

UBDC Summer Training Programme 2017

Introduction to R

Dr Jonathan Minton

University of Glasgow

7 August 2017

Jonathan.minton@glasgow.ac.uk

Copyright Notice

Copyright © 2016 by Jonathan Minton, University of Glasgow, Glasgow G12 8QQ. Telephone 07866 022543. This publication or any part thereof may not be reproduced or transmitted in any form or by any means electronic or mechanical, including photocopying, recording, storage in an information retrieval system or otherwise, without the prior written permission of the author.

Disclaimer

While the author takes great care to ensure the accuracy and quality of these materials, all material is provided without any warranty whatsoever, included but not limited to, the implied warranties of merchantability or fitness for a particular purpose. Further, the author reserves the right to revise this publication and make changes to its content at any time, without obligation to notify any person or entity of such revisions or changes.

Acknowledgements

I would like to thank the following, for initiating, managing, advising, guiding, correcting or otherwise supporting the development of this document and associated course: Prof Nick Bailey, Prof Colin McGinn, Dr Caroline Haig, Ms Rachel Zhang, Dr David McArthur, Dr Sila Katarzyna, Mr Keith Maynard, Dr Mirjam Allik and Ms Claire McCallum. As usual all plaudits should go to them and all further corrections and criticisms, send via the email address above, should go to me.

Last revised: 3 August, 2017

Contents

1. Introduction.....	5
1.1. Approach and conventions used in this course	5
1.2. Course overview	5
2. Data Management and Simple Analysis	7
2.1. Why be interested in data management?	7
2.2. A motivation for surviving the course	8
3. Getting Started	8
3.1. R and RStudio	8
3.2. Installing R.....	9
3.3. Installing RStudio.....	9
3.4. Getting Started with RStudio.....	10
3.5. RStudio Features	12
3.6. Autocomplete features	14
4. Setting up R projects	16
4.1. Creating an R Project	16
4.2. Directory Conventions within an R Project.....	17
4.3. Script conventions	18
4.4. Libraries	18
4.5. Libraries to turn Base R into ‘Wickhamese R’	19
5. An introduction to ‘piped R’	20
5.1. Three ways of saying the same thing	20
5.2. Piping and function arguments.....	22
6. Loading and saving data	25
6.1. Introduction	25
6.2. File types	25
6.3. Data and metadata	28
6.4. Packages for reading and writing data	30
6.5. Getting help about R packages	31
6.6. Example: loading a csv file with leading metadata	33
6.7. Reading Binary Files.....	39
6.7.1. Example 1: Reading Excel Files	39
6.7.2. Example 2: Reading SPSS data	40
6.8. Section summary	42
7. Initial data tidying	43
7.1. Introduction	43

7.2. Example 1: Converting symbols to values	43
7.3. Example 2: Removing commas and removing whitespace	47
7.4. Section summary	50
8. The Tidy Data Twins: tidyr and dplyr	52
8.1. Introduction	52
8.2. The tidy data paradigm	52
8.3. Data Tidying in practice: Place and General Health in the 2001 Census	54
8.4. Working with tidied data	65
8.5. Question 1: Did males or females in Scotland have better self-reported health in 2001?	66
8.6. Question 2: Which places have the highest and lowest proportion of males aged between 50 to 64 who report poor health?	69
8.7. Additional questions	79
8.8. Binding and joining	84
8.8.1. Finding small places using anti_join	84
8.8.2. Binding rows	86
9. Day One Summary	87
10. Day Two	89
10.1. Introduction	89
11. Exploring and Presenting Data Using Tables and Graphs	89
11.1. Exporting tables	91
11.2. Data Visualisation using ggplot2	73
11.2.1. Saving ggplot2 outputs	77
12. An introduction to the split-apply-combine paradigm and plyr	93
12.1. Example: Automating regression modelling	94
12.2. Example: Automating the production of figures and files	97
12.3. Automating the reading and tidying of files	98
12.4. Summary	98
13. Extended Practical: Uncovering aggregation biases in all-cause mortality trends	99
13.1. Background	99
13.2. Uncovering Aggregation Bias	99
13.3. Purpose of this practical	100
13.4. Data source	101
13.5. Data Tidying	102
13.6. Data Analysis	103
13.7. Breakdown of practical sessions	103

1. Introduction

1.1. Approach and conventions used in this course

Welcome to 'Introduction to R'. This course will cover a great deal of material in a relatively short space of time. Rather than presenting a series of unrelated examples of data science and data management in practice, most of this course will follow a single extended example, showing the various conceptual and technical processes required to convert a particular dataset, taken from the 2001 census, from a format where it's difficult to analyse and to work with ('untidy data') to a format where it's much easier to work with and analyse ('tidy data'). By following each of the steps and stages in the journey to make this particular dataset easier to work with and analyse, a much broader series of concepts, tools and ideas about data management and analysis will be introduced, and tools and resources for applying these methods in other contexts will be provided. Although the tools and packages used in this course will be R, the concepts and ideas they implement are much broader and more widely applicable, so even if you choose or are forced to use something other than R after this course, I hope these broader concepts and ideas will still be applicable and useful for you.

Most of this course will involve following through this handbook from start to finish, and completing a series of exercises based around the material presented. Please feel free to talk to me, and to each other, for further support and help. It is likely that, though you may complete each particular exercise, a broader understanding and familiarity with the tools and techniques introduced within this course will only emerge much later, with repeated practice and exposure to a wide range of data management and data science challenges and solutions. I hope this handbook will continue to be useful as a resource for this much longer journey.

Within this text, code will be represented using the `Lucida Console` font, supplementary text and ideas will be presented inside boxes, and exercises to complete will be indicated using **this font (Berlin Sans FB)**.

1.2. Course overview:

- The differences between Statistics and Data Science
- The Data-to-Value Chain
- Differences between R and RStudio
- Understanding the contents of RStudio panes
- RStudio features
- Recognising differences between different parts of R scripts
- How to set up RStudio projects
- R Packages/Libraries
- Wickhamese R and code piping
- Accessing and understanding R help files
- Differences between text and binary files
- Metadata
- Loading text files and binary files
- Initial Data Cleaning/Tidying processes with string manipulation
- Differences between R object types
- User-defined functions
- Code development as pipe building and pipe testing

- The tidy data paradigm
- Getting messy data into a tidy data form
- Saving tidy data
- Rapid data analysis of tidy data
- Extending the pipe: A very quick introduction to data visualisation using ggplot2
- Data visualisation using ggplot2
- Process automation using plyr and the split-apply-combine strategy

2. Data Management and Simple Analysis

2.1. Why be interested in data management?

There's a joke¹ 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 course 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

¹ Note: Statisticians aren't comedians.

data efficiently, you can pass through this stage more quickly, and spend more of your time at stages (3) and (4).

Exercise 2.1: Discuss, in pairs or small groups, one or two examples of projects you've worked on where data management tasks have taken much longer than data analysis tasks, and describe your approach to overcoming these challenges.

2.2. A motivation for surviving the course

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 end of the course will be spent on a practical session which will require using many of the tools introduced in the previous day and a half. 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 course, is to be able to recreate some of the analyses presented in the paper, then hopefully to go beyond some of these analyses.

Exercise 2.2: Click on the link to the paper above and discuss, in pairs or small groups, the key conclusions within the paper, and the data management tasks required to complete the analyses presented.

3. Getting Started

3.1. 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 a 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 it is the product of many minds.

3.2. 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:

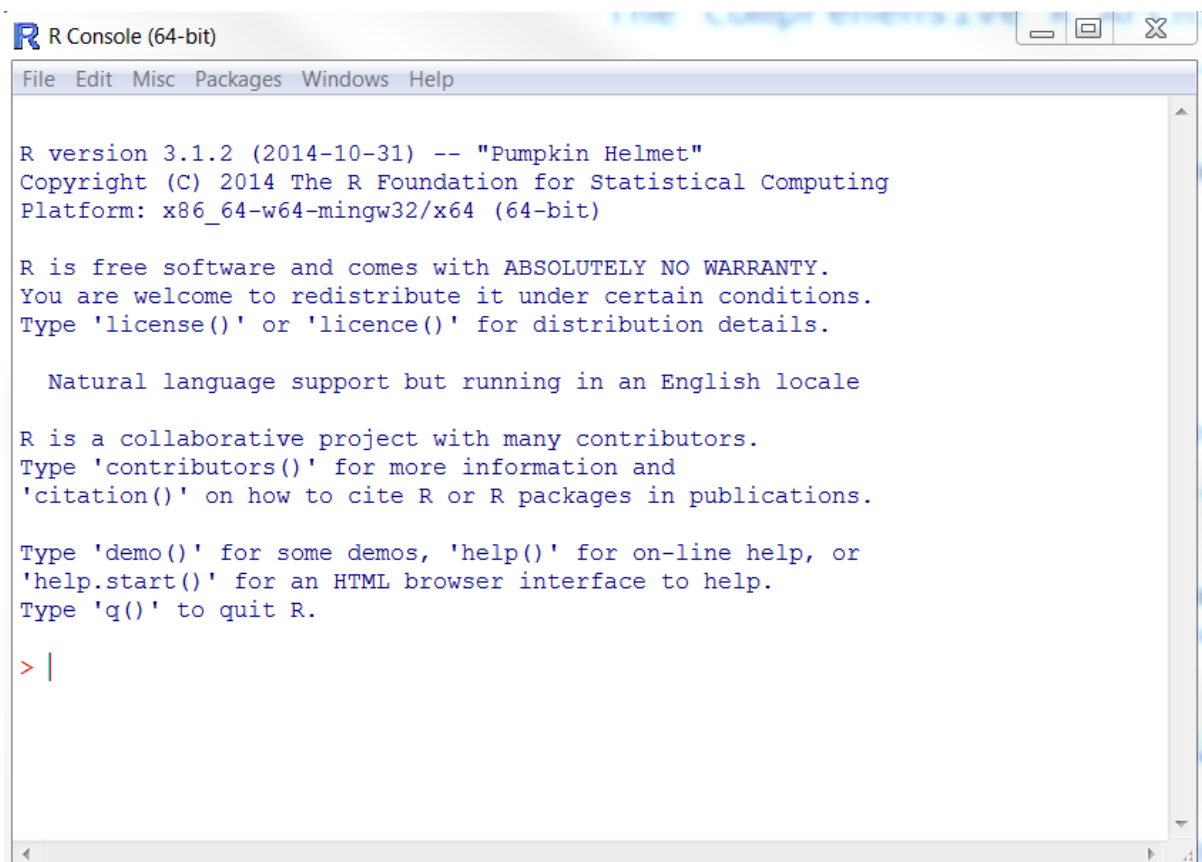


Figure 1 Example of the basic R Graphical User Interface (GUI) in Windows

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.

3.3. 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!

3.4. Getting Started with RStudio

When you first load up RStudio you will probably get something as follows:²

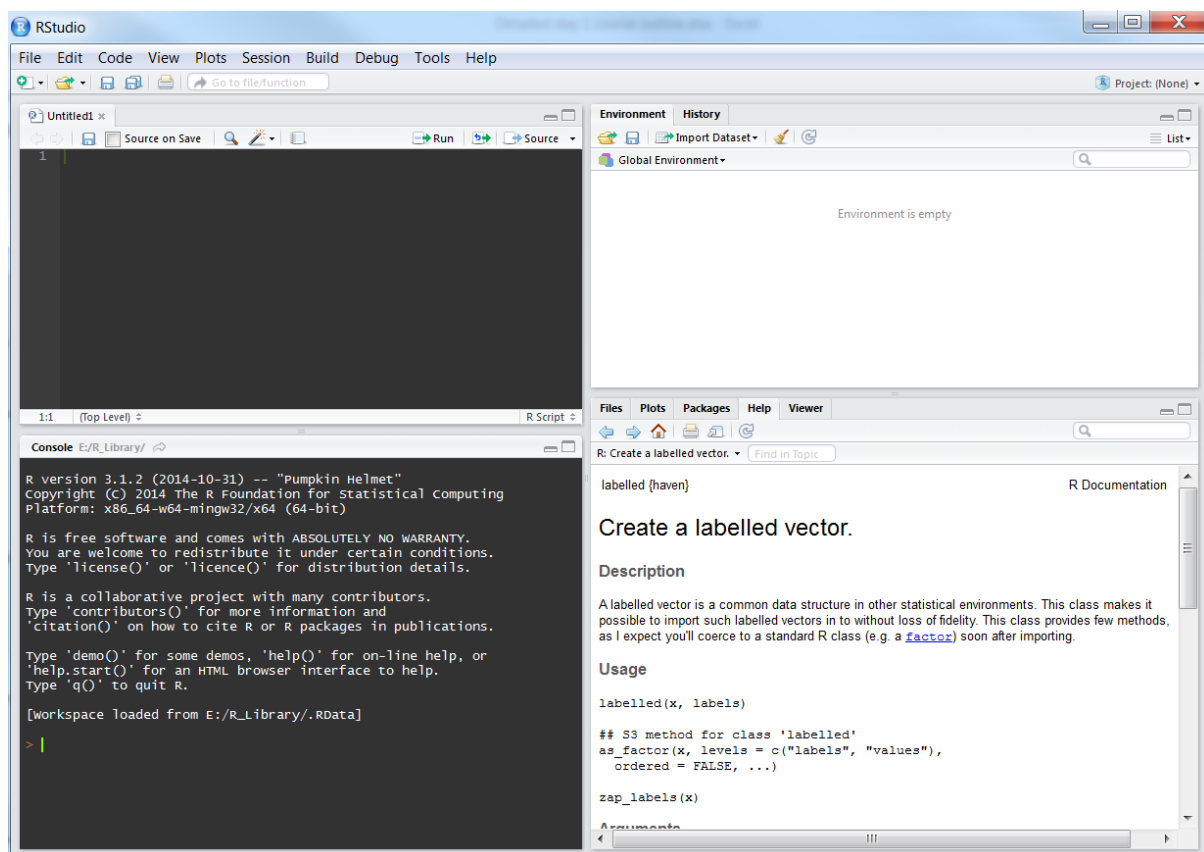


Figure 2 Illustration of Rstudio and its four panes. Top left: Script pane; Bottom left: I/O pane; Bottom right: figures, help and documentation; Top right: history and environment

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 left:** 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 Note you can modify the colour scheme for some of the panes using the 'Global Options' -> 'Appearance' tab within the 'tools' drop down menu. In my session the colour scheme is set to 'Idle Fingers'.

- **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 right:** 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).

Inputs, outputs and side effects

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.)

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.

Exercise 3.1: Open up the R GUI directly, explore and close it, and then open up R within RStudio. Create, but don't save, an R script. Enter some text into the R script, and also directly into the IO pane. Move panes around in RStudio and explore some of the options in the dropdown menu.

3.5. RStudio Features

Exercise 3.2: (Optional): Go into the 'global options' line within the 'Tools' dropdown menu within RStudio, then select 'Appearance', then the 'Idle Fingers' editor theme. The colour coding described below uses this theme, but the principle is broader. If you prefer, select a different theme, or use the default theme.

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:

```

1  # Quick script which shows how mortality contour hurdles have changed
2  # by cohort year.
3
4  # England & Wales?
5
6  rm(list = ls())
7
8
9  require(readr)
10
11 require(plyr)
12 require(tidyr)
13 require(stringr)
14 require(dplyr)
15 require(car)
16
17 #graphics
18 require(lattice)
19 require(latticeExtra)
20
21 require(ggplot2)
22 require(RColorBrewer)
23 require(grid)
24
25 # for smoothing
26 require(fields)
27 require(spatstat)
28
29
30 dta <- read_csv("data/tidy/counts.csv")
31
32 this_dta <- dta %>%
33   filter(country == "GBRTENW" & sex != "total") %>%
34   mutate(birth_year = year - age) %>%
35   filter(birth_year >= 1850 & age >= 50 & age <= 90) %>%
36   arrange(sex, birth_year, age) %>%
37   mutate(
38     cmr = death_count / population_count,
39     lg_cmr = log(cmr, base = 10)
40   ) %>%
41   select(sex, birth_year, age, lg_cmr)
42
43
44 png(filename="figures/shifting_hurdles/shifting_hurdles_spectral.png",
45     width=30, height=20, res=300, units="cm"
46 )

```

Figure 3 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, but 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 symbol. (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.

Exercise 3.3: Create an R script and write out some of the lines in the script above, noting how different parts of the code are highlighted in different ways. If a different colour scheme has been used, work out which text colours indicate which parts of the code. (objects, operators, strings, comments, and so on.) Alternatively, look through the code snippet example presented in Tools -> Options -> Appearance -> Editor theme, and how this changes when different themes are selected.

3.6. Autocomplete features

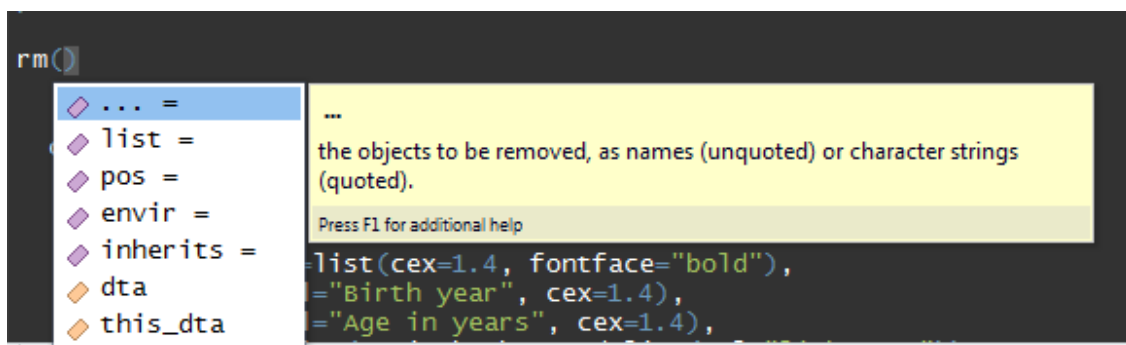
Most of the features in the script pane are only apparent when writing in the pane. For example, if you type

```
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. 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.

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.

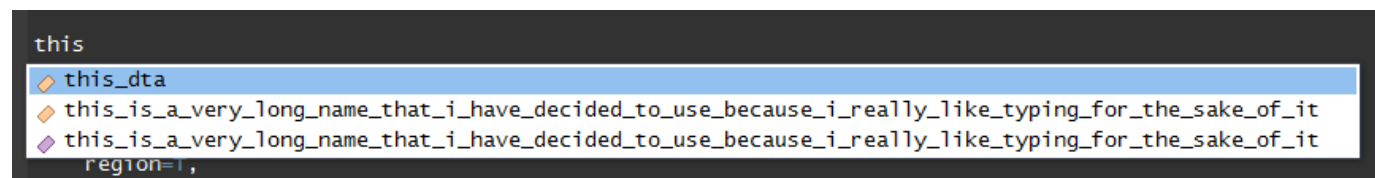
You can also see the yellow box to the right of the drop down list. This contains extra information on the particular highlighted item.

Exercise 3.4: Reproduce the in-text drop-down list shown above.

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,³ 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.


Exercise 3.5: Produce two objects, one with a long name, the other with a short name, and whose names both start with the same characters, and use the drop-down menu to choose between them. Assign the value of one of the objects to the other object, and delete one or both objects.

4. Setting up R projects

4.1. 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 computer 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 your computer, 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.

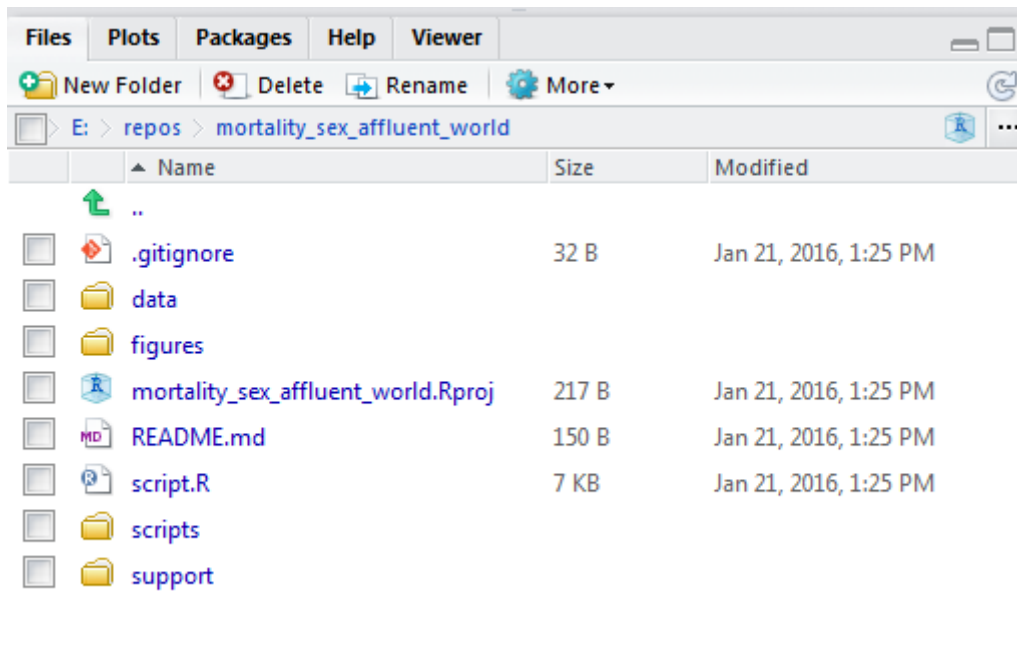
When you have multiple projects, you can use the project icon on the top right to open, close, and switch between projects. Whenever you switch to a new project, there will be a different R session, with different objects and contents, and a different base directory from which R will search for and write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

Exercise 4.1: Use the information above to produce an R project called `aqmen_data_science`.

³ 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.

4.2. 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.

Exercise 4.2: Use at least two different approaches for creating the following subdirectories within your newly created project folder: 1) data; 2) figures; 3) scripts; 4) support. Locate these directories using Windows Explorer.

4.3. 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.

Exercise 4.3: Create a script file script.R and place it in the base project directory, and create a second script file and place it in the scripts subdirectory.

4.4. 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.⁴

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.

4.5. 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.⁵ 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 `magrittr` package.

⁴ 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!

⁵ <http://www.geirfreysson.com/2015/08/hadley-wickham-the-statistics-celebrity-and-r-programmer/>
Accessed 5 Feb 2016

5. An introduction to ‘pipedReader’

5.1. Three ways of saying the same thing

The table overleaf shows three different ways of writing the same code. Each of the formulations is semantically the same, meaning that the same pieces of information are expressed in each case, and the same output will be generated as a result of running the code. However the style with which the operations are expressed are different.

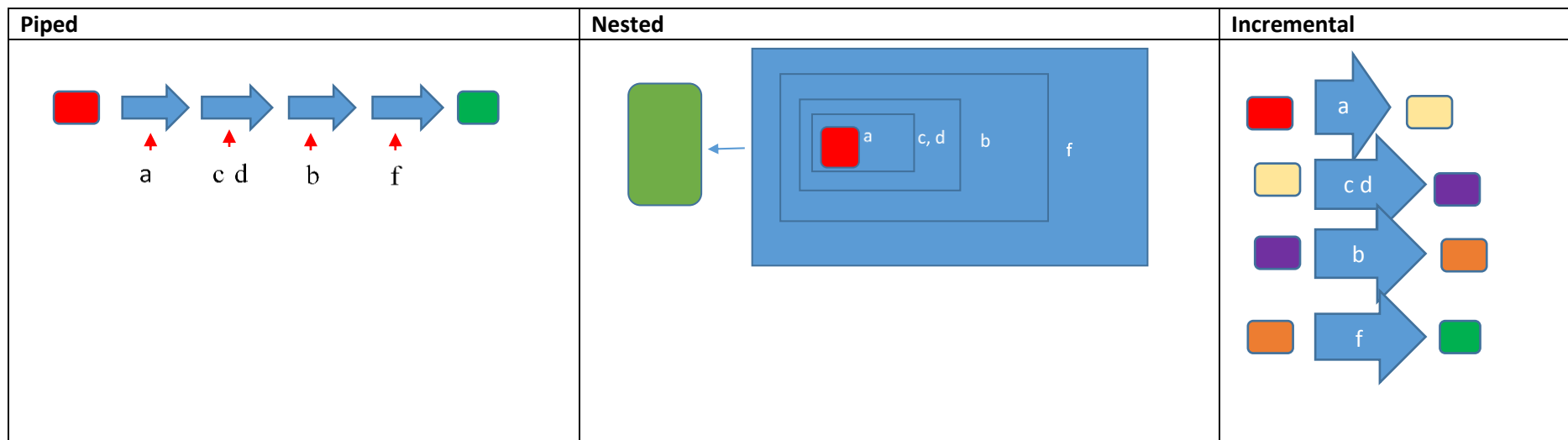
- The first style, which I call **incremental**, involves completing a multi-stage operation one step at a time, over a number of lines. On each line, the output is the first object, which is assigned (with the `<-` operator) the value of the function on its right. The function’s arguments include an input, likely a dataset, and some additional arguments which modify how the function will operate on that input. On each line, a new object is created as an output. The inputs to most lines are the outputs from the previous line, meaning a large number of intermediate outputs (`temp1`, `temp2`, `temp3`) are generated, which are then removed using the `rm` function on the very last line.
- The second style, which I call **nested**, completes the multi-stage operation in a single stage. Functions are contained within functions which are contained within functions. No intermediate outputs are generated, and so no temporary objects need to be removed.
- The third style is a **pipedReader** style. The `%>%` operator is used to pass the output from the previous function as the input to the first argument of the next function. Again, no intermediate objects are created, and so no ‘workings’ need to be removed afterwards.

Each of the three coding styles has an equivalent formulation in English. These are provided on the right column of the table. Additionally, graphical representations of each of the three formulations are also presented.

Of the three English language formulations, I would argue that the ‘pipedReader’ formulation is both easy to understand and closest to how most⁶ native English speakers naturally speak and think. ***The key advance that pipedReader coding provides is a means to write and develop R code in a way that’s much closer to natural language, making it both much easier to read and to write.***

⁶ The exception being ‘tough guy’ fictional characters like Rambo and The Hulk, philosophers of logic, and very young children, who will each tend towards the ‘incremental’ formulation.

Style	Code Example	English Equivalent
<i>Incremental</i>	<pre>temp1 <- function_1(input, arg_a) temp2 <- function_2(temp1, arg_c, arg_d) temp3 <- function_3(temp2, arg_b) output <- function_4(temp3, arg_f) rm(temp1, temp2, temp3)</pre>	<p>Alice is small. Alice walked to the station. The walking was quick. The station is big. The station had a train. The train was late. Alice caught the train.</p>
<i>Nested</i>	<pre>output <- function_4(function_3(function_2(function_1(input, arg_a), arg_c, arg_d), arg_b), arg_f)</pre>	<p>Alice walked to the station to catch a train, which was late, which is big, did so quickly, who is small.</p>
<i>Piped</i>	<pre>input %>% function_1(arg_a) %>% function_2(arg_c, arg_d) %>% function_3(arg_b) %>% function_4(arg_f) -> output</pre>	<p>Alice, who is small, walked quickly to the station, which is big, to catch a train, which was late.</p> <p>Small Alice walked quickly to the big station to catch the late train.</p>



5.2. Piping and function arguments

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.

Exercise 5.1: In pairs or small groups, discuss the example above, focusing on any terminology or concepts which are unfamiliar to you.

Using R help to find which function argument slot accepts the data

Two very important examples of functions whose data slot is not the first argument are the `xtabs` function, for cross tabulations, and the `lm` function, for linear regression modelling,⁷ which are both Base R functions included within R by default; in both cases the first argument slot is reserved for a formula and the second argument slot, named 'data', reserved for the input data.

To see this we can look at the help file for each function, 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`

lm {stats}

R Documentation

Fitting Linear Models

Description

`lm` is used to fit linear models. It can be used to carry out regression, single stratum analysis of variance and analysis of covariance (although `av` may provide a more convenient interface for these).

Usage

```
lm(formula, data, subset, weights, na.action,
   method = "qr", model = TRUE, x = FALSE, y = FALSE, qr = TRUE,
   singular.ok = TRUE, contrasts = NULL, offset, ...)
```

Arguments

<code>formula</code>	an object of class " formula " (or one that can be coerced to that class): a symbolic description of the model to be fitted. The details of model specification are given under 'Details'.
<code>data</code>	an optional data frame, list or environment (or object coercible by as.data.frame to a data frame) containing the variables in the model. If not found in <code>data</code> , the variables are taken from <code>environment(formula)</code> , typically the environment from which <code>lm</code> is called.
<code>subset</code>	an optional vector specifying a subset of observations to be used in the fitting process.
<code>weights</code>	an optional vector of weights to be used in the fitting process. Should be <code>NULL</code> or a numeric vector. If non- <code>NULL</code> , weighted least squares is used with weights <code>weights</code> (that is, minimizing $\sum(w * e^2)$); otherwise ordinary least squares is used. See also 'Details'.
<code>na.action</code>	a function which indicates what should happen when the data contain NAs. The default is set by the <code>na.action</code> setting of options , and is na.fail if that is unset. The 'factory-fresh' default is na.omit . Another possible value is <code>NULL</code> , no action. Value na.exclude can be useful.

⁷ Even if you are not familiar with linear regression, both of these functions are discussed because the formulae to each function are specified in similar ways.

Cross Tabulation

Description

Create a contingency table (optionally a sparse matrix) from cross-classifying factors, usually contained in a data frame, using a formula interface.

Usage

```
xtabs(formula = ~., data = parent.frame(), subset, sparse = FALSE,
      na.action, exclude = c(NA, NaN), drop.unused.levels = FALSE)
```

Arguments

<code>formula</code>	a formula object with the cross-classifying variables (separated by <code>+</code>) on the right hand side (or an object which can be coerced to a formula). Interactions are not allowed. On the left hand side, one may optionally give a vector or a matrix of counts; in the latter case, the columns are interpreted as corresponding to the levels of a variable. This is useful if the data have already been tabulated, see the examples below.
<code>data</code>	an optional matrix or data frame (or similar: see model.frame) containing the variables in the formula <code>formula</code> . By default the variables are taken from <code>environment(formula)</code> .
<code>subset</code>	an optional vector specifying a subset of observations to be used.
<code>sparse</code>	logical specifying if the result should be a sparse matrix, i.e., inheriting from sparseMatrix . Only works for two factors (since there are no higher-order sparse array classes yet).
<code>na.action</code>	a function which indicates what should happen when the data contain NAs.
<code>exclude</code>	a vector of values to be excluded when forming the set of levels of the classifying factors.
<code>drop.unused.levels</code>	a logical indicating whether to drop unused levels in the classifying factors. If this is <code>FALSE</code> and there are unused levels, the table will contain zero marginals, and a subsequent chi-squared test for independence of the factors will not work.

Exercise 5.2: Open up help for `lm` or `xtabs` using either the `'?'` operator or an alternative approach. Locate and review the contents of the help file.

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'.

Piping and the placeholder (`.`) symbol

To use piping with functions that don't take the data as the first argument, 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 the base R way of producing a crosstab of gender against obesity status would be:

```
xtabs(~ obese + gender, data = dta)
```

The piped equivalent of this would be:

```
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.

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.

Exercise 5.3: Run the following code:

```
data(ChickWeight)
```

to load a dataset called `ChickWeight` into the R workspace. Produce a crosstab of `Chick` against `Diet` using `xtabs` using the non-piped formulation. Load `dplyr` and do the same using the piped formulation of the same code. (Note: the `library` or `require` functions will be required.)

Similarly, imagine you want to run a linear 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 placed at the end of all but the last line. In this example, the first two lines both end with `%>%`, but the third line does not. Whenever R encounters the `%>%` operator, it knows the instructions are not yet complete, and to await further instructions before continuing.

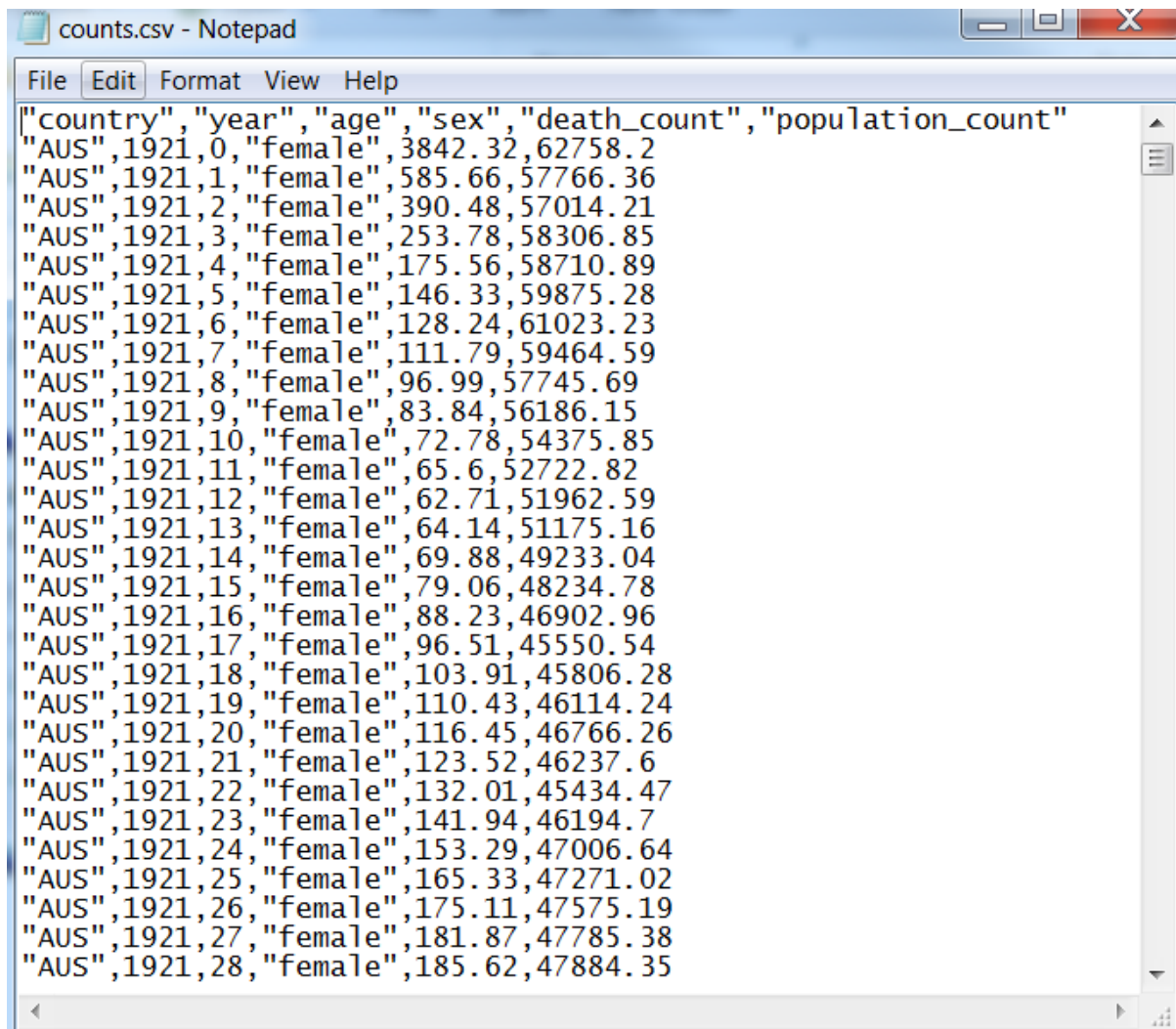
Note here also the use of the `mutate` command, which generates the `bmi` variable given the weight and height variables contained in the `dta` object. The `mutate` function is part of `dplyr`, which we will cover in much more depth soon.

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.

Exercise 5.4: (Optional) Regress Weight against Time using the `ChickWeight` dataset. Save the regression. Produce a multiple regression of weight against time and diet. Compare the regressions.



```
"country","year","age","sex","death_count","population_count"
"AUS",1921,0,"female",3842.32,62758.2
"AUS",1921,1,"female",585.66,57766.36
"AUS",1921,2,"female",390.48,57014.21
"AUS",1921,3,"female",253.78,58306.85
"AUS",1921,4,"female",175.56,58710.89
"AUS",1921,5,"female",146.33,59875.28
"AUS",1921,6,"female",128.24,61023.23
"AUS",1921,7,"female",111.79,59464.59
"AUS",1921,8,"female",96.99,57745.69
"AUS",1921,9,"female",83.84,56186.15
"AUS",1921,10,"female",72.78,54375.85
"AUS",1921,11,"female",65.6,52722.82
"AUS",1921,12,"female",62.71,51962.59
"AUS",1921,13,"female",64.14,51175.16
"AUS",1921,14,"female",69.88,49233.04
"AUS",1921,15,"female",79.06,48234.78
"AUS",1921,16,"female",88.23,46902.96
"AUS",1921,17,"female",96.51,45550.54
"AUS",1921,18,"female",103.91,45806.28
"AUS",1921,19,"female",110.43,46114.24
"AUS",1921,20,"female",116.45,46766.26
"AUS",1921,21,"female",123.52,46237.6
"AUS",1921,22,"female",132.01,45434.47
"AUS",1921,23,"female",141.94,46194.7
"AUS",1921,24,"female",153.29,47006.64
"AUS",1921,25,"female",165.33,47271.02
"AUS",1921,26,"female",175.11,47575.19
"AUS",1921,27,"female",181.87,47785.38
"AUS",1921,28,"female",185.62,47884.35
```

Exercise 6.3: Open up `poverty_dataset.sav` and `counts.csv` as above. Discuss differences between the two files with a colleague.

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

counts.csv - Word

REFERENCES MAILINGS REVIEW VIEW DEVELOPER EndNote X6

Paragraph Styles

AaBbCcDd AaBbCcDd AaBbCc AaBbCc AaB| AaBbCc

Normal No Spac... Heading 1 Heading 2 Title Subtitle

```
"country","year","age","sex","death_count","population_count"
"AUS",1921,0,"female",3842.32,62758.2
"AUS",1921,1,"female",585.66,57766.36
"AUS",1921,2,"female",390.48,57014.21
"AUS",1921,3,"female",253.78,58306.85
"AUS",1921,4,"female",175.56,58710.89
"AUS",1921,5,"female",146.33,59875.28
"AUS",1921,6,"female",128.24,61023.23
"AUS",1921,7,"female",111.79,59464.59
"AUS",1921,8,"female",96.99,57745.69
"AUS",1921,9,"female",83.84,56186.15
"AUS",1921,10,"female",72.78,54375.85
"AUS",1921,11,"female",65.6,52722.82
"AUS",1921,12,"female",62.71,51962.59
"AUS",1921,13,"female",64.14,51175.16
"AUS",1921,14,"female",69.88,49233.04
"AUS",1921,15,"female",79.06,48234.78
"AUS",1921,16,"female",88.23,46902.96
"AUS",1921,17,"female",96.51,45550.54
"AUS",1921,18,"female",103.91,45806.28
"AUS",1921,19,"female",110.43,46114.24
"AUS",1921,20,"female",116.45,46766.26
"AUS",1921,21,"female",123.52,46237.6
"AUS",1921,22,"female",132.01,45434.47
"AUS",1921,23,"female",141.94,46194.7
"AUS",1921,24,"female",153.29,47006.64
"AUS",1921,25,"female",165.33,47271.02
"AUS",1921,26,"female",175.11,47575.19
```

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	<ul style="list-style-type: none"> • Metadata are pre-encoded 	<ul style="list-style-type: none"> • The file access depends on

	<ul style="list-style-type: none"> • File sizes can be smaller 	<p>whether compatible software are available, and remain available: Files may be accessible now, but not in the future</p> <ul style="list-style-type: none"> • If some data are corrupted, all data may become inaccessible.
Text	<ul style="list-style-type: none"> • 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 	<ul style="list-style-type: none"> • 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.

6.3. 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’⁸). 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:

⁸ In this case it seems likely that another hidden escape character, for representing a ‘tab’, separates the variable name from the description

```

CAS008.csv - Notepad
File Edit Format View Help
Table CAS008 whether living in household or communal establishment and age by migration (people)

Table population : All people in area and those having moved from area to within the UK in the past year
Geographical level : Output Area

,, "Lived at same address, or moved from within same area", "Inflow: Lived elsewhere one year ago outside
present area", "Outflow: Moved out of the area to the rest of the UK", "Net migration within the UK"
"60QA000001", "ALL PEOPLE IN HOUSEHOLDS", 54, 4, 3, 1
"60QA000001", "0 to 9", 7, -, -, -
"60QA000001", "10 to 19", 2, -, 1, -1
"60QA000001", "20 to 64", 35, 3, 2, 1
"60QA000001", "65 and over", 10, 1, -, 1
"60QA000001", "ALL PEOPLE IN COMMUNAL ESTABLISHMENTS", -, -, -, -
"60QA000001", "0 to 9", -, -, -, -
"60QA000001", "10 to 19", -, -, -, -
"60QA000001", "20 to 64", -, -, -, -
"60QA000001", "65 and over", -, -, -, -
"60QA000002", "ALL PEOPLE IN HOUSEHOLDS", 100, 74, 11, 57
"60QA000002", "0 to 9", 6, 15, 3, 12
"60QA000002", "10 to 19", 15, 9, 1, 6
"60QA000002", "20 to 64", 63, 50, 7, 39
"60QA000002", "65 and over", 16, -, -, -
"60QA000002", "ALL PEOPLE IN COMMUNAL ESTABLISHMENTS", -, -, 2, -2
"60QA000002", "0 to 9", -, -, -, -
"60QA000002", "10 to 19", -, -, -, -
"60QA000002", "20 to 64", -, -, -, -
"60QA000002", "65 and over", -, -, 2, -2
"60QA000003", "ALL PEOPLE IN HOUSEHOLDS", 72, 4, 2, 2
"60QA000003", "0 to 9", 9, -, -, -
"60QA000003", "10 to 19", 9, -, -, -
"60QA000003", "20 to 64", 48, 4, 2, 2

```

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:

```

icd_10_underlying_single_age_year_race_hispanic_gender_assault.txt - Notepad
File Edit Format View Help
Notes Year Year Code Race Race Code Hispanic Origin Hispanic Origin Code Gender Gender Code Single-Year Ages
Single-Year Ages Code Deaths Population Crude Rate
"1999" "1999" "American Indian or Alaska Native" "1002-5" "Hispanic or Latino" "2135-2" "Female" "F" "< 1 year" "0"
0 6120 Unreliable
"1999" "1999" "American Indian or Alaska Native" "1002-5" "Hispanic or Latino" "2135-2" "Female" "F" "1 year" "1"
0 6291 Unreliable
"1999" "1999" "American Indian or Alaska Native" "1002-5" "Hispanic or Latino" "2135-2" "Female" "F" "2 years" "2"
0 6355 Unreliable
"1999" "1999" "American Indian or Alaska Native" "1002-5" "Hispanic or Latino" "2135-2" "Female" "F" "3 years" "3"
0 5963 Unreliable
"1999" "1999" "American Indian or Alaska Native" "1002-5" "Hispanic or Latino" "2135-2" "Female" "F" "4 years" "4"
0 5923 Unreliable
"1999" "1999" "American Indian or Alaska Native" "1002-5" "Hispanic or Latino" "2135-2" "Female" "F" "5 years" "5"
0 6420 Unreliable
"1999" "1999" "American Indian or Alaska Native" "1002-5" "Hispanic or Latino" "2135-2" "Female" "F" "6 years" "6"
0 6371 Unreliable
"1999" "1999" "American Indian or Alaska Native" "1002-5" "Hispanic or Latino" "2135-2" "Female" "F" "7 years" "7"
1 6369 Unreliable
"1999" "1999" "American Indian or Alaska Native" "1002-5" "Hispanic or Latino" "2135-2" "Female" "F" "8 years" "8"
0 5370 Unreliable
"1999" "1999" "American Indian or Alaska Native" "1002-5" "Hispanic or Latino" "2135-2" "Female" "F" "9 years" "9"
0 8658 Unreliable
"1999" "1999" "American Indian or Alaska Native" "1002-5" "Hispanic or Latino" "2135-2" "Female" "F" "10 years" "10"
1 6506 Unreliable
"1999" "1999" "American Indian or Alaska Native" "1002-5" "Hispanic or Latino" "2135-2" "Female" "F" "11 years" "11"
0 5826 Unreliable
"1999" "1999" "American Indian or Alaska Native" "1002-5" "Hispanic or Latino" "2135-2" "Female" "F" "12 years" "12"
0 5871 Unreliable
"1999" "1999" "American Indian or Alaska Native" "1002-5" "Hispanic or Latino" "2135-2" "Female" "F" "13 years" "13"
0 5697 Unreliable
"1999" "1999" "American Indian or Alaska Native" "1002-5" "Hispanic or Latino" "2135-2" "Female" "F" "14 years" "14"
0 5535 Unreliable
"1999" "1999" "American Indian or Alaska Native" "1002-5" "Hispanic or Latino" "2135-2" "Female" "F" "15 years" "15"
0 5873 Unreliable
"1999" "1999" "American Indian or Alaska Native" "1002-5" "Hispanic or Latino" "2135-2" "Female" "F" "16 years" "16"
0 5603 Unreliable
"1999" "1999" "American Indian or Alaska Native" "1002-5" "Hispanic or Latino" "2135-2" "Female" "F" "17 years" "17"
0 5375 Unreliable
"1999" "1999" "American Indian or Alaska Native" "1002-5" "Hispanic or Latino" "2135-2" "Female" "F" "18 years" "18"
~

```

```

icd_10_underlying_single_age_year_race_hispanic_gender_assault.txt - Notepad
File Edit Format View Help
"2014" "2014" "white" "2106-3" "Not Stated" "NS" "Male" "M" "93 years" "93" 0 Not Applicable Not Applicable
"2014" "2014" "white" "2106-3" "Not Stated" "NS" "Male" "M" "94 years" "94" 0 Not Applicable Not Applicable
"2014" "2014" "white" "2106-3" "Not Stated" "NS" "Male" "M" "95 years" "95" 0 Not Applicable Not Applicable
"2014" "2014" "white" "2106-3" "Not Stated" "NS" "Male" "M" "96 years" "96" 0 Not Applicable Not Applicable
"2014" "2014" "white" "2106-3" "Not Stated" "NS" "Male" "M" "97 years" "97" 0 Not Applicable Not Applicable
"2014" "2014" "white" "2106-3" "Not Stated" "NS" "Male" "M" "98 years" "98" 0 Not Applicable Not Applicable
"2014" "2014" "white" "2106-3" "Not Stated" "NS" "Male" "M" "99 years" "99" 0 Not Applicable Not Applicable
"2014" "2014" "white" "2106-3" "Not Stated" "NS" "Male" "M" "100+ years" "100" 0 Not Applicable Not Applicable
"2014" "2014" "white" "2106-3" "Not Stated" "NS" "Male" "M" "Not Stated" "NS" 2 Not Applicable Not Applicable
"----"
"Dataset: Underlying Cause of Death, 1999-2014"
"Query Parameters:"
"Title: icd_10_underlying_single_age_year_race_hispanic_gender_assault"
"2013 Urbanization: All"
"Autopsy: All"
"Gender: All"
"Hispanic Origin: All"
"ICD-10 Codes: X85-Y09 (Assault)"
"Place of Death: All"
"Race: All"
"Single-Year Ages: All"
"States: All"
"Weekday: All"
"Year/Month: All"
"Group By: Year, Race, Hispanic Origin, Gender, Single-Year Ages"
"Show Totals: False"
"Show Zero Values: True"
"Show Suppressed: False"
"Calculate Rates Per: 100,000"
"Rate Options: Default intercensal populations for years 2001-2009 (except Infant Age Groups)"
"----"
"Help: See http://wonder.cdc.gov/wonder/help/ucd.html for more information."
"----"
"Query Date: Jan 28, 2016 11:14:03 AM"
"----"
"Suggested Citation: Centers for Disease Control and Prevention, National Center for Health Statistics. Underlying Cause of Death"
"1999-2014 on CDC WONDER online Database, released 2015. Data are from the Multiple Cause of Death Files, 1999-2014, as compiled"
"from data provided by the 57 vital statistics jurisdictions through the Vital Statistics Cooperative Program. Accessed at"
"http://wonder.cdc.gov/ucd-icd10.html on Jan 28, 2016 11:14:03 AM"
"----"
Caveats:
"1. Population and rates are labeled 'Not Applicable' when they include a subset of ages 85-100+ because populations are not"
"available for those ages. More information: http://wonder.cdc.gov/wonder/help/ucd.html#Ages 85-100."
"2. Circumstances in Georgia for the years 2008 and 2009 have resulted in unusually high death counts for the ICD-10 cause of"
"death code R99, "other ill-defined and unspecified causes of mortality." Caution should be used in interpreting these data."
"More information: http://wonder.cdc.gov/wonder/help/ucd.html#Georgia-Reporting-Anomalies."
"3. Circumstances in New Jersey for the year 2009 have resulted in unusually high death counts for the ICD-10 cause of death code"
"R99, "other ill-defined and unspecified causes of mortality" and therefore unusually low death counts in other ICD-10 codes,"
"most notably R95, "Sudden Infant Death Syndrome" and X40-X49, "unintentional poisoning." Caution should be used in"
"interpreting these data. More information: http://wonder.cdc.gov/wonder/help/ucd.html#New-Jersey-Reporting-Anomalies."
"4. Circumstances in California resulted in unusually high death counts for the ICD-10 cause of death code R99, "Other"
"ill-defined and unspecified causes of mortality" for deaths occurring in years 2000 and 2001. Caution should be used in"
"interpreting these data. More information: http://wonder.cdc.gov/wonder/help/ucd.html#California-Reporting-Anomalies."
"5. Death rates are flagged as Unreliable when the rate is calculated with a numerator of 20 or less. More information:"
"http://wonder.cdc.gov/wonder/help/ucd.html#Unreliable."
"6. Deaths of persons with Age "Not Stated" are included in "All" counts and rates, but are not distributed among age groups,"
"so are not included in age-specific counts, age-specific rates or in any age-adjusted rates. More information:"
"http://wonder.cdc.gov/wonder/help/ucd.html#Not Stated."

```

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.

Exercise 6.4: (Optional) Discuss examples of binary and text file, and trailing and leading metadata you may have encountered, with a colleague.

6.4. 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, 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 <i>Statistical packages</i>	foreign::read.dta foreign::read.spss foreign::read.ssd	haven::read_dta haven::read_spss haven::read_sas

<i>Excel</i> ⁹	RODBC::odbcConnectExcel gdata::read.xls xlsReadWrite::read.xls xlsx::read.xlsx	readxl::read_excel
Databases	foreign::read.dbf RODBC::odbcConnect	dplyr::src_sqlite

The Scoping Operator ::

The two colons :: in the above 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.

6.5. 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:

⁹ Not supported in Base R

Information on package 'readr'

Description:

```
Package:      readr
Version:      0.1.1
Title:        Read Tabular Data
Description:   Read flat/tabular text files from disk.
Authors@R:    c( person("Hadley", "Wickham", , "hadley@rstudio.com", c("aut",
"cre")), person("Romain", "Francois", role = "aut"), person("R Core
Team", role = "ctb", comment = "Date time code adapted from R"),
person("RStudio", role = "cph"))
Encoding:     UTF-8
Depends:      R (>= 3.0.2)
LinkingTo:    Rcpp, BH
Imports:      Rcpp (>= 0.11.5), curl
Suggests:     testthat, knitr, DiagrammeR
License:      GPL (>= 2)
VignetteBuilder: knitr
Packaged:     2015-05-29 14:19:20 UTC; ripley
Author:       Hadley Wickham [aut, cre], Romain Francois [aut], R Core Team [ctb]
              (Date time code adapted from R), RStudio [cph]
Maintainer:   ORPHANED
NeedsCompilation: yes
Repository:   CRAN
Date/Publication: 2015-05-29 16:26:58
X-CRAN-Original-Maintainer: Hadley Wickham <hadley@rstudio.com>
X-CRAN-Comment: Orphaned on 2015-05-29 as non-cross-platform calls to iconv() were not
corrected by the original maintainer (and have been by the CRAN team).
Built:        R 3.1.3; x86_64-w64-mingw32; 2015-06-14 06:33:35 UTC; windows
```

Index:

collector	Parse character vectors into typed columns.
count_fields	Count the number of fields in each line of a file.
parse_datetime	Parse a character vector of dates or date times.
problems	Retrieve parsing problems.
read_delim	Read a delimited file into a data frame.
read_file	Read a file into a string.
read_fwf	Read a fixed width file.
read_lines	Read lines from a file or string.
read_log	Read common/combined log file.
read_table	Read text file where columns are separated by whitespace.
type_convert	Re-convert character columns in existing data frame.
write_csv	Save a data frame to a csv file.

Further information is available in the following vignettes in directory
 '\\cfsk18.campus.gla.ac.uk/SSD_Home_Data_X/jm383x/My Documents/R/win-library/3.1/readr/doc':

Exercise 6.5: Open up help for the stringr package and the readr package.

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

Read a delimited file into a data frame.

Description

`read_csv` and `read_tsv` are special cases of the general `read_delim`. They're useful for reading the most common types of flat file data. `read_csv2` uses `;` for separators, instead of `,`. This is common in European countries which use `,` as the decimal separator.

Usage

```
read_delim(file, delim, quote = '"', escape_backslash = TRUE,
  escape_double = FALSE, na = "NA", col_names = TRUE, col_types = NULL,
  skip = 0, n_max = -1, progress = interactive())

read_csv(file, col_names = TRUE, col_types = NULL, na = "NA", skip = 0,
  n_max = -1, progress = interactive())

read_csv2(file, col_names = TRUE, col_types = NULL, na = "NA", skip = 0,
  n_max = -1, progress = interactive())

read_tsv(file, col_names = TRUE, col_types = NULL, na = "NA", skip = 0,
  n_max = -1, progress = interactive())
```

Arguments

<code>file</code>	<p>Either a path to a file, a connection, or literal data (either a single string or a raw vector).</p> <p>Files ending in <code>.gz</code>, <code>.bz2</code>, <code>.xz</code>, or <code>.zip</code> will be automatically uncompressed. Files starting with <code>http://</code>, <code>https://</code>, <code>ftp://</code>, or <code>ftps://</code> will be automatically downloaded.</p> <p>Literal data is most useful for examples and tests. It must contain at least one new line to be recognised as data (instead of a path).</p>
<code>delim</code>	Single character used to separate fields within a record.
<code>quote</code>	Single character used to quote strings.
<code>escape_backslash</code>	Does the file use backslashes to escape special characters? This is more general than <code>escape_double</code> as backslashes can be used to escape the delimiter character, the quote character, or to add special characters like <code>\n</code> .
<code>escape_double</code>	Does the file escape quotes by doubling them? i.e. If this option is <code>TRUE</code> , the value <code>""</code> represents a single quote, <code>\</code> .
<code>na</code>	String to use for missing values.
<code>col_names</code>	<p>Either <code>TRUE</code>, <code>FALSE</code> or a character vector of column names.</p> <p>If <code>TRUE</code>, the first row of the input will be used as the column names, and will not be included in the data frame. If <code>FALSE</code>, column names will be generated automatically: <code>X1</code>, <code>X2</code>, <code>X3</code> etc.</p>

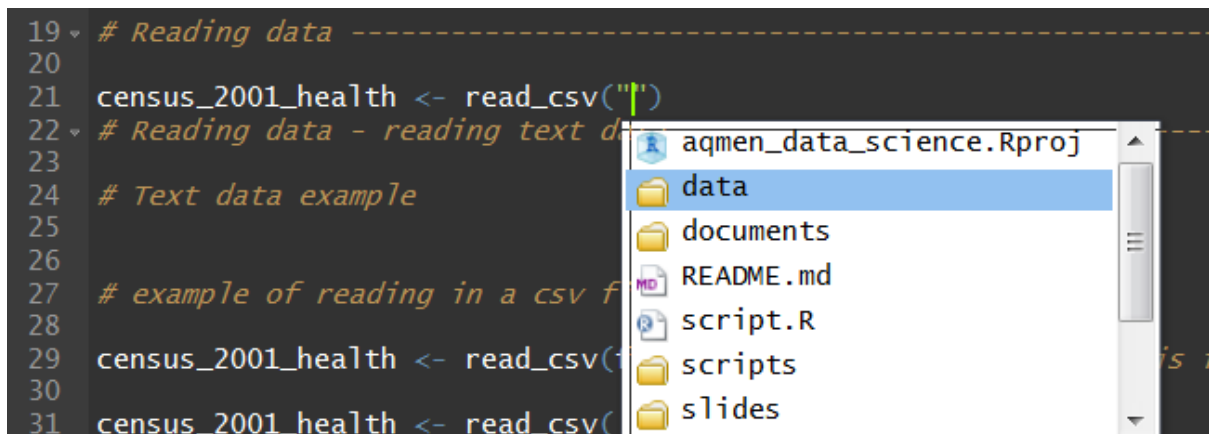
You can see that `read_delim` has 11 arguments. Two of these arguments, `file` and `delim`, do not have default values (indicated by the `=` sign), and so unless both of these arguments are specified the function call will fail. By contrast, you can see that `read_csv` has seven arguments, and of these 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.

6.6. 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:

```
> census_2001_health <- read_csv(file = "data/text/S23.csv")
Error: You have 1 column names, but 16 columns
> |
```

Exercise 6.6: 'Successfully' reproduce this error.

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:

S23.csv - Notepad

File Edit Format View Help

Table S23 SEX AND AGE AND GENERAL HEALTH BY NS-SeC

Table population : All people aged 16 to 74
'Geographical level : Ward to UK

,,,"ALL PEOPLE","Large employers and higher managerial occupations","Higher professional occupations","Lower managerial & professional occupations","Intermediate occupations","Small employers & own account workers","Lower supervisory and technical occupations","Semi-routine occupations","Routine occupations","Never worked","Long-term unemployed","Full -time students","Not classifiable for other reasons "

"SCOTLAND","ALL PEOPLE","ALL AGES",3731079,89354,165600,647288,351251,213577,275865,468922,386376,107857,48204,266483,710302

"SCOTLAND","ALL PEOPLE","16 to 24",566477,2281,11063,44155,58032,6000,36610,85784,65957,25412,4661,224773,1749

"SCOTLAND","ALL PEOPLE","16 to 24 - Good Health",468336,1929,9973,37923,48284,4991,30889,67850,52616,15535,2997,194587,762

"SCOTLAND","ALL PEOPLE","16 to 24 - Fairly Good Health",81109,294,972,5320,8439,837,4800,14776,10881,6294,1374,26535,587

"SCOTLAND","ALL PEOPLE","16 to 24 - Not Good Health",17032,58,118,912,1309,172,921,3158,2460,3583,290,3651,400

"SCOTLAND","ALL PEOPLE","25 to 34",699397,18948,50978,165526,93202,33790,66621,100415,81535,23212,12119,24859,28192

S23.csv - Excel

FILE HOME INSERT PAGE LAYOUT FORMULAS DATA REVIEW VIEW DEVELOPER Jonathan Minton

A1 : Table S23 SEX AND AGE AND GENERAL HEALTH BY NS-SeC

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	Table S23	SEX AND AGE AND GENERAL HEALTH BY NS-SeC											
2													
3		Table population : All people aged 16 to 74											
4		'Geographical level : Ward to UK											
5													
6			ALL PEOPLE	Large emp	Higher pro	Lower ma	Intermedic	Small emp	Lower sup	Semi-routi	Routine oc	Never w	
7	SCOTLAND	ALL PEOPLE	ALL AGES	3731079	89354	165600	647288	351251	213577	275865	468922	386376	10785
8	SCOTLAND	ALL PEOPLE	16 to 24	566477	2281	11063	44155	58032	6000	36610	85784	65957	2541
9	SCOTLAND	ALL PEOPLE	16 to 24 -	468336	1929	9973	37923	48284	4991	30889	67850	52616	1553
10	SCOTLAND	ALL PEOPLE	16 to 24 -	81109	294	972	5320	8439	837	4800	14776	10881	629
11	SCOTLAND	ALL PEOPLE	16 to 24 -	17032	58	118	912	1309	172	921	3158	2460	358
12	SCOTLAND	ALL PEOPLE	25 to 34	699397	18948	50978	165526	93202	33790	66621	100415	81535	2321
13	SCOTLAND	ALL PEOPLE	25 to 34 -	536389	16651	45435	139229	74460	27293	52721	73147	59225	1016
14	SCOTLAND	ALL PEOPLE	25 to 34 -	123638	1942	4661	21759	15150	5357	11083	21428	17151	763
15	SCOTLAND	ALL PEOPLE	25 to 34 -	39370	355	882	4538	3592	1140	2817	5840	5159	541
16	SCOTLAND	ALL PEOPLE	35 to 49	1118333	44280	64482	271495	120479	87366	102371	157641	124737	2489
17	SCOTLAND	ALL PEOPLE	35 to 49 -	775816	37119	54208	213006	89853	65009	73877	106283	79754	881

Exercise 6.7: (Optional) Open the file in notepad and Excel as above.

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:

skip Number of lines to skip before reading data.

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:

```
> census_2001_health <- read_csv(
+   file = "data/text/s23.csv",
+   skip = 5
+ )
=====| 100%      8 MB
warning message:
324653 problems parsing 'data/text/s23.csv'. See problems(...) for more details.
> |
```

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)
```

```
> problems(census_2001_health)
source: local data frame [324,653 x 4]
```

	row	col	expected	actual
1	112	15	an integer	-
2	137	15	an integer	-
3	141	15	an integer	-
4	162	14	an integer	-
5	170	15	an integer	-
6	174	5	an integer	-
7	174	6	an integer	-
8	174	15	an integer	-
9	176	16	an integer	-
10	177	16	an integer	-
..

```
>
```

Exercise 6.8: Reproduce the above code and result

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:

[illegible]

`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.

```
> glimpse(census_2001_health)
observations: 105183
Variables:
$ [EMPTY] (chr) "SCOTLAND", "SCOTLAND", "SCOTLAND", "SCOTLAND", "...
$ [EMPTY] (chr) "ALL PEOPLE", "ALL PEOPLE", "ALL PEOPLE", "ALL PE...
$ [EMPTY] (chr) "ALL AGES", " 16 to 24", "16 to 24 - Good Health"...
$ ALL PEOPLE (int) 3731079, 566477, 468336, 81109, 17032, 699397, 53...
$ Large employers and higher managerial occupations (int) 89354, 2281, 1929, 294, 58, 18948, 16651, 1942, 3...
$ Higher professional occupations (int) 165600, 11063, 9973, 972, 118, 50978, 45435, 4661...
$ Lower managerial & professional occupations (int) 647288, 44155, 37923, 5320, 912, 165526, 139229, ...
$ Intermediate occupations (int) 351251, 58032, 48284, 8439, 1309, 93202, 74460, 1...
$ Small employers & own account workers (int) 213577, 6000, 4991, 837, 172, 33790, 27293, 5357,...
$ Lower supervisory and technical occupations (int) 275865, 36610, 30889, 4800, 921, 66621, 52721, 11...
$ Semi-routine occupations (int) 468922, 85784, 67850, 14776, 3158, 100415, 73147,...
$ Routine occupations (int) 386376, 65957, 52616, 10881, 2460, 81535, 59225, ...
$ Never worked (int) 107857, 25412, 15535, 6294, 3583, 23212, 10166, 7...
$ Long-term unemployed (int) 48204, 4661, 2997, 1374, 290, 12119, 6998, 4111, ...
$ Full-time students (int) 266483, 224773, 194587, 26535, 3651, 24859, 19412...
$ Not classifiable for other reasons (int) 710302, 1749, 762, 587, 400, 28192, 11652, 8882, ...
```

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.

Routine occupations	Never worked	Long-term unemployed	Full-time students	Not classifiable for other reasons
218	56	18	1	531
1581	160	62	14	2384
764	52	22	12	865
577	65	32	2	822
240	43	8	NA	697
463	473	29	9	15616
265	132	11	5	6476
168	215	12	3	6199

Exercise 6.9: Use View to open up the loaded file contents in R, identifying NAs like those above.

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:

```
> census_2001_health <- read_csv(
+   file = "data/text/s23.csv",
+   skip = 5,
+   na = "-"
+ )
> |
```

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

```
> problems(census_2001_health)
[1] row      col      expected actual
<0 rows> (or 0-length row.names)
```

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).

Exercise 6.10: Reproduce the above code and results.

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`:

```
> census_2001_health <- read_csv(
+   file = "data/text/S23.csv",
+   skip = 5,
+   col_types = paste0(rep("c", 16), collapse = "")
+ )
> |
```

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.

The `paste0` function

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.

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


```
> glimpse(census_2001_health)
observations: 105183
variables:
$ [EMPTY] (chr) "SCOTLAND", "SCOTLAND", "SCOTLAND", "SCOTLAND", "SCOTLAND",...
$ [EMPTY] (chr) "ALL PEOPLE", "ALL PEOPLE", "ALL PEOPLE", "ALL PEOPLE", "AL...
$ [EMPTY] (chr) "ALL AGES", " 16 to 24", "16 to 24 - Good Health", "16 to 2...
$ ALL PEOPLE (chr) "3731079", "566477", "468336", "81109", "17032", "699397", ...
$ Large employers and higher managerial occupations (chr) "89354", "2281", "1929", "294", "58", "18948", "16651", "19...
$ Higher professional occupations (chr) "165600", "11063", "9973", "972", "118", "50978", "45435", ...
$ Lower managerial & professional occupations (chr) "647288", "44155", "37923", "5320", "912", "165526", "13922...
$ Intermediate occupations (chr) "351251", "58032", "48284", "8439", "1309", "93202", "74460...
$ Small employers & own account workers (chr) "213577", "6000", "4991", "837", "172", "33790", "27293", "...
$ Lower supervisory and technical occupations (chr) "275865", "36610", "30889", "4800", "921", "66621", "52721"...
$ Semi-routine occupations (chr) "468922", "85784", "67850", "14776", "3158", "100415", "731...
$ Routine occupations (chr) "386376", "65957", "52616", "10881", "2460", "81535", "5922...
$ Never worked (chr) "107857", "25412", "15535", "6294", "3583", "23212", "10166...
$ Long-term unemployed (chr) "48204", "4661", "2997", "1374", "290", "12119", "6998", "4...
$ Full-time students (chr) "266483", "224773", "194587", "26535", "3651", "24859", "19...
$ Not classifiable for other reasons (chr) "710302", "1749", "762", "587", "400", "28192", "11652", "8...
```

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.

Exercise 6.11: Reproduce the above code and results.

6.7. Reading Binary Files

6.7.1. 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.

```
> lookup <- read_excel(
+   path = "data/binary/replication_details.xlsx",
+   sheet = "code_to_country_lookup"
+ )
>
> lookup
Source: local data frame [46 x 4]

   code    long_name in_original_selection include_in_full_selection
1  AUS    Australia           0                1
2  AUT     Austria           1                1
3  BEL     Belgium           0                1
4  BGR     Bulgaria          1                1
5  BLR     Belarus           0                1
6  CAN     Canada            1                1
7  CHE     Switzerland        1                1
8  CHL     Chile             0                1
9  CZE     Czech Republic     1                1
10 DEUTE   East Germany        1                1
..   ...                ...                ...
```

```

>
> problems(lookup)
[1] row      col      expected actual
<0 rows> (or 0-length row.names)
>
> glimpse(lookup)
Observations: 46
Variables:
 $ code                (chr) "AUS", "AUT", "BEL", "BGR", "BLR", "CAN", "...
 $ long_name           (chr) "Australia", "Austria", "Belgium", "Bulgari...
 $ in_original_selection (dbl) 0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0...
 $ include_in_full_selection (dbl) 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1...

```

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.

Exercise 6.12: Open up the excel file in excel and location the `code_to_country_lookup` tab. Note why this tab's contents are likely to load correctly in R while other tabs may not. Successfully reproduce the code above.

6.7.2. 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 as 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:


```

> poverty
Source: local data frame [97 x 11]

  birthrate deathrate infant_death_rate male_life_exp female_life_exp GNP country_group country GNP_thousands
1    24.7      5.7         30.8         69.6         75.5      600      1 Albania      0.600
2    12.5     11.9         14.4         68.3         74.7     2250      1 Bulgaria     2.250
3    13.4     11.7         11.3         71.8         77.7     2980      1 Czechoslovakia 2.980
4    12.0     12.4          7.6         69.8         75.9      NA      1 Former_E_Germany NA
5    11.6     13.4         14.8         65.4         73.8     2780      1 Hungary     2.780
6    14.3     10.2         16.0         67.2         75.7     1690      1 Poland      1.690
7    13.6     10.7         26.9         66.5         72.4     1640      1 Romania     1.640
8    14.0      9.0         20.2         68.6         74.5      NA      1 Yugoslavia    NA
9    17.7     10.0         23.0         64.6         74.0     2242      1 USSR         2.242
10   15.2      9.5         13.1         66.4         75.9     1880      1 Byelorussian_SSR 1.880
..    ...      ...          ...          ...          ...      ...      ...
Variables not shown: high_deathrate (lbl), log10gnp (dbl)
> poverty %>% glimpse
observations: 97
variables:
 $ birthrate      (dbl) 24.7, 12.5, 13.4, 12.0, 11.6, 14.3, 13.6, 14.0, 17.7, 15.2, 13.4, 20.7, 46.6, 28.6, 23.4, 27.4, 32.9...
 $ deathrate      (dbl) 5.7, 11.9, 11.7, 12.4, 13.4, 10.2, 10.7, 9.0, 10.0, 9.5, 11.6, 8.4, 18.0, 7.9, 5.8, 6.1, 7.4, 7.3, 6...
 $ infant_death_rate (dbl) 30.8, 14.4, 11.3, 7.6, 14.8, 16.0, 26.9, 20.2, 23.0, 13.1, 13.0, 25.7, 111.0, 63.0, 17.1, 40.0, 63.0...
 $ male_life_exp   (dbl) 69.6, 68.3, 71.8, 69.8, 65.4, 67.2, 66.5, 68.6, 64.6, 66.4, 66.4, 65.5, 51.0, 62.3, 68.1, 63.4, 63.4...
 $ female_life_exp (dbl) 75.5, 74.7, 77.7, 75.9, 73.8, 75.7, 72.4, 74.5, 74.0, 75.9, 74.8, 72.7, 55.4, 67.6, 75.1, 69.2, 67.6...
 $ GNP             (dbl) 600, 2250, 2980, NA, 2780, 1690, 1640, NA, 2242, 1880, 1320, 2370, 630, 2680, 1940, 1260, 980, 330, ...
 $ country_group   (lbl) 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, ...
 $ country         (chr) "Albania", "Bulgaria", "Czechoslovakia", "Former_E_Germany", "Hungary", "Poland", "Romania", "Yugos...
 $ GNP_thousands   (dbl) 0.600, 2.250, 2.980, NA, 2.780, 1.690, 1.640, NA, 2.242, 1.880, 1.320, 2.370, 0.630, 2.680, 1.940, 1...
 $ high_deathrate   (lbl) 1, 2, 2, 2, 2, 2, 2, 1, 1, 1, 2, 1, 2, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 1, 2, 1, 1, 1, 1, 2, ...
 $ log10gnp         (dbl) 2.778151, 3.352183, 3.474216, NA, 3.444045, 3.227887, 3.214844, NA, 3.350636, 3.274158, 3.120574, 3...

```

Two variables, `country_group` and `high_deathrate`, have a variable type that has not been encountered before, called 'lbl'. 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:

```

> poverty <- poverty %>%
+   mutate(
+     country_group = as_factor(country_group, labels = "values"),
+     high_deathrate = as_factor(high_deathrate, labels = "values")
+   )
> poverty
source: local data frame [97 x 11]
   birthrate deathrate infant_death_rate male_life_exp female_life_exp GNP country_group country GNP_thousands high_deathrate log10gnp
1    24.7      5.7      30.8      69.6      75.5    600 Eastern Europe & USSR Albania      0.600      No 2.778151
2    12.5     11.9      14.4      68.3      74.7   2250 Eastern Europe & USSR Bulgaria    2.250     Yes 3.352183
3    13.4     11.7      11.3      71.8      77.7   2980 Eastern Europe & USSR Czechoslovakia 2.980     Yes 3.474216
4    12.0     12.4       7.6      69.8      75.9    NA Eastern Europe & USSR Former_E_Germany NA      Yes NA
5    11.6     13.4      14.8      65.4      73.8   2780 Eastern Europe & USSR Hungary      2.780     Yes 3.444045
6    14.3     10.2      16.0      67.2      75.7   1690 Eastern Europe & USSR Poland      1.690     Yes 3.227887
7    13.6     10.7      26.9      66.5      72.4   1640 Eastern Europe & USSR Romania     1.640     Yes 3.214844
8    14.0      9.0      20.2      68.6      74.5    NA Eastern Europe & USSR Romania     NA      No NA
9    17.7     10.0      23.0      64.6      74.0   2242 Eastern Europe & USSR USSR      2.242     No 3.350636
10   15.2      9.5      13.1      66.4      75.9   1880 Eastern Europe & USSR Byelorussian_SSR 1.880     No 3.274158
..    ...      ...      ...      ...      ...    ...    ...    ...    ...    ...
> glimpse(poverty)
# A tibble: 97 x 11
#   birthrate deathrate infant_death_rate male_life_exp female_life_exp GNP country_group country GNP_thousands high_deathrate log10gnp
#   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <fctr> <chr> <dbl> <fctr> <dbl>
#1 24.7 5.7 30.8 69.6 75.5 600 Eastern Europe & USSR Albania 0.600 No 2.778151
#2 12.5 11.9 14.4 68.3 74.7 2250 Eastern Europe & USSR Bulgaria 2.250 Yes 3.352183
#3 13.4 11.7 11.3 71.8 77.7 2980 Eastern Europe & USSR Czechoslovakia 2.980 Yes 3.474216
#4 12.0 12.4 7.6 69.8 75.9 NA Eastern Europe & USSR Former_E_Germany NA Yes NA
#5 11.6 13.4 14.8 65.4 73.8 2780 Eastern Europe & USSR Hungary 2.780 Yes 3.444045
#6 14.3 10.2 16.0 67.2 75.7 1690 Eastern Europe & USSR Poland 1.690 Yes 3.227887
#7 13.6 10.7 26.9 66.5 72.4 1640 Eastern Europe & USSR Romania 1.640 Yes 3.214844
#8 14.0 9.0 20.2 68.6 74.5 NA Eastern Europe & USSR Romania NA No NA
#9 17.7 10.0 23.0 64.6 74.0 2242 Eastern Europe & USSR USSR 2.242 No 3.350636
#10 15.2 9.5 13.1 66.4 75.9 1880 Eastern Europe & USSR Byelorussian_SSR 1.880 No 3.274158
#> glimpse(poverty)
# A tibble: 97 x 11
#   birthrate deathrate infant_death_rate male_life_exp female_life_exp GNP country_group country GNP_thousands high_deathrate log10gnp
#   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <fctr> <chr> <dbl> <fctr> <dbl>
#1 24.7 5.7 30.8 69.6 75.5 600 Eastern Europe & USSR Albania 0.600 No 2.778151
#2 12.5 11.9 14.4 68.3 74.7 2250 Eastern Europe & USSR Bulgaria 2.250 Yes 3.352183
#3 13.4 11.7 11.3 71.8 77.7 2980 Eastern Europe & USSR Czechoslovakia 2.980 Yes 3.474216
#4 12.0 12.4 7.6 69.8 75.9 NA Eastern Europe & USSR Former_E_Germany NA Yes NA
#5 11.6 13.4 14.8 65.4 73.8 2780 Eastern Europe & USSR Hungary 2.780 Yes 3.444045
#6 14.3 10.2 16.0 67.2 75.7 1690 Eastern Europe & USSR Poland 1.690 Yes 3.227887
#7 13.6 10.7 26.9 66.5 72.4 1640 Eastern Europe & USSR Romania 1.640 Yes 3.214844
#8 14.0 9.0 20.2 68.6 74.5 NA Eastern Europe & USSR Romania NA No NA
#9 17.7 10.0 23.0 64.6 74.0 2242 Eastern Europe & USSR USSR 2.242 No 3.350636
#10 15.2 9.5 13.1 66.4 75.9 1880 Eastern Europe & USSR Byelorussian_SSR 1.880 No 3.274158

```

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 `lbl` 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.

Exercise 6.13: (Optional) Load the poverty dataset successfully both within R and with SPSS. Try to load the dataset into R using the appropriate functions in two different packages, `foreign` and `haven`, and explore the differences in function arguments and outputs.

6.8. 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.

7. Initial data tidying

Exercise 7.1: Make sure working code used previously has been saved in your script.R file, and check that it works correctly if run in the sequence presented.

7.1. 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.

7.2. 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:

```
> dta <- data.frame(
+   place = rep(c("A", "B"), each = 3),
+   time = 1:3,
+   count = c(4, "-", 1, 6, 2, 4)
+ ) %>% tbl_df
> dta
Source: local data frame [6 x 3]

  place time count
1     A     1     4
2     A     2     -
3     A     3     1
4     B     1     6
5     B     2     2
6     B     3     4
> glimpse(dta)
observations: 6
variables:
 $ place (fctr) A, A, A, B, B, B
 $ time  (int) 1, 2, 3, 1, 2, 3
 $ count (fctr) 4, -, 1, 6, 2, 4
>
```

Exercise 7.2: Create the dta object as above.

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,¹⁰ 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:

¹⁰ 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.

```

> dta2 <- dta %>%
+   mutate(
+     count2 = as.numeric(str_replace(count, "-", "0"))
+   )
> dta2
Source: local data frame [6 x 4]

  place time count count2
1     A   1     4      4
2     A   2     -      0
3     A   3     1      1
4     B   1     6      6
5     B   2     2      2
6     B   3     4      4
> glimpse(dta2)
Observations: 6
Variables:
$ place (fctr) A, A, A, B, B, B
$ time  (int) 1, 2, 3, 1, 2, 3
$ count (fctr) 4, -, 1, 6, 2, 4
$ count2 (dbl) 4, 0, 1, 6, 2, 4
>

```

Exercise 7.3: Reproduce the above code and outcome.

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:

Replace first occurrence of a matched pattern in a string.

Description

Vectorised over `string`, `pattern` and `replacement`. Shorter arguments will be expanded to length of longest.

Usage

```
str_replace(string, pattern, replacement)
```

Arguments

- replacement** replacement string. References of the form `\1`, `\2` will be replaced with the contents of the respective matched group (created by `()`) within the pattern.
- string** input vector. This must be an atomic vector, and will be coerced to a character vector
- pattern** pattern to look for, as defined by a POSIX regular expression. See the "Extended Regular Expressions" section of [regex](#) for details. See [fixed](#), [ignore.case](#) and [perl](#) for how to use other types of matching: fixed, case insensitive and perl-compatible.

Value

character vector.

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:

```

> dta <- dta %>%
+   mutate(
+     count = as.numeric(str_replace(count, "-", "0"))
+   )
> dta
Source: local data frame [6 x 3]

  place time count
1     A     1     4
2     A     2     0
3     A     3     1
4     B     1     6
5     B     2     2
6     B     3     4
> glimpse(dta)
Observations: 6
Variables:
$ place (fctr) A, A, A, B, B, B
$ time  (int) 1, 2, 3, 1, 2, 3
$ count (dbl) 4, 0, 1, 6, 2, 4
> |

```

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:

```

change_dash_to_zero <- function(input){
  output <- input %>%
    str_replace("-", "0") %>%
    as.numeric

  return(output)
}

```

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:

```

> dta <- dta %>%
+   mutate(
+     count = change_dash_to_zero(count)
+   )
> dta
Source: local data frame [6 x 3]

  place time count
1     A     1     4
2     A     2     0
3     A     3     1
4     B     1     6
5     B     2     2
6     B     3     4
> glimpse(dta)
Observations: 6
Variables:
$ place (fctr) A, A, A, B, B, B
$ time   (int) 1, 2, 3, 1, 2, 3
$ count  (dbl) 4, 0, 1, 6, 2, 4

```

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.

Exercise 7.4: Create a user-defined function `change_dash_to_five` which very efficiently messes up all subsequent analyses.

7.3. Example 2: Removing commas and removing whitespace

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

```

> dta <- data.frame(
+   sex = c("male", "male ", " female", "female"),
+   value = c("500", "10 000", "40,000", "25")
+ ) %>% tbl_df
> dta
Source: local data frame [4 x 2]

  sex    value
1 male    500
2 male 10 000
3 female 40,000
4 female   25
> glimpse(dta)
Observations: 4
Variables:
$ sex   (fctr) male, male , female, female
$ value (fctr) 500, 10 000, 40,000, 25

```

Exercise 7.5: Reproduce the above.

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.¹¹ 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:

```
clean_values <- function(input){  
  output <- input %>%  
    str_replace(",", "") %>%  
    str_replace(" ", "") %>%  
    as.numeric  
  
  return(output)  
}
```

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.

Exercise 7.6: Produce the above user-defined function. Reflect on whether the order of the lines within the function affects the function's functionality.

We can test the code as follows:

```
> dta2 <- dta %>%  
+   mutate(  
+     sex2 = str_trim(sex),  
+     value2 = clean_values(value)  
+   )  
>  
> dta2  
Source: local data frame [4 x 4]  
  
   sex  value  sex2 value2  
1  male    500  male    500  
2  male 10 000  male 10000  
3 female 40,000 female 40000  
4 female    25 female    25  
> glimpse(dta2)  
Observations: 4  
variables:  
$ sex      (fctr) male, male , female, female  
$ value    (fctr) 500, 10 000, 40,000, 25  
$ sex2     (chr) "male", "male", "female", "female"  
$ value2   (dbl) 500, 10000, 40000, 25  
> |
```

¹¹ 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.

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:

```
trim_and_factorise <- function(input){
  output <- input %>%
    str_trim %>%
    factor

  return(output)
}
```

Exercise 7.7: Reproduce the above code and results.

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

```
> dta2 <- dta %>%
+   mutate(
+     sex2 = trim_and_factorise(sex),
+     value2 = clean_values(value),
+     test1 = value > 2000,
+     test2 = value2 > 2000
+   )
Warning message:
In Ops.factor(c(4L, 1L, 3L, 2L), 2000) : '>' not meaningful for factors
> dta2
Source: local data frame [4 x 6]

   sex  value  sex2 value2 test1 test2
1 male    500 male    500    NA FALSE
2 male 10 000 male 10000    NA  TRUE
3 female 40,000 female 40000    NA  TRUE
4 female    25 female    25    NA FALSE
> glimpse(dta2)
observations: 4
variables:
$ sex      (fctr) male, male , female, female
$ value    (fctr) 500, 10 000, 40,000, 25
$ sex2     (fctr) male, male, female, female
$ value2   (dbl) 500, 10000, 40000, 25
$ test1    (lg1) NA, NA, NA, NA
$ test2    (lg1) FALSE, TRUE, TRUE, FALSE
> |
```

Exercise 7.8: Reproduce the above code and results.

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`.

```
> dta <- dta %>%
+   mutate(
+     sex = trim_and_factorise(sex),
+     value = clean_values(value)
+   )
> dta
Source: local data frame [4 x 2]

   sex value
1  male   500
2  male 10000
3 female 40000
4 female    25
> glimpse(dta)
Observations: 4
Variables:
$ sex (fctr) male, male, female, female
$ value (dbl) 500, 10000, 40000, 25
> |
```

Exercise 7.9: Reproduce the above code and results.

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/>

Exercise 36: (Optional) Explore the above websites and produce your own regex pattern.

7.4. 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.

Making, breaking and testing pipes

Throughout the extended example below, I will demonstrate a process of iterative testing and developing that I use frequently. I start with an initial input object, in this case `census_2001_health`, and make one change to it at a time. After making this change, I test the new output, to see whether the change is what I expected it to be. I either do this simply by allowing the end of the pipe to 'spill onto' the Console, revealing information about the dimensions of the resulting table, and the contents of the first few rows; or apply one of a number of 'pipe tester' functions, predominantly the 'glimpse' function from within `dplyr`, or the `xtabs` function within base R. Only once the initial data object has reached its target form do I then tend to save the output into a new object. This approach of iterative and incremental building and testing can be surprisingly powerful and easy to apply to a large number of tasks. Rather than creating a large number of objects that carry the results of intermediate project, my R workshop typically contains just two: the initial and final object; or one: the final object, over-written onto the name of the initial object. By having fewer objects in the workspace, there are usually fewer issues with running out of memory and so less chance of R sessions crashing. The only exception to this general pattern of pipe building is if one or more of these intermediate stages is very time and resource intensive, and for example takes a number of minutes or hours to complete; in these cases it is prohibitive to re-run this stage multiple times, and so this intermediate output should be saved.

This pattern of iterative code pipe building can be reversed when looking at and testing code that has been written in a piped format. With the completed code chunk shown above, for example, it is easy to identify the initial input object, and highlight and run that section of the code only, displaying the input on the R console; I could then extend the highlighted selection to include the pipe operator to its right, and all of the code that follows until the next pipe operator, revealing the first intermediate output. I would then continue this process of testing ever longer sections of the pipe until I have reached the end of it, revealing the full process of constructing the output object. For example, using the code chunk above, I would first run:

```
census_2001_health
```

Then followed by:

```
census_2001_health %>%  
  gather("occupational_group", "count", -place, -sex, -age_and_health)
```

And then by:

```
census_2001_health %>%  
  gather("occupational_group", "count", -place, -sex, -age_and_health) %>%  
  separate(age_and_health, into = c("age", "health"), sep = "-", extra = "drop")
```

And so on.

Comments can be added to each line, at the end of each pipe, using the `#` symbol. It is very easy to under-comment code, but it is also possible to over-comment code. By writing code in a way that is clearly laid out and sequential, and using intuitively meaningful function names and object names, the code itself can in a sense become 'self describing', clear enough to other R users, and to yourself in a few months' or years' time, that only a few additional comments are required. One of the advantages of RStudio's autocomplete features is that it reduces the time penalty associated with using longer function and object names, as you can simply tab to select between objects in the R workspace; this means there are fewer excuses to using short but non-descriptive object and function names such as 'data', 'fn', 'tmp', 'stuff', 'xx' and so on.

8. The Tidy Data Twins: tidyr and dplyr

8.1. 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.

8.2. 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)

Exercise 8.1: (Optional) Read the Tidy Data article above. Think about a dataset you have worked with or are working with, and whether it is currently in a tidy data format, and if not which processes and operations are required to get it into a tidy data format.

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 event 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.

8.3. 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` introduced earlier, but now 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:

	[EMPTY]	[EMPTY]	[EMPTY]	ALL PEOPLE	Large employers and higher managerial occupations	Higher professional occupations	Lower managerial & professional occupations	Intermediate occupations	Small employers & own account workers	Lower supervisory and technical occupations	Semi-routine occupations	Routine occupations	Never work
1	SCOTLAND	ALL PEOPLE	ALL AGES	3731079	89354	165600	647288	351251	213577	275865	468922	386376	10781
2	SCOTLAND	ALL PEOPLE	16 to 24	566477	2281	11063	44155	58032	6000	36610	85784	65957	25411
3	SCOTLAND	ALL PEOPLE	16 to 24 - Good Health	468336	1929	9973	37923	48284	4991	30889	67850	52616	15531
4	SCOTLAND	ALL PEOPLE	16 to 24 - Fairly Good Health	81109	294	972	5320	8439	837	4800	14776	10881	6294
5	SCOTLAND	ALL PEOPLE	16 to 24 - Not Good Health	17032	58	118	912	1309	172	921	3158	2460	3583
6	SCOTLAND	ALL PEOPLE	25 to 34	699397	18948	50978	165526	93202	33790	66621	100415	81535	23211
7	SCOTLAND	ALL PEOPLE	25 to 34 - Good Health	536389	16651	45435	139229	74460	27293	52721	73147	59225	10161
8	SCOTLAND	ALL PEOPLE	25 to 34 - Fairly Good Health	123638	1942	4661	21759	15150	5357	11083	21428	17151	7635
9	SCOTLAND	ALL PEOPLE	25 to 34 - Not Good Health	39370	355	882	4538	3592	1140	2817	5840	5159	5411
10	SCOTLAND	ALL PEOPLE	35 to 49	1118333	44280	64482	271495	120479	87366	102371	157641	124737	24891
11	SCOTLAND	ALL PEOPLE	35 to 49 - Good Health	775816	37119	54208	213006	89853	65009	73877	106283	79754	8817
12	SCOTLAND	ALL PEOPLE	35 to 49 - Fairly Good Health	238862	5832	8392	46058	24077	17439	21804	38941	32993	8538
13	SCOTLAND	ALL PEOPLE	35 to 49 - Not Good Health	103655	1329	1882	12431	6549	4918	6690	12417	11990	7539
14	SCOTLAND	ALL PEOPLE	50 to 54	351107	11456	16772	77578	34651	31852	29997	49844	41719	6150
15	SCOTLAND	ALL PEOPLE	55 to 59 - Good Health	211733	8051	12210	55000	23550	21502	10020	20150	12410	1810

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")12
```

¹² This 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.

```

> glimpse(census_2001_health)
Observations: 105183
Variables:
$ place              (chr) "SCOTLAND", "SCOTLAND", "SCOTLAND", "SCOTLAND", "SCOTLAND", "SCOTLAND", "SCOTLAND", "SCOTLAND", "SCOTLAND."
$ sex                (chr) "ALL PEOPLE", "ALL PEOPLE", "ALL PEOPLE", "ALL PEOPLE", "ALL PEOPLE", "ALL PEOPLE", "ALL PEOPLE", "ALL PEOPLE", "ALL PEOP..."
$ age_and_health     (chr) "ALL AGES", "16 to 24", "16 to 24 - Good Health", "16 to 24 - Fairly Good Health", "16 to 24..."
$ ALL PEOPLE         (chr) "3731079", "566477", "468336", "81109", "17032", "699397", "536389", "123638", "39370", "1118..."
$ Large employers and higher managerial occupations (chr) "89354", "2281", "1929", "294", "58", "18948", "16651", "1942", "355", "44280", "37119", "583..."
$ Higher professional occupations (chr) "165600", "11063", "9973", "972", "118", "50978", "54535", "4661", "882", "64482", "54208", "..."
$ Lower managerial & professional occupations (chr) "647288", "44155", "37923", "5320", "912", "165526", "139229", "21759", "4538", "271495", "21..."
$ Intermediate occupations (chr) "351251", "58032", "48284", "8439", "1309", "93202", "74460", "15150", "3592", "120479", "898..."
$ Small employers & own account workers (chr) "21357", "6610", "37888", "400", "172", "33790", "27293", "5357", "40", "87366", "65009", "..."
$ Supervisory and technical occupations (chr) "27585", "6610", "30888", "4800", "921", "66621", "52721", "10383", "28181", "10371", "7387..."
$ Semi-routine occupations (chr) "468922", "58784", "67850", "14776", "3158", "100415", "73147", "21428", "5840", "157641", "1..."
$ Routine occupations (chr) "386376", "65957", "52616", "10881", "2460", "81535", "59225", "17151", "5159", "124737", "79..."
$ Never worked (chr) "107857", "25412", "15535", "6294", "3583", "23212", "10166", "7635", "5411", "24894", "8817", "..."
$ Long-term unemployed (chr) "48204", "4661", "2997", "1374", "290", "12119", "6998", "4111", "1010", "18608", "9788", "67..."
$ Full-time students (chr) "266483", "242773", "194587", "26535", "3651", "24859", "19412", "4479", "968", "14408", "102..."
$ Not classifiable for other reasons (chr) "710302", "1749", "762", "587", "400", "28192", "11652", "8882", "7658", "87572", "27900", "2..."

```

Exercise 8.3 (Optional) **Without reading any further**, consider whether the data above is currently in a tidy data format, and if not what a tidy data structure for this data would look like.

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

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.

Exercise 8.4: Open up help for the `gather` function in the `tidyr` package. Explore the help file and consider why this function is useful for this particular dataset.


```
> census_2001_health %>%
+   gather("occupational_group", "count", -place, -sex, -age_and_health)
Source: local data frame [1,367,379 x 5]
```

	place	sex	age_and_health	occupational_group	count
1	SCOTLAND	ALL	PEOPLE	ALL AGES	ALL PEOPLE 3731079
2	SCOTLAND	ALL	PEOPLE	16 to 24	ALL PEOPLE 566477
3	SCOTLAND	ALL	PEOPLE	16 to 24 - Good Health	ALL PEOPLE 468336
4	SCOTLAND	ALL	PEOPLE	16 to 24 - Fairly Good Health	ALL PEOPLE 81109
5	SCOTLAND	ALL	PEOPLE	16 to 24 - Not Good Health	ALL PEOPLE 17032
6	SCOTLAND	ALL	PEOPLE	25 to 34	ALL PEOPLE 699397
7	SCOTLAND	ALL	PEOPLE	25 to 34 - Good Health	ALL PEOPLE 536389
8	SCOTLAND	ALL	PEOPLE	25 to 34 - Fairly Good Health	ALL PEOPLE 123638
9	SCOTLAND	ALL	PEOPLE	25 to 34 - Not Good Health	ALL PEOPLE 39370
10	SCOTLAND	ALL	PEOPLE	35 to 49	ALL PEOPLE 1118333
..

Exercise 8.5: Reproduce the above analyses and results.

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 think I can make use of a consistency in how these combined age and health terms are labelled: the age group is listed, followed by the '-' symbol, and then the health status. The '-' character is a pattern that regex can be told to look out for; once this pattern has been identified, the next step is to separate the contents of the age_and_health column into two separate columns, age and health, with the age column containing the cell contents before the '-' symbol, and the health column containing the cell contents after the '-' symbol. The separate function within tidyr is designed to do just this.

Exercise 8.6: Open up and explore the help file for the separate function.

The first four arguments to separate are as follows

`separate(data, col, into, sep)`

The first argument is the input dataframe, so receives the contents of the pipe and does not need to be specified formally when using piped code. The second argument, 'col', is the name of the column you want to split; the third argument takes the names of two or more columns that received the split or separated contents from column named in col; and the forth argument defines the symbol or pattern that should form the data split.

Given the above, I add to the pipe as follows:


```
> census_2001_health %>%
+   gather("occupational_group", "count", -place, -sex, -age_and_health) %>%
+   separate(age_and_health, into = c("age", "health"), sep = "-")
Error: values not split into 2 pieces at 1, 2, 6, 10, 14, 18, 22, 26, 30, 31, 35, 39, 43, 47, 51, 55, 59, 60, 64, 68, 72, 76, 80, 84, 88, 89, 93,
97, 101, 105, 109, 113, 117, 118, 122, 126, 130, 134, 138, 142, 146, 147, 151, 155, 159, 163, 167, 171, 175, 176, 180, 184, 188, 192, 196, 200,
204, 205, 209, 213, 217, 221, 225, 229, 233, 234, 238, 242, 246, 250, 254, 258, 262, 263, 267, 271, 275, 279, 283, 287, 291, 292, 296, 300, 304,
308, 312, 316, 320, 321, 325, 329, 333, 337, 341, 345, 349, 350, 354, 358, 362, 366, 370, 374, 378, 379, 383, 387, 391, 395, 399, 403, 407, 408,
412, 416, 420, 424, 428, 432, 436, 437, 441, 445, 449, 453, 457, 461, 465, 466, 470, 474, 478, 482, 486, 490, 494, 495, 499, 503, 507, 511, 515,
519, 523, 524, 528, 532, 536, 540, 544, 548, 552, 553, 557, 561, 565, 569, 573, 577, 581, 582, 586, 590, 594, 598, 602, 606, 610, 611, 615, 619,
623, 627, 631, 635, 639, 640, 644, 648, 652, 656, 660, 664, 668, 669, 673, 677, 681, 685, 689, 693, 697, 698, 702, 706, 710, 714,
```

Exercise 8.7: Reproduce the above code and result.

Rather than separating the column as expected, however, this code produced an error message, saying that a large number of values could not be split: it seems rows 1 and 2 could not be split, but rows 3, 4, and 5 could be split. This error message provides a useful clue if we refer back to the previous output, which shows that the contents of the health_and_age column for the first five rows is:

1. ALL AGES
2. 16 to 24
3. 16 to 24 – Good Health
4. 16 to 24 – Fairly Good Health
5. 16 to 24 – Not Good Health

The rows that failed therefore did not contain the '-' symbol that separate was looking for, and so the function did not know how to split these particular rows. There are at least two ways to resolve this issue:

1. Filter away all rows that do not contain the '-' symbol in the age_and_health column¹³
2. Allow for separate to run even if it were not able to separate any particular row of the column of interest.

If I look at the help file for the separate function, I can see information about the following argument:

extra If `sep` is a character vector, this controls what happens when the number of pieces doesn't match `into`. There are three valid options:

- "error" (the default): throws error if pieces aren't right length
- "drop": always returns `length(into)` pieces by dropping or expanding as necessary
- "merge": only splits at most `length(into)` times

Modifying this extra argument therefore allows the behaviour of separate to be changed. Here is the piped function if this argument is set to 'drop':

```
> census_2001_health %>%
+   gather("occupational_group", "count", -place, -sex, -age_and_health) %>%
+   separate(age_and_health, into = c("age", "health"), sep = "-", extra = "drop")
Source: local data frame [1,367,379 x 6]
```

	place	sex	age	health	occupational_group	count
1	SCOTLAND	ALL PEOPLE	ALL AGES	NA	ALL PEOPLE	3731079
2	SCOTLAND	ALL PEOPLE	16 to 24	NA	ALL PEOPLE	566477
3	SCOTLAND	ALL PEOPLE	16 to 24	Good Health	ALL PEOPLE	468336
4	SCOTLAND	ALL PEOPLE	16 to 24	Fairly Good Health	ALL PEOPLE	81109
5	SCOTLAND	ALL PEOPLE	16 to 24	Not Good Health	ALL PEOPLE	17032
6	SCOTLAND	ALL PEOPLE	25 to 34	NA	ALL PEOPLE	699397
7	SCOTLAND	ALL PEOPLE	25 to 34	Good Health	ALL PEOPLE	536389
8	SCOTLAND	ALL PEOPLE	25 to 34	Fairly Good Health	ALL PEOPLE	123638
9	SCOTLAND	ALL PEOPLE	25 to 34	Not Good Health	ALL PEOPLE	39370
10	SCOTLAND	ALL PEOPLE	35 to 49	NA	ALL PEOPLE	1118333
..

¹³ Completing this task using this approach is left as an optional exercise for you to complete.

As it happens, in this example changing the argument to 'merge' produces exactly the same output: when the split symbol ('-') has not been identified, the second 'into' column is just left empty, meaning its cell contents becomes 'NA' (Not Applicable). This is not exactly what we want, but we are closer than we were.

My next stage in tidying the age and health columns is to remove those rows with missing data. These are the cells marked NA in the health column. These can be removed using the filter function, which takes as its argument one or more conditions which rows in columns must satisfy in order for the row to be returned. Non-R users with some coding experience might assume that, as we are searching for those rows where the health cell does not contain NA, the filter criterion would be something like the following:

```
health != NA
```

Where the ! symbol indicates 'not'. (i.e. "health is not equal to NA".) However the NA value is interpreted differently to most values by R, and instead checking whether a cell contains NA as its contents requires a slightly different approach, usually using the is.na() function. This function returns TRUE where cells contain NA, and FALSE otherwise. What we want, however, is the opposite of this result, so we want to take the output of is.na(), and reverse TRUE to FALSE and FALSE to TRUE. Three ways of doing this are as follows:

```
is.na(health) == FALSE
```

```
!(is.na(health) == TRUE)
```

```
!is.na(health)
```

The most concise and professional way of stating this criterion is the third expression, and you should prove to your own satisfaction that the three above statements are equivalent. Using this criterion to filter the data, we now have the following, which concludes our work on the age and health columns.

Exercise 8.8: (Optional) Learn more about the NA value and how R handles and evaluates this value. Consider why the is.na function is used to evaluate this value.

```
> census_2001_health %>%
+   gather("occupational_group", "count", -place, -sex, -age_and_health) %>%
+   separate(age_and_health, into = c("age", "health"), sep = "-", extra = "drop") %>%
+   filter(!is.na(health))
Source: local data frame [990,171 x 6]
```

	place	sex	age	health	occupational_group	count
1	SCOTLAND	ALL PEOPLE	16 to 24	Good Health	ALL PEOPLE	468336
2	SCOTLAND	ALL PEOPLE	16 to 24	Fairly Good Health	ALL PEOPLE	81109
3	SCOTLAND	ALL PEOPLE	16 to 24	Not Good Health	ALL PEOPLE	17032
4	SCOTLAND	ALL PEOPLE	25 to 34	Good Health	ALL PEOPLE	536389
5	SCOTLAND	ALL PEOPLE	25 to 34	Fairly Good Health	ALL PEOPLE	123638
6	SCOTLAND	ALL PEOPLE	25 to 34	Not Good Health	ALL PEOPLE	39370
7	SCOTLAND	ALL PEOPLE	35 to 49	Good Health	ALL PEOPLE	775816
8	SCOTLAND	ALL PEOPLE	35 to 49	Fairly Good Health	ALL PEOPLE	238862
9	SCOTLAND	ALL PEOPLE	35 to 49	Not Good Health	ALL PEOPLE	103655
10	SCOTLAND	ALL PEOPLE	50 to 54	Good Health	ALL PEOPLE	211733
..

We now need to think carefully about the observational unit structure of the tidy data we are constructing for the variables sex, occupational_group, and especially place. For the sex column the unique values are:

- ALL PEOPLE
- Female

- Male

And similarly for occupational_group the unique values are ALL PEOPLE (all in upper case), followed by a list of occupational categorisations. As people are categorised as either male or female, and in this census table these are mutually exclusive and exhaustive categories, the ALL PEOPLE entries are derivable from rows that we have elsewhere, and so do not need to be included separately. We can therefore filter out ALL PEOPLE from the sex column, by either filtering in 'Male' and 'Female', or filtering out 'ALL PEOPLE'. We can check our reasoning by seeing if both of these filters produce output tables with the same number of rows:

```
> census_2001_health %>%
+   gather("occupational_group", "count", -place, -sex, -age_and_health) %>%
+   separate(age_and_health, into = c("age", "health"), sep = "-", extra = "drop") %>%
+   filter(!is.na(health)) %>%
+   filter(sex != "ALL PEOPLE")
Source: local data frame [660,114 x 6]
```

	place	sex	age	health	occupational_group	count
1	SCOTLAND	Male	16 to 24	Good Health	ALL PEOPLE	243266
2	SCOTLAND	Male	16 to 24	Fairly Good Health	ALL PEOPLE	33821
3	SCOTLAND	Male	16 to 24	Not Good Health	ALL PEOPLE	7688
4	SCOTLAND	Male	25 to 34	Good Health	ALL PEOPLE	269583
5	SCOTLAND	Male	25 to 34	Fairly Good Health	ALL PEOPLE	51541
6	SCOTLAND	Male	25 to 34	Not Good Health	ALL PEOPLE	17662
7	SCOTLAND	Male	35 to 49	Good Health	ALL PEOPLE	392536
8	SCOTLAND	Male	35 to 49	Fairly Good Health	ALL PEOPLE	106497
9	SCOTLAND	Male	35 to 49	Not Good Health	ALL PEOPLE	46686
10	SCOTLAND	Male	50 to 54	Good Health	ALL PEOPLE	108961
..

```
> census_2001_health %>%
+   gather("occupational_group", "count", -place, -sex, -age_and_health) %>%
+   separate(age_and_health, into = c("age", "health"), sep = "-", extra = "drop") %>%
+   filter(!is.na(health)) %>%
+   filter(sex %in% c("Male", "Female"))
Source: local data frame [660,114 x 6]
```

	place	sex	age	health	occupational_group	count
1	SCOTLAND	Male	16 to 24	Good Health	ALL PEOPLE	243266
2	SCOTLAND	Male	16 to 24	Fairly Good Health	ALL PEOPLE	33821
3	SCOTLAND	Male	16 to 24	Not Good Health	ALL PEOPLE	7688
4	SCOTLAND	Male	25 to 34	Good Health	ALL PEOPLE	269583
5	SCOTLAND	Male	25 to 34	Fairly Good Health	ALL PEOPLE	51541
6	SCOTLAND	Male	25 to 34	Not Good Health	ALL PEOPLE	17662
7	SCOTLAND	Male	35 to 49	Good Health	ALL PEOPLE	392536
8	SCOTLAND	Male	35 to 49	Fairly Good Health	ALL PEOPLE	106497
9	SCOTLAND	Male	35 to 49	Not Good Health	ALL PEOPLE	46686
10	SCOTLAND	Male	50 to 54	Good Health	ALL PEOPLE	108961
..

Exercise 8.9: Reproduce the above code and results.

Both the first filter, filtering out "ALL PEOPLE", and the second filter, filtering in either "Male" or "Female", have produced output tables with the same number of rows, so I feel reasonably confident that they have produced the same outputs.

The %in% operator

In the second example above I have a filtered according to the following criterion:

```
sex %in% c("Male", "Female")
```

The %in% operator takes the list of strings to its right, contained in the c() function, and searches for matches to any of these strings in the object to its left, returning TRUE if a match is found and FALSE otherwise. When there are just two or three strings to match, another approach would be:

```
sex == "Male" | sex == "Female"
```

Where the | symbol is R's OR operator, returning TRUE if either criterion is satisfied. Where there

are a large number of strings to match against, however, the `%in%` operator becomes a much more concise way for specifying the criteria. If there were a large number of strings to match against, and these are likely to be searched for multiple times, it can make sense to save the criteria string into an object beforehand, for example:

```
sexes <- c("Male", "Female") #beforehand
sex %in% sexes # within the filter function
```

Exercise 8.10: (Optional) Produce the same output, as in exercise 44, but using an alternative filtering criterion.

The string "ALL PEOPLE" is also used in the `occupational_group` column to refer to all combined occupational groups. Again this information is superfluous as it can be recovered from data available in other rows, so should be filtered out. However, there are more than just two occupational group categories to filter in, so filtering out "ALL PEOPLE" is a much more concise way of expressing the filter criteria:

```
> census_2001_health %>%
+   gather("occupational_group", "count", -place, -sex, -age_and_health) %>%
+   separate(age_and_health, into = c("age", "health"), sep = "-", extra = "drop") %>%
+   filter(!is.na(health)) %>%
+   filter(sex %in% c("Male", "Female")) %>%
+   filter(occupational_group != "ALL PEOPLE")
source: local data frame [609,336 x 6]
```

	place	sex	age	health	occupational_group	count
1	SCOTLAND	Male	16 to 24	Good	Health Large employers and higher managerial occupations	1235
2	SCOTLAND	Male	16 to 24	Fairly	Good Health Large employers and higher managerial occupations	177
3	SCOTLAND	Male	16 to 24	Not	Good Health Large employers and higher managerial occupations	32
4	SCOTLAND	Male	25 to 34	Good	Health Large employers and higher managerial occupations	10579
5	SCOTLAND	Male	25 to 34	Fairly	Good Health Large employers and higher managerial occupations	1113
6	SCOTLAND	Male	25 to 34	Not	Good Health Large employers and higher managerial occupations	187
7	SCOTLAND	Male	35 to 49	Good	Health Large employers and higher managerial occupations	27914
8	SCOTLAND	Male	35 to 49	Fairly	Good Health Large employers and higher managerial occupations	4057
9	SCOTLAND	Male	35 to 49	Not	Good Health Large employers and higher managerial occupations	879
10	SCOTLAND	Male	50 to 54	Good	Health Large employers and higher managerial occupations	7280
..

Perhaps the trickiest decision to make is about the 'place' variable. Looking more carefully at the dataset we see that larger regions and places are written entirely in uppercase, whereas smaller areas are written using a combination of uppercase and lowercase symbols. Here is an example of this using the original table:

	place	sex	age_and_health	ALL PEOPLE	Large employers and higher managerial occupations	Higher professional occupations
170	ABERDEEN CITY	Female	65 to 74 - Not Good Health	9810	10	11
171	ABERDEEN CITY	Female	65 to 74	4084	4	8
172	ABERDEEN CITY	Female	65 to 74 - Good Health	4014	6	3
173	ABERDEEN CITY	Female	65 to 74 - Fairly Good Health	1712	-	-
174	ABERDEEN CITY	Female	65 to 74 - Not Good Health	4044	166	580
175	Ashley	ALL PEOPLE	ALL AGES	792	3	49
176	Ashley	ALL PEOPLE	16 to 24	665	3	47
177	Ashley	ALL PEOPLE	16 to 24 - Good Health	117	-	2
178	Ashley	ALL PEOPLE	16 to 24 - Fairly Good Health	10	-	-
179	Ashley	ALL PEOPLE	16 to 24 - Not Good Health	1282	45	261
180	Ashley	ALL PEOPLE	25 to 34	1100	39	237
181	Ashley	ALL PEOPLE	25 to 34 - Good Health	150	6	21
182	Ashley	ALL PEOPLE	25 to 34 - Fairly Good Health	32	-	3
183	Ashley	ALL PEOPLE	25 to 34 - Not Good Health			

Exercise 8.11: Produce the above output using View.

So long as the producers of the census table were consistent in this convention, we can use it to separate out rows referring to smaller areas from rows referring to larger areas. In my case, as I am concerned about the problems of making inferences based on small population sizes, I have decided to filter to include only the larger places. I know there is a Base R function, called toupper(), which takes a string and converts all characters within it to uppercase. I can make use of this function to build yet another filter:

```
> census_2001_health %>%
+   gather("occupational_group", "count", -place, -sex, -age_and_health) %>%
+   separate(age_and_health, into = c("age", "health"), sep = "-", extra = "drop") %>%
+   filter(!is.na(health)) %>%
+   filter(sex %in% c("Male", "Female")) %>%
+   filter(occupational_group != "ALL PEOPLE") %>%
+   filter(place == toupper(place))
source: local data frame [16,632 x 6]
```

	place	sex	age	health	occupational_group	count
1	SCOTLAND	Male	16 to 24	Good Health	Large employers and higher managerial occupations	1235
2	SCOTLAND	Male	16 to 24	Fairly Good Health	Large employers and higher managerial occupations	177
3	SCOTLAND	Male	16 to 24	Not Good Health	Large employers and higher managerial occupations	32
4	SCOTLAND	Male	25 to 34	Good Health	Large employers and higher managerial occupations	10579
5	SCOTLAND	Male	25 to 34	Fairly Good Health	Large employers and higher managerial occupations	1113
6	SCOTLAND	Male	25 to 34	Not Good Health	Large employers and higher managerial occupations	187
7	SCOTLAND	Male	35 to 49	Good Health	Large employers and higher managerial occupations	27914
8	SCOTLAND	Male	35 to 49	Fairly Good Health	Large employers and higher managerial occupations	4057
9	SCOTLAND	Male	35 to 49	Not Good Health	Large employers and higher managerial occupations	879
10	SCOTLAND	Male	50 to 54	Good Health	Large employers and higher managerial occupations	7280
..

Exercise 8.12: Produce the above analyses and results.

This filter criterion works because, if a place name is already written entirely in uppercase, then its contents will not change if all of its characters are converted to uppercase, so it will still match against itself. All other place names will be changed as a result of being passed to toupper, and so no longer match against themselves, and will be filtered out.

Exercise 8.13: (Optional) Look at the function `tolower()`, and consider whether `filter(place == tolower(place))` would produce a useful result if we wanted only to include places at the smaller spatial scale. If it would not, why? What alternative formulation might work instead?

This simple filter has got us most but not all of the way towards having tidy data, all using the same observational unit. We can see the issue in the first few rows, where the place name is SCOTLAND. In this particular example, a 2001 Census data table for Scotland, 'SCOTLAND' rows in the place variable are like the 'ALL PEOPLE' rows in sex and occupational_group: they should be derivable from the contents of other rows, and so are not required. I therefore perform one final filter of place, removing rows containing SCOTLAND

```
> census_2001_health %>%
+   gather("occupational_group", "count", -place, -sex, -age_and_health) %>%
+   separate(age_and_health, into = c("age", "health"), sep = "-", extra = "drop") %>%
+   filter(!is.na(health)) %>%
+   filter(sex %in% c("Male", "Female")) %>%
+   filter(occupational_group != "ALL PEOPLE") %>%
+   filter(place == tolower(place)) %>%
+   filter(place != "SCOTLAND")
source: local data frame [16,128 x 6]
```

	place	sex	age	health	occupational_group	count
1	ABERDEEN CITY	Male	16 to 24	Good Health	Large employers and higher managerial occupations	49
2	ABERDEEN CITY	Male	16 to 24	Fairly Good Health	Large employers and higher managerial occupations	4
3	ABERDEEN CITY	Male	16 to 24	Not Good Health	Large employers and higher managerial occupations	2
4	ABERDEEN CITY	Male	25 to 34	Good Health	Large employers and higher managerial occupations	509
5	ABERDEEN CITY	Male	25 to 34	Fairly Good Health	Large employers and higher managerial occupations	59
6	ABERDEEN CITY	Male	25 to 34	Not Good Health	Large employers and higher managerial occupations	4
7	ABERDEEN CITY	Male	35 to 49	Good Health	Large employers and higher managerial occupations	1476
8	ABERDEEN CITY	Male	35 to 49	Fairly Good Health	Large employers and higher managerial occupations	191
9	ABERDEEN CITY	Male	35 to 49	Not Good Health	Large employers and higher managerial occupations	43
10	ABERDEEN CITY	Male	50 to 54	Good Health	Large employers and higher managerial occupations	400

We now have a dataset that is very close to the tidy data target form: place refers to mutually exclusive and exhaustive parts of Scotland that are all of a similar scale, sex includes males and females, age contains only mutually exclusive age groups, health only mutually exclusive health statuses, and occupational_group only mutually exclusive occupational groups. Each of these variables is a 'where' variable indicating exactly what the 'count' variable refers to.

A couple of additional data tidying steps are either useful or necessary, however. Firstly, when we split the `age_and_health` column using the "-" symbol, we split strings like "16 to 24 - Good Health" into "16 to 24 " and " Good Health" respectively, we have left a trailing whitespace character at the end of the age column cells, and a leading whitespace character at the start of the health column cells. This means if we were to filter on, for example, `age == "16 to 24"` or `health == "Good Health"`, we would return zero rows in either case, because the trailing and leading whitespace characters mean the strings are different. We can use the `str_trim` function encountered earlier for this:


```

> census_2001_health %>%
+   gather("occupational_group", "count", -place, -sex, -age_and_health) %>%
+   separate(age_and_health, into = c("age", "health"), sep = "-", extra = "drop") %>%
+   filter(!is.na(health)) %>%
+   filter(sex %in% c("Male", "Female")) %>%
+   filter(occupational_group != "ALL PEOPLE") %>%
+   filter(place == toupper(place)) %>%
+   filter(place != "SCOTLAND") %>%
+   mutate(
+     age = str_trim(age),
+     health = str_trim(health)
+   )
Source: local data frame [16,128 x 6]

```

	place	sex	age	health	occupational_group	count
1	ABERDEEN	CITY	Male	16 to 24	Good Health Large employers and higher managerial occupations	49
2	ABERDEEN	CITY	Male	16 to 24	Fairly Good Health Large employers and higher managerial occupations	4
3	ABERDEEN	CITY	Male	16 to 24	Not Good Health Large employers and higher managerial occupations	2
4	ABERDEEN	CITY	Male	25 to 34	Good Health Large employers and higher managerial occupations	509
5	ABERDEEN	CITY	Male	25 to 34	Fairly Good Health Large employers and higher managerial occupations	59
6	ABERDEEN	CITY	Male	25 to 34	Not Good Health Large employers and higher managerial occupations	4
7	ABERDEEN	CITY	Male	35 to 49	Good Health Large employers and higher managerial occupations	1476
8	ABERDEEN	CITY	Male	35 to 49	Fairly Good Health Large employers and higher managerial occupations	191
9	ABERDEEN	CITY	Male	35 to 49	Not Good Health Large employers and higher managerial occupations	43
10	ABERDEEN	CITY	Male	50 to 54	Good Health Large employers and higher managerial occupations	400
..

Exercise 8.14: Reproduce the above analyses and results.

Exercise 8.15: (Optional) Consider two or three alternative situations where `str_trim` may be needed.

Introducing `mutate_each`

In the example above we wanted to perform the same function on two separate columns. If we were to use `mutate` to do this for all of the columns, we would have to write something like

```

mutate(
  place = str_trim(place),
  sex = str_trim(sex),
  age = str_trim(age),
  health = str_trim(health),
  occupational_group = str_trim(occupational_group),
  count = str_trim(count)
)

```

This is not particularly concise code, and creates some scope for coding errors. (For example, in the above I had accidentally typed `sex = str_trim(place)` before noticing this typo and correcting it. If I had not then this error would have led to difficulties later on.) In cases where the same function is being applied to many or all columns of a dataframe, the `mutate_each` function can be used instead. The equivalent way to write out the above using `mutate_each` is

```
mutate_each(funs(str_trim))
```

In `mutate_each`, the function or functions to apply need to be enclosed within a function called `funs`. Additional arguments can be used to specify, either inclusively or exclusively, which columns the function within `funs` should be applied to. For example, if I wanted to apply the function `str_trim` to all columns except `count`, then I would type:

```
mutate_each(funs(str_trim), -count)
```

Exercise 8.16: (Optional) Explore the `mutate_each` function using help.

Our data table is now almost as we want it, except for one thing, which we can spot if we ‘glimpse’ the data:

```
> census_2001_health %>%
+   gather("occupational_group", "count", -place, -sex, -age_and_health) %>%
+   separate(age_and_health, into = c("age", "health"), sep = "-", extra = "drop") %>%
+   filter(!is.na(health)) %>%
+   filter(sex %in% c("Male", "Female")) %>%
+   filter(occupational_group != "ALL PEOPLE") %>%
+   filter(place == toupper(place)) %>%
+   filter(place != "SCOTLAND") %>%
+   mutate_each(funs(str_trim), -count) %>% glimpse
Observations: 16128
Variables:
$ place      (chr) "ABERDEEN CITY", "ABERDEEN CITY", "ABERDEEN CITY", "ABERDEEN C...
$ sex        (chr) "Male", "Male", "Male", "Male", "Male", "Male", "Male", "Male"...
$ age        (chr) "16 to 24", "16 to 24", "16 to 24", "25 to 34", "25 to 34", "2...
$ health     (chr) "Good Health", "Fairly Good Health", "Not Good Health", "Good ...
$ occupational_group (chr) "Large employers and higher managerial occupations", "Large em...
$ count      (chr) "49", "4", "2", "509", "59", "4", "1476", "191", "43", "400", ...
```

Exercise 8.17: Reproduce the above code and results.

The count column is still classified as of type character rather than as a number. If we remember back to when we loaded the data, we made a tactical decision to load each of the then 16 columns of the data in as columns, so as to be able to handle the use in the table of the ‘-’ symbol to represent values of 0. The contents of 12 of the original columns are now contained in just one column, ‘count’, which means that whatever we have to do to turn ‘-’ symbols to ‘0’, and then convert the column type from character to numeric, we now have to do this to just one column rather than many. My function for doing this is as follows:

```
zero_and_numeric <- function(input){
  output <- input %>%
    str_replace("-", "0") %>%
    as.numeric
  return(output)
}
```

Exercise 8.18: Create the zero_and_numeric function as above. Again reflect on whether the order of the lines in the pipe affect the function’s output.

I then add this as a mutate function to the end of the pipe:

```
> census_2001_health %>%
+   gather("occupational_group", "count", -place, -sex, -age_and_health) %>%
+   separate(age_and_health, into = c("age", "health"), sep = "-", extra = "drop") %>%
+   filter(!is.na(health)) %>%
+   filter(sex %in% c("Male", "Female")) %>%
+   filter(occupational_group != "ALL PEOPLE") %>%
+   filter(place == toupper(place)) %>%
+   filter(place != "SCOTLAND") %>%
+   mutate_each(funs(str_trim), -count) %>%
+   mutate(count = zero_and_numeric(count))
Source: local data frame [16,128 x 6]
   place sex    age    health
1 ABERDEEN CITY Male 16 to 24 Fairly Good Health Large employers and higher managerial occupations 49
2 ABERDEEN CITY Male 16 to 24 Fairly Good Health Large employers and higher managerial occupations 4
3 ABERDEEN CITY Male 16 to 24 Not Good Health Large employers and higher managerial occupations 2
4 ABERDEEN CITY Male 25 to 34 Fairly Good Health Large employers and higher managerial occupations 509
5 ABERDEEN CITY Male 25 to 34 Fairly Good Health Large employers and higher managerial occupations 59
6 ABERDEEN CITY Male 25 to 34 Not Good Health Large employers and higher managerial occupations 4
7 ABERDEEN CITY Male 35 to 49 Fairly Good Health Large employers and higher managerial occupations 1476
8 ABERDEEN CITY Male 35 to 49 Fairly Good Health Large employers and higher managerial occupations 191
9 ABERDEEN CITY Male 35 to 49 Not Good Health Large employers and higher managerial occupations 43
10 ABERDEEN CITY Male 50 to 54 Fairly Good Health Large employers and higher managerial occupations 400
.. ... ..
```


I am now, finally, satisfied that the data are in the 'tidy' data format I wanted, and save it to a new object:

```
> tidy_census_2001_health <- census_2001_health %>%
+   gather("occupational_group", "count", -place, -sex, -age_and_health) %>%
+   separate(age_and_health, into = c("age", "health"), sep = "-", extra = "drop") %>%
+   filter(!is.na(health)) %>%
+   filter(sex %in% c("Male", "Female")) %>%
+   filter(occupational_group != "ALL PEOPLE") %>%
+   filter(place == toupper(place)) %>%
+   filter(place != "SCOTLAND") %>%
+   mutate_each(funs(str_trim), -count) %>%
+   mutate(count = zero_and_numeric(count))
```

Exercise 8.19: Reproduce the above code and result, and explore the newly created `tidy_census_2001_health` data frame.

Note that because I have used the `<-` operator to assign the output of these various processes to an object, the output from the end of the pipe does not 'spill into' the console, and so this time the first few lines of the table are not displayed.

8.4. Working with tidied data

A great deal of effort has so far been spent on rearranging and manipulating the contents of a table, in order to make sure the resulting table structure conforms to the 'tidy data' paradigm described above, and to make sure that the contents of cells are error free, and that different variables are of the correct class. In the case of the `tidy_census_2001_health` object we have created, all but the last column, `count`, is of either a character or factor class, with `count` itself being a numeric variable.

There are obviously costs, in terms of time and effort, in producing the tidy data object; the purpose of this section is to illustrate some of the ways these costs, at the data management stage, can be more than compensated for by savings at the analysis stage. By having the data in a tidy data format, I argue, analyses and explorations of the data can now be performed much more easily, allowing us to focus on quickly asking and answering questions of substantive interest to us; tidy data has cleared most of the hurdles and stumbling blocks involved in exploring and interrogating the data, and so greatly reduced the costs, in terms of researcher time and energy, involved in following our curiosity about what the data may reveal about social and health phenomena, and so now makes it much easier to turn data into information, knowledge and insight.

The Data Wrangling 'Cheat Sheet'

RStudio have produced a two page 'cheat sheet' which discusses and summarises the majority of the functions contained within the packages `dplyr` and `tidyr`, and how they can be used together to perform a wide range of data management and data analysis tasks. This document is available from the following location:

<https://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf>

Copies of this document are included with this course's materials, and I recommend having it to hand as a reference. (I have both pages printed at A3 size and displayed on my wall!)

The nearly 80 functions and operators summarised in this sheet can be used to achieve the vast majority of common data management and analyses tasks, and through things like the `.` operator and piping work largely seamlessly with other functions and packages.

Exercise 8.20: Explore and discuss the cheat sheet and some of its functions in groups of two or three.

What follows are a series of examples of data analysis using our tidied census table. In each example, I will ask a question, then describe the process and functions employed to answer it.

8.5. Question 1: Did males or females in Scotland have better self-reported health in 2001?

Exercise 8.21: Without looking at the solution, and with the cheat sheet, work either on your own or with a colleague to develop a solution for answering Question 1.

Analysis code:

```
394 tidy_census_2001_health %>%
395   mutate(
396     good_health = recode(
397       health, recodes = "
398       c('Good Health', 'Fairly Good Health') = 'yes';|
399       'Not Good Health' = 'no'
400     )
401   ) %>%
402   group_by(good_health, sex) %>%
403   summarise(count = sum(count)) %>%
404   spread(good_health, count) %>%
405   mutate(
406     ratio = yes / no,
407     proportion = yes / (yes + no)
408   )
```

Result:

```
Source: local data frame [2 x 5]
   sex    no    yes  ratio proportion
1 Female 213136 1708997 8.01834  0.8891149
2  Male 189647 1619299 8.53849  0.8951616
```

Exercise 8.22: Reflect on the above result. What does it mean?

Conclusion:

It seems that in 2001 both males and females have similar levels of self-reported health, but perhaps for males it was slightly higher. Around 89% of females reported either good or fairly good health, compared with around 90% of males. The ratio of males reporting good health to males reporting bad health was around 8.5, whereas for females it was around 8.0. Appropriate statistical testing should be conducted to calculate whether these differences are statistically significant (given the large number of observations my guess would be that they are).

Exercise 8.23: Consider whether you agree with the above conclusion, and if not what would your conclusion be?

Explanation:

I pass the tidy data object (line 394) to mutate (lines 395-402), which adds another column to the data, good_health, which has two values, 'yes', if individuals report either good health or fairly good health; and 'no', if individuals report not good health. I then group the data by both good_health and sex, and use summarise to sum the counts of these groups. I then re-arrange the output so that 'yes' and 'no' are separate columns, making it easier to use mutate to calculate the yes/no ratio and proportion of yesses for each sex.

Explaining the explanation 1: recode

There are other ways to do it, but I tend to use the recode function for recategorising variables. This function is from the car package, written by the American sociologist John Fox, which predates dplyr and tidyr by many years. However it integrates seamlessly within the mutate command.

The main reason I use recode is because it has a relatively intuitive way of specifying how values should be recoded, within the recodes argument, which seems to be based on SPSS recoding script conventions. The recodes argument takes a string, indicated by opening and closing double quotation marks “. However this string can be split over multiple lines. In the above example the contents of the recodes argument is as follows:

```
recodes = "
  c('Good Health', 'Fairly Good Health') = 'yes';
  'Not Good Health' = 'no'
"
```

I have put a different recoding category on each line. The labels you want to recode *from* are listed on the left of each line, before the = sign, and the labels to recode *to* are on the right of each line, after the = sign. Each recoding sub-argument is separated using the semicolon symbol (;). Note that, because the “ symbol is used to indicate the start and end of the complete string, the single quotation mark symbol ‘ is used to specify each label. Further information is available in the recode help file.

Exercise 8.24: Explore the help file for the recode function. Consider why the single quotation mark is used frequently in recodes arguments rather than the double quotation mark.

Exercise 8.25: (Optional) Look for alternative functions which perform recoding operations like the above. Reflect on their advantages and disadvantages.

Explaining the explanation 2: group_by and summarise

The group_by and summarise functions are part of dplyr, and are immensely powerful but can be initially a bit confusing. We can start to understand more about how they work, in isolation and together, by ‘breaking the pipe’ after the group_by command:

```
> tidy_census_2001_health %>%
+   mutate(
+     good_health = recode(
+       health, recodes = "
+         c('Good Health', 'Fairly Good Health') = 'yes';
+         'Not Good Health' = 'no'
+       "
+     )
+   ) %>%
+   group_by(good_health, sex)
Source: local data frame [16,128 x 7]
Groups: good_health, sex
```

	place	sex	age	health	occupational_group	count	good_health
1	ABERDEEN CITY	Male	16 to 24	Good Health	Large employers and higher managerial occupations	49	yes
2	ABERDEEN CITY	Male	16 to 24	Fairly Good Health	Large employers and higher managerial occupations	4	yes
3	ABERDEEN CITY	Male	16 to 24	Not Good Health	Large employers and higher managerial occupations	2	no
4	ABERDEEN CITY	Male	25 to 34	Good Health	Large employers and higher managerial occupations	509	yes
5	ABERDEEN CITY	Male	25 to 34	Fairly Good Health	Large employers and higher managerial occupations	59	yes
6	ABERDEEN CITY	Male	25 to 34	Not Good Health	Large employers and higher managerial occupations	4	no
7	ABERDEEN CITY	Male	35 to 49	Good Health	Large employers and higher managerial occupations	1476	yes
8	ABERDEEN CITY	Male	35 to 49	Fairly Good Health	Large employers and higher managerial occupations	191	yes
9	ABERDEEN CITY	Male	35 to 49	Not Good Health	Large employers and higher managerial occupations	43	no
10	ABERDEEN CITY	Male	50 to 54	Good Health	Large employers and higher managerial occupations	400	yes
..

The group_by function has not altered the data tabled itself, but added another piece of metadata, or attribute, to it. If you look at the second line of the output you can see the following: Groups: good_health, sex

This Groups attribute shifts the behaviour of all uses of the summarise function, and some uses of the mutate function. But first we need a brief introduction to the summarise function...

If you look at page 2 of the Data Wrangling cheat sheet, you can see the following image near the top of the left column:



This in essence describes what the summarise function does: it takes a number of rows, applies a summary function to the contents of these rows, and returns one row in its place. In our case, the summary function we are using is called 'sum'. It takes the contents of many rows within the counts column, and adds them together. We can get a clearer idea of how summarise and group_by work together by skipping the group_by line of the code, and passing the contents of the first pipe straight to the summarise line:

```
> tidy_census_2001_health %>%
+   mutate(
+     good_health = recode(
+       health, recodes = "
+       c('Good Health', 'Fairly Good Health') = 'yes';
+       'Not Good Health' = 'no'
+     )
+   ) %>%
+   summarise(count = sum(count))
Source: local data frame [1 x 1]

   count
1 3731079
```

So, without the group_by attribute, summarise sums up all of the count rows, returning a total population size.¹⁴ Adding group_by attributes means that summarise instead applies the summary function only to each of the subgroups defined by the attributes. In our case we have chosen to group the data by both sex and good_health status, meaning that the subgroups defined are: 'Male – good health', 'Male – Bad Health'; 'Female – Good Health' and 'Female – Bad Health'.

Note: As the group_by attribute changes the behaviour of other functions, we may want to remove this attribute later in the process. We can do that by passing the output of a pipe to the ungroup function, which removes all existing group_by attributes.

Exercise 8.26: Produce the analyses and results above.

Exercise 8.27: Consider other examples of processes where the group_by function would be useful.

Exercise 8.28: (Optional) Explore the similarities and differences between the mutate_each and summarise_each functions.

8.6. Question 2: Which places have the highest and lowest proportion of males aged between 50 to 64 who report poor health?

Exercise 8.29: (Optional) Without reading any further, try to produce a solution for answering this question.

¹⁴ It should be noted that the resident population of Scotland in the 2001 census was around 5.1 million, whereas the sum of counts from this table is around 3.7 million, suggesting that not all residents were asked this particular census question, or were simply not included within this particular table. If you look at the age categories, you can see that the minimum age group is '16 to 24', and maximum age group is '65 to 74', implying that the table excludes both persons aged either under 16 years or over 74 years of age.

Analysis code:

```
415 tidy_census_2001_health %>%
416   filter(age %in% c("50 to 54", "55 to 59", "60 to 64")) %>%
417   filter(sex == "Male") %>%
418   mutate(
419     good_health = recode(
420       health, recodes = "
421         c('Good Health', 'Fairly Good Health') = 'yes';
422         'Not Good Health' = 'no'
423       "
424     )
425   ) %>%
426   select(place, age, good_health, count) %>%
427   group_by(place, good_health) %>%
428   summarise(count = sum(count)) %>%
429   spread(good_health, count) %>%
430   transmute(place = place, proportion = no / (no + yes)) %>%
431   arrange(proportion)
```

Result:

Lowest to highest:

```
Source: local data frame [32 x 2]
   place proportion
1  SHETLAND ISLANDS 0.1056075
2  ABERDEENSHIRE 0.1061251
3  SCOTTISH BORDERS 0.1127637
4  PERTH & KINROSS 0.1131148
5  ANGUS 0.1180794
6  EAST DUNBARTONSHIRE 0.1191515
7  EAST RENFREWSHIRE 0.1205055
8  ORKNEY ISLANDS 0.1220010
9  MORAY 0.1233741
10 EILEAN SIAR 0.1298984
.. ... ..
```

Highest to lowest:

```
Source: local data frame [32 x 2]
   place proportion
1  GLASGOW CITY 0.3176216
2  NORTH LANARKSHIRE 0.2501308
3  INVERCLYDE 0.2290660
4  WEST DUNBARTONSHIRE 0.2225961
5  SOUTH LANARKSHIRE 0.2039431
6  NORTH AYRSHIRE 0.1982011
7  CLACKMANNANSHIRE 0.1943320
8  EAST AYRSHIRE 0.1907525
9  RENFREWSHIRE 0.1898991
10 DUNDEE CITY 0.1886792
.. ... ..
```

Note: The highest to lowest table is produced using the code above, but with the final line changed from `arrange(proportion)` changed to `arrange(desc(proportion))`

Conclusion:

Rich, rural and remote places like Shetland Islands, Aberdeenshire and the Scottish Borders had the lowest proportion of older working males reporting poor health, with between around 10% and 12% of males in this age group reporting poor health. In contrast most parts of Greater Glasgow had amongst the highest proportions of self-reported poor health amongst older working age males, with between around a quarter and a fifth of males of this age group reporting poorer health. Glasgow City appears to have exceptionally high levels of poor health, with poor health affecting almost a third of males in this age group in 2001.

Exercise 8.30: Reproduce the above code and reflect on why the code works and whether you agree with the conclusion. If not, what would your conclusion be?

Explanation:

I first filtered to include only the sex and age groups of interest. I then recode the health variable in the same way as before, copying and pasting the code from the previous exercise. As only males are now included, the sex variable no longer contains useful distinguishing information, and so I select only the columns place, age, good_health and count (The effect of the select function is also to arrange the selected columns from left to right in the order specified). I group the data by both place and health status, summarise counts in a similar way to before, and spread and transmute the resulting output to produce the proportion of older working age males who report poorer health. Finally, I arrange the output table in either ascending or descending order of the proportions of the area's population with poorer health.

Explaining the Explanation: transmute

In the code above I have used the function transmute rather than the more usual mutate function. The difference between transmute is that, whereas mutate adds columns, usually based on the contents of existing columns, transmute converts columns, removing the original columns that the new columns were based on, as well as non-grouped columns which have not been specified. It is for this reason that I include the place = place within the transmute line, as without writing this transmute would have removed the place variable. In order to understand more about the differences between transmute and mutate, it is helpful to experiment with both functions within your code.

Exercise 8.31: Think of two examples where the transmute function would be preferable to the mutate function.

Addendum to question 2: A very, very quick introduction to data visualisation using ggplot2

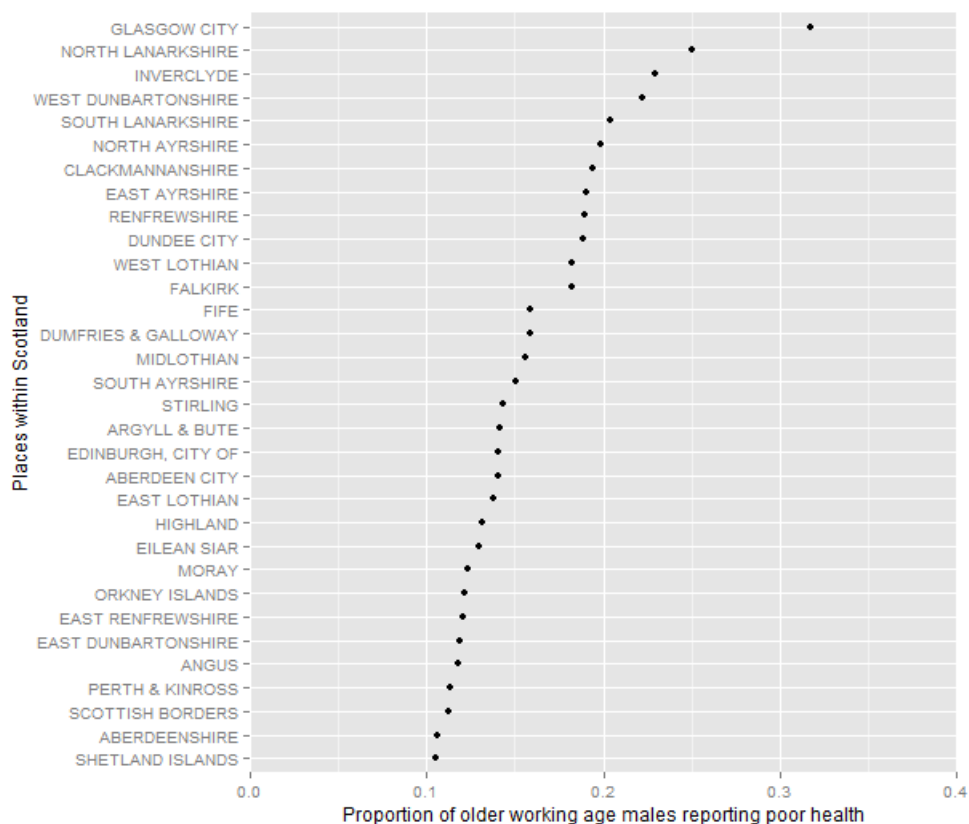
Data tables and analyses produced using dplyr work well, but not perfectly, with ggplot2, a very popular data visualisation package that Hadley Wickham began a few years earlier, and which has since been phenomenally successful and grown greatly in size and popularity. For example, the following code:

```

456 tidy_census_2001_health %>%
457 filter(age %in% c("50 to 54", "55 to 59", "60 to 64")) %>%
458 filter(sex == "Male") %>%
459 mutate(
460   good_health = recode(
461     health, recodes = "
462       c('Good Health', 'Fairly Good Health') = 'yes';
463       'Not Good Health' = 'no'
464     "
465   )
466 ) %>%
467 select(place, age, good_health, count) %>%
468 group_by(place, good_health) %>%
469 summarise(count = sum(count)) %>%
470 spread(good_health, count) %>%
471 transmute(place = place, proportion = no / (no + yes)) %>%
472 arrange(desc(proportion)) %>%
473 ggplot(.) +
474   geom_point(aes(y = reorder(place, proportion), x = proportion)) +
475   labs(x = "Proportion of older working age males reporting poor health", y = "Places within Scotland") +
476   coord_cartesian(xlim = c(0, 0.40))

```

Produces the following image:



Exercise 8.32: Reproduce the above figure. Identify which lines are ggplot2 commands and which are tidyr/dplyr commands. What is the output from dplyr/tidyr, and what would it look like if displayed using View?

The four lines 473 to 476 are all ggplot2 functions, and even without understanding these functions you can see how ggplot2 functions can be thought about as yet another series of extensions to the existing pipe produced using dplyr and tidyr. Like dplyr and tidyr, ggplot2 is modular, allowing one additional instruction to be added (and tested) at a time. Within ggplot2, the '+' operator works much as the %>% operator does within dplyr and tidyr.

Exercise 8.33: (Optional) Using online help, change one or more of the lines of instructions to ggplot2 so the code still runs but the figure produce has been changed.

8.7. Data Visualisation using ggplot2

Exercises 68 and 69 very briefly introduced ggplot2, Hadley Wickham's most popular R package, for data visualisation. The ggplot2 example was introduced at that stage mainly in order to demonstrate that ggplot2 code forms an organic extension of the 'piped coding' approach used in dplyr and tidyr, with each line providing a new instruction which works with the output of the previous instruction, and the + operator chaining together these instructions much as the %>% operator does within dplyr and tidyr. However, ggplot2 also draws from a different paradigm, a way of thinking about and being explicit about the production of data graphics known as the 'grammar of graphics', based on a book of this name by the statistician Leland Wilkinson. A large number of introductions exist to ggplot2, but at its core the grammar of graphics approach is about providing a language for clearly defining *the mapping rules which link values in columns in a data frame to features on a graphical display*. Conceptually, this can be thought about as kind of 'box wiring' exercise, with data frame columns at one side of the box, and graphical features on the other end of the box. An example of this is shown below:

Data Variable		Graphical Aesthetic
% of population providing free care		Position along horizontal axis
% of population with health needs		Position along vertical axis
Areal unit population		Size of bubble
Geographical location (North or south)		Colour of bubble

In this example the data frame, on the left, can be thought to contain four columns, and the values in each of these columns are 'wired' to so they control a different part of the graphic displayed. At a minimum, within ggplot2 the user needs to specify:

- The dataset which contains the variables
- The mapping rules, known as **aesthetics**, which define how the values of the variables control different parts of the graphic
- The **geometrics**: the type of graphic element that is displayed

An illustration of how the above set of mapping rules may look within ggplot2 is as follows:

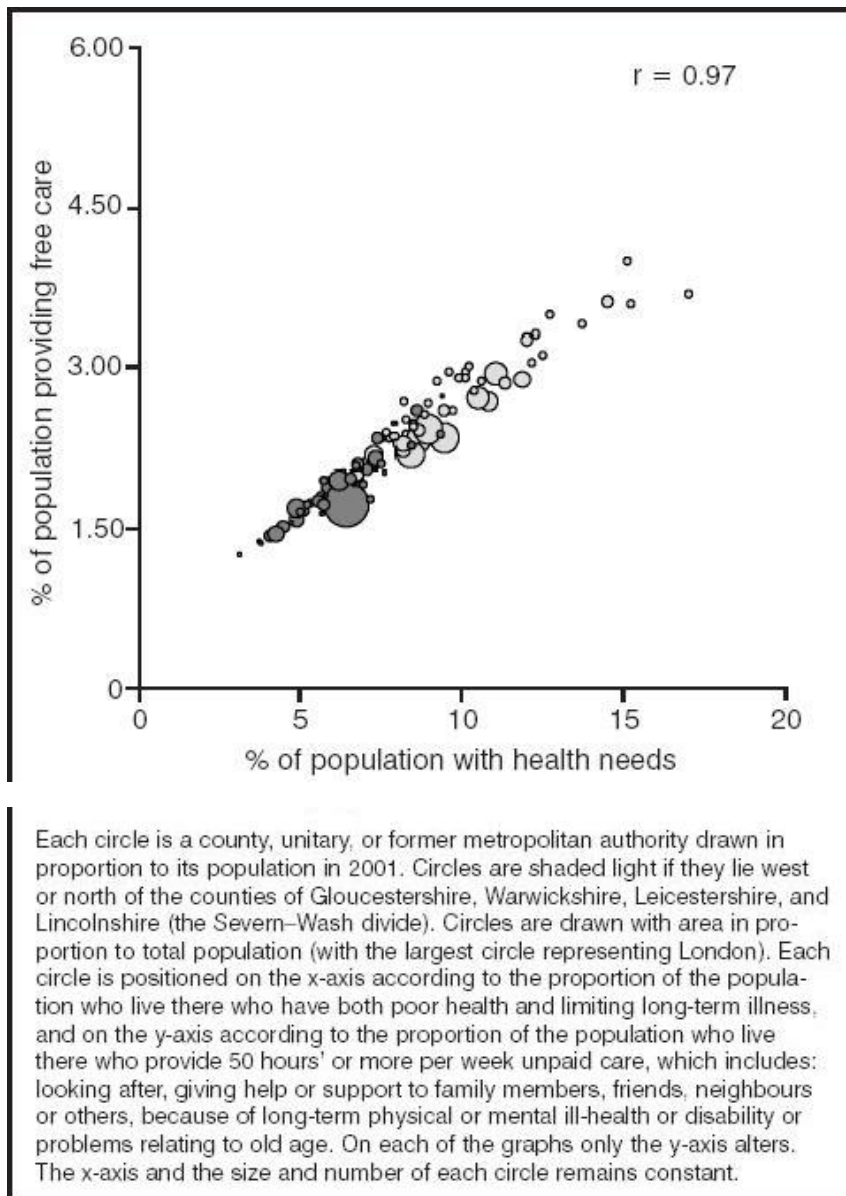
```
589 dta %>%
590   ggplot(.) +
591   geom_point(
592     aes(
593       x = health_care_needs,
594       y = health_care_provision,
595       size = population_size,
596       fill = is_north
597     )
598   )
```

This can be interpreted as follows:

- Line 589: `dta`, ending with the piping operator, is the data frame, which contains at least four variables: `health_care_needs` and `health_care_provision`, which are values from 0 to 1; `size`, which is the population count for the areal unit, and `fill`, which is a binary indicator with one value for areas in the North of England, and another value for the South of England.
- Line 590: `ggplot` is the basic `ggplot2` function which accepts contents of the pipe to its first argument. However, the arguments in this function do not yet have sufficient instructions to be able to produce a plot. Instead, the function needs additional instructions to be able to produce a plot. These additional instructions are added through more `ggplot2` functions, chained together using the `+` operator.
- Lines 591 to 598: this contains the function `geom_point`, which specifies the type of graphic produced. In this case, a point. The argument to this function is passed to its first slot, called 'mapping', which takes an object of class `aes` (aesthetic), created by the `aes` function.
- This `aes` function takes up lines 592 to 597 inclusive. This defines the mapping rules indicated by the box wiring above.
 - `x = health_care_needs` : the horizontal position of the points is determined by the value of rows in the `health_care_needs` column
 - `y = health_care_provision` : the vertical position of the points is determined by the value of the rows in the `health_care_provision` column
 - `size = population_size`: the size of the points is determined by the value of the rows in the `population_size` column. (This effectively turns points into bubbles, and hence is why there is no separate `geom_bubble` function.)
 - `fill = is_north`: the colour or shade within the point/bubble is determined by the value of the rows in the `is_north` column.

The above example is based on the main figure in a 2004 paper by Mary Shaw and Danny Dorling¹⁵, reproduced below, along with the notes presented below that figure:

¹⁵ Shaw M & Dorling D (2004) "Who cares in England and Wales? The Positive Care Law: cross-sectional study" *British Journal of General Practice*, 54 (509): 899-903



This figure was very likely produced in Excel. The grammar of graphics paradigm is largely independent of any particular statistical package or programming language, even though the ggplot2 package is specific to R.

There are a number of ways of providing the same instructions to ggplot2. For example, the mapping rules could have been included in the ggplot function rather than the geom_point function as follows:

```

600 dta %>%
601   ggplot(
602     .,
603     aes(
604       x = health_care_needs,
605       y = health_care_provision,
606       size = population_size,
607       fill = is_north
608     )
609   ) +
610   geom_point()

```

A video introduction to ggplot2 is available on youtube:

<https://www.youtube.com/watch?v=RHu5vgBZ1yQ>

Exercise 8.7.1: (Optional and after the workshop) View the video above.

Quick plots in ggplot2

An alternative way of producing data visualisations using ggplot2 is using the qplot ('quick plot') function:

```

614 qplot(
615   x = health_care_needs,
616   y = health_care_provision,
617   size = population_size,
618   fill = is_north,
619   geom = "point",
620   data = dta
621 )

```

The qplot function is a less formal and explicit way of providing the same set of geom and mapping instructions to ggplot2. In qplot the mapping rules are no longer wrapped up in the aes function, and geom = "point", an argument within the function, performs the same task as + geom_point(), a call to a separate function, in the ggplot2 formulations above. Additionally, the qplot function does not require that the geom be specified; if this argument is not specified then the function will make a guess about the best geom to use. As the name 'quick plot' suggests, the qplot function can be helpful for initial visual exploration of data, though for more explicit and formal control of elements of graphics, the more incremental and layered ggplot formulations are preferable.

Exercise 8.7.2: (Optional) Explore examples of ggplot and qplot functions on the ggplot2 websites below. Consider their relative advantages and disadvantages.

ggplot2: Cheat Sheet, Websites and Books

For ggplot2, the standard R help files are often less useful than for many other functions and

packages, mainly because the concepts and ideas are so inherently visual, whereas the help files are text files. Instead the official website is generally a much more effective resource for learning about ggplot2 in action:

<http://docs.ggplot2.org/0.9.3.1/index.html>

Like the data wrangling cheat sheet, Rstudio have produced a Cheat Sheet for data visualisation using ggplot2:

<https://www.rstudio.com/wp-content/uploads/2015/03/ggplot2-cheatsheet.pdf>

Exercise 8.7.3: Explore and discuss the cheat sheet and website with other delegates.

As suggested by its smaller font size and increased number of columns, the data visualisation cheat sheet contains more content than the data wrangling cheat sheet, not least because ggplot2 is an older, better developed and more complex package than dplyr and tidyr, and so requires more knowledge and experience to use properly. An extremely good book for using ggplot2 is:

- Chang, W, (2013) R Graphics Cookbook (Sebastopol, CA: O'Reilly)

For more advanced and up-to-date questions and solutions about ggplot2, there is also the programming website stackoverflow:

<http://stackoverflow.com/questions/tagged/ggplot2>

Exercise 8.7.4: (Time dependent) Go back to the ggplot2 example presented earlier and consider what each ggplot2 instruction contributes to the overall graphic. Amend the contents of these lines in ways that still produce outputs and assess whether the changes to the code were as expected.

Exercise 8.7.5: (Time dependent) Create a ggplot2 figure based on one of the tables produced in the tidy data analyses covered in day 1.

8.7.1. Saving ggplot2 outputs

The figures created by ggplot2 do not appear in the console on the bottom left pane, but within the bottom right pane in RStudio. Figures are created as 'side effects' from functions (see footnote 3) which do not necessarily generate outputs in their own right; in this case the 'side effect' of successfully running a series of ggplot2 instructions is to create a new 'graphical device', which by default appears in this bottom right pane. This default behaviour can be altered by specifying a different 'graphical device' before creating the graphic, then closing this device afterwards. In Base R this is the main way of creating figures. A full list of these graphical devices is available here:

<https://stat.ethz.ch/R-manual/R-devel/library/grDevices/html/Devices.html>

Within ggplot2 there is a slightly more intuitive way of creating files containing ggplot2 figures, using the ggsave function. This function is run after creating the ggplot2 figure displayed on the bottom right pane, but the way the figure appears in the saved file and the Rstudio graphics device is likely to be different. To understand why, it is important to look at the help file for ggsave and its arguments.

Save a ggplot (or other grid object) with sensible defaults

Description

`ggsave()` is a convenient function for saving a plot. It defaults to saving the last plot that you displayed, using the size of the current graphics device. It also guesses the type of graphics device from the extension.

Usage

```
ggsave(filename, plot = last_plot(), device = NULL, path = NULL,  
        scale = 1, width = NA, height = NA, units = c("in", "cm", "mm"),  
        dpi = 300, limitsize = TRUE, ...)
```

Arguments

<code>filename</code>	File name to create on disk.
<code>plot</code>	Plot to save, defaults to last plot displayed.
<code>device</code>	Device to use (function or any of the recognized extensions, e.g. "pdf"). By default, extracted from filename extension. <code>ggsave</code> currently recognises eps/ps, tex (pictex), pdf, jpeg, tiff, png, bmp, svg and wmf (windows only).
<code>path</code>	Path to save plot to (combined with filename).
<code>scale</code>	Multiplicative scaling factor.
<code>width</code> , <code>height</code>	Plot dimensions, defaults to size of current graphics device.
<code>units</code>	Units for width and height when specified explicitly (in, cm, or mm)
<code>dpi</code>	Resolution used for raster outputs.
<code>limitsize</code>	When <code>TRUE</code> (the default), <code>ggsave</code> will not save images larger than 50x50 inches, to prevent the common error of specifying dimensions in pixels.
<code>...</code>	Other arguments passed on to graphics device

The first argument, `filename`, must include the full path and file extension for the image file to be created. The file extension defines the type of file that is generated, so for example "figure.png" generates a portable network graphic file, and "figure.pdf" generates a portable document format file.

The `dpi` argument defines the image resolution, in dots per inch. Most academic journals require that figures have a resolution of at least 150 DPI, and you can see that by default the resolution is set to 300 DPI.

The `width`, `height` and `units` arguments can be used to specify the dimensions of the image produced, and it is important to be aware that different combinations of resolution and dimension will produce some very different looking images based on the same `ggplot2` call. In particular, it is important to be aware that higher resolutions tend to mean that any text in the display will be displayed at a larger size, as will graphical objects such as points, and if a high `dpi` is used in combination with a small dimension (combination of width and height) then the image may be different to what is expected and desired. More advanced and fine-tuned control of figures within `ggplot2`, largely achieved through the `theme()` function, will likely be required to produce the desired result, and for this the book mentioned previously by Winston Chang is highly recommended.

Exercise 8.7.5: Use `ggsave` to create versions of the example figure produced around Exercise 68, which differ either according to the file type or other arguments to the `ggsave` function. Reflect on which produce better and which produce worse outputs, and why.

8.8. Additional questions

In just two examples I have shown how the same tidy data object can be rearranged and manipulated in order to perform a wide range of analyses. In doing this I have introduced a number of the core features of `dplyr`, in particular the `group_by` and `summarise` functions, as well as further illustrate how `tidyr` functions like `spread` and `gather` can be used not just at the initial data tidying stage but also within later analyses. I have also very briefly introduced the `ggplot2` package as a conceptual extension (though chronologically a forerunner) of the general approach to analysis used here.

How and Why

The ultimate aim with the approach outlined here, having constructed a data table in an appropriate tidy data format, is to allow the process of asking and answering questions about data to be highly iterative, fluid and frictionless. Although all of the analyses presented here could be achieved without the packages and general approach I am advocating, doing so using Base R alone can be a much more technically challenging exercise, producing code that requires a much more advanced understanding of R as a programming language in order to interpret and apply correctly. As analysts of quantitative data, our main interest should be in substantive rather than technical questions: we should feel comfortable asking questions of the data, and ‘to’ the data, without each time feeling a faint sense of dread about the technical challenges involved in formulating those questions in ways that our statistical package or programming language of choice can understand. The friction of the technical challenges of ‘how’ to ask a question reduces over time, as you gain more familiarity with the R programming language. But I argue that concepts like piping and tidy data, and packages like `tidyr` and `dplyr`, help to smooth the journey much more quickly.

Exercise 8.34: Reflect on why, as researchers interested in substantive social phenomena, it can be useful to invest more time in learning about data management techniques, such as those introduced in this course, even if these techniques are not in themselves substantively interesting.

Some additional questions, relating to the `tidy_census_2001_health` dataset, that we might be interested in asking include:

1. What is the sex ratio in different occupational groups? How does this vary by age?
2. Which places have the highest and lowest average ages?
3. Which places are largest and which places are smallest?
4. What is the association between latitude and longitude and self-reported health?
5. Which places saw the greatest improvements in health from 1991 to 2001, and from 2001 to 2011?
6. Is there any kind of statistically or substantively significant ‘gender penalty’ in self-reported health? If so, how does this vary between places, and how has this changed over time?

Some of these questions and parts of questions can be answered using the above dataset alone; some can be partially but not fully answered given the data in this dataset; and other questions will require that the `tidy_census_2001_health` dataset be combined with additional datasets that have

also been prepared in a similar ways. Approaches to combining the tidy_census_2001_health dataset with other similar datasets will be discussed soon. But first I will return briefly to looking at the group_by function, and ways it can be used with the mutate function in addition to the summarise function, as for some types of health data these uses can be extremely helpful.

Exercise 8.35: (Optional and after the course) Try to answer some of the above questions. Email me with code, results and conclusions. Jonathan.minton@glasgow.ac.uk

Using group_by with mutate

To start with, consider the following code and output:

```
> tidy_census_2001_health %>%
+   group_by(place) %>%
+   summarise(count = sum(count)) %>%
+   arrange(desc(count)) %>%
+   mutate(
+     cumulative_count = cumsum(count),
+     cumulative_proportion = cumulative_count / sum(count)
+   )
Source: local data frame [32 x 4]
```

	place	count	cumulative_count	cumulative_proportion
1	GLASGOW CITY	430967	430967	0.1155073
2	EDINBURGH, CITY OF	342431	773398	0.2072853
3	FIFE	254713	1028111	0.2755533
4	NORTH LANARKSHIRE	237357	1265468	0.3391694
5	SOUTH LANARKSHIRE	223181	1488649	0.3989862
6	ABERDEENSHIRE	164674	1653323	0.4431219
7	ABERDEEN CITY	162653	1815976	0.4867160
8	HIGHLAND	152684	1968660	0.5276383
9	RENFREWSHIRE	127993	2096653	0.5619428
10	WEST LoTHIAN	116387	2213040	0.5931367
..

Exercise 8.36: Reproduce the above code and results.

This code first adds up the number of people within each place, then arranges the value in descending order. The output of this is then passed to mutate, containing the following lines:

```
cumulative_count = cumsum(count),
cumulative_proportion = cumulative_count / sum(count)
```

You can see that the result of cumsum is to produce a cumulative sum of count. As the rows have been arranged in descending order each row of count contains a smaller number than the previous row, and so the rate of accumulation decreases with each row.

You can also see that the object created in the second line, cumulative_proportion, depends on the object created by the first line, cumulative_count.¹⁶ Note also the use of sum within the mutate command, returning the sum of count. As no group_by attributes have been set, this sum function will return the sum of count column for all rows.

¹⁶ Think carefully about what would happen if you were to put these two lines in the other order: if cumulative_proportion were the first line, it would be referring to an object that does not yet exist, and so would produce an error.

Substantively, you can see that, of the 32 places comprising Scotland, the ten most populous places comprised nearly 60% of Scotland's population in 2001.

Now consider this variation with an additional call to `group_by`

```
> tidy_census_2001_health %>%
+   group_by(place, sex) %>%
+   summarise(count = sum(count)) %>%
+   group_by(sex) %>%
+   arrange(desc(count)) %>%
+   mutate(
+     cumulative_count = cumsum(count),
+     cumulative_proportion = cumulative_count / sum(count)
+   )
Source: local data frame [64 x 5]
Groups: sex
```

	place	sex	count	cumulative_count	cumulative_proportion
1	GLASGOW CITY	Female	226433	226433	0.1178030
2	EDINBURGH, CITY OF	Female	176516	402949	0.2096364
3	FIFE	Female	131164	534113	0.2778752
4	NORTH LANARKSHIRE	Female	123206	657319	0.3419737
5	SOUTH LANARKSHIRE	Female	116329	773648	0.4024945
6	ABERDEENSHIRE	Female	82090	855738	0.4452023
7	ABERDEEN CITY	Female	81789	937527	0.4877534
8	HIGHLAND	Female	77174	1014701	0.5279036
9	RENFREWSHIRE	Female	66618	1081319	0.5625620
10	WEST LOTHIAN	Female	60096	1141415	0.5938273
...

Exercise 8.37: Reproduce the above code and results.

This time, the cumulative counts, and cumulative proportions, have been produced separately for each sex. If you View this dataset you can see the following, half way down the dataset:

	place	sex	count	cumulative_count	cumulative_proportion
30	EILEAN SIAR	Female	9245	1907594	0.9924360
31	SHETLAND ISLANDS	Female	7598	1915192	0.9963889
32	ORKNEY ISLANDS	Female	6941	1922133	1.0000000
33	GLASGOW CITY	Male	204534	204534	0.1130681
34	EDINBURGH, CITY OF	Male	165915	370449	0.2047872
35	FIFE	Male	123549	493998	0.2730861
36	NORTH LANARKSHIRE	Male	114151	608149	0.3361897
37	SOUTH LANARKSHIRE	Male	106852	715001	0.3952583

Exercise 8.38: View the dataset and reflect on the results.

On the second line, the `group_by` attribute is set, grouping data by both place and sex. The third line, `summarise`, uses these grouping attributes to produce counts for each place and sex combination; *after doing this, summarise then removes the group_by attribute*. I have then set the `group_by` attribute again, for sex only, then arranged the data in descending order by count (implicitly, now, for each sex separately), and calculated the cumulative sum and cumulative proportion again. Because the `group_by` attribute has been set at this stage for sex, both the arrangement of places by size, and the calculations of cumulative counts and proportions, are done separately for each sex.

Here is an example of a slight extension of the above:

```
506 tidy_census_2001_health %>%
507   group_by(place, sex) %>%
508   summarise(count = sum(count)) %>%
509   group_by(sex) %>%
510   arrange(desc(count)) %>%
511   mutate(
512     cumulative_count = cumsum(count),
513     cumulative_proportion = cumulative_count / sum(count)
514   ) %>%
515   select(place, sex, cumulative_proportion) %>%
516   spread(key = sex, value = cumulative_proportion) %>%
517   mutate(
518     rank_female = dense_rank(Female),
519     rank_male = dense_rank(Male)
520   ) %>%
521   arrange(rank_female) %>%
522   mutate(dif_ranks = rank_male - rank_female) %>%
523   view
```

Exercise 8.39: Read through the above code and discuss what you think each line is likely to do. Run the code, one line at a time, to check whether your expectations are correct. Run the code in its entirety and produce some preliminary conclusions based on the results generated.

After calculating cumulative proportions separately for each sex, I have now used `spread` to move the contents of cumulative proportion into two separate columns, Male and Female. I have then used the `dense_rank` function within `dplyr` to rank these cumulative proportions for both males and females. Then I have arranged the places by the rank for female, and used `mutate` to calculate the difference in ranks. Positive scores indicate that the male rank for the place is higher than the female rank, and negative ranks indicate the converse. The result of doing this is the following table:

	place	Female	Male	rank_female	rank_male	dif_ranks
1	GLASGOW CITY	0.1178030	0.1130681	1	1	0
2	EDINBURGH, CITY OF	0.2096364	0.2047872	2	2	0
3	FIFE	0.2778752	0.2730861	3	3	0
4	NORTH LANARKSHIRE	0.3419737	0.3361897	4	4	0
5	SOUTH LANARKSHIRE	0.4024945	0.3952583	5	5	0
6	ABERDEENSHIRE	0.4452023	0.4409114	6	6	0
7	ABERDEEN CITY	0.4877534	0.4856137	7	7	0
8	HIGHLAND	0.5279036	0.5273563	8	8	0
9	RENFREWSHIRE	0.5625620	0.5612849	9	9	0
10	WEST Lothian	0.5938273	0.5924030	10	10	0
11	DUNDEE CITY	0.6231020	0.6787417	11	13	2
12	DUMFRIES & GALLOWAY	0.6517463	0.6213331	12	11	-1
13	FALKIRK	0.6803483	0.6500857	13	12	-1
14	NORTH AYRSHIRE	0.7074125	0.7312369	14	15	1
15	PERTH & KINROSS	0.7333748	0.7052328	15	14	-1
16	EAST AYRSHIRE	0.7569044	0.7547776	16	16	0
17	SOUTH AYRSHIRE	0.7789289	0.7766517	17	17	0
18	EAST DUNBARTONSHIRE	0.8003203	0.8191643	18	19	1
19	ANGUS	0.8212704	0.7980935	19	18	-1
20	SCOTTISH BORDERS	0.8418351	0.8399554	20	20	0
21	WEST DUNBARTONSHIRE	0.8604576	0.8763777	21	22	1
22	EAST RENFREWSHIRE	0.8778290	0.9449088	22	26	4
23	EAST Lothian	0.8951524	0.9111831	23	24	1
24	Argyll & Bute	0.9123708	0.8584247	24	21	-3
25	STIRLING	0.9295579	0.9280526	25	25	0
26	INVERCLYDE	0.9462170	0.9612924	26	27	1
27	MORAY	0.9623663	0.8941505	27	23	-4
28	MIDLothian	0.9783230	0.9768362	28	28	0
29	CLACKMANNANSHIRE	0.9876262	0.9863042	29	29	0
30	EILEAN SIAR	0.9924360	0.9916686	30	30	0
31	Shetland Islands	0.9963889	0.9961464	31	31	0
32	ORKNEY ISLANDS	1.0000000	1.0000000	32	32	0

You can see that there is no difference in gender place ranks for the ten most populous places ($\text{dif_rank} = 0$), but for the 11th place, Dundee City, the female rank is two places higher than the male rank. The largest differences in gender ranks are in East Renfrewshire, where the female rank is four places above the male rank, and in Moray, where the female rank is four places below the male rank. These differences in rank could be indicative of complex and potentially troubling health, economic or migratory issues within some of these places: Could the rank difference in Dundee City be due to men leaving the city because there are few job prospects? Or could it be because men in this city are dying at a particularly high rate compared with women? Similarly, what could the reasons be

for male gender rank in Moray to be so much higher than the female gender rank? Are the sorts of work available in Moray those that tend to attract men much more than women, for example? Could cross-tabulation of occupation in Moray compared with other regions (contained within our dataset) help to answer this question?

Exercise 8.40: Reflect on whether you agree or disagree with the above conclusions.

Exercise 8.41: (Optional and likely after the course) Produce an additional analysis of the data which responds to some of the questions raised by the above analysis. Send me your code, results and conclusions. (jonathan.minton@glasgow.ac.uk)

The purpose of simple and nimble analyses like the above is not to definitively try to answer such questions, but more to raise them in the first place. Questions generate answers, which generate more questions and yet more answers, and in this process knowledge and insight is gained, which can be both helpful as an end in itself and as a means of helping to focus more formal analyses and hypotheses. The purpose of these simple data analyses and tools is to be able to both ask and answer such questions more easily.

Exercise 8.42: Reflect on the above in the context of the relationship between theory and data. (Optional): Search online for the phrase “Merton Middle Range Theory” and consider the role that effective data science and data management skills have in allowing middle range theories to develop quickly and efficiently.

8.9. Binding and joining

Base R's main functions for combining datasets include `merge`, `cbind` (column bind) and `rbind` (row bind). The `dplyr` equivalents of this function include a number of functions involving the word 'join', and the functions `bind_rows` and `bind_cols`. These are detailed on the right hand side of the second page of the Data Wrangling Cheat Sheet.

The join functions, like the concept of tidy data, borrow conceptually from database query languages such as SQL (Structured Query Language), used by database software such as Microsoft Access. So long as two tables contain some columns with the same names and values, the two tables can be joined together. A number of different join types are made available, and you are encouraged to learn more the differences between these join types both by reviewing additional tutorials and guidance, and through careful but also playful experimentation.

A possibly unusual application of one of the join types, the `anti_join`, is now presented.

Exercise 8.43: Look at the Data Wrangling cheat sheet for information on different join types. Consider useful applications of each type of join.

8.9.1. Finding small places using `anti_join`

Remember that our tidied dataset, `tidy_census_2001_health`, contains only the big cities and regions within Scotland. We were able to identify these places because the labels for the large places were written using capital letters, and the other, smaller, places were not. We were able to use a 'trick', using the `toupper` function, to identify and filter in those place names that were written in capital letters, making the process of finding these larger places straightforward. However, smaller place names are not written just using lower case letters, so if we were to try a similar trick using the `tolower` function, which converts all letters in a string to lowercase, the filtering would not be successful, as the first letter of these smaller place names tends to be uppercase, followed by other

letters in lowercase. However even this pattern is not completely consistent, and so if we were to try to construct a pattern to match against smaller place names using something like regex, we would likely end up with a regex expression that looks both horrifically complicated, and still does not work correctly for all instances of smaller place names.

Instead, we can find the smaller places using the `anti_join` function, as follows:

```
529 tidy_census_2001_health_BIG <- census_2001_health %>%
530   gather("occupational_group", "count", -place, -sex, -age_and_health) %>%
531   separate(age_and_health, into = c("age", "health"), sep = "-", extra = "drop") %>%
532   filter(!is.na(health)) %>%
533   filter(sex %in% c("Male", "Female")) %>%
534   filter(occupational_group != "ALL PEOPLE") %>%
535   filter(place == toupper(place)) %>%
536   filter(place != "SCOTLAND") %>%
537   mutate_each(funs(str_trim), -count) %>%
538   mutate(count = change_dash_to_zero(count))
539
540 tidy_census_2001_health_all <- census_2001_health %>%
541   gather("occupational_group", "count", -place, -sex, -age_and_health) %>%
542   separate(age_and_health, into = c("age", "health"), sep = "-", extra = "drop") %>%
543   filter(!is.na(health)) %>%
544   filter(sex %in% c("Male", "Female")) %>%
545   filter(occupational_group != "ALL PEOPLE") %>%
546   mutate_each(funs(str_trim), -count) %>%
547   mutate(count = change_dash_to_zero(count))
548
549
550 tidy_census_2001_health_small <- tidy_census_2001_health_all %>%
551   anti_join(tidy_census_2001_health_BIG) %>%
552   filter(place != "SCOTLAND")
```

Exercise 8.44: Consider which of the above operations are likely to be most computationally intensive, and why. (Optional) Run the code and explore the `tidy_census_2001_health_small` dataset produced. Consider in which circumstances the small area dataset would be preferable to the larger area dataset.

Here, in lines 529-538, I have re-created the `tidy_census_2001_health` dataset query as before, except this time I have named it `tidy_data_2001_health_BIG`. Within lines 540-547 I have run a similar query, creating quasi-tidy data in a similar format, but not filtering in or out any particular places.

I make use of the `anti_join` function within the lines 550-552. If you look quickly at the relevant section of the Data Wrangling cheat sheet, you can see the following brief but helpful description of `anti_join`'s operation:

x1	x2
C	3

dplyr::anti_join(a, b, by = "x1")

All rows in a that do not have a match in b.

We know that when we used piped code, the output of the pipe enters the first argument of the receiving function. In our case, this argument `a` is therefore the `tidy_census_2001_health_all` object, which contains all the rows contained within `b`, `tidy_census_2001_health_BIG`, as well as many others. As the description to `anti_join` indicates, the function finds those rows in `a` that do not match those in `b`. This means it will return an output with fewer rows. This simple command gets us almost exactly what we want. The only remaining task is to then filter out "SCOTLAND" again, as this is at a different observational unit scale to that which we want to use.

There are many other ways that the above operation could be achieved. In particular, you may want to explore the use of the `setdiff` function for achieving the same result, or alternatively the use of the `filter` command.

Exercise 8.45: (Optional and after the course) Produce a variant of the above which includes both small place names and larger place names as two separate columns within the same dataset. Thinking in particular about the `group_by` and `summarise` functions, consider in which contexts this dataset could be useful. Send me your code, results and conclusions. (jonathan.minton@glasgow.ac.uk)

8.9.2. Binding rows

Imagine you had gone to the effort of tidying the equivalent self-assessed health tables, available for the same regions within Scotland, for the 1991 and 2011 censuses as well as the 2001 census. This means that, in addition to the `tidy_census_2001_health` table, you have comparable `tidy_census_1991_health` and `tidy_census_2011_health` tables as well. The names and order of the columns has been set up in exactly the same way in each of these three tables, the occupational categories, sex, and age groups labels have been set up to be identical (for example, the two sexes are not called 'Male' and 'Female' in one table and 'male' and 'female' in another table). You are now very close to being able to combine these tables into a single table which would allow you to explore changes in outcomes over time. However, because time is now something that varies between the three tables, you will have to manually add this as an additional where variable. Something like the following should work:

```
tidy_census_2001_health <- tidy_census_2001_health %>%
  mutate(year = 2001) %>%
  select(place, year, sex, age, health, occupational_group, count)

tidy_census_1991_health <- tidy_census_1991_health %>%
  mutate(year = 1991) %>%
  select(place, year, sex, age, health, occupational_group, count)

tidy_census_2011_health <- tidy_census_2011_health %>%
  mutate(year = 2011) %>%
  select(place, year, sex, age, health, occupational_group, count)

tidy_census_combined_health <- tidy_census_1991_health %>%
  bind_rows(tidy_census_2001_health) %>%
  bind_rows(tidy_census_2011_health) %>%
  arrange(year, place, sex, age)
```

Because the order of the each of the columns has been set the same way for each table, the `bind_rows` function should allow the three tables to be bound on top of each other. The amount of data cleaning and data tidying required to produce this combined health dataset will likely have been considerable, not least because the structure and format of census tables tends to vary with each census. However the resulting table can be very useful for a wide range of analyses, as indicated by the earlier examples. You may wish to save the outcome of all of your hard work so as to be able to skip easily to the more interesting data analysis stage. You can do this using the `write_csv` function:

```
write_csv(tidy_census_combined_health, path = "data/tidied/census_combined_health.csv")
```

Congratulations! You are now statistically and substantively significantly closer to being a data scientist!

9. Final Exercise (Experimental): A Data Science Re-introduction to model building and fitting in the tidyverse

The recent book by Hadley Wickham and Garrett Grolemund, **R for Data Science** attempts to provide a very broad coverage of data science tasks using the tidyverse in R. An interesting aspect of this book – which is also available online at the following address <http://r4ds.had.co.nz/> - is that statistical models are covered only at around two thirds of the way through, after the various technical challenges involved in first getting the data into usable formats are covered in detail. This seems analogous to a cookery course where people are asked to master basic skills – such as how to cook an egg and how to chop an onion – before being given a recipe to follow. This slowness in ‘cutting to the chase’ might feel a bit frustrating in terms of the sense of progress made during the course, but by teaching such basic skills explicitly and in great detail, but by making sure everyone has a good command of the basics you can spend more of your time thinking about more interesting methodological and technical challenges.

Another interesting aspect of the book is that the core concepts and ideas involved in statistical model building are taught and explained in an unusual way. Most of the delegates on this course are, based on background, likely to already have a good command of the principles of statistical inference and regression, but may not have encountered the approach used in this book, which to my mind makes the links between statistical model building and other data-based activity in R much clearer. Given I’ve taken on this course at very short notice (under two work days), for the final part of the course I’d like to work through the exercises in the first two chapters of the model building section of this book (chapters 22 and 23), and get some feedback on the following questions:

- Do you find the approach to describing the purpose and procedures of statistical modelling in R helpful and intuitive?
- Do you think this kind of approach would be an effective introduction to statistical theory and practice for people without the same expertise as you, or should only be taught after a more conventional, perhaps algebra-heavy, introduction?
- Does the approach used make the links between statistical models and programmatic considerations clearer, and if so does this help to demystify the core concepts and aims of statistical models?

We will start with chapter 22, focusing on reproducing the code in the essential exercises and discussing outputs, and see how far we get:

<http://r4ds.had.co.nz/model-intro.html>

Thank you for collaborating with me on this mutual learning exercise. If you have comments and feedback about this aspect of the course please email me at jonathan.minton@glasgow.ac.uk – and if you have any specific criticisms about this aspect of the course please blame me and not UBDC!

10. Day One Summary

Data tidying and data management is an enormous topic, and sadly the challenges involved in getting administrative data into shape do not tend to present themselves in order of difficulty, like a game, or allow certain stages to be skipped, like a cookery or arts and craft show.

Instead, the early stages of data tidying and management, such as identifying and correcting for typos in cells, can be some of the most technically challenging, and unless these challenges are

overcome, the more interesting and productive feeling data analysis tasks later on in data-to-value chain cannot start.

This is the reason why so much material was covered already in this course: without knowing how to start, and how to persist through technical and conceptual challenges, at each of the stages in this long chain, you will become stuck. Within this material I have taken you through almost the complete process for one particular dataset, and through this introduced both some of the thinking and some of the tools needed to keep on keeping on. Even if (as I expect) there was too much material to reasonably absorb in a single day, I hope this document will allow you to revisit, as often as necessary, particular examples and descriptions of processes, until they become familiar to you. In no particular order, some of the books, websites, and other references I would recommend to support you include:

- Fox, J, & Weisberg (2011) An R Companion to Applied Regression, 2nd Ed (London: Sage)
- Matloff, N (2011) The Art of R Programming (San Francisco, CA: No Starch Press)
- Gandrud, C (2013) Reproducible Research with R and RStudio (London, CRC Press)

As the ages of some of these books indicates, the internet is used increasingly to learn and keep up to date on R. Currently, the main forum for learning about R is Stack Overflow, a programmers' community with a rapidly growing archive of R problems and solutions:

<http://stackoverflow.com/>

Data Management and the Research Flywheel

As hinted at in Exercises 70 and 78, my main aim in teaching data management tools and techniques has been to greatly cut down the friction involved in exploring and understanding social phenomena through data analysis, and the ultimate reason for this is a belief that the relationship between evidence and theory should be bidirectional: looking at data makes it easier to develop simple (lower 'middle range') theories, and developing and testing these theories means looking again at the data, leading to the refinement or replacement of existing theories and the development of new theories, which in turn require further analysis of data, to new theories, to new data analysis, and so on and so on. Although a bunch (to use the in this case useful Americanism) of lower middle range theories may have enough in common to warrant being gathered together to form a more general middle middle range theory, and conversely a middle middle range theory may need to be separated (or 'operationalised') to form a series of empirically testable lower middle range theories, the process of generating knowledge and insight from data is often in practice about ensuring that the research flywheel keeps rotating smoothly from data to theory and back again. Effective data management practices and tools exist to allow for smoother running of this flywheel, and through this for better generation of knowledge and insight about social, economic, and health processes.

11. (ADDITIONAL BONUS MATERIAL FOR HOME)

11.1. Introduction

Yesterday we covered a lot of territory, following a particular dataset most of the way through the data-to-knowledge chain. Even with no statistics more complicated than a cumulative sum or a ratio, once we had the dataset in a 'tidy' format, with the first few columns 'where' variables and the last few columns 'what' variables, we were able to answer a range of questions fairly quickly, though not definitively, hopefully meaning that by the end of the session we understood more about some of the complex relationships that exist between health, gender, age, occupation, place and health status in Scotland than we did at the start. By being able to produce answers quickly, and relatively painlessly, using the tidied data, the costs of curiosity fall, and so the returns on being curious increase.

I expect there has been variation in the speed at which people have worked through the material for day one, with some people perhaps having progressed much faster than others. Because of this, I am offering a choice about what to do this morning.

- **Option A:** If you have already completed the various exercises introduced yesterday, or feel fairly confident with that material or simply want a wide breadth of coverage over the two days, two new areas are covered this morning: firstly, we will discuss some methods for outputting tables, and for producing data visualisations using the Wickhamese package `ggplot2`, which was introduced briefly late in day one; secondly, we will look at the split-apply-combine paradigm, and the `plyr` package, as a set of ideas and implementation of those ideas respectively, for the automation of many types of data management and analysis.
- **Option B:** Alternatively, you could continue to work on the exercises introduced yesterday. The introductions to outputting tables, producing figures, and automating processes will be in this handbook for you to take away, and if you wish to you can contact me for further information (jonathan.minton@glasgow.ac.uk).

During the second half of the day, an extended practice will be introduced, in which you will need to reproduce results recently published in a paper which has generated a great deal of attention in the last year. This will involve minimal instruction, both because there are multiple possible solutions to the problems involved, and because defining the tasks involved in doing research is itself a key part of the research process.

12. Exploring and Presenting Data Using Tables and Graphs

Many of the examples encountered yesterday involved starting with our tidied data, containing many thousands of rows, and repeatedly processing this data to produce tables with far fewer rows. This was largely achieved using various permutations of the `group_by` and the `summarise` functions, which together allow for various forms of meaningful (and sometimes not so meaningful) *data reduction* to be performed.

Less is More: The Need for Data Reduction

From a data science perspective, one of the main purposes of statistical analyses is to reduce the data: from many hundreds or thousands of values, to just a handful of values. And one of the main reasons for wanting or needing to reduce the data is that people are fundamentally limited in terms of the number of pieces of data we can make sense of at a single time. Though computers can keep track of millions of items of data with perfect accuracy, human short-term

memory is limited, generally, to between around four and nine items. Without drastically cutting down the number of facts and observations presented to us, data fails to be informative to us, and so the data-to-value chain has not been completed. But without a justifiable and explicit process for choosing how to turn many observations into just a single or a few choice numbers, data risks being misinformative to us, rather than simply uninformative. For example, it is common to find evaluations of events to depend much more heavily on the start and end of the event than on what happens during the middle, hence the importance of making a good ‘first impression’ at job interviews and so on. Similarly, memories and characterisations of people and places can be dominated by events that are salient rather than representative, and often are salient *because* they are non-representative. Intuitive and informal data reduction therefore tends to keep in mind the exceptions to the rules rather than the rules themselves. Summary statistics, backed by rigorously proven and carefully thought through theories of statistical inference, therefore provide an important corrective to natural intuition, making it easier to determine the signal within the noise, and to reduce data intelligently and informatively.

If you look at the Data Wrangling Cheat Sheet, on the second page, there is a column named ‘Summarise Data’, listing a series of summary functions.

Exercise 11.1: Go through each of the summary functions in the Cheat Sheet and identify those which are measures of central tendency, and those which are measures of variability. Consider one or two applications of each of these summary functions in either the tidy data example used in this course or your own work.

The range of summary functions which can be operated through summarise and group_by are not limited to those included in the Cheat Sheet, and many base R and additional functions can also be used. For example, we might want to use the quantile function, as follows, to report the lowest fifth and highest fifth population sizes by place:

```
> tidy_census_2001_health %>%
+   group_by(place) %>%
+   summarise(count = sum(count)) %>%
+   summarise(
+     lowest_quantile = quantile(count, 0.2),
+     highest_quantile = quantile(count, 0.8)
+   )
Source: local data frame [1 x 2]

  lowest_quantile highest_quantile
1          63263.2          160659.2
```

Exercise 11.2:

- Run the above code.
- Looking at the help file for quantile, work out which argument to change to report the median population size, and change the above code so that median values are reported using both the quantile function and the median function.
- (Optional): work out whether two calls to summary are needed.
- (Optional): explore what happens if you change the contents of the summary function to `quantiles = quantile(count, c(0.2, 0.8))`. Does this work as expected and if not, why not?

12.1. Exporting tables

So far, we have produced a large number of tables which display only as outputs within the R console. We have also used the View function, which opens up a tab on the top left pane which can be explored much like an Excel or SPSS table. However we have not yet created any tables which exist as files, accessible outwith our current R session. When we want to save these tables, a number of different approaches need to be taken. A range of options, and packages and functions useful for implementing that option, are summarised in the table on the following page.

Exercise 11.3: Looking at the table, respond to the following:

- Consider the relative advantages and disadvantages of each of the approaches outlined below. Why are they presented in the order in which they are presented?
- Consider why the third approach, outputting to the clipboard, would not be appropriate for large tables.
- What are the differences between `write.csv` and `write_csv`, and why does the `write.csv` example include the argument `row.names = F`?
- (Optional) The `xlsx` package uses Java to interface between R and Excel. Why might this be problematic?
- (Optional) If `table_object` (in the last example) has been created using `dplyr`, why was its class set to `data.frame` before being passed to `xtable`?

Exercise 11.4: Output one or two tables using one or two of the approaches outlined below.

Approach	Packages and functions used	Example code
Take a screengrab of the console output.	Either the 'PrtScn' button, or something like the program 'Snipping Tool'; combined with pasting into something like Powerpoint or Word.	Not applicable.
Copy the text from the console, into something like Word or Excel	Not applicable.	Not applicable.
Output the text of the table to the clipboard	write.csv, write.table and so on, with the file argument set to "clipboard". Use text import wizard to import into Excel.	[snipped] %>% write.csv("clipboard", row.names = F) [snipped] %>% write.table("clipboard", row.names = F)
Output the text of the table to a file	write.csv, write.table and so on, with the file argument set to "clipboard". Use text import wizard to import into Excel.	[snipped] %>% write_csv("table_file_name.csv")
Use R to write directly to Excel worksheets	The xlsx function, in particular functions such as createworkbook and addDataFrame	wb <- createworkbook() addDataFrame(x = table_name, sheet = createSheet(wb, sheetName = "tab_name")) saveWorkbook(wb, file = "tables/excel_workbook.xlsx")
Produce an html table or similar	Packages such as kable or xtable	class(table_object) <- "data.frame" print(xtable(tab), type="html", file="tables/table_name.html")

13. An introduction to the split-apply-combine strategy and ply

The package plyr is, as the name suggests, related to dplyr. The purpose of plyr is to implement the 'split-apply-combine' strategy for data analysis and data management tasks, as described in the following paper:

<https://www.jstatsoft.org/article/view/v040i01/v40i01.pdf>

Exercise 12.1: (After the course): Read the above paper.

As the abstract to this paper states:

Many data analysis problems involve the application of a split-apply-combine strategy, where you break up a big problem into manageable pieces, operate on each piece independently and then put all the pieces back together. This insight gives rise to a new R package that allows you to smoothly apply this strategy, without having to worry about the type of structure in which your data is stored.

Within dplyr, the `group_by` function is the main example of 'splitting', and the `summarise` function is the main example of 'applying'. As the output of a `group_by` followed by `summarise` function is a single data frame rather than collection of separate objects, dplyr also implicitly combines the results of 'split' (`group_by`) and 'apply' (`summarise`) processes. For example, when we write `summarise(count = sum(count))` we are applying the function `sum` to the contents of the variable `count`. By contrast, when we first group by place, we are splitting the data frame pieces according to the value of the place variable.

The functions in the dplyr package therefore apply the split-apply-combine strategy in cases where both the input (the thing being split) and the output (the result of being combined) are data frame class objects. The plyr package can be thought of as a generalisation of this strategy to cases where either the input or the output are of a different R object class. These different possible input and output classes, and the dplyr functions used to handle them, are summarised in table 2 of the paper:

<i>Input \ Output</i>	Array	Data frame	List	Discarded
Array	aapply	adply	alply	a_ply
Data frame	dapply	ddply	dlply	d_ply
List	lapply	ldply	llply	l_ply

Each of the cells in the table above contains a different function. The first character of the function name defines the input expected, and the second character of the function name defines the output expected. In effect, what dplyr does is provide an alternative to the `ddply` function within plyr. Of course, this means there is a lot of functionality that dplyr cannot replace.

Exercise 12.2: (Advanced and after the course): Use `ddply` to produce some of the same results achieved using dplyr code in some of the earlier examples. You may want to then pipe the output of a `ddply` call to the `tbl_df` function. Investigate and understand why.

Because dplyr takes over some of the functionality of plyr for cases where both the input and the output are of data.frame classes, some of dplyr's functions have the same name as plyr's functions. For example, both plyr and dplyr have their own version of the summarise function. When R encounters two functions, from two different packages, that share the same name, it uses the version of the function from the most recently loaded package in preference to those in packages loaded earlier. If we look at the start of the R script we have been using, I have loaded plyr before dplyr, meaning that the dplyr versions of the functions take precedence over the plyr versions. If the packages were loaded in the opposite order, first dplyr and then plyr, then many operations using dplyr would not work. If plyr has been loaded before dplyr, then it can be unloaded using the following (cumbersome) code:

```
detach("package:plyr", unload = T)
```

Using the same name for two functions is an example of what is known as 'overloading'. An example of overloading in ggplot2 is for the + symbol, which chains together instructions much as the %>% does within dplyr; two lines of code cannot be added together in the same way that two numbers can be, but as R can distinguish between two objects that are numeric values (say 4 and 6) and two objects that are instructions, it knows when the + symbol refers to a These symbols are known as operators, but in R operators, even simple ones like +, - and *, are really functions in disguise.

Exercise 12.3: (Optional): Type the following into the console

```
`+`(5, 2)
```

What does this produce and why? How does this result demonstrate that in R operators are really functions in disguise? Can this be applied to the chain operator %>% as well?

13.1. Example: Automating regression modelling

As an example of this, consider the following short script:

```
633 hmd <- read_csv("data/text/counts_germany_combined.csv")
634
635 # want to produce a regression of infant mortality rate against year for
636 # each country
637
638 fn <- function(input){
639   output <- lm(death_rate ~ year + sex, input)
640   return(output)
641 }
642
643 models <- hmd %>%
644   filter(age == 0) %>%
645   filter(sex != "total") %>%
646   mutate(death_rate = death_count / population_count) %>%
647   dply(., .(country), fn)
```

Exercise 12.4: Load the hmd dataset and explore its contents and structure.

The plyr function dply is used on the last line, and takes three arguments.

- The first argument is the input object, in this case the output of the pipe in line 646.
- The second argument defines how the input object should be split. In this example, by country, which is passed an argument to the .() function.

- The third argument, `fn`, is the name of a function which defines the process that should be applied to each piece produced following the split defined by the second argument.

As you can see in the above, I have defined the function to apply on lines 638 to 641. I have unimaginatively named the input argument 'input' and the function output 'output'. The output is produced from the `lm` ('linear model') function call, which takes the input (a dataset) as its second argument, and has as its first argument a formula similar to that used in the `xtabs` function. This formula, `death_rate ~ year + sex`, means 'regress `death_rate` on `year` and `sex`', with the `~` (tilde) symbol separating the response variable on the left side of the expression from the predictor variables on the right side of the expression, and the `+` symbol meaning these two predictor variables should be included independently, without interaction terms.

Exercise 12.5: (Optional) Explore model formula within R to identify how to change the above formula to include an interaction term between `year` and `sex`.

If you look at the help file for the `lm` function you will see that this function does not return a `data.frame` object, but instead a more complex object type of class `lm`. Further information in the help file states that "An object of class `"lm"` is a list...", and you can see that the `dplyr` function used is `dlply`, meaning that the input object is expected to be a `data.frame`, and the output object is expected to be a list.

Exercise 12.6: (Advanced and after the class) Learn more about different object classes in R and understand what is meant by the terms 'ragged' and 'non-ragged', and why `data.frame` objects are defined as 'non-ragged' lists.

List objects are important to understand in some detail, but the basic idea is that they are vectors of objects which may be of different classes, including other lists. This means that lists can contain lists which contain lists, and so on, producing a kind of tree like structure of objects within objects. Elements inside lists can either be accessed by position, or by name. As an example, consider the following:

```
> length(models)
[1] 47
> names(models)
 [1] "AUS"    "AUT"    "BEL"    "BGR"    "BLR"    "CAN"    "CHE"    "CHL"    "CZE"
[10] "DEUT"   "DEUTE"  "DEUTNP" "DEUTW"  "DNK"    "ESP"    "EST"    "FIN"    "FRACNP"
[19] "FRATNP" "GBR_NIR" "GBR_NP"  "GBR_SCO" "GBRCENW" "GBRTENW" "HUN"    "IRL"    "ISL"
[28] "ISR"    "ITA"    "JPN"    "LTU"    "LUX"    "LVA"    "NLD"    "NOR"    "NZL_MA"
[37] "NZL_NM" "NZL_NP" "POL"    "PRT"    "RUS"    "SVK"    "SVN"    "SWE"    "TWN"
[46] "UKR"    "USA"
> models[[3]]

Call:
lm(formula = death_rate ~ year + sex, data = input)

Coefficients:
(Intercept)      year      sexmale 
  2.702884    -0.001356    0.021552 

> models[["BEL"]]

Call:
lm(formula = death_rate ~ year + sex, data = input)

Coefficients:
(Intercept)      year      sexmale 
  2.702884    -0.001356    0.021552
```

The first line identifies that the length of the list produced by `dlply` is 47 elements long. The second command returns the names of each of these elements. These are the names of the countries by

which the data were split earlier. The third line shows how to access the contents of this third element by position: to access elements in lists, double square brackets `[[]]` are used, with the argument defining what to access contained within. The fourth line defines an alternative way of accessing this same element, including the name of the element “BEL” (Belgium) within the square brackets. The contents of `lm` objects can also be passed to the `summary` function, which provides further detail about the statistical significance of coefficients.

```
> summary(models[["BEL"]])

Call:
lm(formula = death_rate ~ year + sex, data = input)

Residuals:
    Min       1Q   Median       3Q      Max
-0.061642 -0.018828 -0.001536  0.016399  0.072493

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  2.703e+00  5.664e-02  47.720  < 2e-16 ***
year        -1.356e-03  2.938e-05 -46.156  < 2e-16 ***
sexmale      2.155e-02  2.958e-03   7.285  2.37e-12 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.02703 on 331 degrees of freedom
Multiple R-squared:  0.8684,    Adjusted R-squared:  0.8676
F-statistic: 1092 on 2 and 331 DF,  p-value: < 2.2e-16
```

Exercise 12.7: Perform the above using `dplyr`'s piping `%>%` operator.

It is also possible to burrow down further through list structures to extract specific elements of lists within lists, either using the double parentheses operator `[[]]`, or the `$` operator, or a combination of the two.

```
> models[["BEL"]] $coefficients[["year"]]
[1] -0.001355853$ 
```

```
> models[["BEL"]][["coefficients"]][["year"]]
[1] -0.001355853
```

Exercise 12.8: (Advanced) Learn more about differences between the `[[]]` and `$` operators.

Exercise 12.9: Access the `lm` object containing the regression results for the USA. How does this compare with those in Belgium?

This ability to ‘burrow’ through complex objects also means that equivalent and comparable objects can be extracted from multiple list objects. If the outputs from these list objects are compatible with other object types, then `plyr` can be used to coerce them to the new object types. For example consider the following code:


```

652 fn <- function(input){
653   output <- input$coefficients
654 }
655
656 ldply(models, fn)

```

Exercise 12.10: Run this code. Without looking below, describe what the function is doing.

Exercise 12.11: (Advanced) Using the output from the above code and `ggplot2`, produce a graph visualising these coefficients for each country. What does this tell you about differences between countries in infant mortality trends?

Note the symmetry in the use of the `plyr` functions: `ldply`, converting from a dataframe to a list, was used to produce the `models` object, and now `ldply`, converting from a list to a dataframe, was used to produce a dataframe of coefficients. If the coefficients were all you were interested in, this means `ddply` could be used to create the coefficients dataframe directly from the `hmd` dataframe, without the intermediate step.

Exercise 12.12: (Advanced and after the class) Use `ddply` combined with a more complex function (`fn`) to produce the coefficients data frame object in a single step. Explore methods for achieving the same using `dplyr`. (Hint: Explore the `'do'` function.) Comment and consider differences between the two approaches and their relative merits for this task. Send me your code, comments and results (jonathan.minton@glasgow.ac.uk).

13.2. Example: Automating the production of figures and files

Eagle-eyed readers will have noticed that one of the acceptable output types for `plyr` function is no output ('discarded'), produced using the `a_ply`, `d_ply` and `l_ply` functions. Although performing some analyses only to 'discard' the output may not seem like a useful activity, if we remember that some functions' main purpose lie in their 'side effects' rather than their output within the R environment, then the purpose of these functions becomes clearer. In particular, graphics are produced as 'side effects' of running `ggplot2` code and similar, and files are written as a 'side effect' of the `write_csv` function.

Consider now the following code example:

```

667 fn <- function(input){
668   this_country <- unique(input$country)
669
670   ggplot(input) +
671     geom_line(aes(x = year, y = death_rate, group = sex, linetype = sex)) +
672     labs(x = "Year", y = "Infant mortality rate", title = this_country)
673
674   ggsave(
675     paste0("figures/countries/", this_country, ".png"),
676     width = 8, height = 8, dpi = 150
677   )
678   NULL
679 }
680
681 dir.create("figures/countries", recursive = T)
682
683 hmd %>%
684   filter(age == 0) %>%
685   filter(sex != "total") %>% |
686   mutate(death_rate = death_count / population_count) %>%
687   d_ply(., .(country), fn, .progress = "text")

```

This code will create 47 different figures, each uniquely labelled, within the directory “figures/countries”. Because this takes an appreciable amount of time, an additional argument `.progress = “text”`, has been used to display a progress bar. This is not necessary but can be useful for showing that the code is executing correctly.

Exercise 12.13: Run the above code.

Exercise 12.14: (Advanced) Produce figures showing the mortality rate in under fives based on the above code.

The debug function

One reason I tend to use the above pattern when working with `plyr` function – first defining the function and then running `plyr`, rather than defining the function within the `plyr` function – is that it allows the debug function to be applied to the function to allow it to be tested and developed. The debug function takes a function name as its argument, and allows you to see and interact with the operations of the function one command at a time. This means that you can check that the inputs you expect `plyr` to pass to the function are those that are actually passed, the commands within the function work as you expect them to, and the output from each call to the function are what you expect them to be. This makes it much easier to pinpoint bugs in code.

In the above examples, to start debugging write

`debug(fn)`

And to finish debugging write

`undebug(fn)`

Exercise 10.15: (Advanced) Use the debug feature with one of the examples above.

13.3. Automating the reading and tidying of files

Though beyond the scope of this course, another useful application of `dplyr` is in reading, formatting, and rearranging large amounts of data contained in separate files. An example of this is the `hmd` dataset we’ve been using, which was produced by tidying and combining a large number of files.

Exercise 12.16: (Advanced and after the course) Register on the website mortality.org and explore the directory structure and file contents of zipped mortality data available from that website. Consider the processes required to convert from that format to the format used in the `hmd` dataframe used before. For further information please see my technical report: <http://www.demogr.mpg.de/papers/technicalreports/tr-2015-001.pdf>

13.4. Summary

In this second day we have introduced graphics using `ggplot2` and automated data management and analysis using `plyr`. Both of these should be considered conceptual extensions to the tidy data and data management and analysis processes and methods introduced in the first day. Though I expect few people will have completed all the exercises by this stage, this handbook is available to take home and to complete at home. If you would like any further help or support with this material feel free to contact me: jonathan.minton@glasgow.ac.uk . Further suggestions and comments about this course are welcome.

14. Extended Practical: Uncovering aggregation biases in all-cause mortality trends

14.1. Background

A recent paper by two economists, Anne Case and Angus Deaton, has generated a great deal of media attention in the USA. It focuses on changing mortality rates amongst middle-aged White Non-Hispanic (WNH) populations in the USA, from 1989 to 2013, arguing firstly that mortality rates within this age group have increased, while those in other developed world populations tended to fall; and secondly that external causes of death which are symptomatic of despair - in particular suicides, poisonings and drug related deaths – explain much of such trends.

A link to the paper is provided here:

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

Links to the paper's Altmetric page, and in particular reports about its findings on websites and newspapers, are below:

<https://pnas.altmetric.com/details/4715117>

<https://pnas.altmetric.com/details/4715117/news>

14.2. Uncovering Aggregation Bias

Alongside dozens of stories in mainstream media outlets, the statistician and blogger Andrew Gelman also discussed the paper in his blog:

<http://andrewgelman.com/>

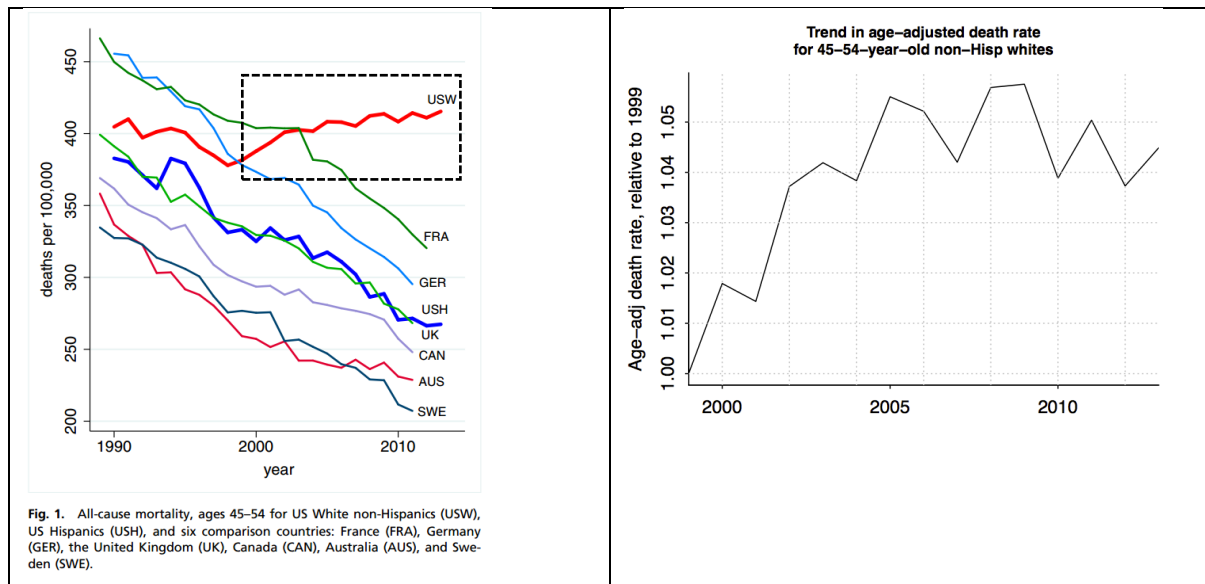
Whereas the mainstream media outlets tended to take the findings presented in Case & Deaton's paper on trust, Gelman sought to test the methodological robustness of one of the key findings: that mortality rates for 45-54 year old Non-Hispanic Whites (hereafter NHWs) in the USA have been increasing whereas mortality rates in other comparable countries have been decreasing.

Gelman's concern was that the mortality rate within this age group was not age adjusted sufficiently. In particular, the average age within the 10 year age strata may not have been constant over time, but instead may have increased, which could have explained part of the rise in mortality rates. Additionally males and females within the age group were grouped together, even though male and female all-cause mortality rate levels and trends may be quite different. Put more technically, the trends reported may have suffered from at least one of two types of *aggregation bias*, caused by:

- Grouping everyone between the ages of 45 and 54 years of age together in a single category;
- Grouping males and females together.

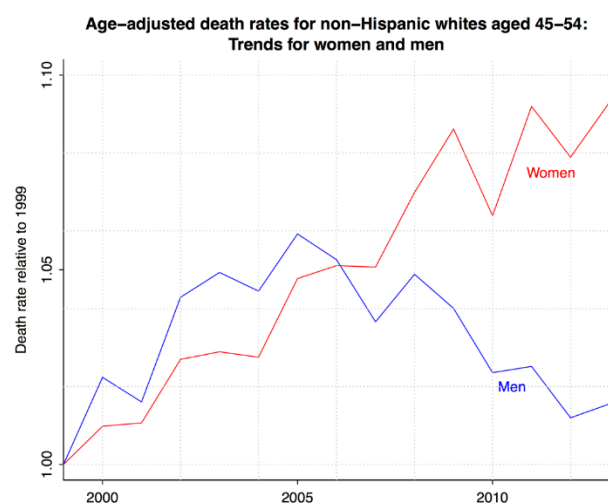
For the years 1999 to 2013, the death count and population count data are available by age in single years, rather than age in 10 year strata, as well as separately for males and females. This meant that both of these types of aggregation bias can be explored.

When Gelman did this he found both age-based and sex-based aggregation biases in the findings for all-cause mortality in the 45-54 age group. By extracting the data for males and females separately, and by age in single years, he was able to produce an age-adjusted trend in mortality risk in this age group from 1999 onwards. Below, the figure 1 from Case & Deaton's paper is shown on the left-hand side, and Gelman's adjustment shown on the right-hand side. A dashed line is used to indicate WNH trends from 1999 onwards.



As you can see, from 1999 onwards the trends continue to go up in Case & Deaton's figure, but in Gelman's figure they remain broadly static from around 2004 onwards. This qualitative difference in trends is the result of the age-aggregation bias.

Gelman showed that the trends in this age group are also qualitatively different for males and females, with the broadly static trend overall caused by a decline in mortality risk for males and a near equal-but-opposite increase for females, as shown below:



Within the following blog entry, Gelman also shows the mortality trends for different ages in single years, for different age groups, and for different ethnic groups:

<http://andrewgelman.com/2015/11/10/death-rates-have-been-increasing-for-middle-aged-white-women-decreasing-for-men/>

14.3. Purpose of this practical

The purpose of this extended practical session is to apply the techniques and methods taught over the last day and a half to explore the US mortality trends highlighted by Case and Deaton; the aggregation biases uncovered by Gelman; and other differences between sexes, age groups and

ethnic groups which emerged over the period 1999 to 2013. Doing this will involve the following tasks:

1. Navigating through the Centre for Disease Control WONDER database, accessible here: <http://wonder.cdc.gov/>
2. Using the WONDER database query system to extract data which will allow the following results to be reproduced:
 - a. the unadjusted trend reported in Case & Deaton's paper (all-cause, males and females combined, age group 45-54 years);
 - b. the age/sex disaggregations, mortality rates for ages in single years, and age-adjusted trends reported by Gelman;
 - c. Equivalent mortality trends in other ethnic and age groups
3. Formatting the data extracted and developing and saving code which will make it easy to look at and compare trends for a wide range of different population groups (different ethnicities, different age groups, different sexes)
4. Performing a series of analyses of the tidied data which replication both the results presented in Case & Deaton's paper, and Gelman's blog posts.

14.4. Data source

When you use the WONDER database, you will have to agree to a series of conditions about appropriate data usage. Because of this, the data have not been pre-extracted and pre-loaded for you, and so you will have to each agree to the terms and extract the data.

To access the relevant data, select:

1. 'Detailed Mortality' (within Mortality/Underlying Cause of Death). This should open up a page called 'About Underlying Cause of death, 1999-2014'
2. Review the Data Use Restrictions and conditions, and if you agree to these conditions, click 'I Agree'.
3. This will open up a request form, including a section called '1. Organize table layout:'. Make sure to select the following options, including giving a meaningful name for the data file to extract:

1. Organize table layout: Send Help

Group Results By

- And By** Year
- And By** Age Groups
- And By** Gender
- And By** Race
- And By** Hispanic Origin

Notes:

- Group Results By "15 Leading Causes" to see the top 15 rankable causes selected from the corresponding 113 or 130 Cause List. [More information.](#)

Measures (Default measures always checked and included. Check box to include any others.)

- ☒ Deaths
- ☒ Population
- ☒ Crude Rate

For crude rates:

- ☐ 95% Confidence Interval
- ☐ Standard Error
- ☐ Age Adjusted Rate
- ☐ 95% Confidence Interval
- ☐ Standard Error
- ☐ Percent of Total Deaths

Title

4. Scroll down to 'Select Demographics', and select the option 'Single-Year Ages'
5. Scroll down to the bottom of the page, and select the option 'export results', then click 'send'. A progress bar will appear labelled 'Processing Request', and after a few minutes a .txt file will be downloaded. **Note: The Progress Bar is faulty, and will continue to tell you to wait even after the file has been downloaded.**
6. **While the text file is downloading**, please read the above material carefully to learn more about the substantive topics which can be explored using this dataset.

- The dataset extracted should look as follows and be around 6MB in size. Move this file to your R project directory and create an R script for data tidying and analysis.

Notes	Year	Year	Code	Single-Year	Ages	Single-Year	Ages	Code	Gender	Gender	Code
Deaths	"1999"	"1999"	"< 1 year"	"0"	"Female"	"F"	"American Indian or Alaska Native"				
Unreliable	"1999"	"1999"	"< 1 year"	"0"	"Female"	"F"	"American Indian or Alaska Native"				
861.6	"1999"	"1999"	"< 1 year"	"0"	"Female"	"F"	"American Indian or Alaska Native"				
Applicable	"1999"	"1999"	"< 1 year"	"0"	"Female"	"F"	"American Indian or Alaska Native"				
"Total"	"1999"	"1999"	"< 1 year"	"0"	"Female"	"F"	"Asian or Pacific Islander"	"A-PI"			
	"1999"	"1999"	"< 1 year"	"0"	"Female"	"F"	"Asian or Pacific Islander"	"A-PI"			
	"1999"	"1999"	"< 1 year"	"0"	"Female"	"F"	"Asian or Pacific Islander"	"A-PI"			
"Total"	"1999"	"1999"	"< 1 year"	"0"	"Female"	"F"	"Asian or Pacific Islander"	"A-PI"			
	"1999"	"1999"	"< 1 year"	"0"	"Female"	"F"	"Black or African American"	"2054-			
	"1999"	"1999"	"< 1 year"	"0"	"Female"	"F"	"Black or African American"	"2054-			
1365.4	"1999"	"1999"	"< 1 year"	"0"	"Female"	"F"	"Black or African American"	"2054-			
"Total"	"1999"	"1999"	"< 1 year"	"0"	"Female"	"F"	"Black or African American"	"2054-			
	"1999"	"1999"	"< 1 year"	"0"	"Female"	"F"	"White" "2106-3"	"Hispanic or L			
	"1999"	"1999"	"< 1 year"	"0"	"Female"	"F"	"White" "2106-3"	"Not Hispanic			
	"1999"	"1999"	"< 1 year"	"0"	"Female"	"F"	"White" "2106-3"	"Not Stated"			
"Total"	"1999"	"1999"	"< 1 year"	"0"	"Female"	"F"	"White" "2106-3"				
"Total"	"1999"	"1999"	"< 1 year"	"0"	"Female"	"F"	"White" "2106-3"				
	"1999"	"1999"	"< 1 year"	"0"	"Male"	"M"	"American Indian or Alaska Native"	"1002-			
	"1999"	"1999"	"< 1 year"	"0"	"Male"	"M"	"American Indian or Alaska Native"	"1002-			
"Total"	"1999"	"1999"	"< 1 year"	"0"	"Male"	"M"	"American Indian or Alaska Native"	"1002-			
	"1999"	"1999"	"< 1 year"	"0"	"Male"	"M"	"Asian or Pacific Islander"	"A-PI"	"Hispa		
	"1999"	"1999"	"< 1 year"	"0"	"Male"	"M"	"Asian or Pacific Islander"	"A-PI"	"Not H		
	"1999"	"1999"	"< 1 year"	"0"	"Male"	"M"	"Asian or Pacific Islander"	"A-PI"	"Not S		
"Total"	"1999"	"1999"	"< 1 year"	"0"	"Male"	"M"	"Asian or Pacific Islander"	"A-PI"			
	"1999"	"1999"	"< 1 year"	"0"	"Male"	"M"	"Black or African American"	"2054-5"			
	"1999"	"1999"	"< 1 year"	"0"	"Male"	"M"	"Black or African American"	"2054-5"			
	"1999"	"1999"	"< 1 year"	"0"	"Male"	"M"	"Black or African American"	"2054-5"			
"Total"	"1999"	"1999"	"< 1 year"	"0"	"Male"	"M"	"Black or African American"	"2054-5"			

- Scroll to the end of the file and note the contents of the last few lines with the data/metadata distinction discussed yesterday in mind.

Hint: The dataset is a text file, but not a comma-separated value text file. Instead is a tab delimited file. Look at the function `read_delim`, and consider using the argument `delim = "\t"` within this function.

14.5. Data Tidying

The first, but perhaps most time consuming, task will be to take the above output and create a 'tidy' dataset with the following format:

- Year
- Age
- Ethnic Group
- Sex
- Death Count
- Population Count

The Ethnic Group category needs to include three mutually exclusive groups:

- White Non-Hispanics,
- Hispanics, and
- Black Non-Hispanics.

Note: This will require that information from both the Race and Hispanic Origin categories be combined.

Hint: Remember the `recode` function in the `car` package. Also look at the `ifelse` function and consider whether the combined category is best achieved in one or multiple steps).

Hint: Some of the raw data column names involve spaces. Remember that these can be accessed by placing the ``` symbol (top left button, below the escape key, on most keyboards) immediately before and after the object name. Using base R they can also be renamed to something easier to work within using the `names()` function.

14.6. Data Analysis

With the data in the format above there are a large number of potential analyses and comparisons which can be performed, and I hope you are intrigued by the data enough to want to explore it further after the workshop. Within the workshop here are six questions/tasks, covering both the post 1999 part of the figure presented in Case & Deaton's paper, and also some of Gelman's analysis:

1. Produce the death rate for White Non-Hispanics between the ages of 45 and 54 years inclusive for each year from 1999 to 2014.
2. Do the above, but for males and females separately.
3. Repeat the above, but index male and female death rates to their 1999 values
4. Produce the death rate trends, indexed to 1999 values, for each sex and age in single years
5. Perform the age adjustment: i.e produce the average age/sex specific trend in all-cause mortality within this period.
6. (Additional) What has happened to the male and female black/white mortality gap between 1999 and 2014?

Some hints and tips

- Indexing to 1999 values can be achieved using something like: `death_trend = death_rate / death_rate[year == 1999]`. The square brackets are another way of filtering, which can be performed within a `mutate` or `summarise` function.
- It is possible to use the `spread` and `gather` functions instead of the `[]` operator to index values to 1999 levels.
- Indexing to 1999 values can be achieved using the `group_by` function followed by the `mutate` function, rather than the `summarise` function,
- It may be helpful to present death rates per 100 000 population rather than per head.
- If graphing using `ggplot2`, the `group`, `linetype` and `colour` attributes within the `aes` function can all be useful for showing male and female trends separately.
- For visualising trends over time for separate groups using `ggplot2`, look at the `facet_wrap` and `facet_grid` functions.

14.7. Breakdown of practical sessions

The extended practical will include two sessions separated by a break. The tasks to achieve in each of these sessions are as follows:

- First session: Download the data. Skim the blog entries and paper. Produce the tidy data.
- Second session: Complete the analysis

A solution will be provided to the first session at the end of the second session, and a solution to the second session at the end of the second session.

Good luck!

Possible follow up activities

If you are interested in the problems – both methodological and socioeconomic – uncovered by the results of Case, Deaton and Gelman, then here are some additional analyses which you could perform to learn more:

- Compare mortality rates in the USA with age/sex equivalent trends in other countries. (n.b. the counts.csv dataset will be helpful for this. It contains death count and population count data from the Human Mortality Database).
- Compare differences in trends between Urban and Rural areas.
- Look at cause-specific mortality trends and their contribution to overall mortality trends. In particular, look at the contribution of external causes of deaths to overall trends within particular groups.
- Use plyr to automate the production and visualisation of trends for different groups.