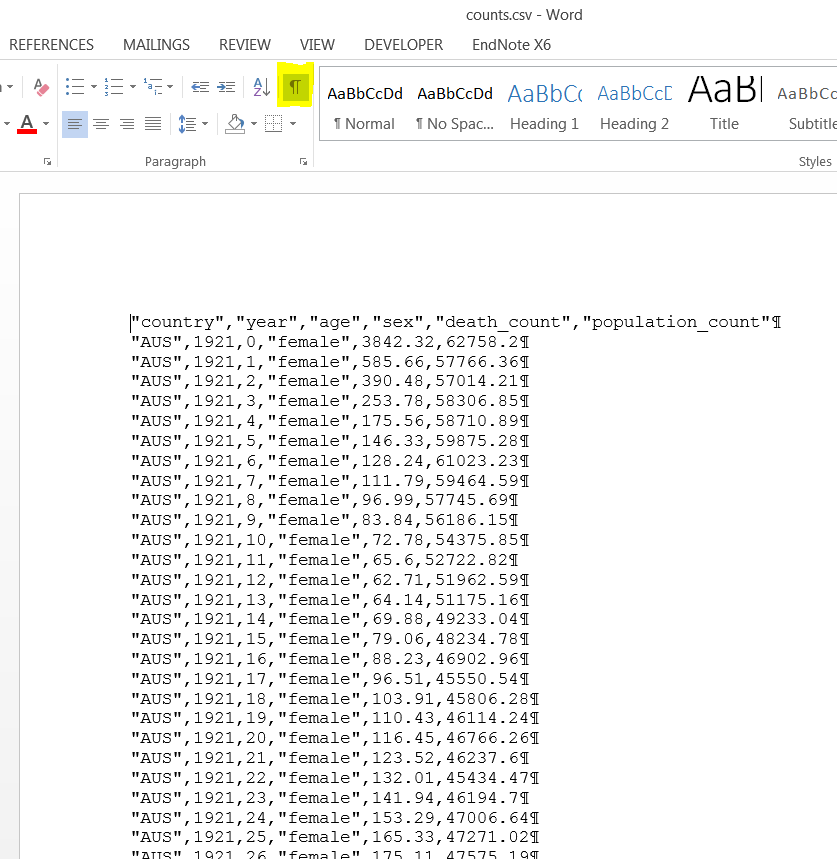


Some of the information here is human readable, in particular the first few lines of code which seems to begin with variable names followed by variable descriptions. However after this first section the data is very difficult to interpret, containing a number of strange characters and symbols that do not clearly correspond to particular values and cells in the data tables that are displayed in SPSS. The user is therefore dependent on SPSS, a proprietary and relatively expensive piece of software, or other software which can read this file format, in order to make sense of this dataset.

In contrast, here is what happens when I try to open up a file called ‘counts.csv’ in notepad:



The first line of this text file clearly defines the names of the variables, and all subsequent lines define one additional row of data. ‘csv’ means ‘comma separated values’, and it is clear within each line that there are six pieces of information, each separated by a comma. However, that’s not the only information this file contains. By opening the file up in Word, and selecting the option I’ve highlighted in the image below, you can see that the file contains an additional symbol, a hidden character known as a carriage return. This hidden character is at the end of each line, and tells the computer where the end of each line is located



There are therefore four types of information contained in this file:

* The names of the variables, contained on the first line
* The data itself, contained from the second line onwards
* Commas, which separate values within a line
* The new line character return symbol, which tells the computer where each line ends.

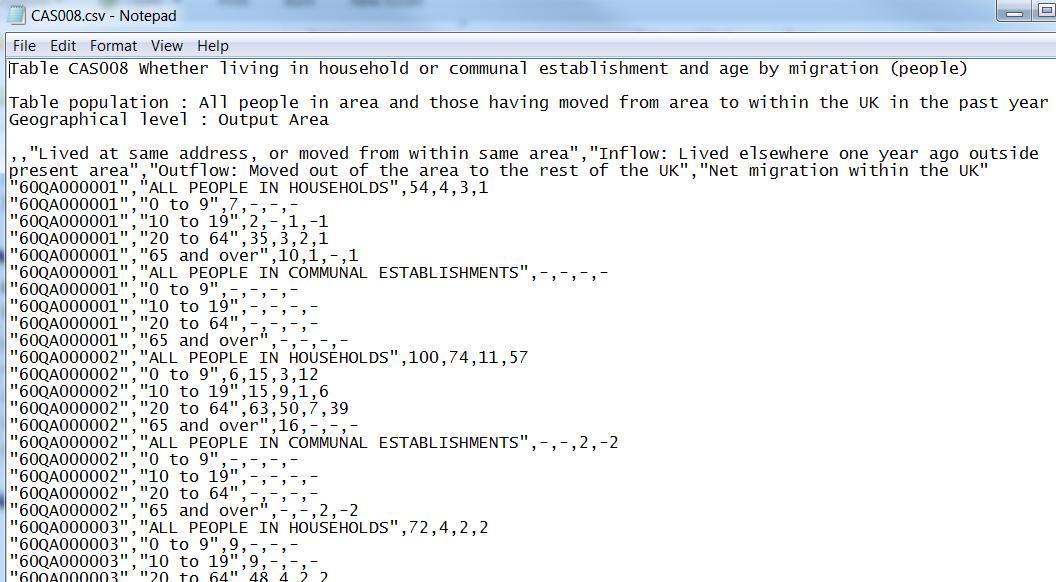
The first file is an example of a ‘binary’ file, and the second is an example of a ‘text’ file. There are advantages and disadvantages to both file types, summarised in the table below:

|  |  |  |
| --- | --- | --- |
| **File Type** | **Advantages** | **Disadvantages** |
| Binary | * Metadata are pre-encoded * File sizes can be smaller | * The file access depends on whether compatible software are available, and remain available: Files may be accessible now, but not in the future * If some data are corrupted, all data may become inaccessible. |
| Text | * Human readable and so easy to understand the nature of the data * Accessible on any machine running any software * An ideal format for archiving data so they remain accessible over time | * Variable meta-data need to be specified elsewhere * Total file size can be larger |

It has become increasingly important, for example when applying for large research grants, to have comprehensive data management plans, which describe how data are to be made accessible to future researchers with appropriate data security clearances. As a rule of thumb data and scripts stored as text files are much more effective for ensuring that your data remain accessible over time, and access is not in effect restricted only to those researchers and research institutions able to afford particular versions of potentially expensive proprietary software. A particular concern with using binary files is with backwards compatibility: future versions of SPSS or Stata may not always remain compatible with the .sav and .dta files which they currently save in, and so data stored in such formats may become inaccessible after a number of years. For these reasons the writing and reading of data as text files is to be encouraged as part of any forward thinking data management plan. Both for this reason, and because the data formats are inherently simpler, most of this section on file reading and writing will be focused on working with text files, though towards the end of the section the principles learned will be applied to reading some types of binary file as well.

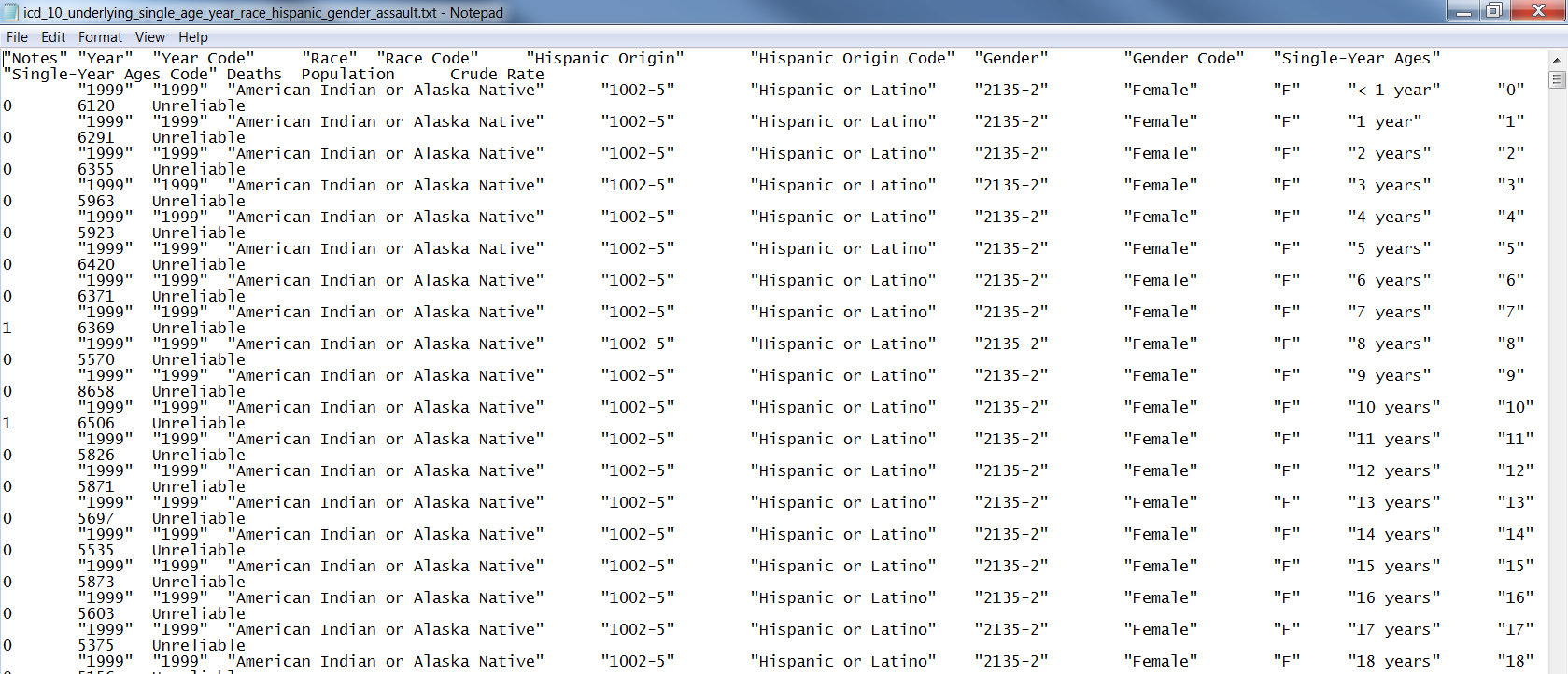
## Data and metadata

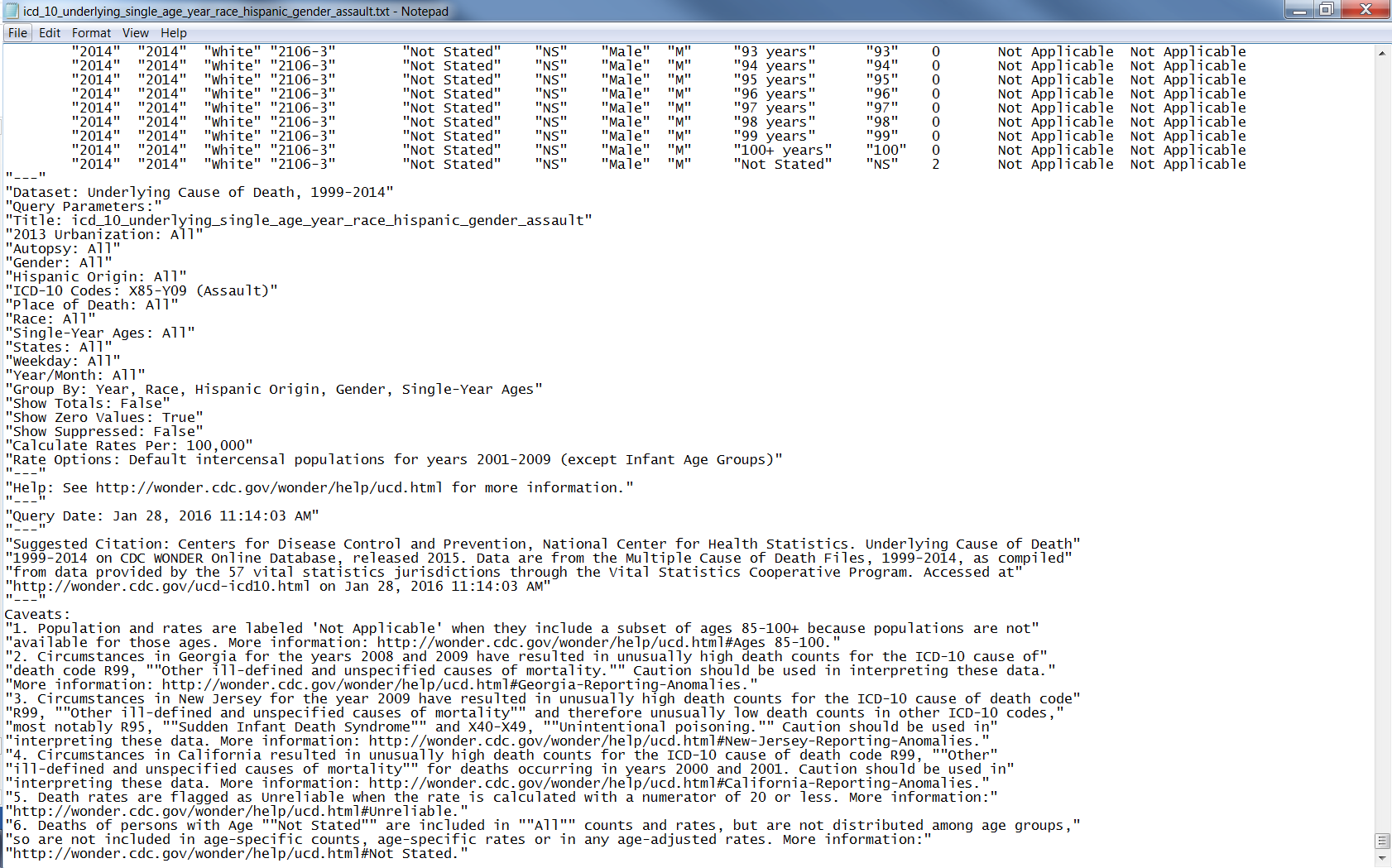
Within the table above a distinction was made between ‘data’ and ‘metadata’. Examples of metadata within the .sav dataset shown before include the description of the SPSS version in which the file was saved in (‘Windows Release 15.0.0’) and descriptions of particular variables (e.g. ‘BIRTHRAT Birth rate per 1000 population’[[7]](#footnote-7)). Many text files also contain a mixture of both data (‘rectangles’ of values) and metadata (descriptions and contextual information about the data) within the same file. For example, here is a file from the 2001 census for Scotland, containing data available at output area level:



You can see here that the first five lines of the file are metadata, providing a human-readable description of the particular census table which has been viewed. This is an example of ‘leading metadata’, and when reading this file into R or similar, these first few lines will have to be skipped, and the appropriate function arguments used to tell the machine where the ‘proper data’ begins.

Similarly, it is not unusual for there to be ‘trailing metadata’ at the end of a text file. The following two images show the first and last few lines of a file downloaded from the CDC Wonder database:





You can see that the last few lines of this file contain a range of metadata, including the ICD codes used, the query, information on how to cite the data source, and caveats about data quality.

In both cases of leading and trailing metadata, it is important to be able to know how to tell R how to distinguish between the metadata and data, and to work with and store these different pieces of information separately.

## Packages for reading and writing data

The table below shows a number of packages which are available for reading and writing different types of data. In each case, the package is listed first, followed by two colons,[[8]](#footnote-8) then the functions within that package which perform that particular read operation. Most of these packages have write operations that operate in similar ways.

|  |  |  |
| --- | --- | --- |
| **File type** | **Base R/Historic** | **Wickhamese** |
| **Text files** e.g. comma-separated values and tab delimited values | utils::read.csv  utils::read.table  utils::read.delim | readr::read\_csv  readr::read\_table |
| **Binary files**  *Statistical packages*  *Excel[[9]](#footnote-9)* | foreign::read.dta  foreign::read.spss  foreign::read.ssd  RODBC::odbcConnectExcel  gdata::read.xls  xlsReadWrite::read.xls  xlsx::read.xlsx | haven::read\_dta  haven::read\_spss  haven::read\_sas  readxl::read\_excel |
| **Databases** | foreign::read.dbf  ROBDC::odbcConnect | dplyr::src\_sqlite |

## Getting help about R packages

You can get a help file listing all functions contained in a function using

library(help = XXX)

where XXX refers to the package name (which has to be entered as a string, with “ at the start and end). For example, if you type

library(help = “readr”)

The following information about the readr package opens up in the Rstudio script window:



The particular functions available in this package are listed below the section marked ‘Index:’ In the case of this particular package, not all functions are listed. In particular, functions such as read\_csv and read\_tsv are not indexed. This is because these functions are both simple modifiers for the read\_delim function, filling in some details about default values for some of read\_delim’s arguments (i.e. the particular type of delimiter used). You can see this by looking for help on read\_csv:

?read\_csv



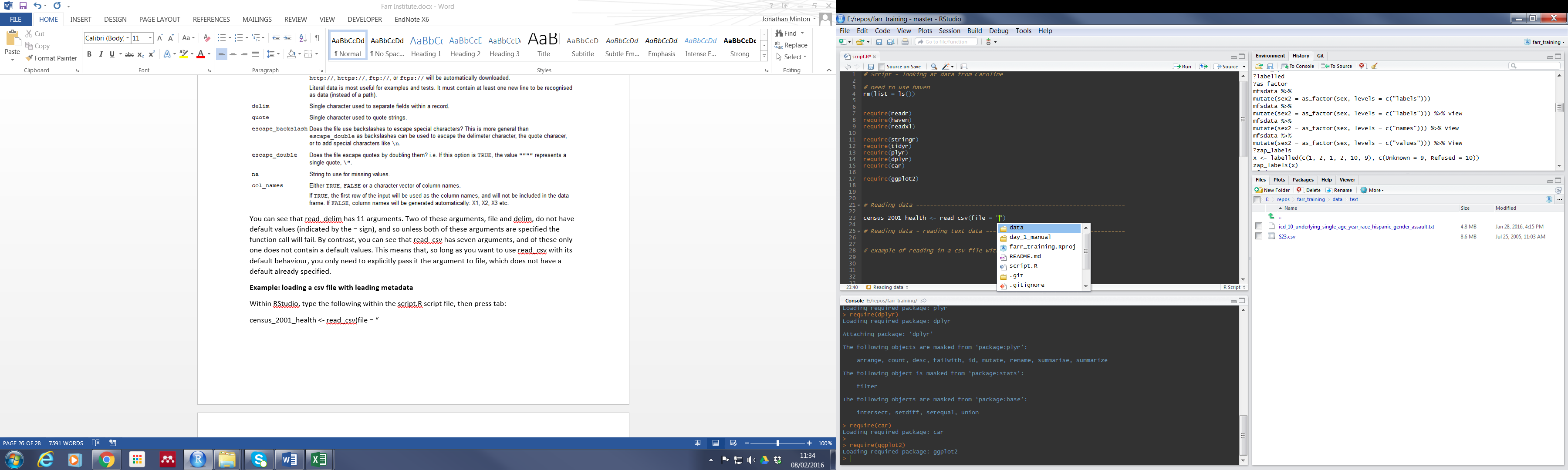
You can see that read\_delim has 11 arguments. Two of these arguments, file and delim, do not have default values (indicated by the = sign), and so unless both of these arguments are specified the function call will fail. By contrast, you can see that read\_csv has seven arguments, and of these only one does not contain a default values. This means that, so long as you want to use read\_csv with its default behaviour, you only need to explicitly pass it the argument to file, which does not have a default already specified.

## Example: loading a csv file with leading metadata

Within RStudio, type the following within the script.R script file, then press tab:

census\_2001\_health <- read\_csv(file = “

Pressing the tab button should open up a drop-down menu within the script file, allowing you to select the file, and directory location relative to the base directory, that you want to load.



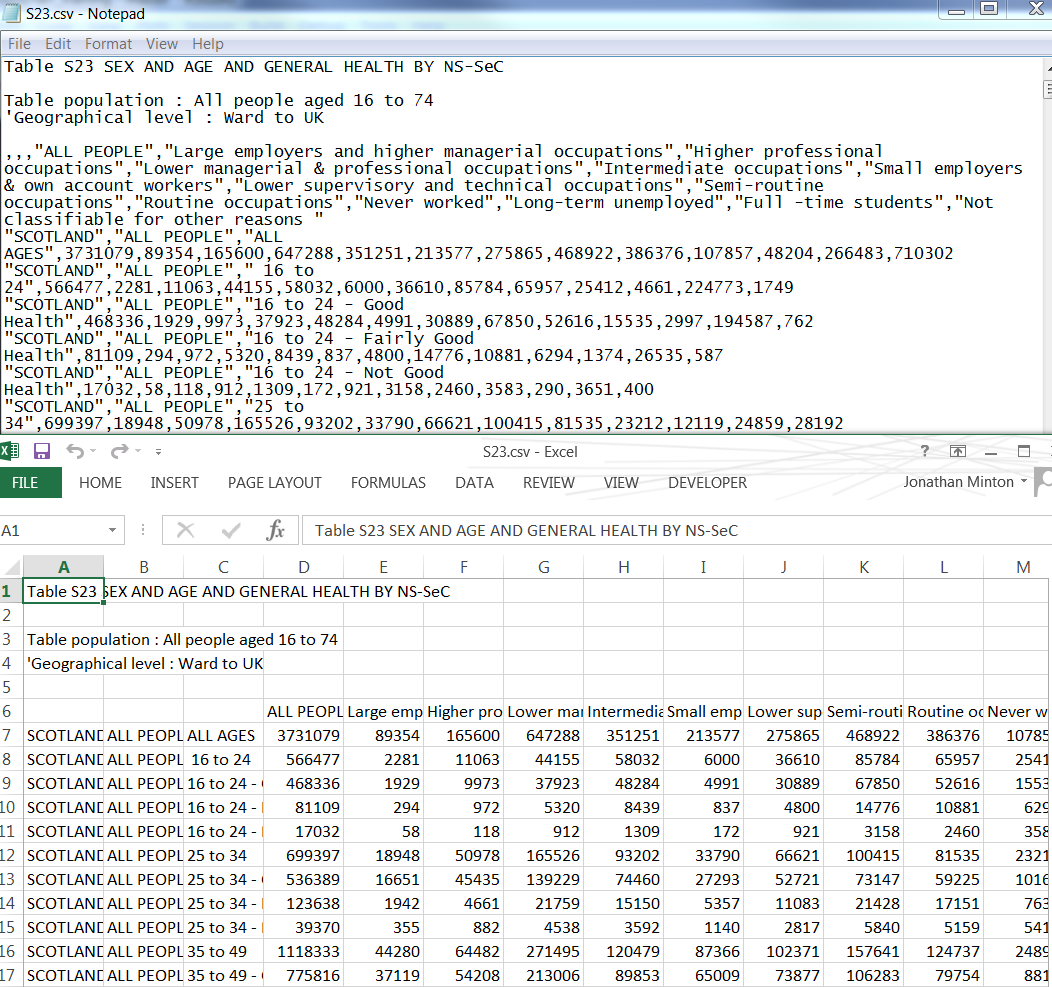
To start with, use the drop down options to specify that you want to load the following file: “data/text/S23.csv”. Then, make sure the function is closed.

census\_2001\_health <- read\_csv(file = "data/text/S23.csv")

You can select text in the script window for passing to the Console either by highlighting the selected text, then clicking the ‘Run’ button at the top right of the Script window. Or, with the cursor at the end of the line, you can press Ctrl + Return. Either way, you should get the following anticlimactic outcome:



This means that the default arguments in read\_csv were not right for the dataset you are trying to load in. To get past this error message you need to learn more about the structure of the data you are trying to load. You can do this by opening up the file outside RStudio. The file opens both in Notepad and Excel (because Excel recognises the file format), producing something like the following:

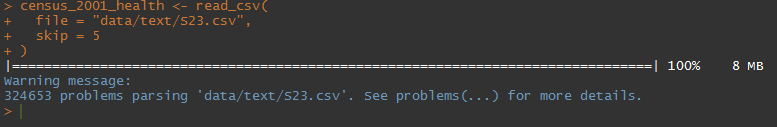


You can see from this that the first five lines of the file are metadata. They contain information about the name and population described by the data below. Although these metadata are useful to humans, they confuse R because it was expecting to find commas, indicating different columns in the data column, from the first row onwards. It saw no commons on this first row, and so thought the data would comprise a single column. It then identifies these commas later in the file, recognises an inconsistency, and generates an error.

If you look within the arguments for read\_csv and read\_delim, you will find the following:



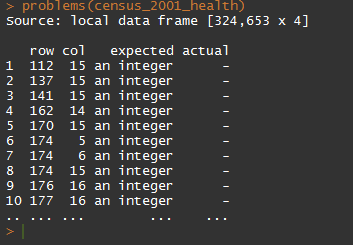
This argument allows you to tell read\_csv to skip a finite number of rows before starting to read in the data. You can see from both the notepad and excel representations of the file that the first five lines contain metadata, with the data itself, beginning with the column names, starting from the sixth line onwards. You therefore know you need to specify the skip argument too:



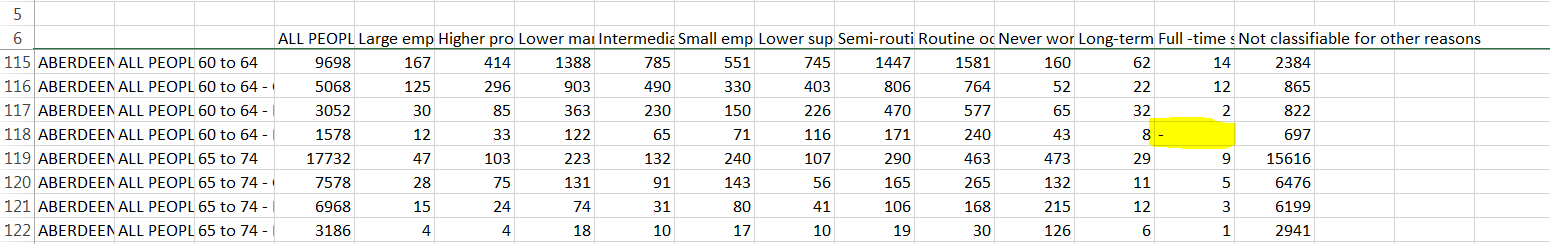
You have now been able to load the data into the object census\_2001\_health. However, read\_csv has now produced a warning message. For more information, it recommends applying the problems function to the object you have created. Either of the following equivalent expressions should work:

census\_2001\_health %>% problems

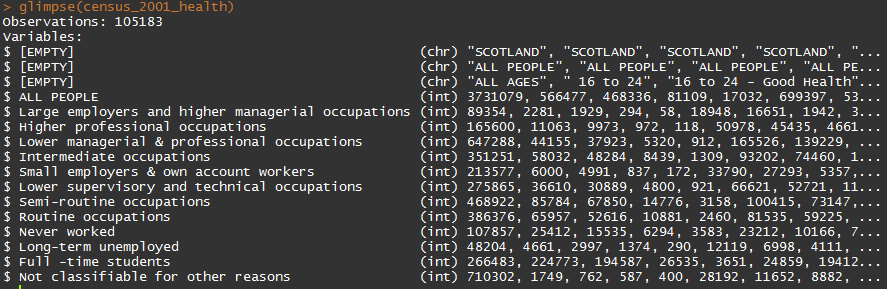
problems(census\_2001\_health)



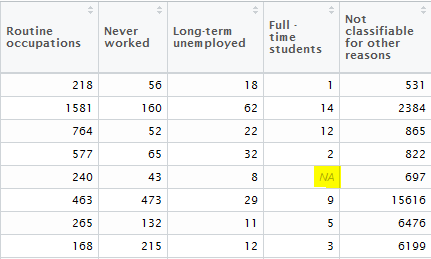
What this says is that the type of data readr expected some cells to contain was not the same as the type of data those cells actually contained. The first problem it identifies is on the 112th row. Given we have skipped the first five rows, and the sixth row contains the column names, this implies the issue is one line 118 of the data file itself. Looking quickly at the .csv file in Excel, we can see the following:



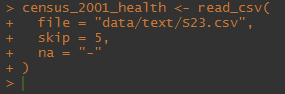
read\_csv has recognised that most of the cells in each of cells from the 4th column onwards are numbers; in particular, integers (i.e. no decimal places). The character – is obviously not a number, and so does not fit within the expected data type. You can learn more about the data types R assumed each input to contain using the glimpse function, or simply typing the object name, which displays the first few rows of data.



The names in brackets describe the data types that read\_csv has assumed each column of data to contain. The first three columns have been classified as ‘chr’, meaning ‘character’. The latter columns are all classified as ‘int’, meaning ‘integer’. The read\_csv function has decided on these character types for each of the columns without you having to specify them; it has made an educated guess, by looking at the contents of the first few rows of each of the columns and trying to select accordingly. However, although most of the cell contents from the 4th column onwards are integer values, ‘-‘ is not. You can see how R has interpreted these ‘integer’ cells containing the ‘-‘ symbol by using the View function, and scrolling down to row 112.



You can see here that, rather than reproducing the ‘-‘ symbol faithfully in the data, it has instead converted it to the value NA. NA means ‘not applicable’, and so indicates missing data. If read\_csv’s interpretation of this character as representing missing values were correct, then it is still not good practice to rely on the function’s default behaviour, and instead you should formally specify that this character has that particular meaning. You can do this using the ‘na’ argument in read\_csv and related functions. To do this, add the na argument to the earlier function call:

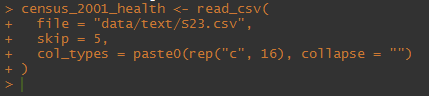


Note that this time a warning message was not generated. If you now run the problems function on the object, you get the following:



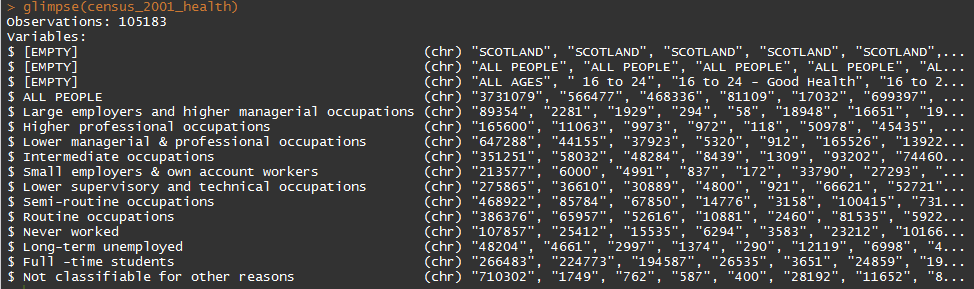
Whereas before there were more than 320 000 ‘problems’ identified when loading the data, now there are none, as all of the ‘problems’ were of the same type (not knowing how to interpret ‘-‘ as an integer value).

However, in this particular dataset, the 2001 census, the ‘-‘ character does not represent missing values, but instead the number zero. Telling R that this character needs to be converted to the number zero is more difficult than simply telling it that the ‘-‘ character represents missing values, and involves some additional data tidying processes which are described in the next section. In the context of this section, focused on loading in files, dealing with this sort of data requires that a different argument, col\_types, be used instead of na:



The col\_types argument accepts a string of characters, which must be as long as the number of columns to be read in. Each of these characters is shorthand for the data type of a particular column. For example, if there are just four columns in the dataset, and the first two are characters and the latter two are integers, then you could write col\_types = “ccii” (character, character, integer, integer). Instead, there are 16 columns, and so the string needs to be 16 characters in length. My approach in the above is to tell read\_csv that all of the columns should be read in as characters rather than numbers, because character columns can cope with the ‘-‘ symbol whereas integer columns cannot; this in effect overrides the intelligent default behaviour and automatic type identification used in the function, and is for finer grained control of the data. The string specifying the col\_types argument must therefore be a string of 16 characters, each containing the letter ‘c’, i.e. “cccccccccccccccc”. If the string is of the wrong length, i.e. not equal to the number of columns, then the function will fail.[[10]](#footnote-10)

If you now look at the contents and structure of the data object using glimpse, the output is slightly different:



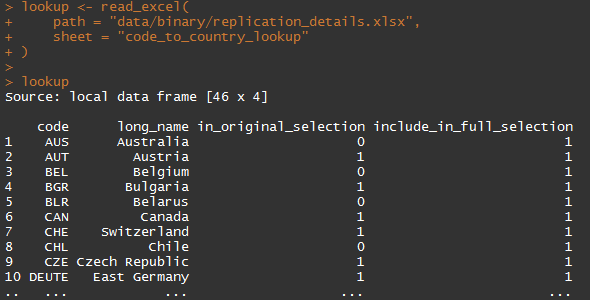
All variables now have (chr) listed as their type, meaning they are characters rather than numbers. This is further indicated by the fact each of the numbers is now encapsulated within the “ symbol. Although the contents of these cells look like numbers to us, for R they are now simply characters, and cannot be interpreted as numeric values without further type conversion. We will now leave this example within this section for now, before returning to it again in the next section on string cleaning.

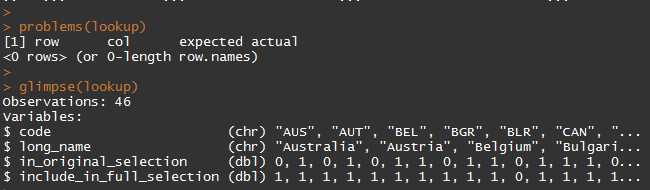
## Reading Binary Files

### Example 1: Reading Excel Files

Within the directory data/binary/ there is an Excel workbook called ‘replication\_details.xlsx’. This contains a number of distinct worksheets, each with a different name. Most of the contents of the worksheets is not set up to be easy for R to read or write to, but the contents of the sheet code\_to\_country\_lookup should be relatively easy for R to read using the readxl package. The main function within this package is imaginatively titled read\_excel. This function deliberately has a very similar set of arguments to it as readr, the main exceptions being that it ‘path’ instead of ‘file’ as its first argument, and a ‘sheet’ argument for specifying either the name or location of the worksheet within the workbook you have specified using ‘path’.

If you look within the replication\_details.xlsx file you will see that the code\_to\_country\_lookup worksheet contains no leading or trailing lines for metadata, it is simply a rectangular table of values.





The ‘glimpse’ function indicates that the variables in\_original\_selection and include\_in\_full\_selection are both encoded as ‘double’ type variables. This data type is restrictive than integers, used in the previous example, but also uses more data to represent the same amount of information. We can also see that in this case the variables are binary indicators, indicating whether a row should be included or excluded. In the next section we will look further at type conversion, an example of which would be converting these variables from binary to logical character type.

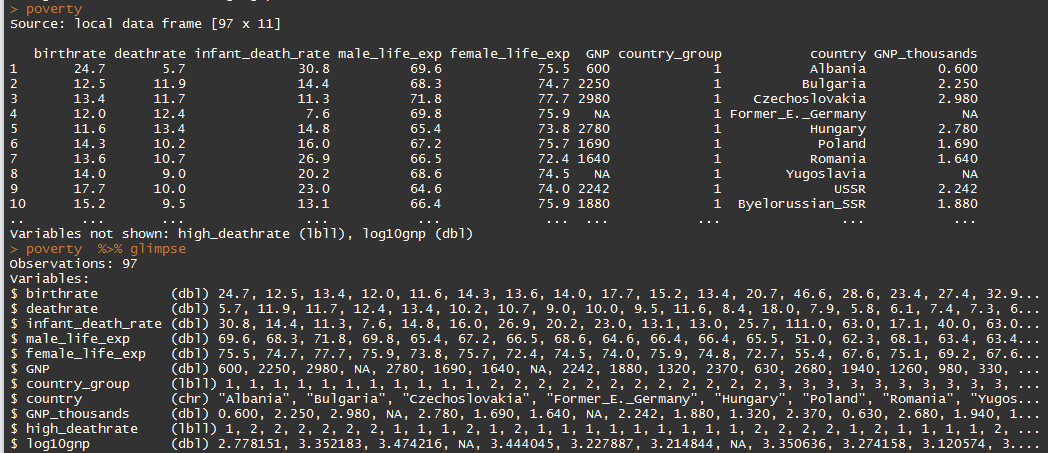
### Example 2: Reading SPSS data

SPSS saves data in two main file formats, .sav and .por. Each of these file formats differs slightly between SPSS versions. The contents of either of these file formats tend to be more complex than for a simple text file, containing various forms of embedded metadata providing information such descriptions of the variables, and value-label lookup information for particular variables. Although all of this extra information is designed to make datasets easier to work with within SPSS, they can create additional challenges when reading the data into R, because both SPSS and R work with data in different ways.

To illustrate some of the challenges involved in working with SPSS data within R, use the read\_spss function within the haven package to load the file ‘poverty\_dataset.sav’ into R

poverty <- read\_spss(path = "data/binary/poverty\_dataset.sav")

This seems to have been very straightforward. However, looking at the first few rows of the data, and glimpsing the data, show that the data generated is slightly unusual:



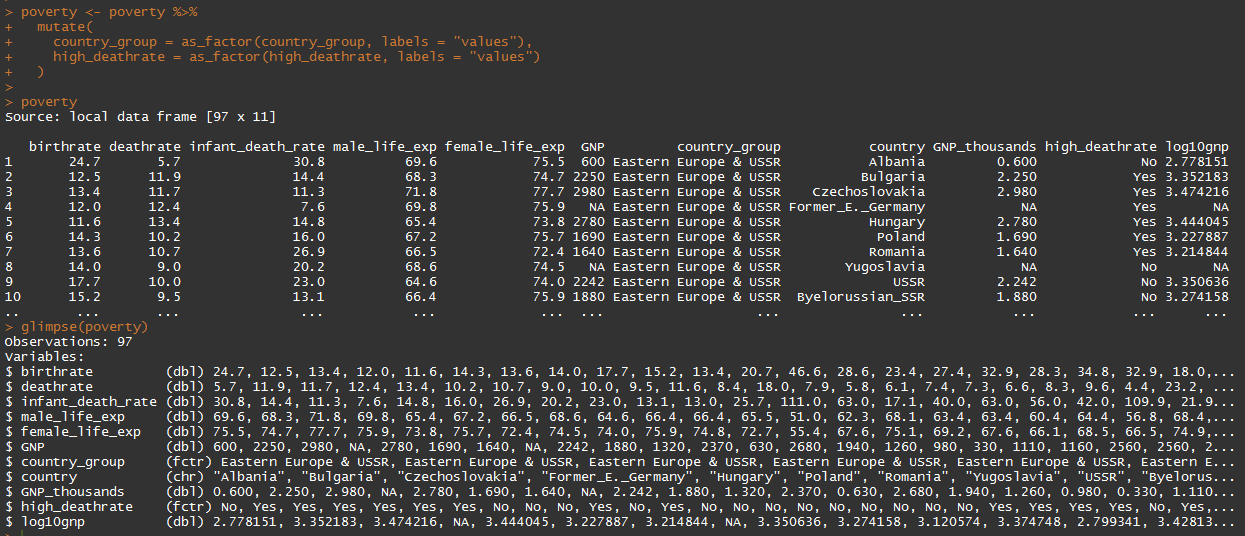
Two variables, country\_group and high\_deathrate, have a variable type that has not been encountered before, called ‘lbll’. This is short for ‘label’, and is not a standard R variable type, meaning that without further processing many functions expecting standard R variables will not work with it.

Labelled variables within SPSS comprise at least two components:

* **Data**: A series of numeric values (e.g. 1, 1, 1, 2, 1, 3, etc)
* **Metadata**: A lookup table, which converts from numeric values to more informative labels (e.g. 1 = ‘underweight’; 2 = ‘normal’; 3 = ‘overweight but not obese’; 4 = ‘obese’)

The reason for this separation between numeric data and lookup tables is that it can result in much smaller file sizes. Instead of having to repeatedly write out the same label name thousands of times in a file, a simple and much smaller number, like ‘1’, can be used in its place, and the associated label can be re-attached to this number for display within SPSS.

At the moment, R is only displaying the numeric values for these two labelled variables, rather than the associated labels. However, these lookup tables were contained in the .sav file, and have been loaded into R using the read\_spss function. R has its own native variable type, called ‘factor’, which works in much the same way as ‘label’ variables work within SPSS. Labels can usually, but not always, be converted into factors using the as\_factor function. In the case of this dataset, the following will work:



You can see here that the value labels for country\_group and high\_deathrate are now displayed. The ‘glimpse’ function also shows that both variables are now of type ‘fctr’ (factor) rather than the lbll as indicated previously.

|  |
| --- |
| **Note:** A bug with the current version of as\_factor seems to be that, if the table of value-label lookups for a label begin with the value ‘0’ rather than the value ‘1’, as\_factor will not work properly. Please ensure that the lowest value used for a label is ‘1’ before attempting to read this data into R. |

## Section summary

This section has looked at a number of related packages for loading in data from a variety of formats. I have argued that, as a rule of thumb, text data files should be used in preference to ‘binary’ files as they have benefits both of simplicity and of accessibility across platforms and over time. However we have also looked at examples of how to load in data from Excel workbooks and worksheets, using the readxl package, and of loading data in SPSS .sav format, using the haven packages. As summarised earlier, there are a number of different packages available for reading and writing data in a variety of formats, which ostensibly perform similar tasks. The packages haven, readxl, and readr have been used because they work in similar ways, according to a similar logic, and fit with the tidyr and dplyr packages which the next section will discuss. However there may be tasks that older packages like foreign can accomplish more effectively, not least because they are much older packages and have therefore undergone more testing. A strength but also a challenge when first learning R is that there are almost always many different ways of achieving the same outcome; it is usually useful to learn how to do the same thing in at least two different ways, both to avoid becoming too dependent on a single package and approach, and to learn more about the R language.

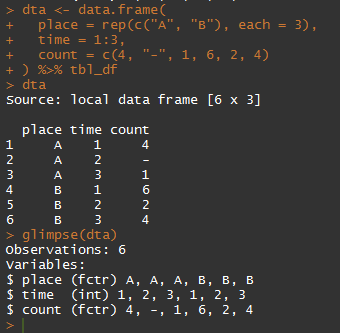
# Initial data tidying

## Introduction

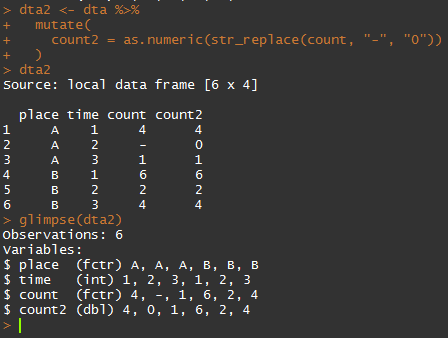
In the previous section we have introduced a number of packages and approaches for loading different types of data into R. In the process of doing this we have encountered a number of different data types, as indicated by abbreviations such as (int) and (dbl) used to define particular variables by the glimpse function. We have also identified certain issues whereby the cell contents of particular variables contain symbols which R cannot easily process. In particular, we have a case from a 2001 census table in which the ‘-‘ is used to represent the value 0 rather than missing data. We left this particular example by forcing read\_csv to read all variables in as characters, which perform no type conversion, in order to have cell contents and symbols which can be more appropriately processed at a later stage. This section will provide an introduction to stringr, a character string processing package which I use to perform this type of character string tidying and processing.

## Example 1: Converting symbols to values

To start with, let’s create a small fictitious dataset which shares with the census data table the problem of 0 being represented by the ‘-‘ symbol:



The ‘-‘ character needs to be converted to the value ‘0’ in the count variable, and character type for count then needs to be converted, correctly,[[11]](#footnote-11) to numeric. I think I have an approach for performing this task, but because I do not want to permanently alter the contents of the dataframe until I am sure the code works, I first create and explore a second dataset, called dta2:



The ‘mutate’ command, described in more detail later, adds a new column, count2, to the end of the dataframe. This new column is created as a result of the following code:

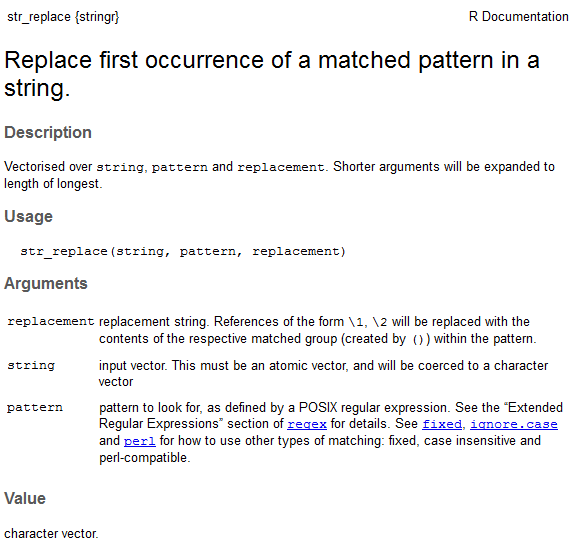
as.numeric(str\_replace(count, “-“, “0”))

This is an example of a ‘function within a function’, common to base R, which I have argued against in favour of piping previously. However piping is already used in this script, to feed the dta object to the mutate function, and having piping within piping can be problematic. To help think through what the code within the mutate function is doing, however, let’s try to write out the above code in a ‘piped’ format, converting from ‘inside-to-outside’ to ‘left-to-right’:

count %>% str\_replace(. , “-“, “0”) %>% as.numeric

We can now see that ‘count’ is the initial input, which is passed to the function str\_replace as its first argument (represented by the . symbol). The second and third arguments to str\_replace are set to “-“ and “0” respectively. Finally, the output from this function is passed to as.numeric, a function which performs type conversion from other data types to numeric data.

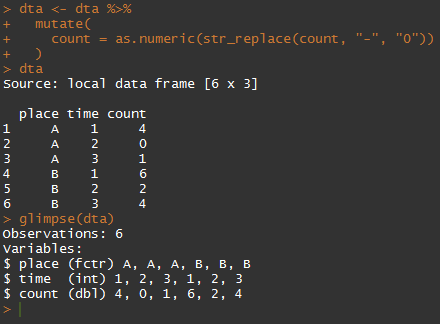
Let’s now look at the str\_replace function help:



This shows that the second argument is labelled ‘pattern’, and the third argument is labelled ‘replacement’. What the function does is identify instances of the ‘pattern’ in the string, and replaces them with the contents of the ‘replacement’ argument. So, in our example, it replaces the symbol ‘-‘ with the symbol ‘0’.

The ‘Value’ section of the help file describes the type of variable that the function returns. We see that this is a character vector, but we know that, once the ‘-‘ has been replaced by a number, all the characters in the vector should be numeric. We would like to be able to perform mathematical operations on these numbers, for example to compare two values to work out which is greater, and to do this we need to convert this output from a character vector to a numeric vector. This is what the final part of the code, ‘as.numeric’, does.

Having confirmed that our code works as intended, we have no need for the original ‘count’ variable, as it contains the data in the wrong format. Instead, we can overwrite it as follows:

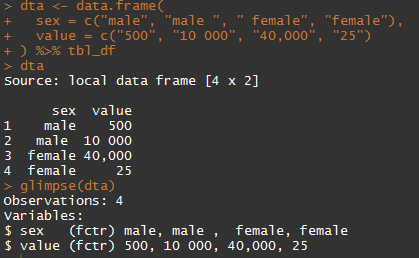


These permanent alternations to the data should only be performed once you are sure the operations work as you expect them to, hence the creation of the dta2 object earlier. These ‘scratchpad’ objects can be removed using the rm function.

|  |
| --- |
| **User Defined Functions**  A very important feature of R is the ability to create user-defined functions. Here is an example which performs the above task:    Confusingly, user-defined functions are created using a function called ‘function’. This function takes as its inputs zero or more input arguments, and returns one object as an output (although the output object may itself contain a number of separate objects within). What the code above says is:   1. Create a function which takes one argument, called input. 2. Take this input argument, and first perform str\_replace on it, followed by as.numeric. Save the result of these two operations into an object called output. 3. Return the output object 4. Call the function ‘change\_dash\_to\_zero’   The piece of code within the two curly brackets { and } is known as the function ‘body’. Within the function body you can refer to the objects you have accepted and defined as function arguments (‘input’ in this case), as well as a limited number of other objects created elsewhere. Note that by using a user defined function, you are now able to write out the above code using piping rather than with functions-within-functions as before. To call the function within mutate, we can write:    The result of this operation is exactly as before. The ‘count’ object is passed as the main argument to the change\_dash\_to\_zero function, as so adopts the name ‘input’ within this function. For relatively simple operations like the above, there are not great benefits to using a user-defined function. But for much larger and more complex operations the benefits of having such functions can be very large both in terms of the efficiency and readability of code. |

## Example 2: Removing commas and removing whitespace

Consider the following dataframe, again called dta for lack of imagination:

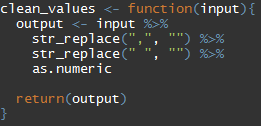


Although we recognise that two sexes are in the ‘sex’ variable, because there is a space after the second ‘male’, and a space before the first ‘female’, R thinks these are distinct labels. If we later try to produce statistics or other summaries by sex, therefore, it will present summary statistics for ‘male ‘ as well as ‘male’ and ‘ female’ as well as ‘female’, thinking there are four sex groups in the data rather than two.

A similar issue exists within the ‘value’ vector. In the second row, a space is used to separate values above 1000 from those below 1000; and in the third row, the comma symbol is used in the same way.[[12]](#footnote-12) This means that R does not recognise the values as numeric, and would not know, for example, that the third row has a higher value than the second row.

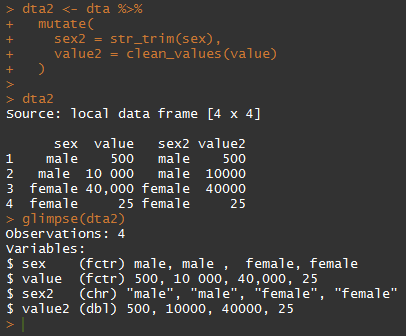
The str\_replace function used previously can also be used to help with the value column. In principle, this function, or the related str\_replace\_all function, can be used to remove trailing and leading whitespace too. But for now the convenience function str\_trim will be used instead.

Let’s use a user-defined function for the value column cleaning task:

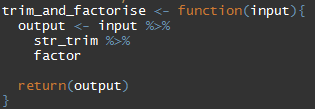


The second argument, for both calls to str\_replace, is “”. The “ symbols define the start and end of the replacement string, and so the contents of this string is … nothing. Replacing a one character string with a zero character strings is how str\_replace can be used to remove characters.

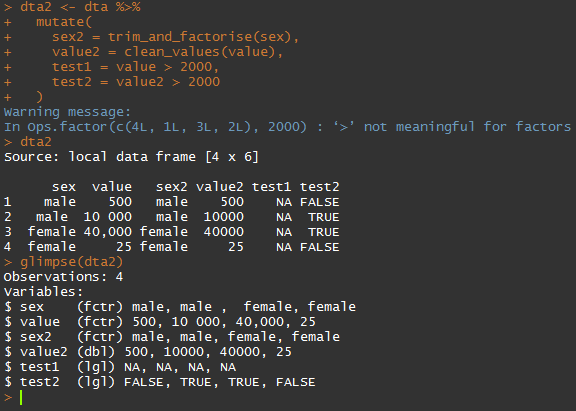
We can test the code as follows:



Compared with its input, ‘sex’, the output ‘sex2’ has trimmed the whitespace successfully. And compared with its input, ‘value’, the output ‘value2’ has successfully removed commas and spaces, and converted the variable type to numeric (as indicated by the dbl label within glimpse). The function str\_trim has also, however, converted the sex input from factor (fctr) to character (chr), which we may not have wanted. As the output of str\_trim is a character string, we would then have to explicitly convert it back to a factor using the factor function. Let’s create a new user defined function which does this:



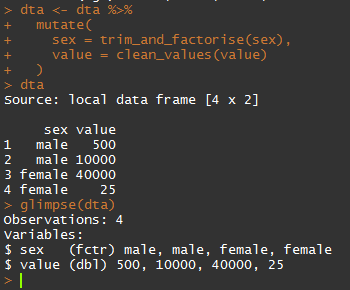
Using this function, and the clean\_values function created earlier, we can now create another version of dta2 to test our operations:



In this example I have also created two new variables, test1 and test2. The first of these, test1, is the result of comparing each of the values in ‘value’ with the number 2000; it returns TRUE if the corresponding value is greater than 2000, and FALSE otherwise. The second of these does the same to the tidied ‘value2’ variable. We can see that the code has generated a warning message, saying that this comparison, involving the > (greater than) operator, is not meaningful for factors. Looking at the result of test1, we can see that all comparisons have returned the value NA (not applicable), effectively meaning that the comparison has failed. By contrast, test2 has returned ‘TRUE’ where value2 is greater than 2000, and ‘FALSE’ otherwise, demonstrating that the contents of value2 are of a variable format where numeric comparisons have been made.

More subtly, we can also see that sex2 is now a factor rather than a character variable, and so the trim\_and\_factorise function appears to have worked correctly.

Having performed these tests, and seen that the new variables pass them, we can now overwrite the sex and value variables in dta.



|  |
| --- |
| **Going further: the terrible power of regex**  Although the tasks described above are common in data tidying, in order to make the most of stringr functions a knowledge of something called ‘regex’ will be needed. Regex, short for ‘regular expression’, is a standardised way of representing particular patterns of symbols within character strings. Regular expressions are the inputs to the ‘pattern’ argument for stringr functions, and although some of these regex patterns, such as the “-“ string used earlier, are very simple, others can be very complex and initially appear baffling. Further details about regex are available from the following links:  <https://en.wikipedia.org/wiki/Regular_expression>  <http://www.regular-expressions.info/>  Many regex ‘testers’ exist online, and these can be a good way to learn more:  <http://regexr.com/> |

## Section summary

This section has provided a very brief introduction to some common data tidying operations that can be performed using functions in the stringr function, as well as being a hands-on introduction to user defined functions as well as data testing. The next section will more formally introduce functions like mutate (already used and encountered) as well as related functions within the dplyr and tidyr packages.

# The Tidy Data Twins: tidyr and dplyr

## Introduction

The material covered so far may have seemed both tough and tedious: a lot of new packages and approaches needed to be learned, with a lot of potential for errors and to go wrong, and all you have to show for it are some R objects that don’t look that different to those you started with. However, the tools you learnt to use earlier form the foundation of all further data science and analysis work, as well as the raw material, carefully organised and arranged. With this foundational and logistical work in place, you can now get on with building something new.

To help you with effective construction of statistical analyses and preparation of datasets for such analysis, Hadley Wickham has provided both a series of design principles, known as the ‘tidy data paradigm’ and a pair of toolkits, tidyr and dplyr, which can help achieve these designs more easily. This section will start by briefly summarising the tidy data paradigm, followed by some examples of data tidying. It will conclude by discussing and demonstrating various functions and tools which work with one or more tidy datasets to produce further derived datasets and analyses, including the production of summary datasets, which provide a data science entry point to doing statistical analyses.

## The tidy data paradigm

Wickham’s article on tidy data, called simply ‘Tidy Data’, is available from the following location:

<http://vita.had.co.nz/papers/tidy-data.pdf>

Wickham argues that, in tidy data:

1. Each variable forms a column
2. Each observation forms a row
3. Each type of observational unit forms a table

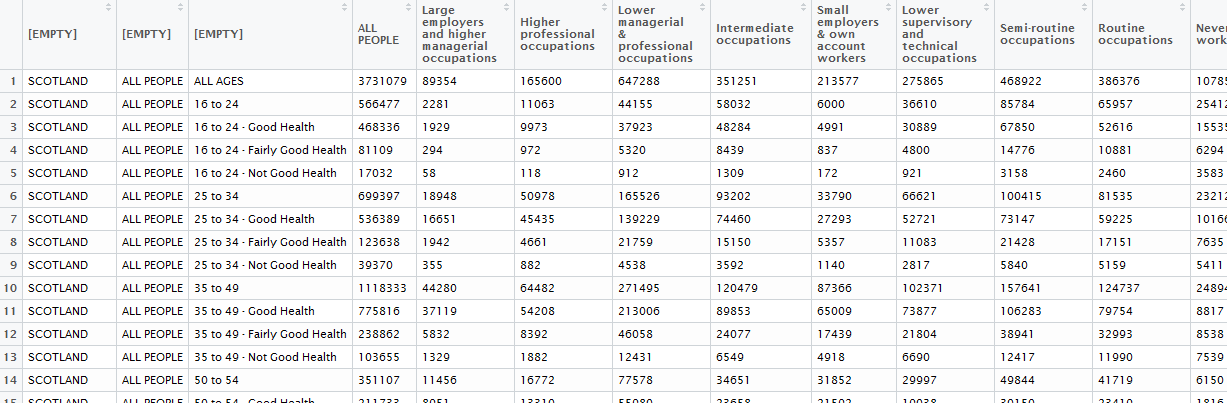
(source: p. 3 of above)

The article provides a number of examples and illustrations of both ‘untidy’ and ‘tidy’ datasets, and the types of processing required to change data from ‘untidy’ to ‘tidy’ structural forms. You are strongly encouraged to read through these examples, even though examples of data tidying given use slightly older methods and approaches to those described here.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **‘What’ and ‘Where’: My way of thinking about ‘Tidy Data’**  My own way of thinking about tidy data is to think about data tables as comprising two distinct types of variable:   * ‘Where’ or ‘Locator’ variables * ‘What’ variables   The purpose of the Where variables is to define, precisely and unambiguously, what the What variables refer to. They provide a ‘location’ at which and for which some characteristics (the Whats) were recorded. A simple example of this is the following:   |  |  |  | | --- | --- | --- | | latitude | longitude | Height above sea level | | 0.00 | 0.00 | 25 | | 25.00 | 0.00 | 21 | | 15.00 | 12.00 | 6 |   In this example, the first two columns of the table are the Where variables, and the third column is the ‘What’ variable. The Where variables required are defined by the type of observational unit (the third point of the definition above), and observations recorded using the same type of observation unit can be combined as additional columns within the same table. For example:   |  |  |  |  | | --- | --- | --- | --- | | **latitude** | **longitude** | **Height above sea level** | **Rainfall** | | 0.00 | 0.00 | 25 | Low | | 25.00 | 0.00 | 21 | Low | | 15.00 | 12.00 | 6 | High |   Just as observations (the ‘What’ variables) can be located spatially, so they can be located temporally as well:   |  |  |  |  | | --- | --- | --- | --- | | **latitude** | **longitude** | **Season** | **Rainfall** | | 0.00 | 0.00 | Winter | Low | | 25.00 | 0.00 | Winter | Low | | 15.00 | 12.00 | Winter | High | | 0.00 | 0.00 | Summer | Medium | | 25.00 | 0.00 | Summer | High | | 15.00 | 12.00 | Summer | Low |   In this above example, there are now three Where variables and one What variable. This table therefore uses a different type of observational unit than the previous two, and so are not compatible. A table using yet another type of observational unit is as follows:   |  |  | | --- | --- | | **Country** | **Rainfall** | | USA | Low | | Scotland | High | | France | High | | England | Medium |   Now there is just one Where variable and one What variable. Whenever two tables use different types of observational unit the process of mapping What variables is usually non-trivial at best. Often, the mapping of What variables between tables is a one-directional process. For example, Rainfall levels recorded at latitude, longitude and season can probably be mapped onto rainfall levels at latitude and longitude only by taking an all season average for that particular location: the What variable can be mapped from the more detailed to the less detailed Locational system, but not necessarily the other way around. However even this is not always the case. For example, if we assume that the height above sea level is seasonally invariant, then this variable, from the table that used the {latitude, longitude} Where variables, can probably be joined onto a table that used {latitude, longitude, season} where variables.  The relevance of this way of thinking hopefully becomes clearer if we introduce the following two tables. Firstly Table A   |  |  |  | | --- | --- | --- | | **Person\_id** | **Date\_of\_birth** | **Sex** | | 1 | 11/6/1943 | Male | | 2 | 7/2/1990 | Female | | 3 | 23/9/1938 | female |   And now Table B   |  |  |  |  | | --- | --- | --- | --- | | **Hospital\_id** | **Date of visit** | **Patient\_id** | **Event\_type** | | A | 19/9/2014 | 2 | Emergency | | A | 6/2/2015 | 1 | Inpatient | | B | 4/11/2012 | 3 | Emergency |   In Table A the first column is a ‘where’ variable: the second and third columns refer to characteristics of the variables that will (usually) not change over time. In Table B the first three columns are Where variables, and the last column is a What variable. For some types of analysis, such as producing survival or time to even statistics, it may be important to, for example, join the data of birth field from Table A to the appropriate rows in Table B. The Tidy Data paradigm makes thinking about these and many processes easier. |

## Data Tidying in practice: Place and General Health in the 2001 Census

Because we encountered it earlier, let’s now return to the object census\_2001\_health earlier, with the Tidy Data paradigm in mind. Using the View function, we can explore the first few columns and rows of this table, and scroll through the rest of the table:

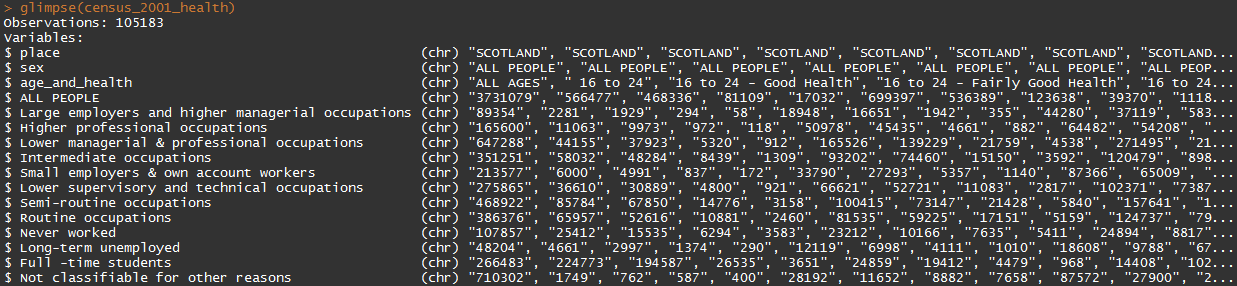


This data is not ‘tidy’, and quite a lot of work is required to make it so. Before we can work further with the dataset, we should first give the first three columns unique names (Currently their names are all listed as ‘EMPTY’), because this makes subsequent dplyr and tidyr functions much easier to use. This is best achieved using the Base R names function:

names(census\_2001\_health)[1:3] <- c(“place”, “sex”, “age\_and\_health”)

The above code introduces a number of features of R which have not yet been discussed. The ‘names’ function extracts the ‘names’ attribute from the dataframe object, which is returned as a character vector. [1:3] locates the first three elements only of this vector. These three elements are assigned, respectively, the character strings “place”, “sex” and “age\_and\_health” respectively.

If we now glimpse the table we can see that these first three variables have been named successfully.



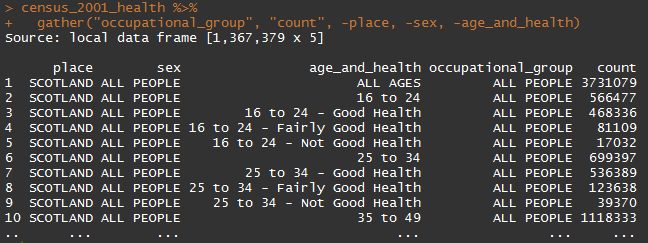
Although at this stage you may have to trust me on this, the Tidy Data target form for this table should be something like:

* Place: large region within Scotland
* Sex: male or female
* Age group
* Health status
* Occupational background
* Count

Of these variables, all but the last of the variables are ‘Where’ variables, ‘locating’ what particular count values refer to. Currently there are three main challenges to overcome to make this data ‘tidy’:

1. Inconsistent observational units within single variables – for example ‘place’ includes ‘SCOTLAND’, a number of large regions within Scotland, and smaller areas within each of these large areas. The sum of all counts, therefore, is likely to be three times the population of Scotland. Just one of these observational unit systems should be used. A similar but less severe issue is within the 2nd column, which has ‘male’, ‘female’ and ‘all people’, which would likely result in double counting of population sizes.
2. Two types of ‘where’ variable are located within a single column: age group and health are both provided within the third column. Ideally, the two pieces of information contained within them should be separated into two distinct variables, ‘age group’ and ‘health status’.
3. Different categories for a single variable are located in multiple columns: from the forth column onwards. These all refer to different occupational categories.

Because it looks most like progress, we will tackle the last of these three challenges first. To do this, we use the ‘gather’ function in the tidyr package.

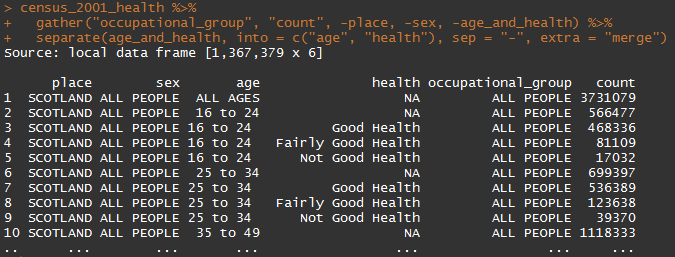


The gather function ‘gathers’ the contents of a number of columns, and puts their contents into two columns: the first argument to gather is the name of the first column, known as the ‘key’, and the second argument to gather is the name of the second column it creates, known as the ‘value’. I have called the key ‘occupational group’, and the value ‘count’. Subsequent arguments to gather define which columns should be gathered into these two columns, and you can either define the gathered contents inclusively, by listing the variables that SHOULD go in these columns, or exclusively, by listing the variables that SHOULD NOT go into these columns. Of the 16 columns in the original dataframe, I wanted all but three to go into these two columns, as they each define different occupational groups. I therefore define the gathered contents exclusively rather than inclusively. I do this using the – symbol, typing

-place, -sex, -age\_and\_health

As additional arguments to the gather function. By implication, gather gathers together everything except these three columns.

The next thing I want to do is separate the contents of the age\_and\_health variable into separate age columns and health columns. Fortunately, I can make use of a consistency in terms of how this column’s contents are labelled: firstly, the age group is listed, then there is – symbol, and then the health status is listed. Another of tidyr’s functions, separate,



Some of the problems with this dataset include:

1. The first three columns are not named, and are listed as [EMPTY] [EMPTY] [EMPTY]

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)
7. In this case it seems likely that another hidden escape character, for representing a ‘tab’, separates the variable name from the description [↑](#footnote-ref-7)
8. These two colons are known in R as the ‘scoping operator’, and are used by R to specify particular versions of functions located in particular packages. In R it is possible to have two functions with the same name. This creates an ambiguity in R, which it usually resolves by assuming you meant to call the function from the most recently loaded package. The scoping operator allows you to override this default behaviour, and can also be useful when you need to be more explicit in your code about which packages you are using for which functions. [↑](#footnote-ref-8)
9. Not supported in Base R [↑](#footnote-ref-9)
10. The horrible looking expression paste0(rep(“c”, 16), collapse = “”) is a way of generating this long string of c’s to the right length. The number 16 indicates the number of times the character should be repeated, and the collapse argument turns the output of ‘rep’ from a vector containing 16 identical one character strings ‘c’ into a single string containing ‘c’ 16 times. For now, it is not important to know how this expression works, just what it does and how it can be modified. [↑](#footnote-ref-10)
11. As, in this toy example, the count variable is a factor rather than a character, technically ‘count’ is already numeric, but comprised of integer values and a key-value lookup table, as with SPSS labels introduced earlier. If you are interested and have the time try to think about how this might be problematic in this case. [↑](#footnote-ref-11)
12. In practice, only one of these symbols, a space or a comma, is likely to be used. But this is intended as a particularly ‘bad’ example. [↑](#footnote-ref-12)