Using R at Grattan Institute

Will Mackey and Matt Cowgill

2020-03-03

Contents

Welcome!

This guide is designed for everyone who uses - or would like to use - R at Grattan Institute.

The goal of this guide is to make it easy - even fun! - to use R at Grattan, whether you're doing analysis or reviewing someone else's analysis.

The guide does two main things:

1. Sets out some guidelines and good practices when using R at Grattan.
2. Shows you how to use R to undertake some of the analytical tasks you're likely to undertake at Grattan.

As a guide to using R, this website is helpful but incomplete. We can't possibly cover - or anticipate - all the skills you might need to know. If you make it to the end of this guide and want to learn more, start by reading R for Data Science by Hadley Wickham and Garrett Grolemund. It's free.

Because the guide is intended for everyone who uses R at Grattan, there may be some material that is too basic for more experienced users, or material that goes over the heads of beginners. That's OK - just skip those bits for now.

Any complaints or comments about this guide can be sent to Will or Matt, respectively.

This site was written in R with RMarkdown and the bookdown package.

Part I

What is R and why do we use it?

Chapter 1

Introduction to R

Most people reading this guide will know what R is. But if you don't - that's OK!

If you have used R before and are comfortable enough with it, you might want to skip to the next page. This page is intended for people who are unfamiliar with R. Soon, this will be you!

1.1 What is R?

R is a programming language. *That sounds scarier than it really is!* Don't freak out. R is software you use to work with data, just like Excel, Stata, or SPSS.

R is available for Windows, macOS and Linux. One of R's best features: it's free!

One of R's strengths is that it was designed by and for statisticians, data scientists, and other people who work with data. This can also be one of its weaknesses - statisticians aren't always the best at designing software that's easy to use out of the box.

R has a lot in common with other statistical software like SAS, Stata, SPSS or Eviews. You can use those software packages to read data, manipulate it, generate summary statistics, estimate models, and so on. You can use R for all those things and more. Everything you can do in Excel, you can (and generally should!) do in R. (See the next page for more on why we usually use R rather than Excel.)

You interact with R by writing code. This is a little different to Stata or SPSS (or Excel), which allow you to do at least part of your analyses by clicking on menus and buttons. This means the initial learning curve for R can be a

little steeper than for something like SPSS, but there are great benefits to a code-based approach to data analysis (see the next page for more on this).

R is quite old, having been first released publicly in 1995, but it's also growing and changing rapidly. A lot of developments in R come in the form of new add-on pieces of software - known as 'packages' - that extend R's functionality in some way. We cover packages more later in this page.

To analyse data with R, you will typically write out a text file containing code. This file - which we'll call a script - should be able to be read and executed by R from start to finish. Your script is like a recipe from a cookbook - it tells R all the steps that are needed to go from the raw ingredients (your data) to the finished product (the graphs or other finished product).

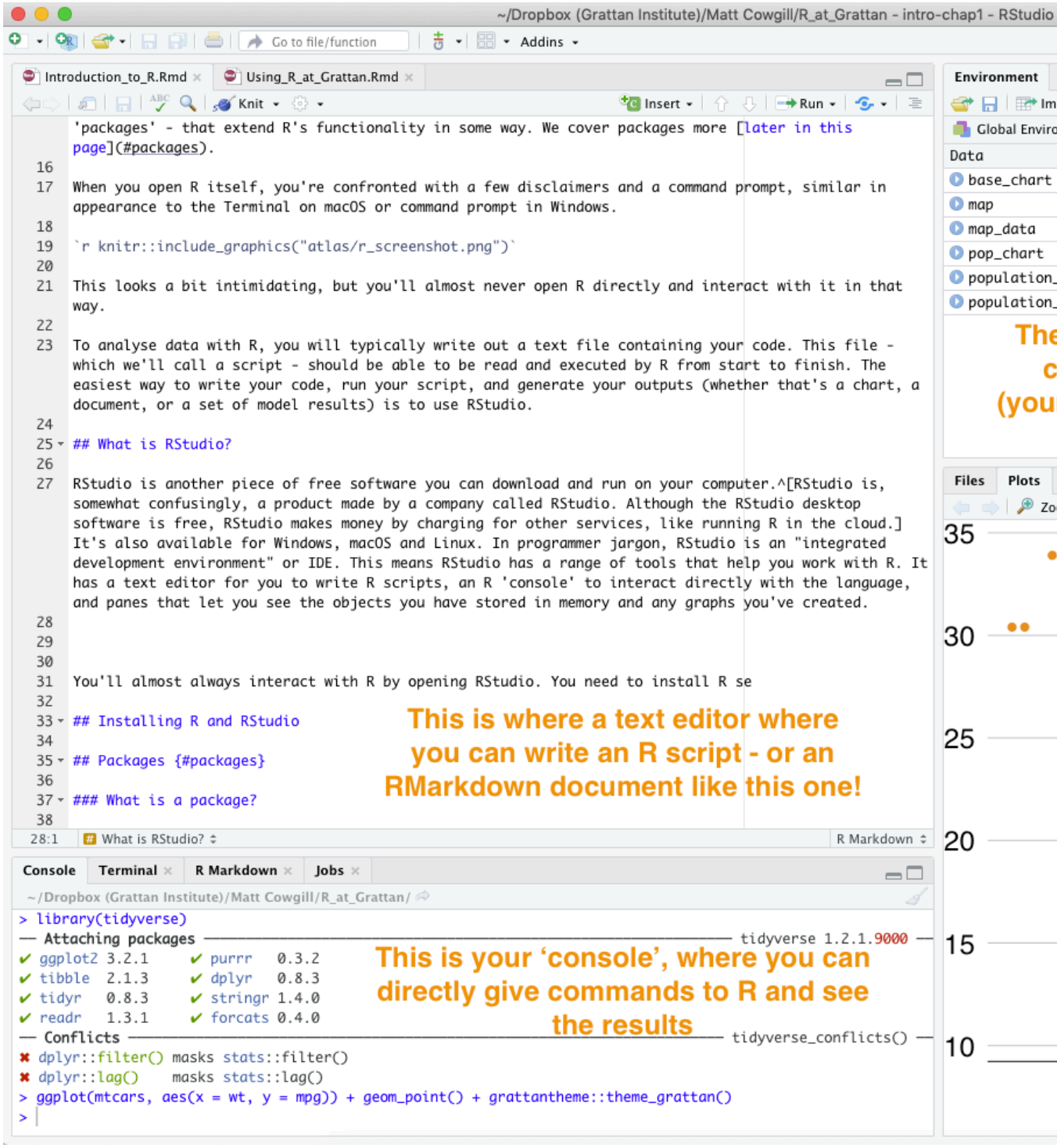
The easiest way to write your code, run your script, and generate your outputs (whether that's a chart, a document, or a set of model results) is to use RStudio.

1.2 What is RStudio?

RStudio is another piece of free software you can download and run on your computer.[1] Like R itself, RStudio is available for Windows, macOS and Linux.

In programmer jargon, RStudio is an "integrated development environment" or IDE. Translated to English, this means RStudio has a range of tools that help you work with R. It has a text editor for you to write R scripts, an R 'console' to interact directly with the language, and panes that let you see the objects you have stored in memory and any graphs you've created, among other things.

[1] RStudio is, somewhat confusingly, a product made by a company called RStudio. Although the RStudio desktop software is free, RStudio makes money by charging for other services, like running R in the cloud. When we refer to RStudio, we're referring to the desktop software unless we make it clear that we mean the company.

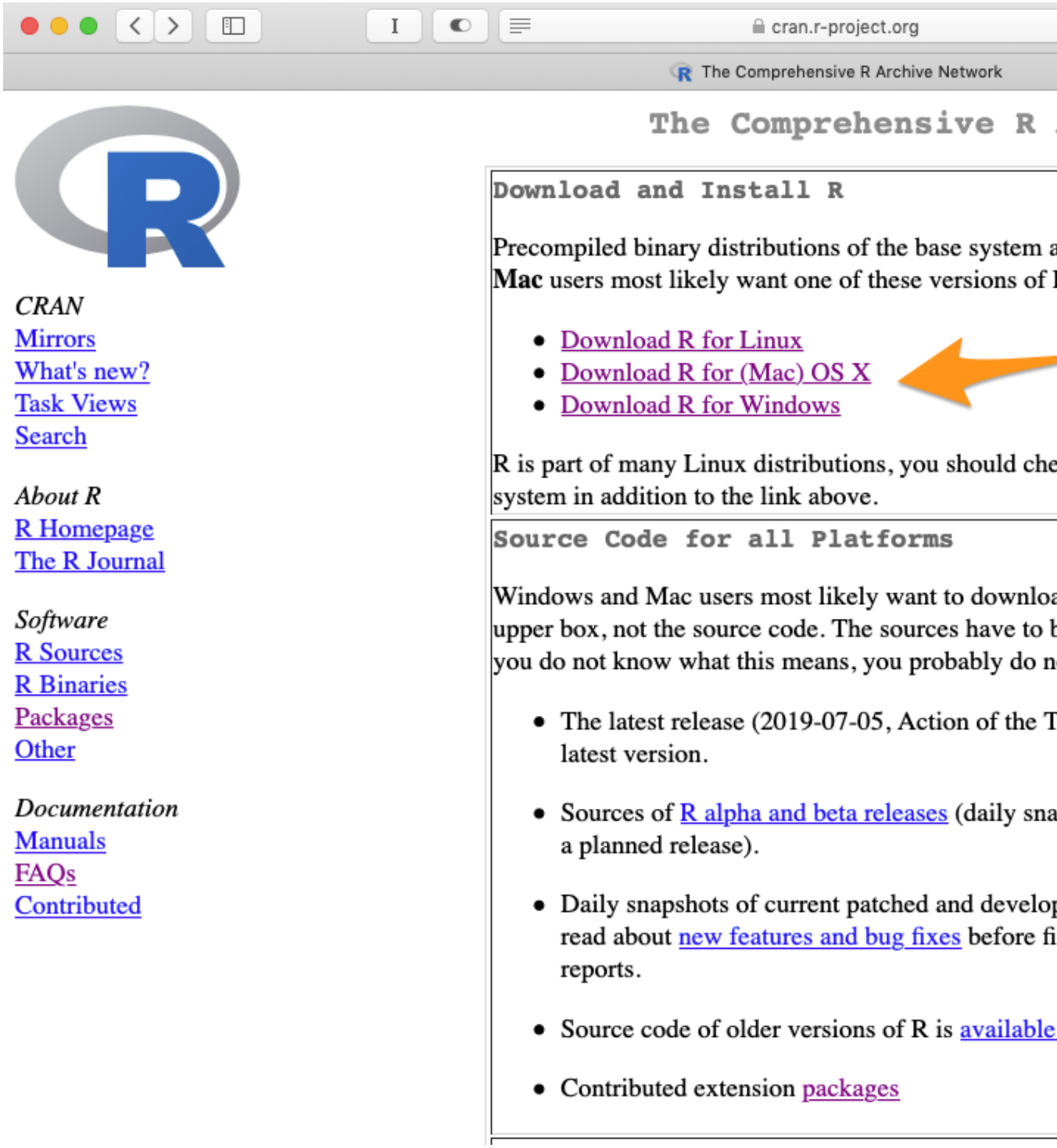

~/Dropbox (Grattan Institute)/Matt Cowgill/R_at_Grattan - intro-chap1 - RStudio
Go to file/function
Addins
Introduction_to_R.Rmd
Using_R_at_Grattan.Rmd
Knit
Insert
Run
'packages' - that extend R's functionality in some way. We cover packages more [later in this page](#packages).
When you open R itself, you're confronted with a few disclaimers and a command prompt, similar in appearance to the Terminal on macOS or command prompt in Windows.
`r knitr::include_graphics("atlas/r_screenshot.png")`
This looks a bit intimidating, but you'll almost never open R directly and interact with it in that way.
To analyse data with R, you will typically write out a text file containing your code. This file - which we'll call a script - should be able to be read and executed by R from start to finish. The easiest way to write your code, run your script, and generate your outputs (whether that's a chart, a document, or a set of model results) is to use RStudio.
What is RStudio?
RStudio is another piece of free software you can download and run on your computer.^[RStudio is, somewhat confusingly, a product made by a company called RStudio. Although the RStudio desktop software is free, RStudio makes money by charging for other services, like running R in the cloud.] It's also available for Windows, macOS and Linux. In programmer jargon, RStudio is an "integrated development environment" or IDE. This means RStudio has a range of tools that help you work with R. It has a text editor for you to write R scripts, an R 'console' to interact directly with the language, and panes that let you see the objects you have stored in memory and any graphs you've created.
You'll almost always interact with R by opening RStudio. You need to install R se
Installing R and RStudio
Packages {#packages}
What is a package?
This is where a text editor where you can write an R script - or an RMarkdown document like this one!
28:1
What is RStudio?
R Markdown
Console
Terminal
R Markdown
Jobs
~/Dropbox (Grattan Institute)/Matt Cowgill/R_at_Grattan/
> library(tidyverse)
Attaching packages
tidyverse 1.2.1.9000
ggplot2 3.2.1
purrr 0.3.2
tibble 2.1.3
dplyr 0.8.3
tidyr 0.8.3
stringr 1.4.0
readr 1.3.1
forcats 0.4.0
This is your 'console', where you can directly give commands to R and see the results
Conflicts
tidyverse_conflicts()
dplyr::filter() masks stats::filter()
dplyr::lag() masks stats::lag()
> ggplot(mtcars, aes(x = wt, y = mpg)) + geom_point() + grattantheme::theme_grattan()
Environment
Global Enviro
Data
base_chart
map
map_data
pop_chart
population_
population_
Files
Plots
35
30
25
20
15
10

You'll almost always interact with R by opening RStudio.

1.3 Installing R and RStudio

Although you'll usually work with R by opening RStudio, you need to install both R and RStudio separately.

Install R by going to CRAN, the Comprehensive R Archive Network. CRAN is a community-run website that houses R itself as well as a broad range of R packages.


cran.r-project.org
The Comprehensive R Archive Network
The Comprehensive R
CRAN
Mirrors
What's new?
Task Views
Search
About R
R Homepage
The R Journal
Software
R Sources
R Binaries
Packages
Other
Documentation
Manuals
FAQs
Contributed
Download and Install R
Precompiled binary distributions of the base system a
Mac users most likely want one of these versions of
Download R for Linux
Download R for (Mac) OS X
Download R for Windows
R is part of many Linux distributions, you should che
system in addition to the link above.
Source Code for all Platforms
Windows and Mac users most likely want to downloa
upper box, not the source code. The sources have to b
you do not know what this means, you probably do n
The latest release (2019-07-05, Action of the T
latest version.
Sources of R alpha and beta releases (daily sna
a planned release).
Daily snapshots of current patched and develop
read about new features and bug fixes before fi
reports.
Source code of older versions of R is available
Contributed extension packages

You want to download the latest base R release, as a 'binary'. Don't worry, you don't need to know what a binary is.

For macOS, the page will look like this:

CRAN
Mirrors
What's new?
Task Views
Search

About R
R Homepage
The R Journal

Software
R Sources
R Binaries
Packages
Other

Documentation
Manuals
FAQs
Contributed

R for M

This directory contains binaries for a base distribution and pa
OS 8.6 to 9.2 (and Mac OS X 10.1) are no longer supported
systems (which is R 1.7.1) here. Releases for old Mac OS X
found in the old directory.

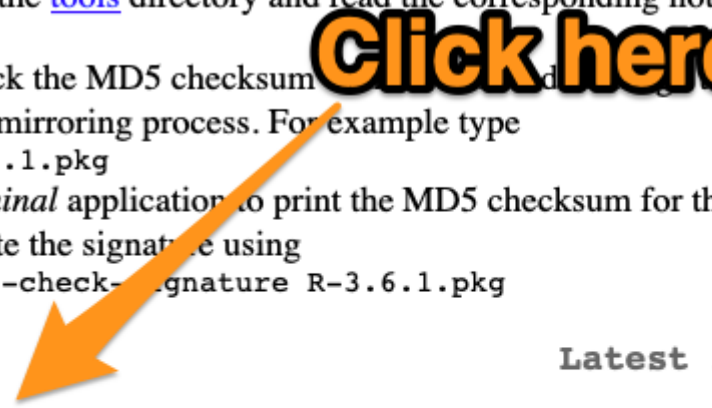
Note: CRAN does not have Mac OS X systems and cannot c
when assembling binaries, please use the normal precautions

As of 2016/03/01 package binaries for R versions older than
such versions should adjust the CRAN mirror setting accordi

R 3.6.1 "Action of the To

Important: since R 3.4.0 release we are now providing bina
toolkit to provide support for OpenMP and C++17 standard f
tools from the tools directory and read the corresponding not

Please check the MD5 checksum
during the mirroring process. For example type
`md5 R-3.6.1.pkg`
in the *Terminal* application to print the MD5 checksum for th
also validate the signature using
`pkgutil --check-signature R-3.6.1.pkg`



Latest

R-3.6.1.pkg
MD5-hash: 279e6662103dfe6a625b4573143cb995
SHA1-
hash: 4e932f8e5013870d2a9179b54eaee277f41657b0
(ca. 76MB)

R 3.6.1 binary for OS X
Contains R 3.6.1 frame
Tcl/Tk 8.6.6 X11 librar
are optional and can be
are only needed if you
package documentation

For Windows, you'll need to click on the 'base' version, and then click again to start the download.

CRAN
Mirrors
What's new?
Task Views
Search

About R
R Homepage
The R Journal

Software
R Sources
R Binaries
Packages
Other

Documentation
Manuals
FAQs
Contributed

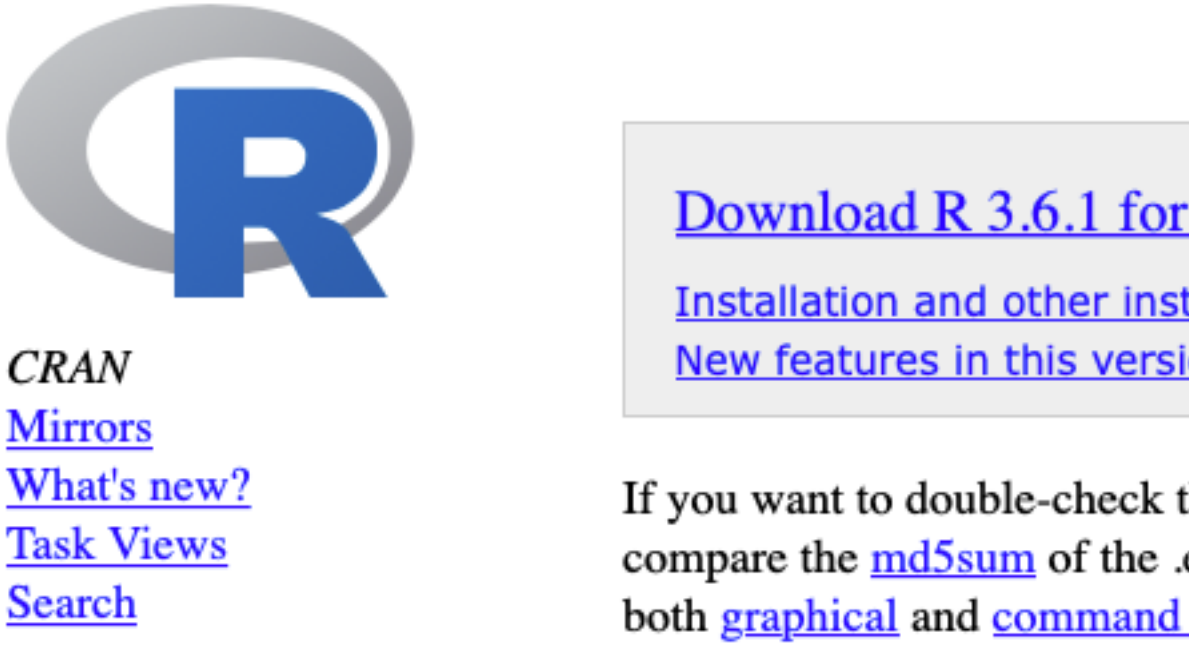
Subdirectories:

base	Binaries for base distrib
contrib	Binaries of contributed There is also informatio and corresponding envi
old contrib	Binaries of contributed managed by Uwe Ligge
Rtools	Tools to build R and R Windows, or to build R

Please do not submit binaries to CRAN. Pack
questions / suggestions related to Windows bi

You may also want to read the R FAQ and R

Note: CRAN does some checks on these bina
downloaded executables.



Once you've installed R, you'll need to install RStudio. Go to the RStudio website and install the latest version of RStudio Desktop (open source license).

Once they're both installed, get started by opening RStudio.

1.4 Learning more about R

This guide will show you how to use R at Grattan. But it is not a comprehensive tool for learning R. The book R For Data Science by Garrett Grolemund and Hadley Wickham is a great resource that will help you go from being a beginner to being able to do real-world analysis. The book is available for free online. There's even an active R for Data Science community online that shares tips and solutions to R problems.

1.5 `#r_at_grattan`

A great way to learn about R is to ask a Grattan person! Pose a question in the `#r_at_grattan` channel in Grattan Slack and someone will be sure to answer it.

At Grattan, none of us is a programmer first and foremost. We're a motley crew of economists, lawyers, doctors, scientists and philopsophers who have learned how to code so we can work with data. Don't feel bad if you don't know what you're doing yet – we've all been there and are happy to help you get up to speed.

Chapter 2

Why use R?

We can break this question into two parts:

1. Why use script-based software to analyse data?
2. Why use R, specifically?

2.1 Why use script-based software?

It's important for our analyses to be **reproducible**. This means that all of the steps that were taken to go from your raw data to your final outputs are clearly set out and can be reproduced if necessary.

Reproducibility is very important for quality control ("QC"), particularly of complex analyses - if it's not clear what you've done, it's hard for someone to check your work. It also makes things easier for you in the future - coming back to an old analysis a few months or years down the track is much easier if it's reproducible. At Grattan, most of us rotate from program to program periodically – your colleagues will probably need to revisit your work at some point in the future, and they'll thank you if it's in a reproducible script.

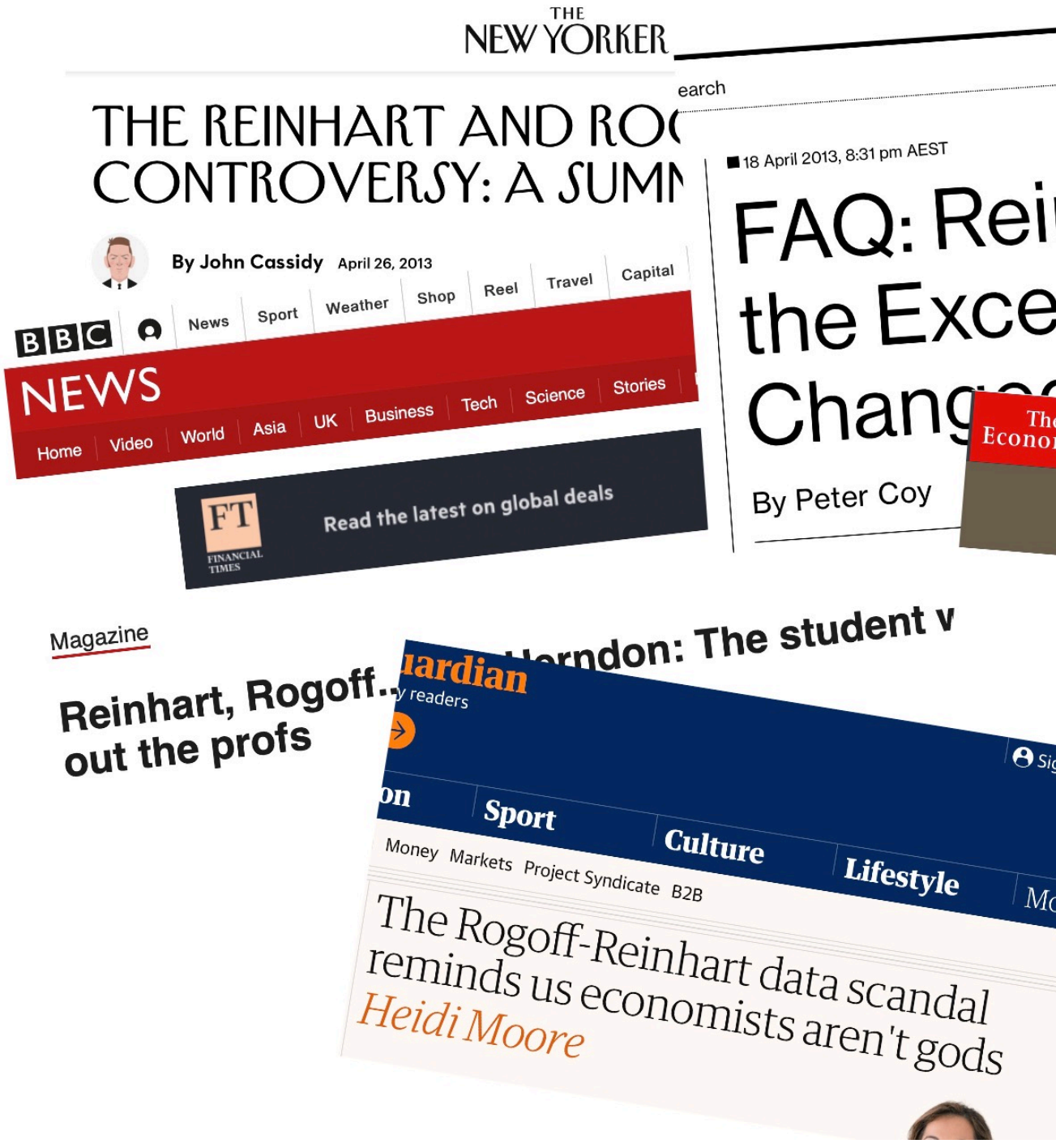
Script-based analyses are more likely to be reproducible.[1] A script sets out all the steps that were taken from reading in data, to tidying it, to estimating models or summary statistics and generating output.

Analysis that isn't script based, like work done in Excel, is almost never reproducible. It is generally unclear what steps were taken, in which order, to go

[1] Using a script-based approach doesn't guarantee that your analysis will be truly reproducible. If your work involves some steps that aren't documented in the script - such as data "cleaning" in Excel - then it is not fully reproducible. If your script will only run on your machine - because there are undocumented options, for example - it is not reproducible.

from the raw data to the output. It isn't even always clear in a spreadsheet what is the 'raw data' and what has been modified in some way.

Using scripts makes us less susceptible to the sort of errors famously made by the economists Reinhart and Rogoff in their Excel-based analysis of the effect of public debt on economic growth. It's still quite possible to make errors in a script-based analysis, but those errors are easier to find when the analysis is more transparent.


THE NEW YORKER
earch
THE REINHART AND RO
CONTROVERSY: A SUMM
By John Cassidy April 26, 2013
18 April 2013, 8:31 pm AEST
FAQ: Rei
the Exce
Chang
By Peter Coy
BBC
News Sport Weather Shop Reel Travel Capital
NEWS
Home Video World Asia UK Business Tech Science Stories
FT FINANCIAL TIMES
Read the latest on global deals
Magazine
Reinhart, Rogoff..
out the profs
uardian
y readers
Sport Culture Lifestyle
Money Markets Project Syndicate B2B
The Rogoff-Reinhart data scandal
reminds us economists aren't gods
Heidi Moore

Script-based analysis software also allows us to:

- Work with larger data sets;
- Work with data in a broader range of formats;
- Easily combine different data sets;
- Automate tasks and build from previous analyses; and
- Estimate a broad range of statistical models.

2.2 Why use R specifically?

Doing reproducible, script-based, research doesn't necessarily involve using R. It's perfectly possible to do reproducible work in Stata or Python (though somewhat harder in SPSS, where data is often manipulated by clicking things).

We use R specifically because:

- It's free!
- It's open source.
- It's powerful, particularly when it comes to statistics and data science.
- It's flexible and customisable.
- It has an active community extending its capabilities all the time and providing support online.
- It can be used to make publication-quality graphs.
- It's becoming the norm in academic research and common in the corporate world.

Part II

Using R the Grattan way

Chapter 3

Organising an R project at Grattan

All our work at Grattan, whether it's in R or some other software, should heed the "hit by a bus" rule. If you're not around, colleagues should be able to access, understand, verify, and build on the work you've done.

Organising your analysis in a predictable, consistent way helps to make your work reproducible by others, including yourself in the future. This is really important! If your analysis is messy, you're more likely to make errors, and less likely to spot them. Other people will find it hard to check your analysis and you'll find it harder to return to it down the track.

This page sets out some guidelines for organising your work in R at Grattan. It covers:

- Using RStudio projects and relative filepaths;
- Using a consistent subfolder structure; and
- Naming your scripts and keeping them manageable.

Using a consistent coding style also helps make our work more shareable; that's covered on the next page.

3.1 Use RStudio projects, not `setwd()`

In Excel, your data, code and output generally all live together in one file. In R, you have a script, which will usually load some data, do something to it, and save some output. You end up with multiple files - the raw data, your script, and some output. Your R script is like a recipe in a cookbook - when R is

cooking your recipe, it needs to know where to find your ingredients (the data) and put the finished product (your delicious analysis).

When it's executing your script, R needs to know where to read files from and save files to on your computer. By default, it uses your working directory. Your working directory is shown at the top of your console in RStudio, or you can find out what it is by running the command `getwd()`.

You can tell R which folder to use as your working directory by using the command `setwd()`, as in `setwd("~/Desktop/some random folder")` or `setwd("C:\Users\mcowgill\Documents\Somerandomfolder")`. **This is a bad idea that you should avoid!** If anyone - including you - tries to run your script on a different machine, with a different folder structure, it probably won't work. If people can't get past the first line when they're trying to run your script, there's an annoying and unnecessary hurdle to reproducing and checking your analysis.

In the words of Jenny Bryan:

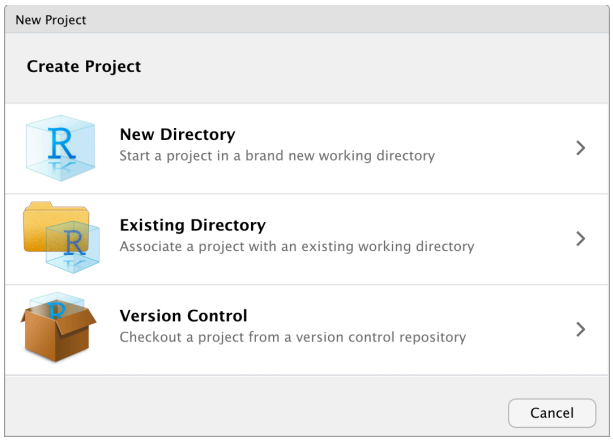
> if the first line of your R script is `setwd("C:\Users\jenny\path\that\only\I\have")` I will come into your office and SET YOUR COMPUTER ON FIRE.

Seems fair.

Creating a 'project' in RStudio sets your working directory in a way that's portable across machines.

3.1.1 How to create a project

Creating an RStudio project is straightforward: **click File, then New Project**. You can then choose to start your project in a new directory, or an existing directory. Simple!



RStudio will then create a file with an .Rproj extension in the folder you've chosen.

3.1.2 Opening a project

When you want to work on a particular project, just open the `.Rproj` file, or click File -> Open project in RStudio. Your working directory will be set to the directory that contains the .Rproj file.

3.2 Use relative filepaths

A benefit of using RStudio projects is that you can use relative filepaths rather than machine-specific filepaths. Machine-specific filepaths not only stop you from sharing your work with others, they're also super annoying for you! Who wants to type out a full filepath everytime you load or save a file?

Bad, machine-specific filepaths, boo, hiss

```
hes <- read_csv("/Users/mcowgill/Desktop/hes1516.csv")
hes <- read_csv("C:\Users\mcowgill\Desktop\hes1516.csv")
grattan_save("/Users/mcowgill/Desktop/images/expenditure_by_income.pdf")
```

Instead, use relative filepaths. These are filepaths that are relative (hence the name) to your project folder, which you set by creating an RStudio project.

Good, relative filepaths, cool, yay

```
hes <- read_csv("data/HES/hes1516.csv")
grattan_save("atlas/expenditure_by_income.pdf")
```

The first example above tells R to look in the 'data' subdirectory of your project folder, and then the 'HES' subdirectory of 'data', to find the 'hes1516.csv' file. This file path isn't specific to your machine, so your code is more shareable this way.

At Grattan, we even have our own R package, called grattandata that helps load certain types of data in R in a way that makes your script portable and reusable by colleagues. We cover that more in the Reading Data chapter.

3.3 Keep your stuff together

Your script(s), data, and output should generally all live in the same place. [1] That place should be the project folder - that's the folder that contains the

[1] This isn't always possible, like when you're working with restricted-access microdata. But unless there's a really good reason why you can't keep your data together with the rest of

.Rproj file that was created when you created an RStudio project (you did that, right? Scroll back up the page if not).

Don't just put everything in your project folder itself. This can get really overwhelming and confusing, particularly for anyone trying to understand and check your work. Instead, separate your code, your source data, and your output into subfolders.

A good structure is to have a subfolder for:

- your code - called 'R'
- your source data - called 'data'
- your graphs - called 'atlas', like in our LaTeX projects
- your non-graph output, like formatted tables, called 'output'

Sometimes your data folder might have subfolders - 'raw' for data that you've done nothing to, and 'clean' for data you've modified in some way. Don't keep 'raw' data together in the same place as data you've modified.

Other folder structures are OK and might make more sense for your project. The important thing is to **have** a folder structure, and to use a structure that is easily comprehensible to anyone else looking at your analysis.

3.4 Keep your scripts manageable

Unless your project is very simple, it's not a good idea to put all your work into one R script. Instead, break your analysis into discrete pieces and put each piece in its own file. Number the files to make it clear what order they're supposed to be run in.

Here's a useful structure:

- 01_import.R
- 02_tidy.R
- 03_model.R
- 04_visualise.R

You don't need to use these filenames. Think about what works best for your project.

It should be clear what each script is trying to do. Use meaningful filenames that clearly indicate the overarching purpose of the script. Use comments to explain why you're doing things. Err on the side of over-commenting, rather than under-commenting (we cover this more elsewhere in this guide). At the end of each script, you can save the script's output, and then load the file you create in the next script.[^Alternatively, `source()` the previous script.]

---

your work, you should do it.

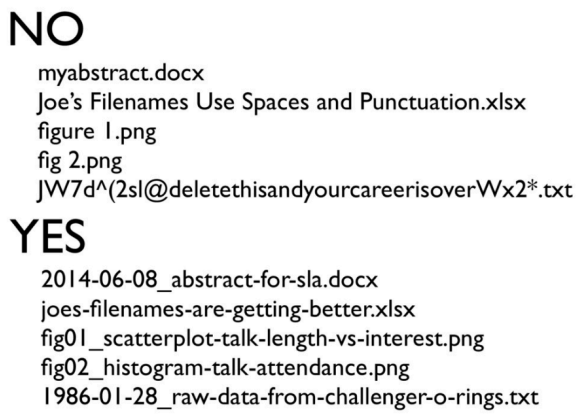
3.5 Make your filenames readable by both machines and humans

Have another look at the example filenames set out above:

- 01_import.R
- 02_tidy.R
- 03_model.R
- 04_visualise.R

They're sortable - they start with a number. They don't have spaces, so any and all software should be able to handle them. And, even though they're short and minimal, they give humans a good idea about what the files do. These are what you should strive for when choosing filenames.

For more on good principles for naming files, see this excellent presentation by Jenny Bryan, which includes the following examples:



Don't create multiple versions of the same script (like `analysis_FINAL_002_MC.R` and `analysis_FINALFINAL_003_MC_WM.R`.) We're all familiar with this hellish scenario: you do some work in a Word document (shudder, shudder, the horror, etc.), email it to a colleague, the colleague edits it and sends it back with a tweaked filename, like `cool_word_doc_002.docx`. Soon enough your hard drive and email client is cluttered with endless iterations of your document. Try to avoid replicating this nightmare in R.

If you do end up with multiple versions, put everything other than the latest version in a subfolder of your "R" folder, called "R/archive". To avoid a horrible mess of `analysis_FINAL_002.R` type documents cluttering up your folder, consider using Git for version control.

3.6 Include a README file

Your analysis workflow might seem completely obvious to you. Let's say that in one script you load raw ABS microdata, run a particular script to clean it up, save the cleaned data somewhere, then load that cleaned data in a second script to produce a summary table, then use a third script to produce a graph based on the summary table. Easy!

Except that might not seem easy or self-explanatory to anyone who comes along and tries to figure out how your analysis works, including you in the future.

Make things easier by including a short text file - called README - in the project folder. This should explain the purpose of the project, the key files, and (if it isn't clear) the order in which they should be run. If you got the data from somewhere non-obvious, explain that in the README file.

3.7 Keep your workspace clean

Sometimes R doesn't behave the way you expect it to. You might run a script and find it works fine, then run it again and find it's producing some strange output. This can be the result of changes in your R environment. You can set different options in R, which can (silently!) affect things. Or maybe you had some different objects - data, functions - defined in your environment the second time round that you didn't have originally, or some extra packages loaded.

To avoid this situation, keep your workspace tidy. When you load a script, do it in a fresh R session.

But... don't clean your workspace within your analysis script. People sometimes do this using this command:

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

This removes all objects from your environment. But it doesn't completely clear your R environment, and it doesn't do anything to any packages you have loaded. As Jenny Bryan puts it, this command is "a red flag, because it is indicative of a non-reproducible workflow."

3.8 Quick guide to starting a project

When you're starting a new project:

1. Open RStudio;
2. Click 'File -> New project'
3. Click 'New Directory'

4. Click 'New Project'
5. Give your new project a name, choose where it should go, and click 'Create Project'
6. Create subfolders in your project folder using the 'New Folder' button (by default in the lower-right pane of RStudio) - start with an 'R' folder
7. Click 'File -> New File -> R Script'
8. Save your R script in your R folder.

Now you've got a good shell of a project - a dedicated folder, with an associated RStudio project, and at least one subfolder. This is a good base to start your work.

Chapter 4

Grattan coding style

This page sets out the core elements of coding style we use at Grattan. If you're new to R, don't stress about remembering - or even understanding - everything on this page. Just be aware that we have a coding style, and come back to this when you're a bit further along.

The benefits of a common coding style are well explained by Hadley Wickham:

> Good style is important because while your code only has one author, it'll usually have multiple readers. This is especially true when you're writing code with others. In that case, it's a good idea to agree on a common style up-front.

Below we describe the **key** elements of Grattan coding style, without being too tedious about it all. There are many elements of coding style we don't cover in this guide; if you're unsure about anything, consult the `tidyverse` guide.

You should also see the Using R at Grattan page for guidelines about setting up your project.

A core principle for coding at Grattan is that your code should be **readable by humans**.

4.1 Load packages first

Our analysis scripts will almost always involve loading some packages. These should be laoded at the top of a script, in one block like this:

```
library(tidyverse)
library(grattantheme)
```

If you're loading a package from Github, it's a good idea to leave a comment to say where it came from, like this:

```
library(tidyverse)
library(grattantheme)
library(strayr) # remotes::install_github("mattcowgill/strayr")
```

Don't scatter `library()` calls throughout your script - put them all at the top.

The only thing that should come before loading your packages is the script preamble.

4.2 Script preamble

Describe what your script does in the first few lines using comments or within an RMarkdown document.

Good

```
# This script reads ABS data downloaded from TableBuilder and combines into a single d
```

Your preamble might also pose a research question that the script will answer.

Good

```
# Do women have higher levels of educational attainment than men, within the same geog
```

Your preamble shouldn't be a terse, inscrutable comment.

Bad

```
# make ABS ed data graph
```

If it's hard to concisely describe what your script does in a few lines of plain English, that might be a sign that your script does too many things. Consider breaking your analysis into a series of scripts. See Organising R Projects at Grattan for more.

Your preamble should anticipate and answer any questions other people might have when reviewing your script. For example:

Good

```
# This script calculates average income by age group and sex using the ABS Household E
```

The preamble should pertain the the code contained in the specific script. If you have comments or information about your analysis as a whole, put it in your README file.

4.3 Use comments

Comments are necessary where the code *alone* doesn't tell the full story. Comments should tell the reader **why** you're doing something, rather than just **what** you're doing.

For example, comments are important when groups are coded with numbers rather than character strings, because this might not be obvious to someone reading your script:

Necessary to comment

```
data %>%
  filter(gender == 1,    # Keep only male observations
         age == "05")    # Keep only 35-39 year-olds.
```

Without the comment, readers of your code might not be aware that `1` in this dataset corresponds to `male`, or that `age == "05"` refers to 35-39 year olds. Without the comment, the code is not self-explanatory.

If your code *is* self-explanatory, you can include or omit comments as you see fit.

Not necessary (but okay if included)

```
# We want to only look at women aged 35-39
data %>%
  filter(gender == "Female",
         age >= 35 & age <= 39)
```

You should also include comments where your code is more complex and may not be easily understood by the reader. If you're using a function from a package that isn't commonly used at Grattan, include a comment to explain what it does.

Err on the side of commenting more, rather than less, throughout your code. Something may seem obvious to you when you're writing your code, but it might not be obvious to the person reading your code, even if that person is you in the future. Better to over-comment than under-comment.

Comments can go above code chunks, or next to code - there are examples of both above.

4.4 Breaking your script into chunks

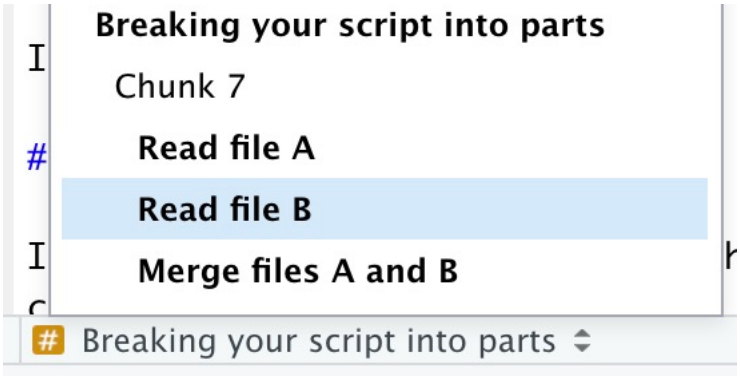
It's useful to break a lengthy script into chunks with `-----`.

Good

```
# Read file A -----

a <- read_csv("data/a.csv")

# Read file B -----

b <- read_csv("data/b.csv")

# Combine files A and B ----

c <- bind_rows(a, b)
```

(In practice, you'll have more than one line of code in each block.)

This helps you, and others, navigate your code better, using the navigation tool built in to RStudio. In the script editor pane of RStudio, at the bottom left, there's a little navigation tool that helps you easily jump between named sections of your script.



Breaking your script into chunks with ----- also makes your code easier to read.

4.5 Assigning values to objects

In R, you work with objects. An object might be a data frame, or a vector of numbers or letters, or a list. Functions can be objects, too.

Use the <- operator to assign values to objects. Here are some **good** examples:

```
schools <- read_csv("data/schools_data.csv")

three_letters <- c("a", "b", "c")
```

```
lf <- labour_force %>%
  filter(status != "NILF")
```

Avoid `->`, `=` and `assign()`. Here are some **bad** examples::

```
schools = read_csv("data/schools_data.csv")

assign("three_letters", c("a", "b", "c"))

labour_force %>%
  filter(status != "NILF") -> lf
```

All these bad operators will work, but they are best avoided. The `=` operator is avoided for reasons of visual consistency, style, and to avoid confusion. `assign()` is avoided because it can lead to unexpected behaviour, and is usually not the best way to do what you want to do. The `->` operator is avoided because it's easy to miss when skimming over code.

The `<<-` operator should also be avoided.

4.6 Naming objects and variables

It's important to be consistent when naming things. This saves you time when writing code. If you use a consistent naming convention, you don't need to stop to remember if your object is called `ed_by_age` or `edByAge` or `ed.by.age`. Having a consistent naming convention across Grattan also makes it easy to read and QC each other's code.

Grattan uses *words separated by underscores* `_` (aka 'snake_case') to name objects and variables. This is common practice across the Tidyverse. Object names should be descriptive and not-too-long. This is a trade-off, and one that's sometimes hard to get right. However, using snake_case provides consistency:

Good object names

```
sa3_population
gdp_growth_vic
uni_attainment
```

Bad object names

```
sa3Pop
GDPgrowthVIC
uni.attainment
```

Variable names face a similar trade-off. Again, try to be descriptive and short using snake_case:

Good variable names

```
gender
gdp_growth
highest_edu
```

Bad variable names

```
s801LHSAA
gdp.growth
highEdu
chaosVar_name.silly
var2
```

When you load data from outside Grattan, such as ABS microdata, variables will often have bad names. It is worth taking the time at the top of your script to rename your variables, giving them consistent, descriptive, short, snake_case names. An easy way to do this is using `clean_names()` function from the `janitor` package:

```
df_with_bad_names <- data.frame(firstColumn = c(1:3),
                                Second.column = c(4:6))

df_with_good_names <- janitor::clean_names(df_with_bad_names)

df_with_good_names
```

```
##   first_column second_column
## 1            1             4
## 2            2             5
## 3            3             6
```

The most important thing is that your code is internally consistent - you should stick to one naming convention for all your objects and variables. Using snake_case, which we strongly recommend, reduces friction for other people reading and editing your code. Using short names saves effort when coding. Using descriptive names makes your code easier to read and understand.

4.7 Spacing

Giving your code room to breathe greatly helps readability for future-you and others who will have to read your code. Code without ample whitespace is hard to read, justasitishardertoreadEnglishsenteceswithoutspaces.

4.7.1 Assign and equals

Put a space each side of an assign operator `<-`, equals `=`, and other 'infix operators' (`==`, `+`, `-`, and so on).

Good

```
uni_attainment <- filter(data, age == 25, gender == "Female")
```

Bad

```
uni_attainment<-filter(data,age==25,gender=="Female")
```

Exceptions are operators that *directly connect* to an object, package or function, which should **not** have spaces on either side: `::`, `$`, `@`, `[`, `[[`, etc.

Good

```
uni_attainment$gender
uni_attainment$age[1:10]
readabs::read_abs()
```

Bad

```
uni_attainment $ gender
uni_attainment$ age [ 1 : 10]
readabs :: read_abs()
```

4.7.2 Commas

Always put a space *after* a comma and not before, just like in regular English.

Good

```
select(data, age, gender, sa2, sa3)
```

Bad

```
select(data,age,gender,sa2,sa3)
select(data ,age ,gender ,sa2 ,sa3)
```

4.7.3 Parentheses

Do not use spaces around parentheses in most cases:

Good

```
mean(x, na.rm = TRUE)
```

Bad

```
mean (x, na.rm = TRUE)
mean( x, na.rm = TRUE )
```

For spacing rules around `if`, `for`, `while`, and `function`, see the Tidyverse guide.

4.8 Short lines, line indentation and the pipe %>%

Keeping your lines of code short and indenting them in a consistent way can help make reading code much easier. If you are supplying multiple arguments to a function, it's generally a good idea to put each argument on a new line - hit enter/return after the comma, like in the `rename` and `filter` examples below. Indentation makes it clear where a code block starts and finishes.

Using pipes (`%>%`) instead of nesting functions also makes things clearer.[1] The pipe should always have a space before it, and should generally be followed by a new line, as in this example:

Good: short lines and indentation

```
young_qual_income <- data %>%
  rename(gender = s801LHSAA,
         uni_attainment = high.ed) %>%
  filter(income > 0,
         age >= 25 & age <= 34) %>%
  group_by(gender,
           uni_attainment) %>%
  summarise(mean_income = mean(income,
                               na.rm = TRUE))
```

Without indentation, the code is harder to read. It's not clear where the chunk starts and finishes, and which bits of code are arguments to which functions.

Bad: short lines, no indentation

```
young_qual_income <- data %>%
rename(gender = s801LHSAA,
uni_attainment = high.ed) %>%
filter(income > 0,
age >= 25 & age <= 34) %>%
group_by(gender, uni_attainment) %>%
summarise(mean_income = mean(income, na.rm = TRUE))
```

Long lines are also bad and hard to read. **Bad: long lines**

[1] The pipe is from the `magrittr` package and is used to chain functions together, so that the output from one function becomes the input to the next function. The pipe is loaded as part of the `tidyverse`.

```
young_qual_income <- data %>% rename(gender = s801LHSAA, uni_attainment = high.ed) %>% filter(inc
```

When you want to take the output of a function and pass it as the input to another function, use the pipe (%>%). Don't write ugly, inscrutable code like this, where multiple functions are wrapped around other functions.

War-crime bad: long lines without pipes

```
young_qual_income<-summarise(group_by(filter(rename(data,gender=s801LHSAA,uni_attainment=high.ed)
```

Writing clear code chunks, where functions are strung together with a pipe (%>%), makes your code much more expressive and able to be read and understood. This is another reason to favour R over something like Excel, which pushes people to piece together functions into Frankenstein's monsters like this:

```
=IF($G16 = "All day", INDEX(metrics!$D$8:$H$66, MATCH(INDEX(correspondence!$B$2:$B$23, MATCH('cor
```

I just threw up in my mouth a little bit.

The pipe function %>% can make code more easy to write and read. The pipe can create the temptation to string together lots and lots of functions into one block of code. This can make things harder to read and understand.

Resist the urge to use the pipe to make code blocks too long. A block of code should generally do one thing, or a small number of things.

4.9 Omit needless code

Don't retain code that ultimately didn't lead anywhere. If you produced a graph that ended up not being used, don't keep the code in your script - if you want to save it, move it to a subfolder named 'archive' or similar. Your code should include the steps needed to go from your raw data to your output - and not extraneous steps. If you ask someone to QC your work, they shouldn't have to wade through 1000 lines of code just to find the 200 lines that are actually required to produce your output.

When you're doing data analysis, you'll often give R interactive commands to help you understand what your data looks like. For example, you might view a dataframe with `View(mydf)` or `str(mydf)`. This is fine, and often necessary, when you're doing your analysis. **Don't keep these commands in your script**. These type of commands should usually be entered straight into the R console, not in a script. If they're in your script, delete them.

Chapter 5

What are packages?

R comes with a lot of functions - commands - built in to do a broad range of tasks. You could, if you really wanted, import a dataset, clean it up, estimate a model, and make a plot just using the functions that come with R - known as 'base R'[1]. But using packages will make your life easier.

Like R itself, packages are free and open source. You can install them from within RStudio.

5.1 How to install packages

You'll typically install packages using the console in RStudio. That's the part of the window that, by default, sits in the bottom-left corner of the screen.

In our work at Grattan, we use packages from two different source: the Comprehensive R Archive Network (CRAN) and Github. The main difference you need to know about is that we use different commands to install packages from these two sources.

To install a package from CRAN, we use the command `install.packages()`.

For example, this code will install the `ggplot2` package from CRAN:

```
install.packages("ggplot2")
```

The easiest way to install a package from Github is to use the function `install_github()`. Unfortunately, this function doesn't come with base R. The `install_github()` function is part of the `remotes` package. To use it, we first need to install `remotes` from CRAN:

[1] Technically some of the 'built-in' functions are part of packages, like the `tools`, `utils` and `stats` packages that come with R. We'll refer to all these as base R.

```
install.packages("remotes")
```

Now we can install packages from Github using the `install_github()` function from the `remotes` package. For example, here's how we would install the Grattan ggplot2 theme, which we'll discuss later in this website:

```
remotes::install_github("mattcowgill/grattantheme", dependencies = TRUE, upgrade = "alw
```

5.2 Get set up: install packages for Grattan

Just starting out or setting up a new machine? Run this block of code to get yourself all set up:

```
cran_packages <- c("devtools", "tidyverse", "readabs", "janitor", "grattan",
                   "rio")

install.packages(cran_packages)

github_packages <- c("grattan/grattantheme", "grattan/grattandata",
                     "wfmackey/absmapsdata")

remotes::install_github(github_packages,
                        dependencies = TRUE,
                        upgrade = "always")
```

5.3 Using packages

Before using a function that comes from a package, you need to tell R where to look for the function. There are two main ways to do that.

We can either load (aka 'attach') the package by using the `library()` function. We typically do this at the top of a script.

```
library(remotes)

# Now that the `remotes` package is loaded, we can use its `install_github()` function

install_github("mattcowgill/grattantheme")
```

Or, we can use two colons - `::` - to tell R to use an individual function from a package without loading it:

```
remotes::install_github("mattcowgill/grattantheme")
```

It usually makes sense to load a package with `library()`, unless you only need to use one of its function once or twice. There's no harm to using the `::` operator even if you have already loaded a package with `library()`. This can remove ambiguity both for R and for humans reading your code, particularly if you're using an obscure function - it makes it clearer where the function comes from.

5.4 Upgrading packages

It's generally a good idea to keep your packages up-to-date. The easiest way to do this is to run this code:

```
devtools::update_packages()
```

This will upgrade all your packages - including those you've installed from CRAN and Github.

When you run the above command, it will prompt you to ask which packages you want to update - press 1 for 'All'.

If it asks you 'Do you want to install from sources the package which needs compilation?' type 'no' and press enter.[^Nothing against installing from source, but this part of the guide is aimed at people who are not familiar with R and may not have the tools installed to build from source.]

5.5 Downgrading packages

Sometimes, when packages change, their functions evolve. The arguments to a function might change, or a function might be phased out ('deprecated') in favour of another. You can usually just adapt your workflow to the package's new version without much fuss. If you find this isn't the case, and you want to downgrade to an earlier version of a package, it's straightforward. Just use the `install_version()` function, like this:

```
devtools::install_version("devtools", "1.13.3")
```

It's rare that you'd need to downgrade. Better to stay up to date, and adapt your code when necessary to changes in packages.

Chapter 6

Packages commonly used at Grattan

Some packages we use at Grattan - like the `tidyverse` collection of packages - are very popular among R users. Some - like the `grattantheme` package - are specific to Grattan Institute. Others - like the `readabs` package - are made by Grattan people, useful at Grattan, but also used outside of the Institute.

6.1 The tidyverse!

The `tidyverse` is central to our work at Grattan. The `tidyverse` is a collection of related R packages for importing, wrangling, exploring and visualising data in R. The packages are designed to work well together. The main packages in the `tidyverse` include:

- *ggplot2* for making beautiful, customisable graphs
- *dplyr* for manipulating data frames
- *tidyr* for tidying your data
- *readr* for importing data from a broad range of formats
- *purrr* for functional programming
- *stringr* for manipulating strings of text

All these packages (and more!) will automatically be loaded for you when you run the command[1]:

[1] There's no need to install or load the individual `tidyverse` packages - like `dplyr` - separately. Just install them all together, and load them with the single `library(tidyverse)` command. That way, you don't need to remember which functions come from `tidyr` and which from `dplyr` - they're all just `tidyverse` functions.

```
library(tidyverse)
```

```
## -- Attaching packages ---------------------------- tidyverse 1.3.0 --
```

```
## v ggplot2 3.3.0.9000     v purrr   0.3.3
## v tibble  2.1.3          v dplyr   0.8.4
## v tidyr   1.0.2          v stringr 1.4.0
## v readr   1.3.1          v forcats 0.4.0
```

```
## -- Conflicts ------------------------------- tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

A range of other packages are installed on your machine as part of the `tidyverse`. These include:

- *readxl* for importing Excel spreadsheets into R
- *haven* for importing Stata, SAS and SPSS data
- *lubridate* for working with dates
- *rvest* for scraping websites

Although these packages are installed as part of the `tidyverse`, they aren't loaded automatically when you run `library(tidyverse)`. You'll need to load them individually, like:

```
library(lubridate)
library(readxl)
```

6.1.1 Why do we use the tidyverse?

The `tidyverse` makes life easier!

The core `tidyverse` packages, like `ggplot2`, `dplyr`, and `tidyr`, are extremely popular. The `tidyverse` is probably the most popular 'dialect' of R. This means that any problem you encounter with the `tidyverse` will have been encountered many times before by other R users, so a solution will only be a Google search away.

The `tidyverse` packages are all designed to work well together, with a consistent underlying philosophy and design. This means that coding habits you learn with one `tidyverse` package, like `dplyr`, are also applicable to other packages, like `tidyr`.

They're designed to work with data frames[2], a rectangular data object that will be familiar to spreadsheet users that is very intuitive and convenient for the sort of work we do at Grattan. In particular, the `tidyverse` is built around

[2]The tidyverse works with 'tibbles', which are a tidyverse-specific variant of the data frame. Don't worry about the difference between tibbles and data frames.

the concept of *tidy data*, which has a specific meaning in this context that we'll come to later. The fact that `tidyverse` packages are all built around one type of data object makes them easier to work with.

The creator of the `tidyverse`, Hadley Wickham, places great value on code that is expressive and comprehensible to humans. This means that code written in the `tidyverse` idiom is often able to be understood even if you haven't encountered the functions before. For example, look at this chunk of code:

```
my_data %>%
  filter(age >= 30) %>%
  mutate(relative_income = income / mean(income))
```

Without knowing what `my_data` looks like, and even if you haven't encountered these functions before, this should be reasonably intuitive. We're taking some data, and then[3] only keeping observations that relate to people aged 30 and older, then calculating a new variable, `relative_income`. The name of a `tidyverse` function - like `filter`, `group_by`, `summarise`, and so on - generally gives you a pretty good idea what the function is going to do with your data, which isn't always the case with other approaches.

Here's one way to do the same thing in base R:

```
transform(my_data[my_data$age >= 30, ],
          relative_income = income / mean(income))
```

The base R code gets the job done, but it's clunkier, less expressive, and harder to read. A core principle of coding at Grattan is that you should strive to make your work **human readable**.

Code written with `tidyverse` functions is often faster than its base R equivalents. But most of our work at Grattan is with small-to-medium sized datasets (with fewer than a million rows or so), so speed isn't usually a major concern anyway.[4]

The most valuable resource we deal with at Grattan is our time. Computers are cheap, people are not. If your code executes quickly, but it takes your colleague many hours to decipher it, the cost of the extra QC time more than outweighs the saving through faster computation. The `tidyverse` packages strike a balance between expressive, comprehensible code and computational efficiency that suits the nature of our work at Grattan. This balance is the right one for most of our work, most of the time.

Most R scripts at Grattan should start with `library(tidyverse)`. Most of your work will be in data frames, and most of the time the `tidyverse` contains the core tools you'll need to do that work.

[3] you can read the pipe, `%>%`, as 'and then'

[4] When working with very large datasets, it might be worth gaining speed using other packages, such as `data.table`. Fortunately, using the `dtplyr` package you can get most of the speed benefits of `data.table` and stick to familiar `dplyr` syntax.

6.2 Grattan-specific packages

A range of Grattan people have written packages that come in handy at Grattan.

- *grattan* The `grattan` package, created by Hugh Parsonage, contains two broad sets of functions. One set of functions (sometimes known by the nickname "Grattax") is used for modelling the personal income tax system. Another set of functions ("Grattools") are useful for a lot of our work, like converting dates to financial years (`grattan::date2fy()`) or a version of `dplyr::ntile()` that uses weights (`grattan::weighted_ntile()`).
- *grattantheme* The `grattantheme` package, by Matt Cowgill and Will Mackey, helps to make your ggplot2 charts Grattan-y. We cover the package extensively in the data visualisation chapter.
- *grattandata* The `grattandata` package, by Matt Cowgill and Jonathan Nolan, is used to load microdata from the Grattan microdata repository. We cover this in the reading data chapter.

Install these Grattan-specific packages using this block of code:

6.3 Other commonly-used packages

There are other packages we commonly use at Grattan, including some developed by Grattan staff. These include:

- *absmapsdata* This package, by Will Mackey, is very handy for working with spatial data. You'll want it if you're going to be making maps.
- *readabs* The `readabs` package, by Matt Cowgill, provides an easy way to download, tidy, and import ABS time series data in R.

Chapter 7

Getting help with R

The most important skill you need to learn to use R well is how to get help. This is a great list of steps to try:

This blog post explains those steps at a bit more length.

The most important step is often breaking your problem down into a small, reproducible example - a `reprex` in R jargon. Often, the process of making a reprex can make your problem appear more clearly to you, so you'll solve it yourself before you even have to ask someone else!

7.1 Getting help with R problems at Grattan

The channel `#r_at_grattan` on Grattan Slack is a great place to get answers to R questions

This guide Package vignettes Matt, Will, others

7.2 Getting help with R problems in general

A guide to Googling well

7.3 Resources for learning R

R4DS, et al.

Part III

Load, manipulate and visualise data

Chapter 8

Reading data

8.1 Importing data

8.1.1 Reading CSV files

8.1.1.1 `read_csv()`

The `read_csv()` function from the `tidyverse` is quicker and smarter than `read.csv` in base R.

Pitfalls: 1. read_csv is quicker because it surveys a sample of the data

We can also compress `.csv` files into `.zip` files and read them *directly* using `read_csv()`:

```
read_csv("data/my_data.zip")
```

This is useful for two reasons:

1. The data takes up less room on your computer; and
2. The original data, which shouldn't ever be directly edited, is protected and cannot be directly edited.

8.1.1.2 `data.table::fread()`

The `fread` function from `data.table` is quicker than both `read.csv` and `read_csv`.

8.1.2 `grattandata::read_microdata()`

8.1.3 `readxl::read_excel()`

8.1.4 `rio`

8.1.5 `readabs`

8.2 Reading common files:

- TableBuilder CSVSTRINGs
- HES household file
- SIH
- LSAY and derivatives

See data directory for a list of microdata available to Grattan.

8.3 Appropriately renaming variables

As shown in the style guide

Add `rename_abs` function to a common Grattan package?

8.4 Getting to tidy data

`pivot_long()` and `pivot_wide()` *Make sure these are stable btw*

Chapter 9

Different data types

9.1 Tidy data

Other data structures

9.2 Dates with `lubridate::`

The `lubridate::` package

9.3 Strings with `stringr::`

- Replacing values
- Matching values
- Separating columns

9.4 Factors with `forcats::`

- Dangers with factors

Chapter 10

Data transformation with `dplyr`

This section focusses on transforming rectangular datasets.

The `dplyr` verbs and concepts covered in this chapter are also covered in this video by Garrett Grolemund (a co-author of *R for Data Science* with Hadley Wickham).

10.1 Set up

```
library(tidyverse)
```

The `sa3_income` dataset will be used for all key examples in this chapter.[1] It is a long dataset from the ABS that contains the average income and number of workers by Statistical Area 3, occupation and sex between 2011 and 2016.

```
sa3_income <- read_csv("data/sa3_income.csv")
```

```
## Parsed with column specification:
## cols(
##   sa3 = col_double(),
##   sa3_name = col_character(),
##   sa3_sqkm = col_double(),
##   sa3_income_percentile = col_double(),
##   sa4_name = col_character(),
##   gcc_name = col_character(),
##   state = col_character(),
```

[1] From ABS Employee income by occupation and sex, 2010-11 to 2016-16

```
##   occupation = col_character(),
##   occ_short = col_character(),
##   prof = col_character(),
##   gender = col_character(),
##   year = col_double(),
##   median_income = col_double(),
##   average_income = col_double(),
##   total_income = col_double(),
##   workers = col_double()
## )
```

```
head(sa3_income)
```

```
## # A tibble: 6 x 6
##    year sa3_name  state gender income workers
##   <dbl> <chr>     <chr> <chr>   <dbl>   <dbl>
## 1  2011 Belconnen ACT   Men    54105.   67774
## 2  2012 Belconnen ACT   Men    56724.   69435
## 3  2013 Belconnen ACT   Men    58918.   69697
## 4  2014 Belconnen ACT   Men    60525.   68613
## 5  2015 Belconnen ACT   Men    60964.   63428
## 6  2016 Belconnen ACT   Men    63389.   69828
```

10.2 The pipe: %>%

You will almost always want to perform more than one of the operations described below on your dataset. One way to perform multiple operations, one after the other, is to 'nest' them inside. This nesting will be *painfully* familiar to Excel users.

Consider an example of baking and eating a cake.[2] You take the ingredients, combine them, then mix, then bake, and then eat them. In a nested formula, this process looks like:

```
eat(bake(mix(combine(ingredients))))
```

In a nested formula, you need to start in the *middle* and work your way out. This means anyone reading your code – including you in the future! – needs to start in the middle and work their way out. But because we're used to left-right reading, we're not particularly good at naturally interpreting nested functions like this one.

This is where the 'pipe' can help. The pipe operator `%>%` (keyboard shortcut: `cmd + shift + m`) takes an argument on the left and 'pipes' it into the function

[2] XXX cannot remember the source for this example; probably Hadley? Jenny Bryan? Maybe somenone else?

on the right. Each time you see `%>%`, you can read it as 'and then'.

So the you could express the baking example as:

```
ingredients %>% # and then
  combine() %>% # and then
  mix() %>% # and then
  bake() %>% # and then
  eat() # yum!
```

Which reads as:

> take the `ingredients`, then `combine`, then `mix`, then `bake`, then `eat` them.

This does the same thing as `eat(bake(mix(combine(ingredients))))`. But it's much nicer and more natural to read, and to *write*.

Another example: the function `paste` takes arguments and combines them together into a single string. So you could use the pipe to:

```
"hello" %>% paste("dear", "reader")
```

```
## [1] "hello dear reader"
```

which is the same as

```
paste("hello", "dear", "reader")
```

```
## [1] "hello dear reader"
```

Or you could define a vector of numbers and pass[3] them to the `sum()` function:

```
my_numbers <- c(1, 2, 3, 5, 8, 13)

my_numbers %>% sum()
```

```
## [1] 32
```

Or you could skip the intermediate step altogether:

```
c(1, 2, 3, 5, 8, 13) %>%
  sum()
```

```
## [1] 32
```

This is the same as:

```
sum(c(1, 2, 3, 5, 8, 13))
```

```
## [1] 32
```

[3] 'pass' can also be used to mean 'pipe'.

The benefits of piping become more clear when you want to perform a few sequential operations on a dataset. For example, you might want to `filter` the observations in the `sa3_income` data to only `NSW`, before you `group_by gender` and `summarise` the `income` of these grops (these functions are explained in detail below). All of these functions take 'data' as the first argument, and are designed to be used with pipes.

Like the income differential it shows, writing this process as a nested function is outrageous and hard to read:

```
summarise((group_by(filter(sa3_income, state == "NSW"), gender)), av_mean_income = mean
```

```
## # A tibble: 2 x 2
##   gender av_mean_income
##   <chr>           <dbl>
## 1 Men            58202.
## 2 Women          41662.
```

The original common way to avoid this unseemly nesting in `R` was to assign each 'step' its own object, which is definitely clearer:

```
data1 <- filter(sa3_income, state == "NSW")
data2 <- group_by(data1, gender)
data3 <- summarise(data2, av_mean_income = mean(income))
data3
```

```
## # A tibble: 2 x 2
##   gender av_mean_income
##   <chr>           <dbl>
## 1 Men            58202.
## 2 Women          41662.
```

And using pipes make the steps clearer still:

1. take the `sa3_income` data, and then %>%
2. `filter` it to only NSW, and then %>%
3. `group` it by gender, and then %>%
4. `summarise` it

```
sa3_income %>%  # and then
  filter(state == "NSW") %>% # and then
  group_by(gender) %>% # and then
  summarise(av_mean_income = mean(income))
```

```
## # A tibble: 2 x 2
##   gender av_mean_income
##   <chr>           <dbl>
## 1 Men            58202.
## 2 Women          41662.
```

10.3 Select variables with `select()`

The `select` function takes a dataset and **keeps** or **drops** variables (columns) that are specified.

For example, look at the variables that are in the `sa3_income` dataset (using the `names()` function):

```
names(sa3_income)
```

```
## [1] "year"     "sa3_name" "state"    "gender"   "income"   "workers"
```

If you wanted to keep just the `state` and `income` variables, you could take the `sa3_income` dataset and select just those variables:

```
sa3_income %>%
  select(state, income)
```

```
## # A tibble: 4,019 x 2
##    state income
##    <chr>  <dbl>
##  1 ACT   54105.
##  2 ACT   56724.
##  3 ACT   58918.
##  4 ACT   60525.
##  5 ACT   60964.
##  6 ACT   63389.
##  7 ACT   53139.
##  8 ACT   54515.
##  9 ACT   58132.
## 10 ACT   56247.
## # ... with 4,009 more rows
```

Or you could use `-` (minus) to remove the `state` and `sa3_name` variables:[4]

```
sa3_income %>%
  select(-state, -sa3_name)
```

```
## # A tibble: 4,019 x 4
##     year gender income workers
##    <dbl> <chr>   <dbl>   <dbl>
##  1  2011 Men    54105.   67774
##  2  2012 Men    56724.   69435
##  3  2013 Men    58918.   69697
##  4  2014 Men    60525.   68613
##  5  2015 Men    60964.   63428
##  6  2016 Men    63389.   69828
##  7  2011 Men    53139.     666
```

[4] This is the same as **keeping everything except** the `state` and `sa3_name` variables.

```
##  8  2012 Men    54515.     647
##  9  2013 Men    58132.     641
## 10  2014 Men    56247.     561
## # ... with 4,009 more rows
```

10.3.1 Selecting groups of variables

Sometimes it can be useful to keep or drop variables with names that have a certain characteristic; they begin with some text string, or end with one, or contain one, or have some other pattern altogether.

You can use patterns and 'select helpers'[5] from the Tidyverse to help deal with these sets of variables.

For example, if you want to keep just the SA3 and state variables – ie the variables that start with `"s"` – you could:

```
sa3_income %>%
  select(starts_with("s"))
```

```
## # A tibble: 4,019 x 2
##    sa3_name      state
##    <chr>         <chr>
##  1 Belconnen     ACT
##  2 Belconnen     ACT
##  3 Belconnen     ACT
##  4 Belconnen     ACT
##  5 Belconnen     ACT
##  6 Belconnen     ACT
##  7 Canberra East ACT
##  8 Canberra East ACT
##  9 Canberra East ACT
## 10 Canberra East ACT
## # ... with 4,009 more rows
```

Or, instead, if you wanted to keep just the variables that contain `"er"`, you could:

```
sa3_income %>%
  select(contains("er"))
```

```
## # A tibble: 4,019 x 2
##    gender workers
##    <chr>    <dbl>
##  1 Men      67774
##  2 Men      69435
```

[5] Explained in useful detail by the Tidyverse people at https://tidyselect.r-lib.org/reference/select_helpers.html

```
##  3 Men      69697
##  4 Men      68613
##  5 Men      63428
##  6 Men      69828
##  7 Men        666
##  8 Men        647
##  9 Men        641
## 10 Men        561
## # ... with 4,009 more rows
```

And if you wanted to keep **both** the `"s"` variables and the `"er"` variables, you could:

```
sa3_income %>%
  select(starts_with("s"), contains("er"), )
```

```
## # A tibble: 4,019 x 4
##    sa3_name      state gender workers
##    <chr>         <chr> <chr>    <dbl>
##  1 Belconnen     ACT   Men      67774
##  2 Belconnen     ACT   Men      69435
##  3 Belconnen     ACT   Men      69697
##  4 Belconnen     ACT   Men      68613
##  5 Belconnen     ACT   Men      63428
##  6 Belconnen     ACT   Men      69828
##  7 Canberra East ACT   Men        666
##  8 Canberra East ACT   Men        647
##  9 Canberra East ACT   Men        641
## 10 Canberra East ACT   Men        561
## # ... with 4,009 more rows
```

The full list of these handy select functions are provided with the `?tidyselect::select_helpers` documentation, listed below:

- `starts_with()`: Starts with a prefix.
- `ends_with()`: Ends with a suffix.
- `contains()`: Contains a literal string.
- `matches()`: Matches a regular expression.
- `num_range()`: Matches a numerical range like x01, x02, x03.
- `one_of()`: Matches variable names in a character vector.
- `everything()`: Matches all variables.
- `last_col()`: Select last variable, possibly with an offset.

10.4 Filter with `filter()`

The `filter` function takes a dataset and **keeps** observations (rows) that meet the **conditions**.

`filter` has one required first argument – the data – and then as many 'conditions' as you want to provide.

10.4.1 Conditions; logical operations; `TRUE` or `FALSE`

The **conditions** are logical operations, meaning they are a statement that return either `TRUE` or `FALSE` in the computer's mind.[6]

We know, for instance, that `12` is equal to `12` and that `1 + 2` does not equal `12`. Which means if we type `12 == 12` or `1 + 2 == 12` into the console it should give `FALSE`:

```
12 == 12
```

```
## [1] TRUE
```

```
1 + 2 == 12
```

```
## [1] FALSE
```

Or, we can see if `1 + 2` is equal `5` or `9` or `3` by providing a vector of those numbers:

```
1 + 2 == c(5, 9, 3)
```

```
## [1] FALSE FALSE  TRUE
```

This works for character strings, too:

```
"apple" == c("orange", "apple", 7)
```

```
## [1] FALSE  TRUE FALSE
```

A lot of what we do in 'data science' is based on these `TRUE` and `FALSE` conditions.

10.4.2 Filtering data with `filter`

Turning back to the `sa3_income` data, if you just wanted to see observations people in `NT`:

```
sa3_income %>%
  filter(state == "NT")
```

[6] Computers' mind, more likely.

```
## # A tibble: 123 x 6
##     year sa3_name      state gender income workers
##    <dbl> <chr>         <chr> <chr>   <dbl>   <dbl>
##  1  2011 Alice Springs NT    Men    52602.   23663
##  2  2012 Alice Springs NT    Men    55050.   24065
##  3  2013 Alice Springs NT    Men    57251.   24218
##  4  2014 Alice Springs NT    Men    58403.   24566
##  5  2015 Alice Springs NT    Men    60084.   24562
##  6  2016 Alice Springs NT    Men    64330.   22048
##  7  2011 Barkly        NT    Men    50517.    2272
##  8  2012 Barkly        NT    Men    52474.    2321
##  9  2013 Barkly        NT    Men    55006.    2364
## 10  2014 Barkly        NT    Men    56543.    2234
## # ... with 113 more rows
```

Or you might just want to look at high-income (`income > 70,000`) areas from Victoria in 2015:

```
sa3_income %>%
  filter(state == "Vic",
         income > 70000,
         year == 2015)
```

```
## # A tibble: 3 x 6
##    year sa3_name           state gender income workers
##   <dbl> <chr>              <chr> <chr>   <dbl>   <dbl>
## 1  2015 Bayside            Vic   Men    77175.   62460
## 2  2015 Stonnington - East Vic   Men    70652.   27922
## 3  2015 Stonnington - West Vic   Men    70234.   47597
```

Each of the commas in the `filter` function represent an 'and' `&`. So you can read the steps above as:

> take the `sa3_income` data and filter to keep only the observations that are from Victoria, and that have a average income above 70k, and are from the year 2015.

Sometimes you might want to relax a little, keeping observations from one category **or** another. You can do this with the **or** symbol: `|`[7]

```
sa3_income %>%
  filter(state == "Vic" | state == "Tas",
         income > 100000,
         year == 2015 | year == 2016)
```

```
## # A tibble: 0 x 6
## # ... with 6 variables: year <dbl>, sa3_name <chr>, state <chr>, gender <chr>,
## #   income <dbl>, workers <dbl>
```

[7] On the keyboard: `shift` + `backslash`

Which reads:

> take the sa3_income data and filter to keep only the observations that are from Victoria OR NSW, and that have a average income above 100k, and are from the year 2015 OR 2016.

10.4.3 Grouped filtering with group_by()

The group_by function groups a dataset by given variables. This effectively generates one dataset per group within your main dataset. Any function you then apply – like filter() – will be applied to *each* of the grouped datasets.

For example, you could filter the sa3_income dataset to keep just the observation with the highest average income:

```
sa3_income %>%
  filter(income == max(income))
```

```
## # A tibble: 1 x 6
##    year sa3_name     state gender  income workers
##   <dbl> <chr>        <chr> <chr>    <dbl>   <dbl>
## 1  2015 West Pilbara WA    Men    107844.   22928
```

To keep the observations that have the highest average incomes *in each state*, you can group_by state, then filter:[8]

```
sa3_income %>%
  group_by(state) %>%
  filter(income == max(income))
```

```
## # A tibble: 8 x 6
## # Groups:   state [8]
##    year sa3_name                 state gender  income workers
##   <dbl> <chr>                    <chr> <chr>    <dbl>   <dbl>
## 1  2013 Molonglo                 ACT   Men     92947.     227
## 2  2016 North Sydney - Mosman    NSW   Men     90668.   74702
## 3  2016 Christmas Island         NT    Men     84474.     621
## 4  2015 Gladstone                Qld   Men     97282.   48026
## 5  2015 Outback - North and East SA    Men     71791.   15849
## 6  2016 West Coast               Tas   Men     58116.   11117
## 7  2016 Bayside                  Vic   Men     78624.   64541
## 8  2015 West Pilbara             WA    Men    107844.   22928
```

From the description of the tibble above, you can learn that your data has 8 unique groups of state:

```
## # Groups:       state [8]
```

[8] Wow they are all men!

Or you could keep the observations with the *lowest* average incomes in *each state and year*:[9]

```
sa3_income %>%
  group_by(state, year) %>%
  filter(income == min(income))
```

```
## # A tibble: 48 x 6
## # Groups:   state, year [48]
##     year sa3_name                state gender income workers
##    <dbl> <chr>                   <chr> <chr>   <dbl>   <dbl>
##  1  2014 Cocos (Keeling) Islands NT    Men    32652.      45
##  2  2011 Belconnen               ACT   Women  43235    22708
##  3  2014 Belconnen               ACT   Women  48399.   22750
##  4  2015 Belconnen               ACT   Women  48814.   20577
##  5  2016 Belconnen               ACT   Women  50756.   22982
##  6  2012 Gungahlin               ACT   Women  45241    13647
##  7  2013 North Canberra          ACT   Women  45844.   11965
##  8  2012 Great Lakes             NSW   Women  32590     4730
##  9  2015 Lord Howe Island        NSW   Women  34173.      75
## 10  2011 Lower Murray            NSW   Women  30800.    2122
## # ... with 38 more rows
```

The dataset remains grouped after your function(s). To explicitly 'ungroup' your data, add the `ungroup` function to your chain (the 'Groups' note has disappeared in the below):

```
sa3_income %>%
  group_by(state, year) %>%
  filter(income == min(income)) %>%
  ungroup()
```

```
## # A tibble: 48 x 6
##     year sa3_name                state gender income workers
##    <dbl> <chr>                   <chr> <chr>   <dbl>   <dbl>
##  1  2014 Cocos (Keeling) Islands NT    Men    32652.      45
##  2  2011 Belconnen               ACT   Women  43235    22708
##  3  2014 Belconnen               ACT   Women  48399.   22750
##  4  2015 Belconnen               ACT   Women  48814.   20577
##  5  2016 Belconnen               ACT   Women  50756.   22982
##  6  2012 Gungahlin               ACT   Women  45241    13647
##  7  2013 North Canberra          ACT   Women  45844.   11965
##  8  2012 Great Lakes             NSW   Women  32590     4730
##  9  2015 Lord Howe Island        NSW   Women  34173.      75
## 10  2011 Lower Murray            NSW   Women  30800.    2122
## # ... with 38 more rows
```

[9] Wow they are all women!

10.5 Edit and add new variables with `mutate()`

To add new variables to your dataset, use the `mutate` function. Like all `dplyr` verbs, the first argument to `mutate` is your data. Then define variables using a `new_variable_name = x` format, where `x` can be a single number or character string, or simple operation or function using current variables.

To add a new variable to the `sa3_income` dataset that shows the log the number of workers:

```
sa3_income %>%
  mutate(log_workers = log(workers))
```

```
## # A tibble: 4,019 x 7
##     year sa3_name      state gender income workers log_workers
##    <dbl> <chr>         <chr> <chr>   <dbl>   <dbl>       <dbl>
##  1  2011 Belconnen     ACT   Men    54105.   67774        11.1
##  2  2012 Belconnen     ACT   Men    56724.   69435        11.1
##  3  2013 Belconnen     ACT   Men    58918.   69697        11.2
##  4  2014 Belconnen     ACT   Men    60525.   68613        11.1
##  5  2015 Belconnen     ACT   Men    60964.   63428        11.1
##  6  2016 Belconnen     ACT   Men    63389.   69828        11.2
##  7  2011 Canberra East ACT   Men    53139.     666         6.50
##  8  2012 Canberra East ACT   Men    54515.     647         6.47
##  9  2013 Canberra East ACT   Men    58132.     641         6.46
## 10  2014 Canberra East ACT   Men    56247.     561         6.33
## # ... with 4,009 more rows
```

To edit a variable, redefine it in `mutate`. For example, if you wanted to take the last two digits of year:

```
sa3_income %>%
  mutate(year = as.integer(year - 2000))
```

```
## # A tibble: 4,019 x 6
##     year sa3_name      state gender income workers
##    <int> <chr>         <chr> <chr>   <dbl>   <dbl>
##  1    11 Belconnen     ACT   Men    54105.   67774
##  2    12 Belconnen     ACT   Men    56724.   69435
##  3    13 Belconnen     ACT   Men    58918.   69697
##  4    14 Belconnen     ACT   Men    60525.   68613
##  5    15 Belconnen     ACT   Men    60964.   63428
##  6    16 Belconnen     ACT   Men    63389.   69828
##  7    11 Canberra East ACT   Men    53139.     666
##  8    12 Canberra East ACT   Men    54515.     647
##  9    13 Canberra East ACT   Men    58132.     641
## 10    14 Canberra East ACT   Men    56247.     561
## # ... with 4,009 more rows
```

10.5.1 Using if_else() or case_when()

Sometimes you want to create a new variable based on some sort of condition. Like, if the number of workers in an sa3 is more than 2,000, set the new many_workers variable to TRUE, and set it to FALSE otherwise.

This kind of operation can be thought of as if_else: if (some condition), do this, otherwise do that.

That's what the if_else() function does. It takes three arguments: a condition, a value if that condition is true, and a value if that condition is false.

You can use the if_else() function when you are creating new variables in a mutate command:

```
sa3_income %>%
  mutate(many_workers = if_else(workers > 2000, "Many workers", "Not many workers"))
```

```
## # A tibble: 4,019 x 7
##     year sa3_name      state gender income workers many_workers
##    <dbl> <chr>         <chr> <chr>   <dbl>   <dbl> <chr>
##  1  2011 Belconnen     ACT   Men    54105.   67774 Many workers
##  2  2012 Belconnen     ACT   Men    56724.   69435 Many workers
##  3  2013 Belconnen     ACT   Men    58918.   69697 Many workers
##  4  2014 Belconnen     ACT   Men    60525.   68613 Many workers
##  5  2015 Belconnen     ACT   Men    60964.   63428 Many workers
##  6  2016 Belconnen     ACT   Men    63389.   69828 Many workers
##  7  2011 Canberra East ACT   Men    53139.     666 Not many workers
##  8  2012 Canberra East ACT   Men    54515.     647 Not many workers
##  9  2013 Canberra East ACT   Men    58132.     641 Not many workers
## 10  2014 Canberra East ACT   Men    56247.     561 Not many workers
## # ... with 4,009 more rows
```

Which reads:

> Take the sa3_income dataset, and then add a variable that says 'Many workers' if there are more than 2,000 workers, and 'Not many workers' if there are fewer-or-equal than 2,000 workers.

With the if_else function, you take one conditional statement and return something based on that. But **often** you don't want to be so binary; you want to do this if this is true, that if that is true, and the other if the other is true, etc.

This could be done by nesting if_else statements:

```
sa3_income %>%
  mutate(worker_group = if_else(workers > 2000, "More than 2000 workers",
                                if_else(workers > 1000, "1000-2000 workers",
```

```
                                        if_else(workers > 500, "500-1000 workers",
                                                "500 workers or less"))))
```

```
## # A tibble: 4,019 x 7
##     year sa3_name      state gender income workers worker_group
##    <dbl> <chr>         <chr> <chr>   <dbl>   <dbl> <chr>
##  1  2011 Belconnen     ACT   Men    54105.   67774 More than 2000 workers
##  2  2012 Belconnen     ACT   Men    56724.   69435 More than 2000 workers
##  3  2013 Belconnen     ACT   Men    58918.   69697 More than 2000 workers
##  4  2014 Belconnen     ACT   Men    60525.   68613 More than 2000 workers
##  5  2015 Belconnen     ACT   Men    60964.   63428 More than 2000 workers
##  6  2016 Belconnen     ACT   Men    63389.   69828 More than 2000 workers
##  7  2011 Canberra East ACT   Men    53139.     666 500-1000 workers
##  8  2012 Canberra East ACT   Men    54515.     647 500-1000 workers
##  9  2013 Canberra East ACT   Men    58132.     641 500-1000 workers
## 10  2014 Canberra East ACT   Men    56247.     561 500-1000 workers
## # ... with 4,009 more rows
```

But that syntax can be a bit difficult to read. You can do this in a clearer way using `case_when`:

```
sa3_income %>%
  mutate(worker_group = case_when(
    workers > 20000 ~ "More than 20,000 workers",
    workers > 10000 ~ "More than 10,000 workers",
    workers >  5000 ~ "More than 5,000 workers",
    workers <= 5000 ~ "5,000 or fewer workers"
  ))
```

```
## # A tibble: 4,019 x 7
##     year sa3_name      state gender income workers worker_group
##    <dbl> <chr>         <chr> <chr>   <dbl>   <dbl> <chr>
##  1  2011 Belconnen     ACT   Men    54105.   67774 More than 20,000 workers
##  2  2012 Belconnen     ACT   Men    56724.   69435 More than 20,000 workers
##  3  2013 Belconnen     ACT   Men    58918.   69697 More than 20,000 workers
##  4  2014 Belconnen     ACT   Men    60525.   68613 More than 20,000 workers
##  5  2015 Belconnen     ACT   Men    60964.   63428 More than 20,000 workers
##  6  2016 Belconnen     ACT   Men    63389.   69828 More than 20,000 workers
##  7  2011 Canberra East ACT   Men    53139.     666 5,000 or fewer workers
##  8  2012 Canberra East ACT   Men    54515.     647 5,000 or fewer workers
##  9  2013 Canberra East ACT   Men    58132.     641 5,000 or fewer workers
## 10  2014 Canberra East ACT   Men    56247.     561 5,000 or fewer workers
## # ... with 4,009 more rows
```

The `case_when` function takes the first condition (LHS) and applies some value (RHS) if it is true. It then moves to the next condition, and so on. Once an observation has been classified – eg an observation has more than 20,000 workers

– it is ignored in proceeding conditions.

Ending a `case_when` statement with `TRUE ~ [some value]` is a catch all, and will apply the RHS `[some value]` to any observations that did not meet an explicit condition. For example, you could end the worker classification with:

```
sa3_income %>%
  mutate(worker_group = case_when(
    workers > 20000 ~ "More than 20,000 workers",
    workers > 10000 ~ "More than 10,000 workers",
    workers >  5000 ~ "More than 5,000 workers",
    TRUE ~ "5,000 or fewer workers"
  ))
```

```
## # A tibble: 4,019 x 7
##     year sa3_name      state gender income workers worker_group
##    <dbl> <chr>         <chr> <chr>   <dbl>   <dbl> <chr>
##  1  2011 Belconnen     ACT   Men    54105.   67774 More than 20,000 workers
##  2  2012 Belconnen     ACT   Men    56724.   69435 More than 20,000 workers
##  3  2013 Belconnen     ACT   Men    58918.   69697 More than 20,000 workers
##  4  2014 Belconnen     ACT   Men    60525.   68613 More than 20,000 workers
##  5  2015 Belconnen     ACT   Men    60964.   63428 More than 20,000 workers
##  6  2016 Belconnen     ACT   Men    63389.   69828 More than 20,000 workers
##  7  2011 Canberra East ACT   Men    53139.     666 5,000 or fewer workers
##  8  2012 Canberra East ACT   Men    54515.     647 5,000 or fewer workers
##  9  2013 Canberra East ACT   Men    58132.     641 5,000 or fewer workers
## 10  2014 Canberra East ACT   Men    56247.     561 5,000 or fewer workers
## # ... with 4,009 more rows
```

Meaning, for any observation that did not have workers more than 20,000 or more than 10,000 or more than 5,000, assign the value `"5,000 or fewer workers"`.

Observations that do not meet a condition will be set to `NA`:

```
sa3_income %>%
  mutate(worker_group = case_when(
    workers > 10e6 ~ "More than 10 million workers"
  ))
```

```
## # A tibble: 4,019 x 7
##     year sa3_name      state gender income workers worker_group
##    <dbl> <chr>         <chr> <chr>   <dbl>   <dbl> <chr>
##  1  2011 Belconnen     ACT   Men    54105.   67774 <NA>
##  2  2012 Belconnen     ACT   Men    56724.   69435 <NA>
##  3  2013 Belconnen     ACT   Men    58918.   69697 <NA>
##  4  2014 Belconnen     ACT   Men    60525.   68613 <NA>
##  5  2015 Belconnen     ACT   Men    60964.   63428 <NA>
##  6  2016 Belconnen     ACT   Men    63389.   69828 <NA>
```

```
##  7  2011 Canberra East ACT   Men    53139.     666 <NA>
##  8  2012 Canberra East ACT   Men    54515.     647 <NA>
##  9  2013 Canberra East ACT   Men    58132.     641 <NA>
## 10  2014 Canberra East ACT   Men    56247.     561 <NA>
## # ... with 4,009 more rows
```

Like any `if` or `if_else`, you can provide more than one condition to your conditional statement:

```
sa3_income %>%
  mutate(women_group = case_when(
    gender == "Women" & workers > 20000 ~ "More than 20,000 women",
    gender == "Women" & workers > 10000 ~ "More than 10,000 women",
    gender == "Women" & workers >  5000 ~ "More than 5,000 women",
    gender == "Women"                   ~ "5,000 or fewer women",
    TRUE ~ "Men"
  ))
```

```
## # A tibble: 4,019 x 7
##     year sa3_name      state gender income workers women_group
##    <dbl> <chr>         <chr> <chr>   <dbl>   <dbl> <chr>
##  1  2011 Belconnen     ACT   Men    54105.   67774 Men
##  2  2012 Belconnen     ACT   Men    56724.   69435 Men
##  3  2013 Belconnen     ACT   Men    58918.   69697 Men
##  4  2014 Belconnen     ACT   Men    60525.   68613 Men
##  5  2015 Belconnen     ACT   Men    60964.   63428 Men
##  6  2016 Belconnen     ACT   Men    63389.   69828 Men
##  7  2011 Canberra East ACT   Men    53139.     666 Men
##  8  2012 Canberra East ACT   Men    54515.     647 Men
##  9  2013 Canberra East ACT   Men    58132.     641 Men
## 10  2014 Canberra East ACT   Men    56247.     561 Men
## # ... with 4,009 more rows
```

10.5.2 Grouped mutates with `group_by()`

Like filtering, you can add or edit variables on grouped data. For example, you could get the average number of workers in each SA3 over the 6 years:

```
sa3_income %>%
  group_by(sa3_name, gender) %>%
  mutate(av_workers = mean(workers))
```

```
## # A tibble: 4,019 x 7
## # Groups:   sa3_name, gender [672]
##     year sa3_name      state gender income workers av_workers
##    <dbl> <chr>         <chr> <chr>   <dbl>   <dbl>      <dbl>
##  1  2011 Belconnen     ACT   Men    54105.   67774     68129.
```

```
##  2  2012 Belconnen     ACT   Men    56724.   69435     68129.
##  3  2013 Belconnen     ACT   Men    58918.   69697     68129.
##  4  2014 Belconnen     ACT   Men    60525.   68613     68129.
##  5  2015 Belconnen     ACT   Men    60964.   63428     68129.
##  6  2016 Belconnen     ACT   Men    63389.   69828     68129.
##  7  2011 Canberra East ACT   Men    53139.     666       641.
##  8  2012 Canberra East ACT   Men    54515.     647       641.
##  9  2013 Canberra East ACT   Men    58132.     641       641.
## 10  2014 Canberra East ACT   Men    56247.     561       641.
## # ... with 4,009 more rows
```

Above, the `mean()` function is applied separately to each unique group of `sa3_name` and `gender`, taking one average for women in Queanbeyan, one average for men in Queanbeyan, and so on.

Grouping a dataset does not prohibit operations that don't utilise the grouping. For example, you could get each year's workers relative to the SA3/gender average in the same call to `mutate`:

```
sa3_income %>%
  group_by(sa3_name, gender) %>%
  mutate(av_workers = mean(workers),
         worker_diff = workers / av_workers)
```

```
## # A tibble: 4,019 x 8
## # Groups:   sa3_name, gender [672]
##     year sa3_name      state gender income workers av_workers worker_diff
##    <dbl> <chr>         <chr> <chr>   <dbl>   <dbl>      <dbl>       <dbl>
##  1  2011 Belconnen     ACT   Men    54105.   67774     68129.       0.995
##  2  2012 Belconnen     ACT   Men    56724.   69435     68129.       1.02
##  3  2013 Belconnen     ACT   Men    58918.   69697     68129.       1.02
##  4  2014 Belconnen     ACT   Men    60525.   68613     68129.       1.01
##  5  2015 Belconnen     ACT   Men    60964.   63428     68129.       0.931
##  6  2016 Belconnen     ACT   Men    63389.   69828     68129.       1.02
##  7  2011 Canberra East ACT   Men    53139.     666       641.       1.04
##  8  2012 Canberra East ACT   Men    54515.     647       641.       1.01
##  9  2013 Canberra East ACT   Men    58132.     641       641.       1.00
## 10  2014 Canberra East ACT   Men    56247.     561       641.       0.875
## # ... with 4,009 more rows
```

See that the data remains grouped after the `mutate`. You can explicitly `ungroup()` afterwards:

```
sa3_income %>%
  group_by(sa3_name, gender) %>%
  mutate(av_workers = mean(workers),
         worker_diff = workers / av_workers) %>%
  ungroup()
```

```
## # A tibble: 4,019 x 8
##     year sa3_name      state gender income workers av_workers worker_diff
##    <dbl> <chr>         <chr> <chr>   <dbl>   <dbl>      <dbl>       <dbl>
##  1  2011 Belconnen     ACT   Men    54105.   67774     68129.       0.995
##  2  2012 Belconnen     ACT   Men    56724.   69435     68129.       1.02
##  3  2013 Belconnen     ACT   Men    58918.   69697     68129.       1.02
##  4  2014 Belconnen     ACT   Men    60525.   68613     68129.       1.01
##  5  2015 Belconnen     ACT   Men    60964.   63428     68129.       0.931
##  6  2016 Belconnen     ACT   Men    63389.   69828     68129.       1.02
##  7  2011 Canberra East ACT   Men    53139.     666       641.       1.04
##  8  2012 Canberra East ACT   Men    54515.     647       641.       1.01
##  9  2013 Canberra East ACT   Men    58132.     641       641.       1.00
## 10  2014 Canberra East ACT   Men    56247.     561       641.       0.875
## # ... with 4,009 more rows
```

10.6 Summarise data with `summarise()`

Summarising is a useful way to assess and present data. The `summarise` function collapses your data down into a single row, performing the operation(s) you provide:

```
sa3_income %>%
  summarise(mean_income = mean(income),
            total_workers = sum(workers))  # this is a silly statistic
```

```
## # A tibble: 1 x 2
##   mean_income total_workers
##         <dbl>         <dbl>
## 1      50272.     117002608
```

Summarising is usually only useful when combined with `group_by`.

10.6.1 Grouped summaries with `group_by()`

Grouped summaries can help change the *detail* of your data. In the original `sa3_income` data, there is a unique `workers` observation for each year, SA3 and gender. If you wanted to aggregate that information up see the total number of workers for each year and SA3:

```
sa3_income %>%
  group_by(year, sa3_name) %>%
  summarise(workers = sum(workers))
```

```
## # A tibble: 2,010 x 3
## # Groups:   year [6]
```

```
##     year sa3_name                                 workers
##    <dbl> <chr>                                      <dbl>
##  1  2011 Adelaide City                              18048
##  2  2011 Adelaide Hills                             59794
##  3  2011 Albany                                     43811
##  4  2011 Albury                                     50490
##  5  2011 Alice Springs                              31563
##  6  2011 Armadale                                   56088
##  7  2011 Armidale                                   27957
##  8  2011 Auburn                                     57298
##  9  2011 Augusta - Margaret River - Busselton       35852
## 10  2011 Bald Hills - Everton Park                  36273
## # ... with 2,000 more rows
```

After the `summarise` function, the dataset grouping remains but is reduced by one – so the right-hand-side grouping is lost. This enables a common combination to find a proportion of a group. For example, if you

Common functions to use with summarise

Grouped summaries generate summary statistics for grouped data. It uses the same `summarise` function, but is preceded with a `group_by`. For example, if you want to find the average income for women and men:

```
sa3_income %>%
  group_by(gender) %>%
  summarise(mean_income = mean(income))
```

```
## # A tibble: 2 x 2
##   gender mean_income
##   <chr>        <dbl>
## 1 Men         58780.
## 2 Women       41760.
```

Or the total workers in each year and state by gender:

```
sa3_income %>%
  group_by(year, state, gender) %>%
  summarise(workers = sum(workers))
```

```
## # A tibble: 96 x 4
## # Groups:   year, state [48]
##     year state gender workers
##    <dbl> <chr> <chr>    <dbl>
##  1  2011 ACT   Men     265281
##  2  2011 ACT   Women    88632
##  3  2011 NSW   Men    4438272
##  4  2011 NSW   Women  1415914
##  5  2011 NT    Men     140946
```

```
##  6  2011 NT    Women    44413
##  7  2011 Qld   Men    2859150
##  8  2011 Qld   Women   918841
##  9  2011 SA    Men     997160
## 10  2011 SA    Women   325980
## # ... with 86 more rows
```

10.7 Arrange with `arrange()`

'doesn't add or subtract to your data'

Sorting data in one way or another can be useful. Use the `arrange` function to sort data by the provided variable(s). Like with `select`, you can use the minus sign `-` to reverse the order.

For example, to find the areas in 2016 with the **least** workers:

```
sa3_income %>%
  filter(year == 2016) %>%
  arrange(workers)
```

```
## # A tibble: 670 x 6
##     year sa3_name                  state gender income workers
##    <dbl> <chr>                     <chr> <chr>   <dbl>   <dbl>
##  1  2016 Lord Howe Island          NSW   Women  37944       74
##  2  2016 Urriarra - Namadgi        ACT   Women  86672.      90
##  3  2016 Christmas Island          NT    Women  57640      141
##  4  2016 Canberra East             ACT   Women  52091.     182
##  5  2016 Lord Howe Island          NSW   Men    40292      255
##  6  2016 Urriarra - Namadgi        ACT   Men    86747.     296
##  7  2016 Christmas Island          NT    Men    84474.     621
##  8  2016 Barkly                    NT    Women  52552.     704
##  9  2016 Canberra East             ACT   Men    58035.     711
## 10  2016 Daly - Tiwi - West Arnhem NT    Women  50096.    1075
## # ... with 660 more rows
```

You can provide more than one variable. To sort the data first by `state` and, within each state, by the most workers (ie sorting by negative workers):

```
sa3_income %>%
  filter(year == 2016) %>%
  arrange(state, -workers)
```

```
## # A tibble: 670 x 6
##     year sa3_name       state gender income workers
##    <dbl> <chr>          <chr> <chr>   <dbl>   <dbl>
##  1  2016 Belconnen      ACT   Men    63389.   69828
```

```
##  2  2016 Tuggeranong    ACT   Men    66921.   65248
##  3  2016 Gungahlin      ACT   Men    66714.   55176
##  4  2016 North Canberra ACT   Men    62258.   37481
##  5  2016 Woden Valley   ACT   Men    66853.   24690
##  6  2016 Belconnen      ACT   Women  50756.   22982
##  7  2016 Tuggeranong    ACT   Women  52058.   21949
##  8  2016 South Canberra ACT   Men    72437.   20998
##  9  2016 Gungahlin      ACT   Women  50908.   18134
## 10  2016 Weston Creek   ACT   Men    67242.   15500
## # ... with 660 more rows
```

10.7.1 `lead` and `lag` functions with `arrange`

Having your data arranged in the way you want lets you use the `lead` (looking forward) and `lag` (looking backward) functions.

Both the `lead` and `lag` functions take a variable as their only requried argument. The default number of lags or leads is `1`, and this can be changed with the second argument. For example:

```
sa3_income %>%
  mutate(last_workers = lag(workers))
```

```
## # A tibble: 4,019 x 7
##     year sa3_name      state gender income workers last_workers
##    <dbl> <chr>         <chr> <chr>   <dbl>   <dbl>        <dbl>
##  1  2011 Belconnen     ACT   Men    54105.   67774           NA
##  2  2012 Belconnen     ACT   Men    56724.   69435        67774
##  3  2013 Belconnen     ACT   Men    58918.   69697        69435
##  4  2014 Belconnen     ACT   Men    60525.   68613        69697
##  5  2015 Belconnen     ACT   Men    60964.   63428        68613
##  6  2016 Belconnen     ACT   Men    63389.   69828        63428
##  7  2011 Canberra East ACT   Men    53139.     666        69828
##  8  2012 Canberra East ACT   Men    54515.     647          666
##  9  2013 Canberra East ACT   Men    58132.     641          647
## 10  2014 Canberra East ACT   Men    56247.     561          641
## # ... with 4,009 more rows
```

If you wanted to see the growth rate of income over time, you could `arrange` then `group_by` your data before creating an `income_growth` variable that is `income / lag(income)`.

```
sa3_income %>%
  arrange(sa3_name, gender, year) %>%
  group_by(sa3_name, gender) %>%
  mutate(income_growth = income / lag(income) - 1)
```

```
## # A tibble: 4,019 x 7
## # Groups:   sa3_name, gender [672]
##     year sa3_name      state gender income workers income_growth
##    <dbl> <chr>         <chr> <chr>   <dbl>   <dbl>         <dbl>
##  1  2011 Adelaide City SA    Men    48760.   13737        NA
##  2  2012 Adelaide City SA    Men    49974.   13730         0.0249
##  3  2013 Adelaide City SA    Men    52975.   13955         0.0601
##  4  2014 Adelaide City SA    Men    54818.   13782         0.0348
##  5  2015 Adelaide City SA    Men    54185.   13930        -0.0115
##  6  2016 Adelaide City SA    Men    56689.   15300         0.0462
##  7  2011 Adelaide City SA    Women  38359.    4311        NA
##  8  2012 Adelaide City SA    Women  40409.    4219         0.0534
##  9  2013 Adelaide City SA    Women  41287.    4281         0.0217
## 10  2014 Adelaide City SA    Women  42872     4200         0.0384
## # ... with 4,009 more rows
```

10.8 Putting it all together

You will often use a combination of the above `dplyr` functions to get your data into shape.

For example, say you want to get the total workers and total income in each state and year by gender. You could start with the `sa3_income` dataset, and then filter to year 2016, then create a new variable equal to `workers * income`, then group by year, state and gender before you summarise to get the statistics you want. With pipes, it could look something like:

```
sa3_income %>%
  filter(year == 2016) %>%
  mutate(total_income = workers * income) %>%
  group_by(year, state, gender) %>%
  summarise(total_workers = sum(workers),
            mean_income = mean(income),
            total_income = sum(total_income))
```

```
## # A tibble: 16 x 6
## # Groups:   year, state [8]
##     year state gender total_workers mean_income  total_income
##    <dbl> <chr> <chr>          <dbl>       <dbl>         <dbl>
##  1  2016 ACT   Men           293558      67901.  19336462167.
##  2  2016 ACT   Women          97565      58222.   5180698359.
##  3  2016 NSW   Men          4952353      62207. 314145522637
##  4  2016 NSW   Women        1575308      45003.  72633515399.
##  5  2016 NT    Men           157954      70961.  11488404531.
##  6  2016 NT    Women          48107      54143.   2607187917.
```

```
##  7  2016 Qld   Men          3110067      61794. 194644704512.
##  8  2016 Qld   Women         994436      44251.  44474845486.
##  9  2016 SA    Men          1041747      58602.  60710695691.
## 10  2016 SA    Women         340699      43034.  14705553952
## 11  2016 Tas   Men           316727      54427.  17485898880.
## 12  2016 Tas   Women         104040      40685.   4305640148.
## 13  2016 Vic   Men          3926751      59814. 236830412049.
## 14  2016 Vic   Women        1264225      42816.  55273320473.
## 15  2016 WA    Men          1756314      72582. 127679046129.
## 16  2016 WA    Women         540767      48537.  26179412110.
```

Or say you want to see the annual growth rate of female workers in the SA3 with the highest female income. You could filter to keep women, and then group by SA3, then get the highest income for each of SA3, then ungroup and filter to keep only the SA3 with the highest income, then arrange by year and get the annual worker growth:

```
sa3_income %>%
  filter(gender == "Women") %>%
  group_by(sa3_name) %>%
  mutate(highest_income = max(income)) %>%
  ungroup() %>%
  filter(highest_income == max(highest_income)) %>%
  arrange(year) %>%
  mutate(worker_growth = workers / lag(workers) - 1)
```

```
## # A tibble: 6 x 8
##    year sa3_name        state gender income workers highest_income worker_growth
##   <dbl> <chr>           <chr> <chr>   <dbl>   <dbl>          <dbl>         <dbl>
## 1  2011 Urriarra - Nam~ ACT   Women  48525.      84         86672.        NA
## 2  2012 Urriarra - Nam~ ACT   Women  51648.      96         86672.         0.143
## 3  2013 Urriarra - Nam~ ACT   Women  61858.     124         86672.         0.292
## 4  2014 Urriarra - Nam~ ACT   Women  72980.      99         86672.        -0.202
## 5  2015 Urriarra - Nam~ ACT   Women  68534.      72         86672.        -0.273
## 6  2016 Urriarra - Nam~ ACT   Women  86672.      90         86672.         0.25
```

10.9 Joining datasets with `left_join()`

Joining one dataset with another is incredibly useful and can be a difficult concept to grasp. The concept of joining one dataset to another is well introduced in Chapter 13 of R for Data Science:

> It's rare that a data analysis involves only a single table of data. Typically you have many tables of data, and you must combine them to answer the questions that you're interested in. Collectively, multiple

> tables of data are called **relational data** because it is the relations, not just the individual datasets, that are important.

The `dplyr` package 'Join two tbls together' page provides a comprehensive summary of all join types. We will explore the key use of joins in our line of work – `left_join` – below.

A 'left' join takes your main dataset, and adds variables from a new dataset based on a matching condition **that's unhelpful, fix**. If an observation in the new dataset is not found in the main dataset, it is ignored.

It is probably easier to show this with an example. Say that we had the income percentiles of each SA3 in each year from a different data source:

```
sa3_percentiles <- read_csv("data/sa3_percentiles.csv")
```

```
## Parsed with column specification:
## cols(
##   sa3_name = col_character(),
##   year = col_double(),
##   sa3_income_percentile = col_double()
## )
```

Chapter 11

Analysis

Chapter 12

Data Visualisation

This chapter explores data visualisation broadly, and how to 'do' data visualisation in R specifically.

The next chapter – the Visualisation Cookbook – gives more practical advice for the charts you might want to create.

12.1 Introduction to data visualisation

You can use data visualisation to **examine and explore** your data, and to **present** a finding to your audience. Both of these elements are important.

When you start using a dataset, you should look at it.[1] Plot histograms of variables-of-interest to spot outliers. Explore correlations between variables with scatter plots and lines-of-best-fit. Check how many observations are in particular groups with bar charts. Identify variables that have missing or coded-missing values. Use faceting to explore differences in the above between groups, and do it interactively with non-static plots.

These **exploratory plots** are just for you and your team. They don't need to be perfectly labelled, the right size, in the Grattan palette, or be particularly interesting. They're built and used only to help you and your team explore the data. Through this process, you can become confident your data is *what you think it is*.

When you choose to **present a visualisation to a reader**, you have to make decisions about what they can and cannot see. You need to highlight or omit

[1] From Kieran Healy's *Data Vizualization: A Practical Introduction*: 'You should look at your data. Graphs and charts let you explore and learn about the structure of the information you collect. Good data visualizations also make it easier to communicate your ideas and findings to other people.'

particular things to help them better understand the message you are presenting.

This requires important *technical* decisions: what data to use, what 'stat' to present it with — *show every data point, show a distribution function, show the average or the median?* — and on what scale — *raw numbers, on a log scale, as a proportion of a total?*.

It also requires *aesthetic* decisions. What colours in the Grattan palette would work best? Where should the labels be placed and how could they be phrased to succinctly convey meaning? Should data points be represented by lines, or bars, or dots, or balloons, or shades of colour?

All of these decisions need to made with two things in mind:

1. Rigour, accuracy, legitimacy: the chart needs to be honest.
2. The reader: the chart needs to help the reader understand something, and it must convince them to pay attention.

At the margins, sometimes these two ideas can be in conflict. Maybe a 70-word definition in the middle of your chart would improve its technical accuracy, but it could confuse the average reader and reduce the chart's impact.

Similarly, a bar chart is often the safest way to display data. Like our prose, our charts need to be designed for an interested teenager. But we need to *earn* their interest. If your reader has seen four similar bar charts in a row and has stopped paying attention by the fifth, your point loses its punch.[2]

The way we design charts – much like our writing – should always be honest, clear and engaging to the reader.

This chapter shows how you can do this with R. It starts with the 'grammar of graphics' concepts of a package called `ggplot`, and explains how to make those charts 'Grattan-y'. The next chapter gives you the when-to-use and how-to-make particular charts.

12.2 Set-up and packages

This section uses the package `ggplot2` to visualise data, and `dplyr` functions to manipulate data. Both of these packages are loaded with `tidyverse`. The `scales` package helps with labelling your axes.

The `grattantheme` package is used to make charts look Grattan-y. The `absmapsdata` package is used to help make maps.

```
library(tidyverse)
library(grattantheme)
```

[2] 'Bar charts are evidence that you are dead inside' – Amanda Cox, data editor for the New York Times.

```
library(ggrepel)
library(scales)
```

```
# note: to be added to grattantheme; remove this when done
grattan_label <- function(..., size = 18) {

  .size = size / ggplot2::.pt

geom_label(...,
           fill = "white",
           label.padding = unit(0.01, "lines"),
           label.size = 0,
           size = .size)
}
```

For most charts in this chapter, we'll use the `sa3_income` data summarised below.[3] It is a long dataset containing the median income and number of workers by SA3, occupation and gender between 2010 and 2015. We will also create a `professionals` subset that only includes people in professional occupations in 2015:

```
sa3_income <- read_csv("data/sa3_income.csv")

professionals <- sa3_income %>%
  select(-sa4_name, -gcc_name) %>%
  filter(year == 2015,
         occupation == "Professionals",
         !is.na(median_income),
         !gender == "Persons")

# Show the first six rows of the new dataset
head(professionals)
```

```
## # A tibble: 6 x 14
##     sa3 sa3_name sa3_sqkm sa3_income_perc~ state occupation occ_short prof
##   <dbl> <chr>       <dbl>            <dbl> <chr> <chr>      <chr>     <chr>
## 1 10102 Queanbe~    6511.               74 NSW   Professio~ Professi~ Prof~
## 2 10102 Queanbe~    6511.               74 NSW   Professio~ Professi~ Prof~
## 3 10102 Queanbe~    6511.               74 NSW   Professio~ Professi~ Prof~
## 4 10103 Snowy M~   14283.                7 NSW   Professio~ Professi~ Prof~
## 5 10103 Snowy M~   14283.                7 NSW   Professio~ Professi~ Prof~
## 6 10103 Snowy M~   14283.                7 NSW   Professio~ Professi~ Prof~
## # ... with 6 more variables: gender <chr>, year <dbl>, median_income <dbl>,
## #   average_income <dbl>, total_income <dbl>, workers <dbl>
```

[3] From ABS Employee income by occupation and gender, 2010-11 to 2015-16

12.3 Concepts

The `ggplot2` package is based on the **g**rammar of **g**raphics. …

The main ingredients to a `ggplot` chart are:

- **Data**: what data should be plotted.
  - e.g. `data`
- **Aesthetics**: what variables should be linked to what chart elements.
  - e.g. `aes(x = population, y = age)` to connect the `population` variable to the `x` axis, and the `age` variable to the `y` axis.
- **Geoms**: how the data should be plotted.
  - e.g. `geom_point()` will produce a scatter plot, `geom_col` will produce a column chart, `geom_line()` will produce a line chart.

Each plot you make will be made up of these three elements. The full list of standard geoms is listed in the `tidyverse` documentation.

`ggplot` also has a 'cheat sheet' that contains many of the often-used elements of a plot, which you can download here.

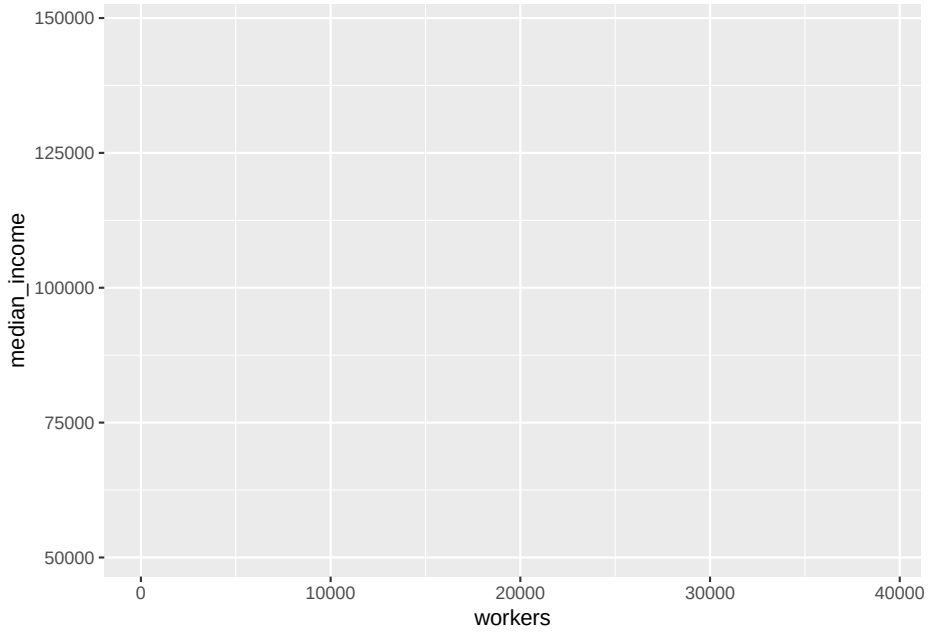
For example, you can plot a column chart by passing the `sa3_income` dataset into `ggplot()` ("make a chart with this data"). This completes the first step – data – and produces an empty plot:

```
professionals %>%
        ggplot()
```

Next, set the `aes` (aesthetics) to `x = state` ("make the x-axis represent state"), `y = pop` ("the y-axis should represent population"), and `fill = year` ("the fill colour represents year"). Now `ggplot` knows where things should *go*.

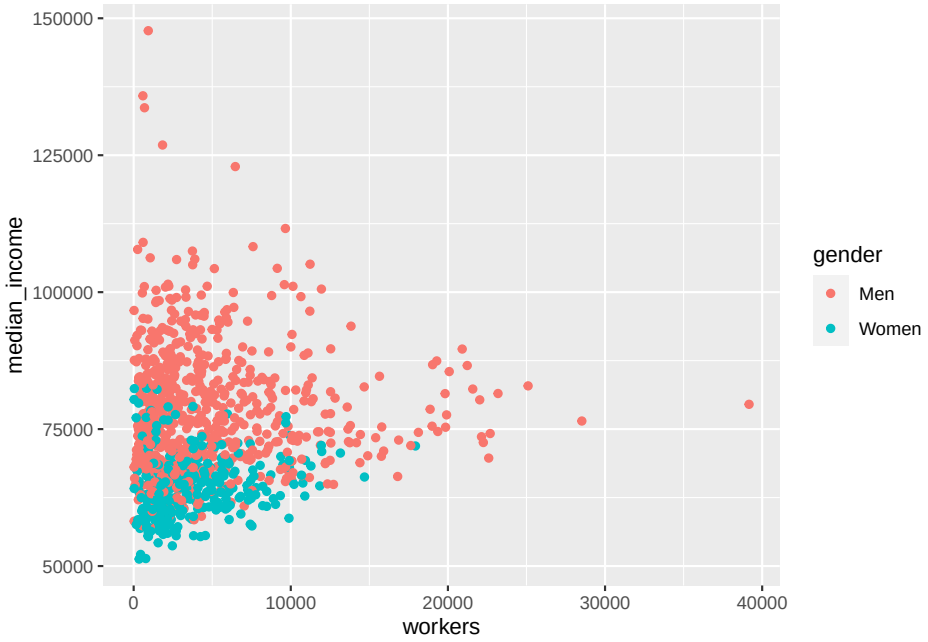
If we just plot that, you'll see that `ggplot` knows a little bit more about what we're trying to do. It has the states on the x-axis and range of populations on the y-axis:

```
professionals %>%
        ggplot(aes(x = workers,
                   y = median_income,
                   colour = gender))
```



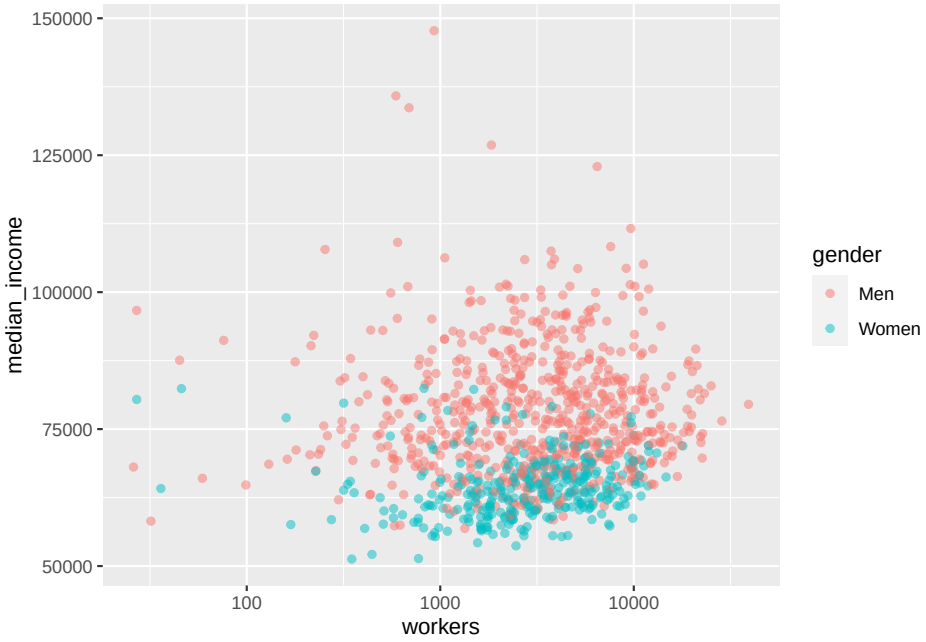
Now that `ggplot` knows where things should go, it needs to how to *plot* them on the chart. For this we use `geoms`. Tell `ggplot` to take the things it knows and plot them as a column chart by using `geom_col`:

```
professionals %>%
        ggplot(aes(x = workers,
                   y = median_income,
                   colour = gender)) +
        geom_point()
```



Great! There are a couple of quick things we can do to make the chart a bit clearer. There are points for each group in each year, which we probably don't need. So filter the data before you pass it to `ggplot` to just include 2015: `filter(year == 2015)`. There will still be lots of overlapping points, so set the opacity to below one with `alpha = 0.5`. The `workers` x-axis can be changed to a log scale with `scale_x_log10`.

```
professionals %>%
        ggplot(aes(x = workers,
                   y = median_income,
                   colour = gender)) +
        geom_point(alpha = .5) +
        scale_x_log10()
```



That looks a bit better. The following sections in this chapter will cover a broad range of charts and designs, but they will all use the same building-blocks of `data`, `aes`, and `geom`.

The rest of the chapter will explore:

- Exploratory data visualisation
- Grattanising your charts and choosing colours
- Saving charts according to Grattan templates
- Making bar, line, scatter and distribution plots
- Making maps and interactive charts
- Adding chart labels

12.4 Exploratory data visualisation

Plotting your data early in the analysis stage can help you quickly identify outliers, oddities, things that don't look quite right.

12.5 Making Grattan-y charts

The `grattantheme` package contains functions that help *Grattanise* your charts. It is hosted here: https://github.com/mattcowgill/grattantheme

You can install it with `remotes::install_github` from the package:

```
install.packages("remotes")
remotes::install_github("mattcowgill/grattantheme")
```

The key functions of `grattantheme` are:

- `theme_grattan`: set size, font and colour defaults that adhere to the Grattan style guide.
- `grattan_y_continuous`: sets the right defaults for a continuous y-axis.
- `grattan_colour_continuous`: pulls colours from the Grattan colour palette for `colour` aesthetics.
- `grattan_fill_continuous`: pulls colours from the Grattan colour palette for `fill` aesthetics.
- `grattan_save`: a save function that exports charts in correct report or presentation dimensions.

This section will run through some examples of *Grattanising* charts. The `ggplot` functions are explored in more detail in the next section.

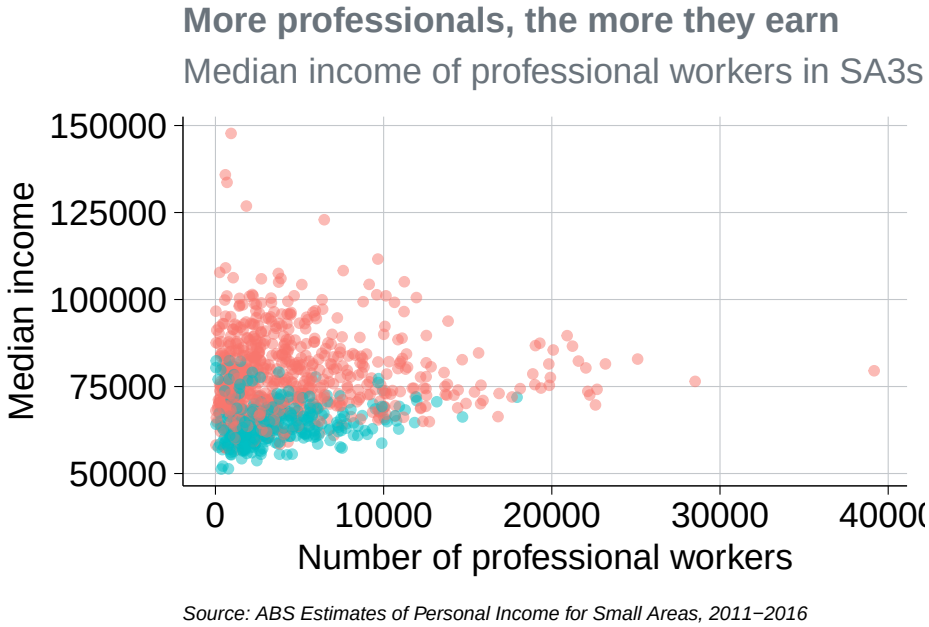
12.5.1 Making Grattan charts

Start with a scatterplot, similar to the one made above:

```
base_chart <- professionals %>%
        ggplot(aes(x = workers,
                   y = median_income,
                   colour = gender)) +
        geom_point(alpha = .5) +
        labs(title = "More professionals, the more they earn",
             subtitle = "Median income of professional workers in SA3s",
             x = "Number of professional workers",
             y = "Median income",
             caption = "Source: ABS Estimates of Personal Income for Small Areas, 2011-2016")

base_chart
```

Let's make it Grattany. First, add `theme_grattan` to your plot:

```
base_chart +
        theme_grattan(chart_type = "scatter")
```



Then use `grattan_y_continuous` to adjust the y-axis. This takes the same

arguments as the standard `scale_y_continuous` function, but has Grattan defaults built in. Use it to set the labels as dollars (with `scales::dollar()`) and to give the y-axis some breathing room (starting at $50,000 rather than the minimum point). Also add `scale_x_log10` to make the x-axis a log10 scale, telling it to format the labels as numbers with commas (using `scales::comma()`).[4]

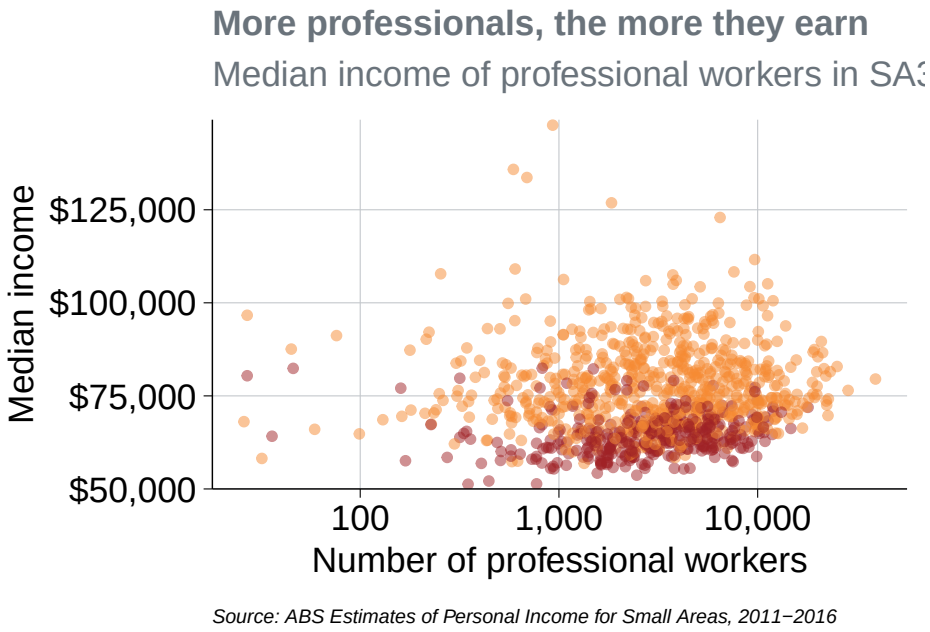
```
base_chart +
        theme_grattan(chart_type = "scatter") +
        grattan_y_continuous(labels = dollar, limits = c(50e3, NA)) +
        scale_x_log10(labels = comma)
```

Source: ABS Estimates of Personal Income for Small Areas, 2011–2016

To define `colour` colours, use `grattan_colour_manual` with the number of colours you need (two, in this case):

```
prof_chart <- base_chart +
        theme_grattan(chart_type = "scatter") +
        grattan_y_continuous(labels = dollar, limits = c(50e3, NA)) +
        scale_x_log10(labels = comma) +
        grattan_colour_manual(2)

prof_chart
```

[4]The `dollar` and `comma` commands are functions, but can be used without `()`. Using `dollar()` or `comma()` works too, and you can provide arguments that adjust their output: eg `dollar(suffix = "million")`



Nice chart! Now you can save it and share it with the world.

12.5.2 Saving Grattan charts

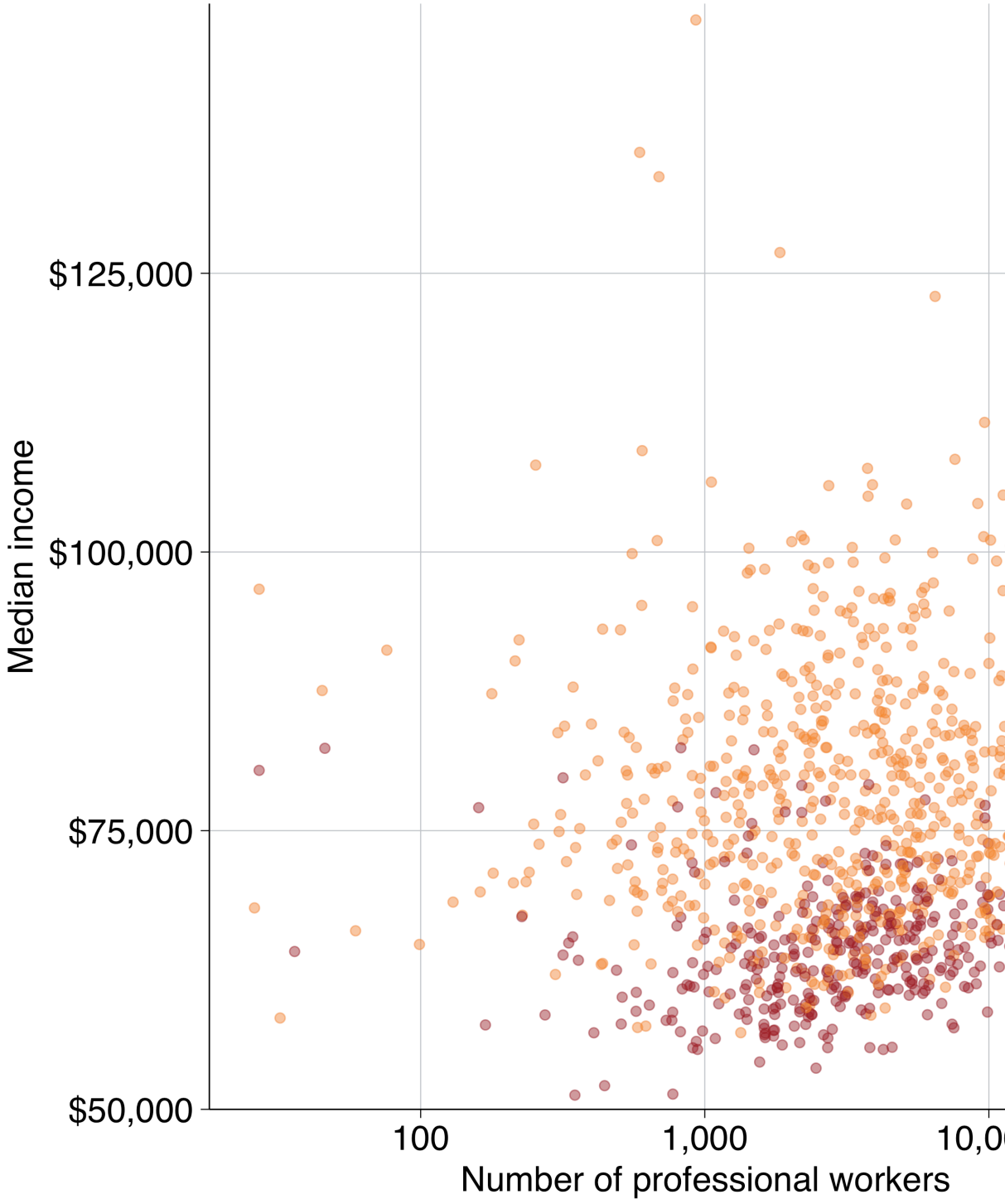
The `grattan_save` function saves your charts according to Grattan templates. It takes these arguments:

- `filename`: the path, name and file-type of your saved chart. eg: `"atlas/professionals_chart.pdf"`.
- `object`: the R object that you want to save. eg: `prof_chart`. If left blank, it grabs the last chart that was displayed.
- `type`: the Grattan template to be used. This is one of:
  - `"normal"` The default. Use for normal Grattan report charts, or to paste into a 4:3 PowerPoint slide. Width: 22.2cm, height: 14.5cm.
  - `"normal_169"` Only useful for pasting into a 16:9 format Grattan PowerPoint slide. Width: 30cm, height: 14.5cm.
  - `"tiny"` Fills the width of a column in a Grattan report, but is shorter than usual. Width: 22.2cm, height: 11.1cm.
  - `"wholecolumn"` Takes up a whole column in a Grattan report. Width: 22.2cm, height: 22.2cm.
  - `"fullpage"` Fills a whole page of a Grattan report. Width: 44.3cm, height: 22.2cm.
  - `"fullslide"` Creates an image that looks like a 4:3 Grattan PowerPoint slide, complete with logo. Width: 25.4cm, height: 19.0cm.
  - `"fullslide_169"` Creates‘ an image that looks like a 16:9 Grattan

PowerPoint slide, complete with logo. Use this to drop into standard presentations. Width: 33.9cm, height: 19.0cm
  - `"blog"` Creates a 4:3 image that looks like a Grattan PowerPoint slide, but with less border whitespace than 'fullslide'."
  - `"fullslide_44" Creates` an image that looks like a 4:4 Grattan PowerPoint slide. This may be useful for taller charts for the Grattan blog; not useful for any other purpose. Width: 25.4cm, height: 25.4cm.
  - Set `type = "all"` to save your chart in all available sizes.
- `height`: override the height set by `type`. This can be useful for really long charts in blogposts.
- `save_data`: exports a `csv` file containing the data used in the chart.
- `force_labs`: override the removal of labels for a particular `type`. eg `force_labs = TRUE` will keep the y-axis label.

To save the `prof_chart` plot created above as a whole-column chart for a **report**:

```
grattan_save("atlas/professionals_chart_report.pdf", prof_chart, type = "wholecolumn")
```

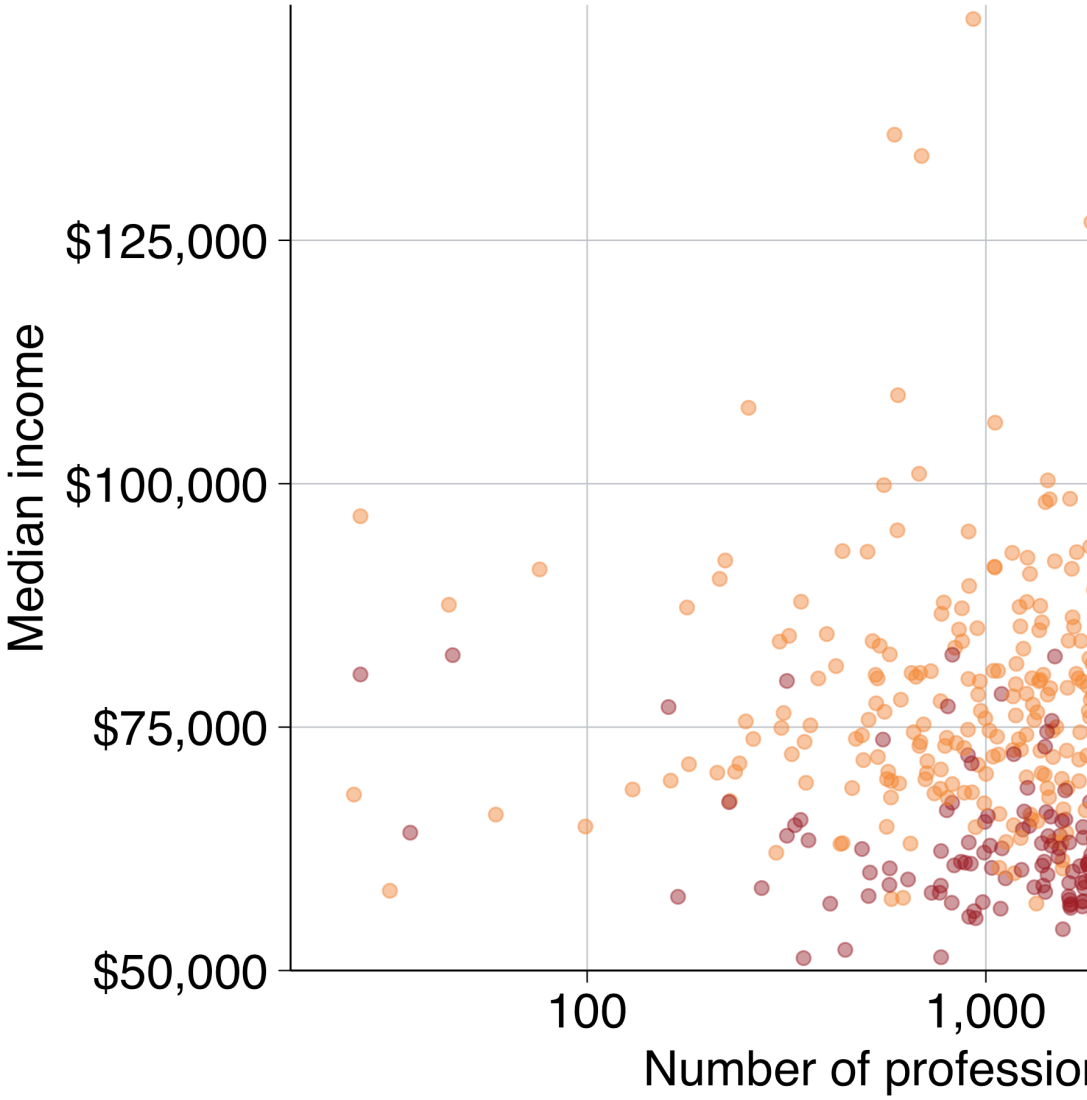

$125,000
$100,000
$75,000
$50,000
Median income
100
1,000
Number of professional workers

To save it as a **presentation** slide instead, use `type = "fullslide"`:

```
grattan_save("atlas/professionals_chart_presentation.pdf", prof_chart, type = "fullslide")
```

More professionals, the more they earn

Median income of professional workers in SA3s



Source: ABS Estimates of Personal Income for Small Areas, 2011-2016

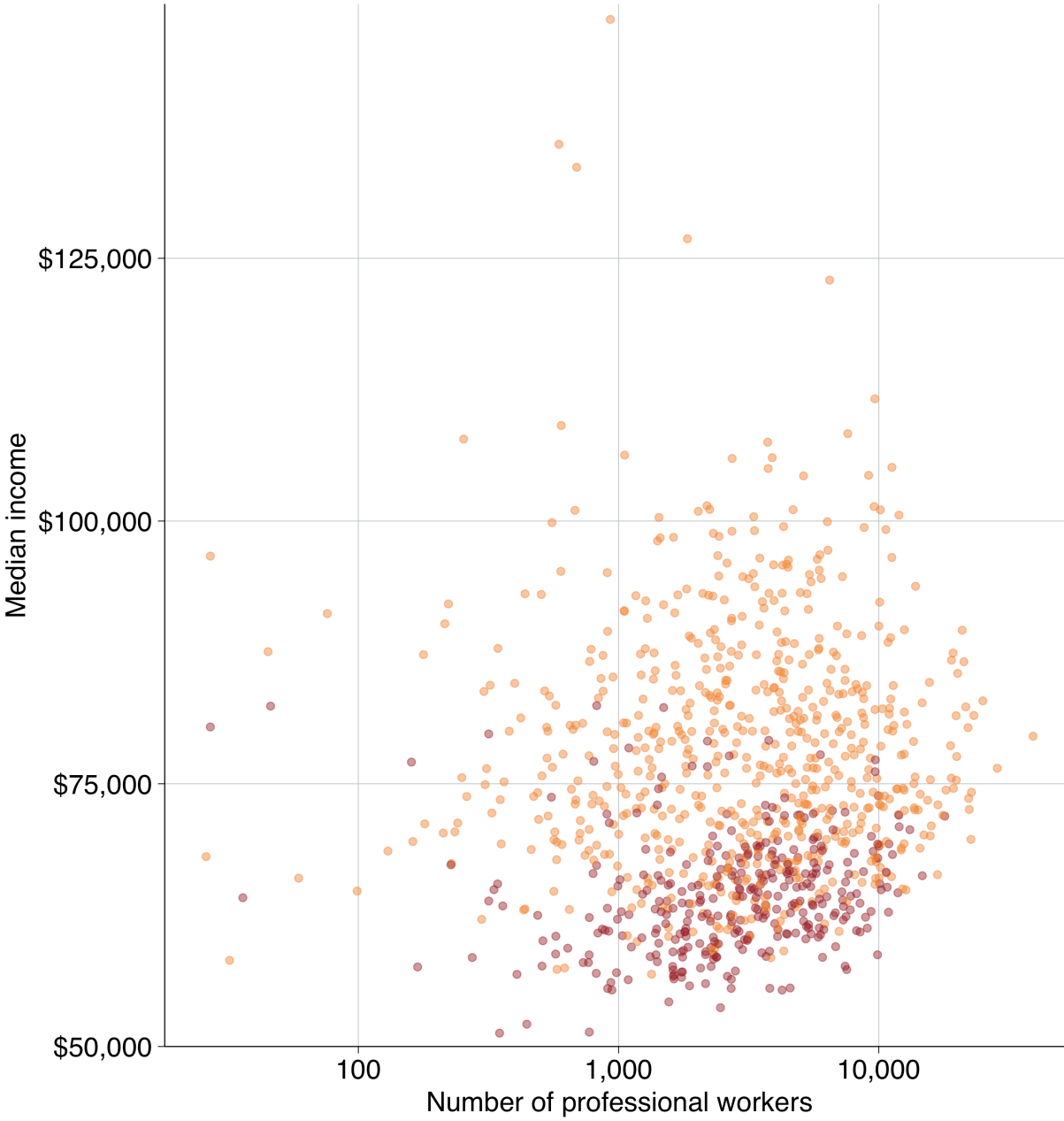
Or, if you want to emphasise the point in a *really tall* chart for a **blogpost**, you can use `type = "blog"` and adjust the `height` to be 50cm. Also note that because this is for the blog, you should save it as a `png` file:

```
grattan_save("atlas/professionals_chart_blog.png", prof_chart,
             type = "blog", height = 30)
```

More professionals, the more they earn

GRATTAN Institu

Median income of professional workers in SA3s



Source: ABS Estimates of Personal Income for Small Areas, 2011-2016

And that's it! The following sections will go into more detail about different chart types in R, but you'll mostly use the same basic `grattantheme` formatting you've used here.

12.6 Adding labels

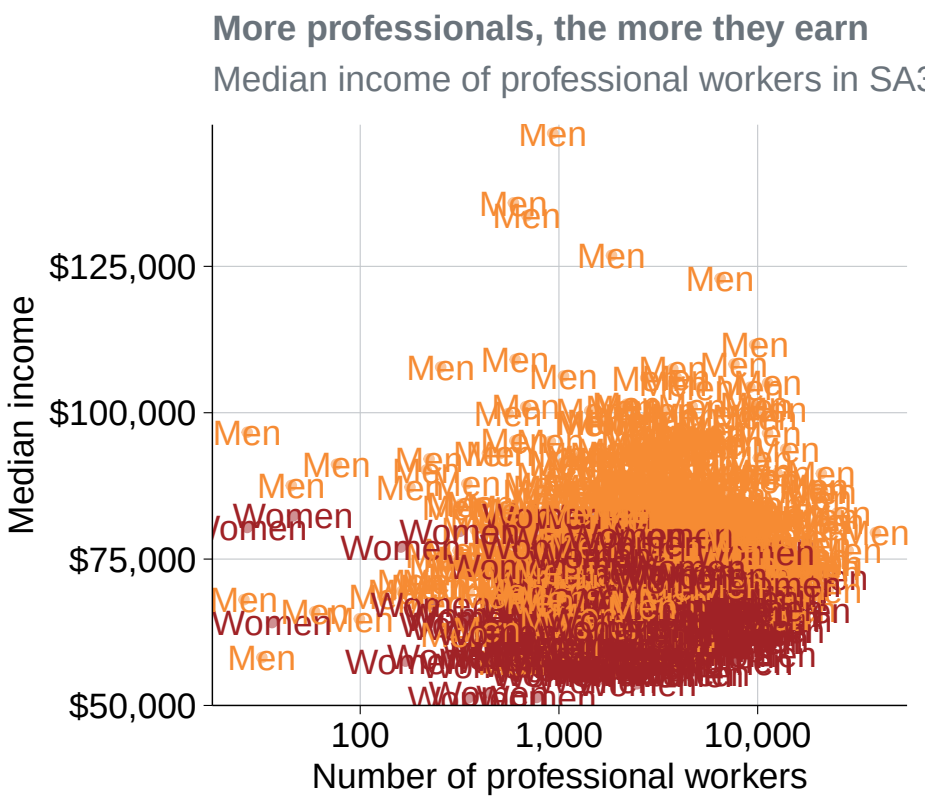
Labels can be a bit finicky – especially compared to labelling charts visually in PowerPoint. …

Labels can be done in two broad ways:

1. Labelling every single data point on your chart. Grattan charts rarely do this.
2. Labelling some of the data points on your chart. This is how you label Grattan charts: label on item in a group and let the reader join the dots.

We'll look at the first approach so you can get a feel for how the labelling geoms – `geom_label` and `geom_text` (and some useful extensions) – work. It won't be pretty.

```
prof_chart +
  geom_text(aes(label = gender))
```



Great! That looks *terrible*. `geom_text` is labelling each individual point because it has been told to do so. Just like `geom_point`, it takes the `x` and `y` aesthetics of each observation, then plots the `label` at that location. But we just want to label one of the points for `female` and one for `male`.

To do this, we can create a new dataset that just contains one observation each. Here, you're filtering the dataset to include *only* the female/male observations that have the most people:

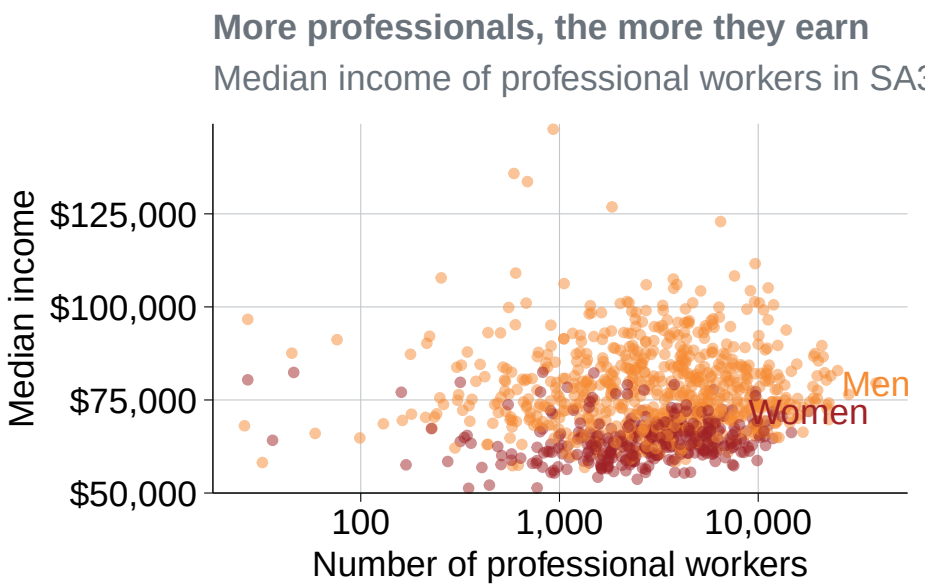
```
label_data <- professionals %>%
  group_by(gender) %>%
  filter(workers == max(workers)) %>%
  ungroup()

label_data
```

```
## # A tibble: 2 x 14
##     sa3 sa3_name sa3_sqkm sa3_income_perc~ state occupation occ_short prof
##   <dbl> <chr>       <dbl>            <dbl> <chr> <chr>      <chr>     <chr>
## 1 11703 Sydney ~     25.1               84 NSW   Professio~ Professi~ Prof~
```

```
## 2 11703 Sydney ~     25.1               84 NSW   Professio~ Professi~ Prof~
## # ... with 6 more variables: gender <chr>, year <dbl>, median_income <dbl>,
## #   average_income <dbl>, total_income <dbl>, workers <dbl>
```

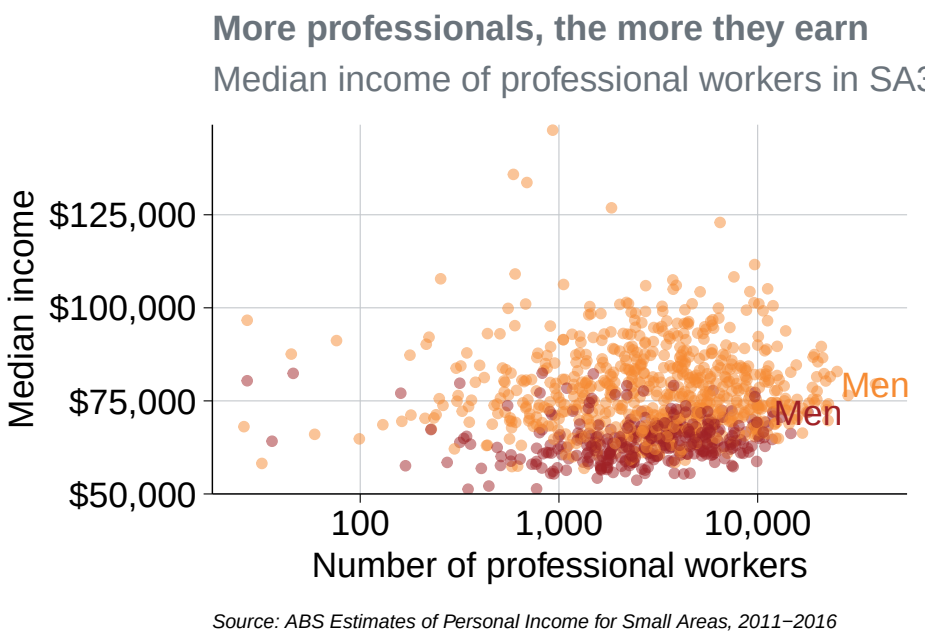
And then tell `geom_text` to look at *that* dataset:

```
prof_chart +
  geom_text(data = label_data,
            aes(label = gender))
```



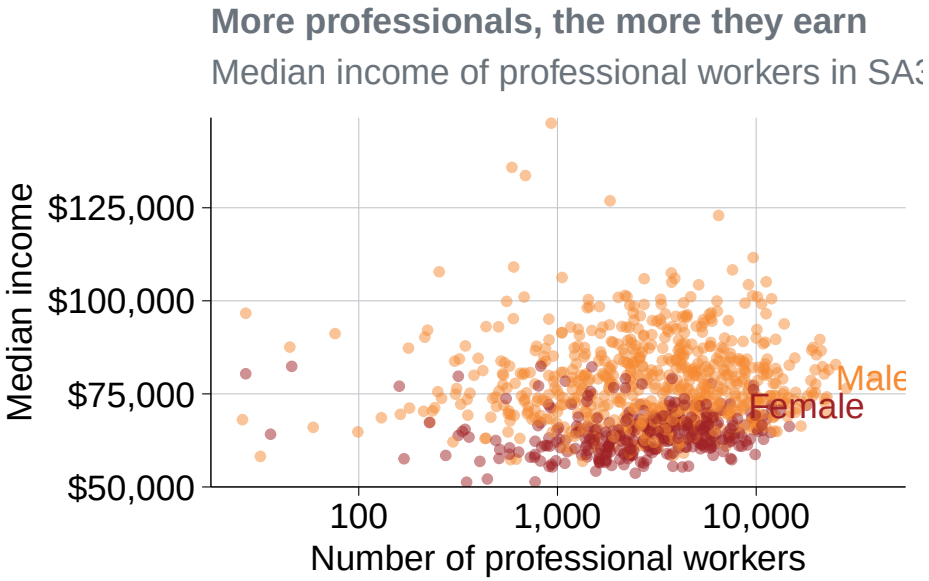
Okay, not bad. The labels go off the chart. You could fix this by shortening the labels either inside the `label_data`:

```
label_data_short <- label_data %>%
  mutate(gender_label = if_else(gender == "Females",
                                "Women",
                                "Men"))

prof_chart +
  geom_text(data = label_data_short,
            aes(label = gender_label))
```



Or you could adjust the label values directly inside the aesthetics call. Note that this means you have to provide a vector that is the same length as the number of observations in the data (a length of two, in this case).

```
prof_chart +
  geom_text(data = label_data,
            aes(label = c("Female", "Male")))
```



Source: ABS Estimates of Personal Income for Small Areas, 2011–2016

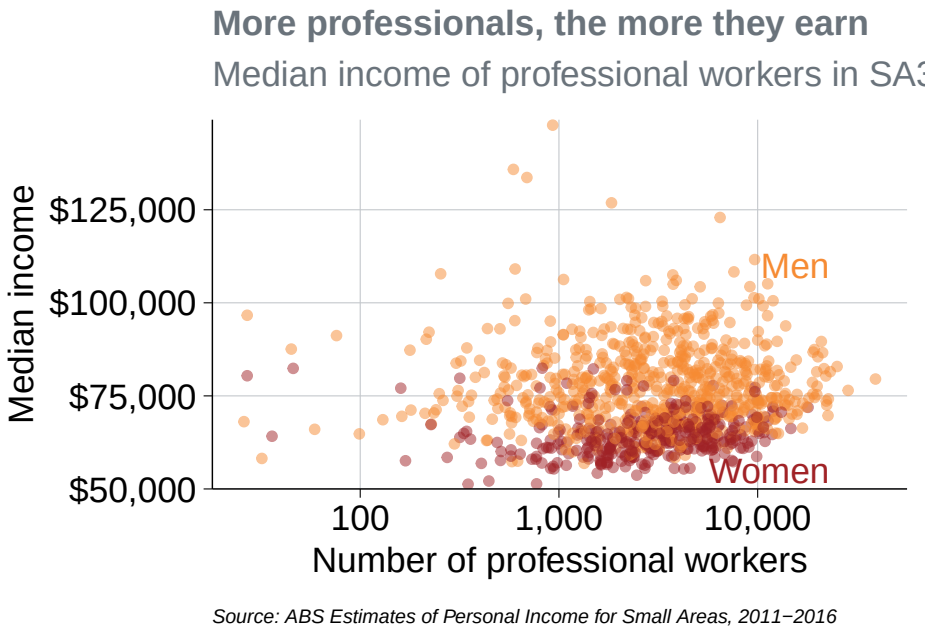
To have more freedom over *where* your labels are placed, you can create a dataset yourself. Add the `x` and `y` values for your labels, and the label names.[5]

```
self_label <- tribble(
  ~gender, ~workers,   ~median_income,
  "Women",    23000,           55000,
  "Men",      23000,          110000)


self_label
```

```
## # A tibble: 2 x 3
##   gender workers median_income
##   <chr>    <dbl>         <dbl>
## 1 Women    23000         55000
## 2 Men      23000        110000
```

```
prof_chart +
  geom_text(data = self_label,
            aes(label = gender),
            hjust = 1)
```

[5]We are using the `tribble` function here to make it a little bit clearer what values apply to which gender. The 'normal' way to create a tibble is with the `tibble` function: `tibble(x = c(10, 100), y = c(100, 10))`, etc.



[cover `annotate`]

Chapter 13

Chart cookbook

This section takes you through a few often-used chart types.

13.1 Set up

```
library(tidyverse)
library(grattantheme)
library(ggrepel)
library(absmapsdata)
library(sf)
library(scales)
library(janitor)
# this might be hairy; should get `grattools` happening:
library(grattan)
```

```
# note: to be added to grattantheme; remove this when done
grattan_label_repel <- function(..., size = 18) {

  .size = size / ggplot2::.pt

  geom_label_repel(...,
                   fill = "white",
                   label.padding = unit(0.1, "lines"),
                   label.size = 0,
                   size = .size)
}
```

```
grattan_label <- function(..., size = 18) {

  .size = size / ggplot2::.pt

  geom_label(...,
                   fill = "white",
                   label.padding = unit(0.1, "lines"),
                   label.size = 0,
                   size = .size)
}
```

The sa3_income dataset will be used for all key examples in this chapter.[1] It is a long dataset from the ABS that contains the median income and number of workers by Statistical Area 3, occupation and sex between 2010 and 2016.

```
sa3_income <- read_csv("data/sa3_income.csv") %>%
  filter(!is.na(median_income),
         !is.na(average_income))
```

```
## Parsed with column specification:
## cols(
##   sa3 = col_double(),
##   sa3_name = col_character(),
##   sa3_sqkm = col_double(),
##   sa3_income_percentile = col_double(),
##   sa4_name = col_character(),
##   gcc_name = col_character(),
##   state = col_character(),
##   occupation = col_character(),
##   occ_short = col_character(),
##   prof = col_character(),
##   gender = col_character(),
##   year = col_double(),
##   median_income = col_double(),
##   average_income = col_double(),
##   total_income = col_double(),
##   workers = col_double()
## )
```

```
head(sa3_income)
```

```
## # A tibble: 6 x 16
##     sa3 sa3_name sa3_sqkm sa3_income_perc~ sa4_name gcc_name state occupation
##   <dbl> <chr>       <dbl>            <dbl> <chr>    <chr>    <chr> <chr>
## 1 10102 Queanbe~    6511.               80 Capital~ Rest of~ NSW   Clerical ~
```

[1] From ABS Employee income by occupation and sex, 2010-11 to 2016-16

```
## 2 10102 Queanbe~    6511.              76 Capital~ Rest of~ NSW   Clerical ~
## 3 10102 Queanbe~    6511.              78 Capital~ Rest of~ NSW   Clerical ~
## 4 10102 Queanbe~    6511.              76 Capital~ Rest of~ NSW   Clerical ~
## 5 10102 Queanbe~    6511.              74 Capital~ Rest of~ NSW   Clerical ~
## 6 10102 Queanbe~    6511.              79 Capital~ Rest of~ NSW   Clerical ~
## # ... with 8 more variables: occ_short <chr>, prof <chr>, gender <chr>,
## #   year <dbl>, median_income <dbl>, average_income <dbl>, total_income <dbl>,
## #   workers <dbl>
```

13.2 Bar charts

Bar charts are made with `geom_bar` or `geom_col`. Creating a bar chart will look something like this:

```
ggplot(data = <data>) +
  geom_bar(aes(x = <xvar>, y = <yvar>),
     stat = <STAT>,
     position = <POSITION>
  )
```

It has two key arguments: `stat` and `position`.

First, `stat` defines what kind of *operation* the function will do on the dataset before plotting. Some options are:

- `"count"`, the **default**: count the number of observations in a particular group, and plot that number. This is useful when you're using microdata. When this is the case, there is no need for a `y` aesthetic.
- `"sum"`: sum the values of the `y` aesthetic.
- `"identity"`: directly report the values of the `y` aesthetic. This is how PowerPoint and Excel charts work.

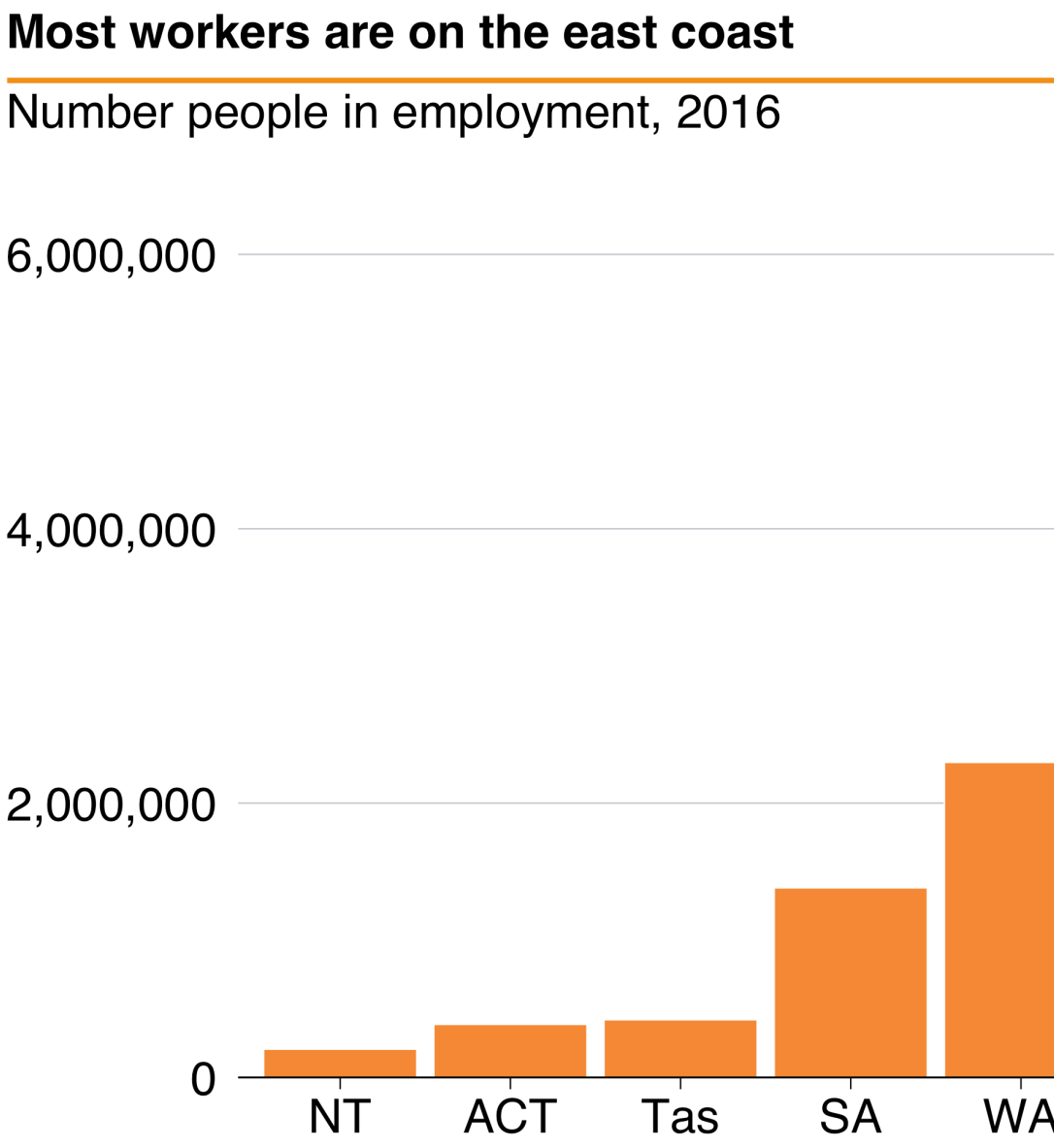
You can use `geom_col` instead, as a shortcut for `geom_bar(stat = "identity)`.

Second, `position`, dictates how multiple bars occupying the same x-axis position will positioned. The options are:

- `"stack"`, the default: bars in the same group are stacked atop one another.
- `"dodge"`: bars in the same group are positioned next to one another.
- `"fill"`: bars in the same group are stacked and all fill to 100 per cent.

13.2.1 Simple bar plot

This section will create the following vertical bar plot showing number of workers by state in 2016:



Notes: Only includes people who submitted a tax return in 2016-16.
Source: ABS (2018)

First, create the data you want to plot.

```
data <- sa3_income %>%
  filter(year == 2016) %>%
  group_by(state) %>%
  summarise(workers = sum(workers))

data
```

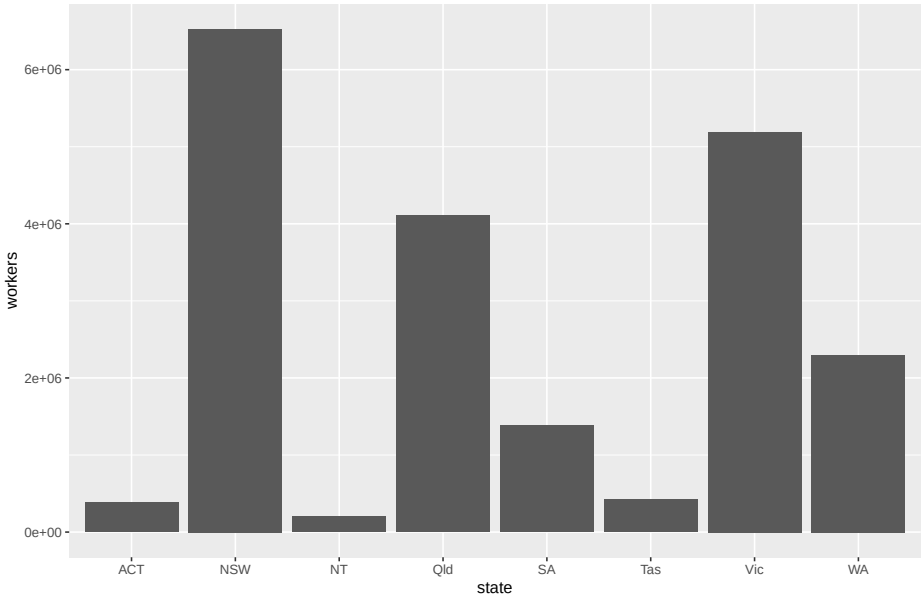
```
## # A tibble: 8 x 2
##   state workers
##   <chr>   <dbl>
## 1 ACT    386989
## 2 NSW   6527661
## 3 NT     206061
## 4 Qld   4104503
## 5 SA    1382446
## 6 Tas    420767
## 7 Vic   5190976
## 8 WA    2297081
```

Looks awesomesauce: you have one observation (row) for each state you want to plot, and a value for their number of workers.

Now pass the nice, simple table to `ggplot` and add aesthetics so that `x` represents `state`, and `y` represents `workers`. Then, because the dataset contains the *actual* numbers you want on the chart, you can plot the data with `geom_col`:[2]

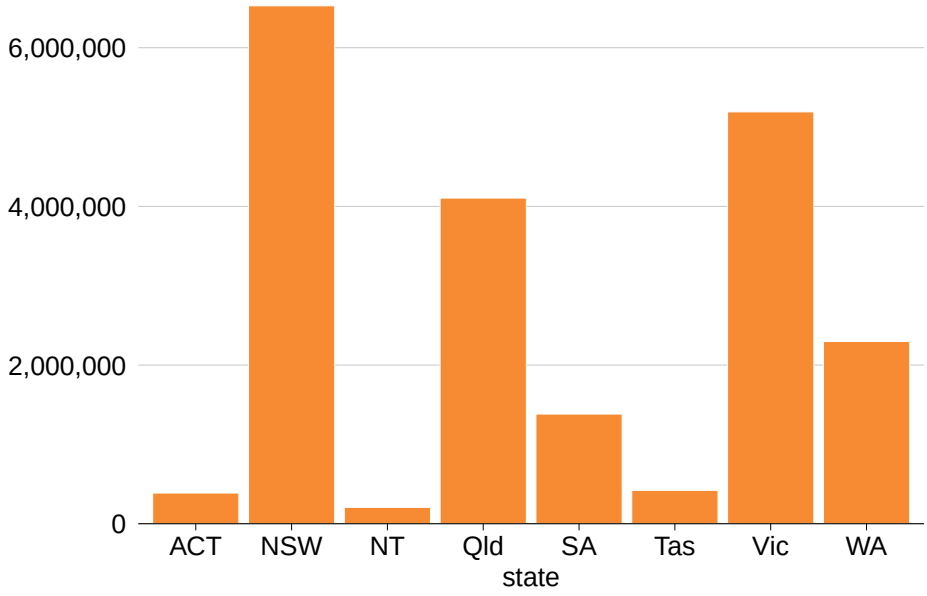
```
data %>%
  ggplot(aes(x = state,
             y = workers)) +
  geom_col()
```

[2]Remember that `geom_col` is just shorthand for `geom_bar(stat = "identity")`



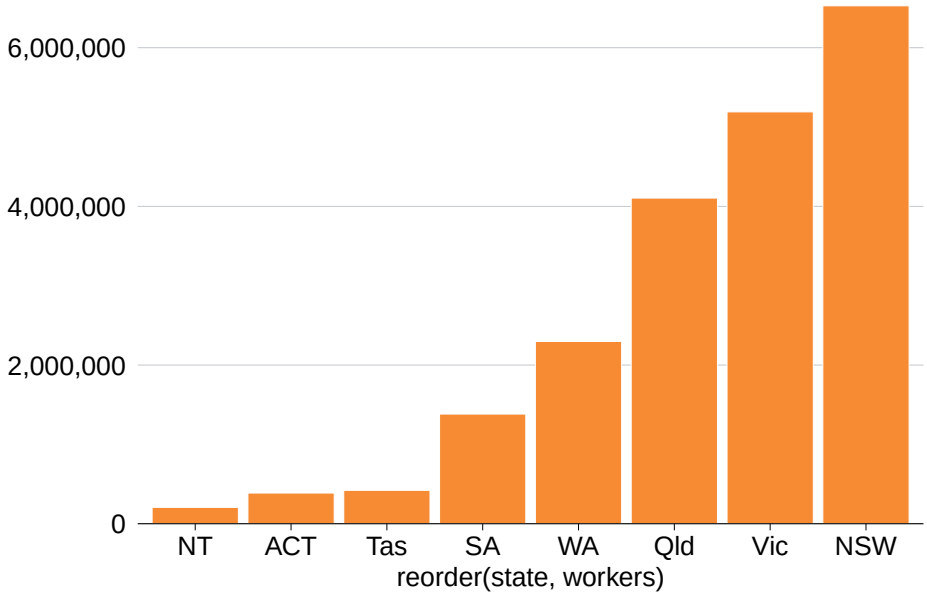
Make it Grattany by adjusting general theme defaults with `theme_grattan`, and use `grattan_y_continuous` to change the y-axis. Use labels formatted with commas (rather than scientific notation) by adding `labels = comma`.

```
data %>%
  ggplot(aes(x = state,
             y = workers)) +
  geom_col() +
  theme_grattan() +
  grattan_y_continuous(labels = comma)
```



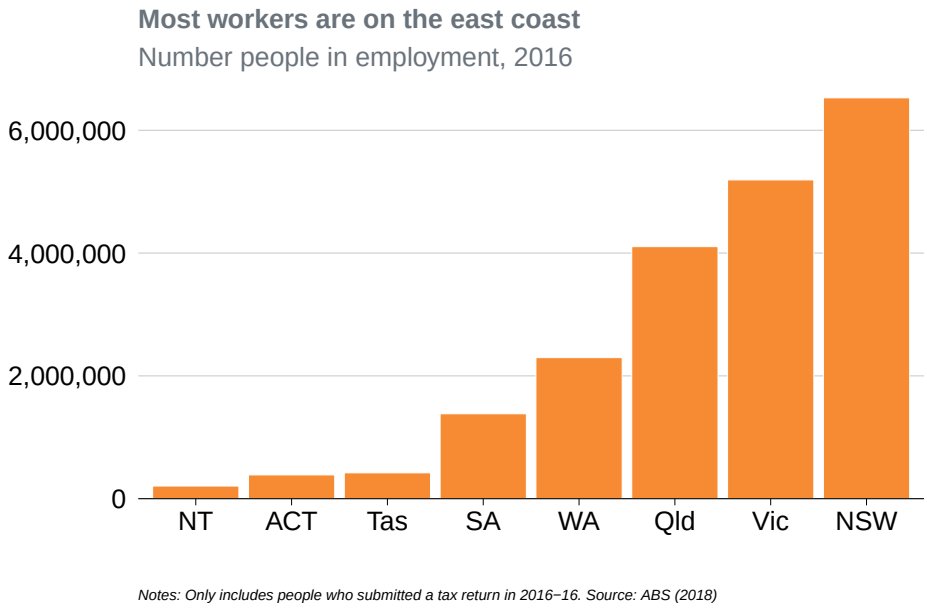
To order the states by number of workers, you can tell the `x` aesthetic that you want to `reorder` the `state` variable by `workers`:

```
data %>%
  ggplot(aes(x = reorder(state, workers), # reorder states by workers
             y = workers)) +
  geom_col() +
  theme_grattan() +
  grattan_y_continuous(labels = comma)
```



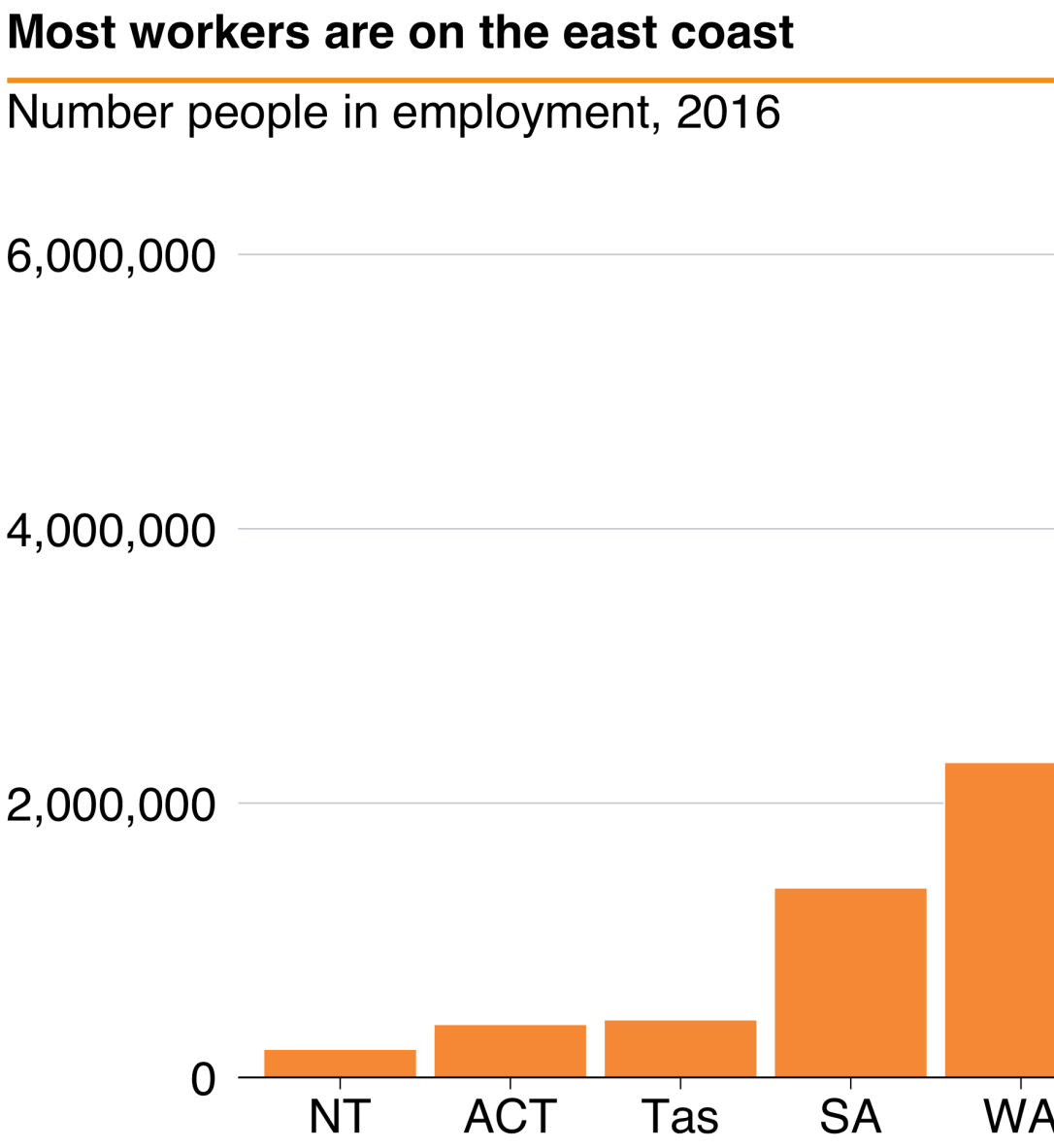
You can probably drop the x-axis label – people will understand that they're states without you explicitly saying it – and add a title and subtitle with `labs`:

```
simple_bar <- data %>%
  ggplot(aes(x = reorder(state, workers),
             y = workers)) +
  geom_col() +
  theme_grattan() +
  grattan_y_continuous(labels = comma) +
  labs(title = "Most workers are on the east coast",
       subtitle = "Number people in employment, 2016",
       x = "",
       caption = "Notes: Only includes people who submitted a tax return in 2016-16. So

simple_bar
```



Looks wonderful! Now you can export as a full-slide Grattan chart using `grattan_save`:

```
grattan_save("atlas/simple_bar.pdf", simple_bar, type = "fullslide")
```



Notes: Only includes people who submitted a tax return in 2016-16.
Source: ABS (2018)

13.2.2 Bar plot with multiple series

This section will create a horizontal bar plot showing average income by state and gender in 2016:

First create the dataset you want to plot, getting the average income by state and gender in the year 2016:

```
data <- sa3_income %>%
  filter(year == 2016) %>%
  group_by(state, gender) %>%
  summarise(average_income = sum(total_income) / sum(workers))

data
```

```
## # A tibble: 16 x 3
## # Groups:   state [8]
##    state gender average_income
##    <chr> <chr>           <dbl>
##  1 ACT   Men            78141.
##  2 ACT   Women          65548.
##  3 NSW   Men            69750.
##  4 NSW   Women          53191.
##  5 NT    Men            75246.
##  6 NT    Women          58527.
##  7 Qld   Men            65108.
##  8 Qld   Women          48458.
##  9 SA    Men            60244.
## 10 SA    Women          47533.
## 11 Tas   Men            56345.
## 12 Tas   Women          45158.
## 13 Vic   Men            64908.
## 14 Vic   Women          49264.
## 15 WA    Men            76677.
## 16 WA    Women          51578.
```

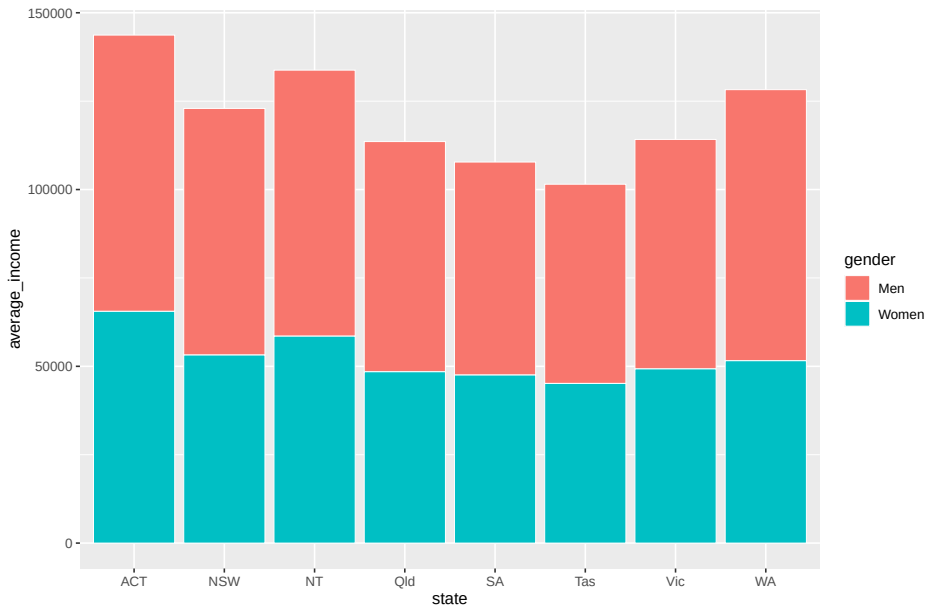
Looks too good to be true: you have one observation (row) for each state × gender group you want to plot, and a value for their average income. Put `state` on the x-axis, `average_income` on the y-axis, and split gender by fill-colour (`fill`).

Pass the data to `ggplot`, give it the appropriate `x` and `y` aesthetics, along with `fill` (the fill colour[3]) representing `gender`. And because you have the *actual* values for `average_income` you want to plot, use `geom_col`:[4]

[3]The aesthetic `fill` represents the 'fill' colour – the colour that fills the bars in your chart. The `colour` aesthetic controls the colours of the *lines*.

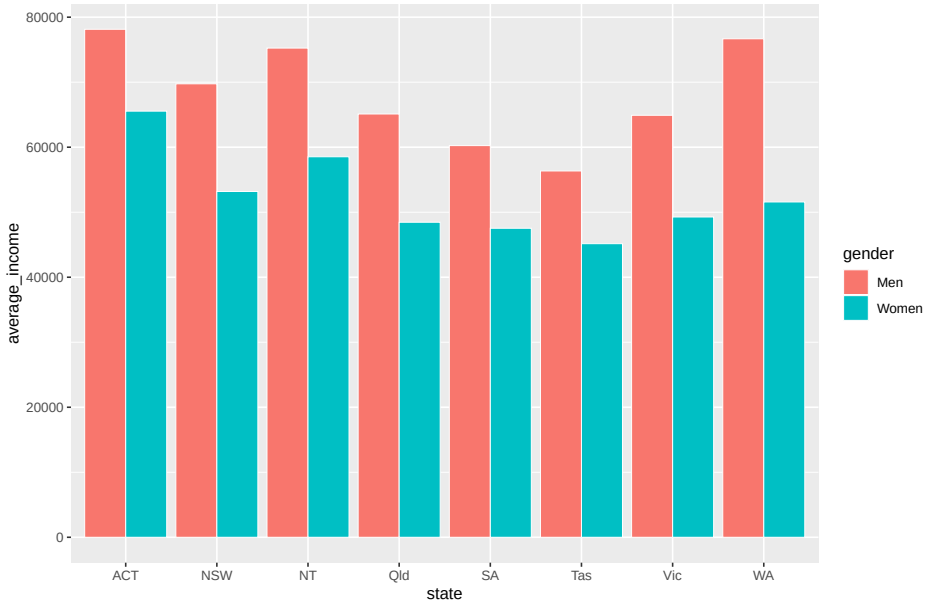
[4]`geom_col` is shorthand for `geom_bar(stat = "identity"`

```
data %>%
  ggplot(aes(x = state,
             y = average_income,
             fill = gender)) +
  geom_col()
```



The two series – women and men – created by `fill` are stacked on-top of each other by `geom_col`. You can tell it to plot them next to each other – to 'dodge' – instead with the `position` argument *within* `geom_col`:

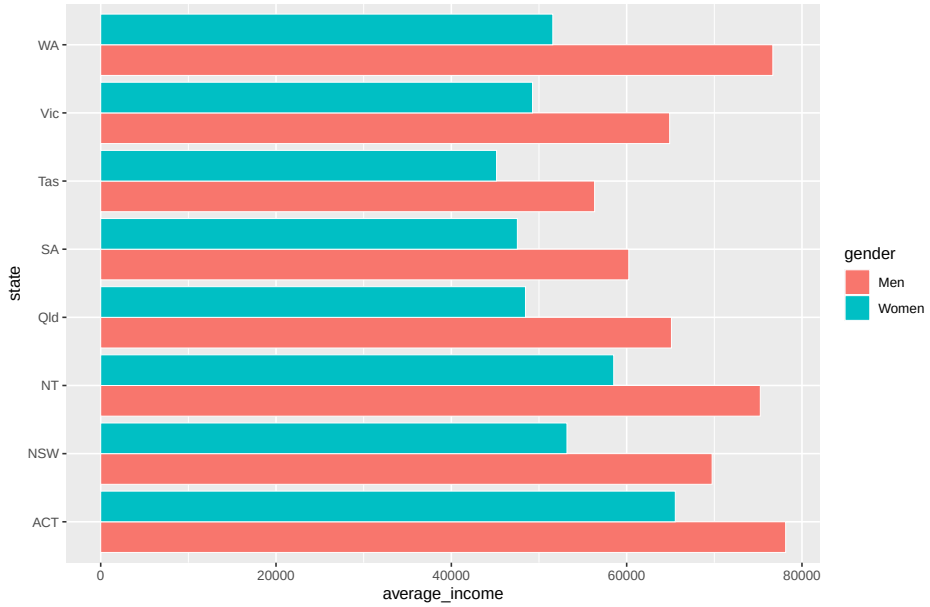
```
data %>%
  ggplot(aes(x = state,
             y = average_income,
             fill = gender)) +
  geom_col(position = "dodge") # 'dodge' the series
```



To flip the chart – a useful move when you have long labels – add `coord_flip` (ie 'flip the x and y coordinates of the chart').

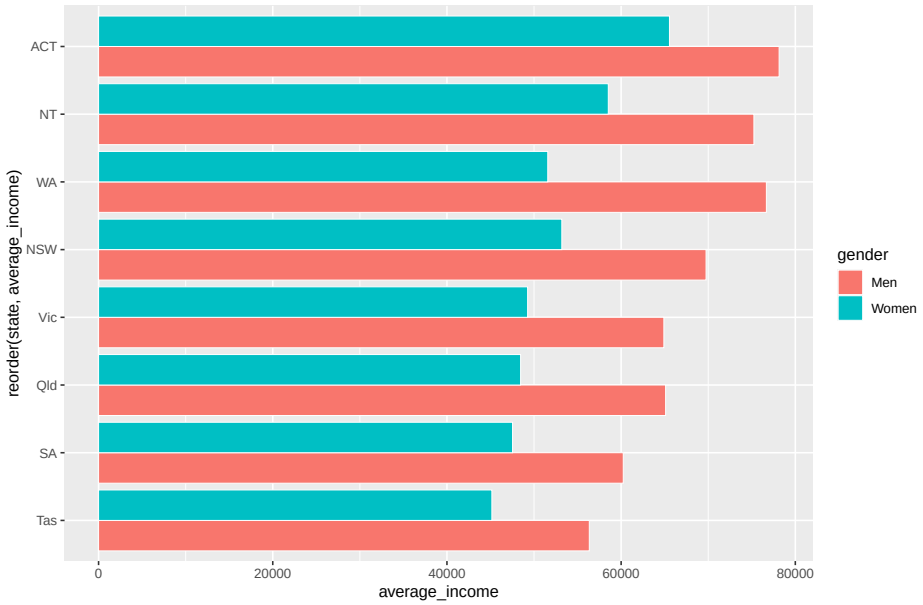
However, while the *coordinates* have been flipped, the underlying data hasn't. If you want to refer to the `average_income` axis, which now lies horizontally, you would still refer to the `y` axis (eg `grattan_y_continuous` still refers to your `y` aesthetic, `average_income`).

```
data %>%
  ggplot(aes(x = state,
             y = average_income,
             fill = gender)) +
  geom_col(position = "dodge") +
  coord_flip() # rotate the chart
```



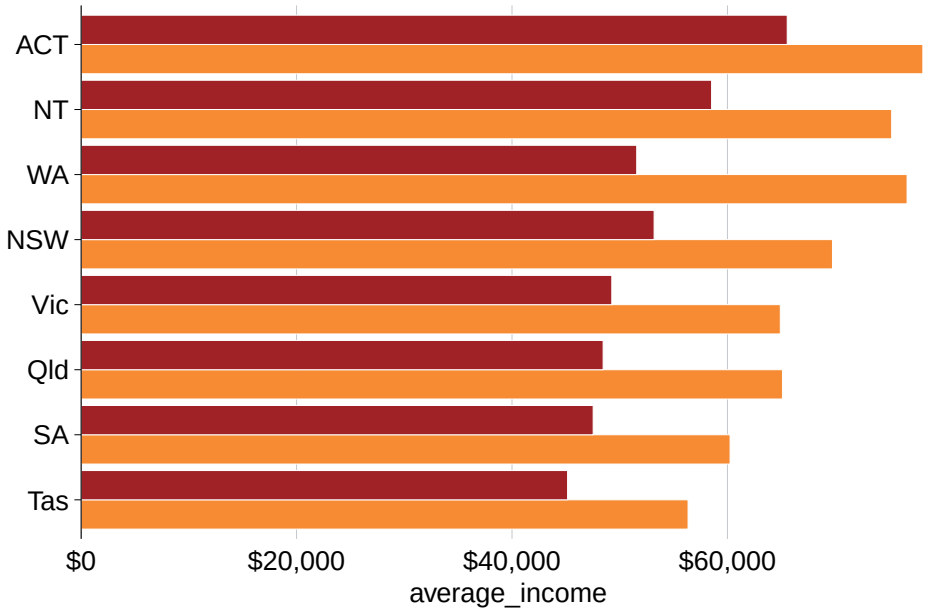
And reorder the states by average income, so that the state with the highest (combined) average income is at the top. This is done with the `reorder(var_to_reorder, var_to_reorder_by)` function when you define the `state` aesthetic:

```
data %>%
  ggplot(aes(x = reorder(state, average_income), # reorder
             y = average_income,
             fill = gender)) +
  geom_col(position = "dodge") +
  coord_flip()
```



Wonderful – that's how you want our *data* to look. Now you can Grattanise it. Note that `theme_grattan` needs to know that the coordinates were flipped so it can apply the right settings. Also tell `grattan_fill_manual` that there are two fill series.

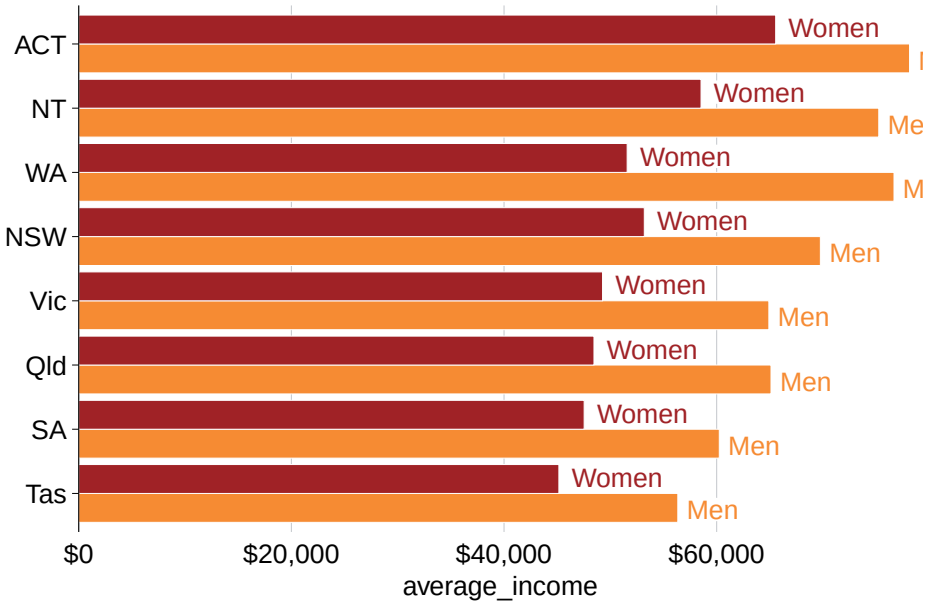
```
data %>%
  ggplot(aes(x = reorder(state, average_income),
             y = average_income,
             fill = gender)) +
  geom_col(position = "dodge") +
  coord_flip() +
  theme_grattan(flipped = TRUE) + # grattan theme
  grattan_y_continuous(labels = dollar) + # y axis
  grattan_fill_manual(2) # grattan fill colours
```



You can use `grattan_label` to **label your charts** in the Grattan style. This function is a 'wrapper' around `geom_label` that has settings that we tend to like: white background with a thin margin, 18-point font, and no border. It takes the standard arguments of `geom_label`.

Section 12.6 shows how labels are treated like data points: they need to know where to go (`x` and `y`) and what to show (`label`). But if you provide *every point* to your labelling `geom`, it will plot every label:

```
data %>%
  ggplot(aes(x = reorder(state, average_income),
             y = average_income,
             fill = gender)) +
  geom_col(position = "dodge") +
  coord_flip() +
  theme_grattan(flipped = TRUE) +
  grattan_y_continuous(labels = dollar) +
  grattan_fill_manual(2) +
  grattan_label(aes(colour = gender,  # colour the text according to gender
                    label = gender),  # label the text according to gender
            position = position_dodge(width = 1),  # position dodge with width 1
            hjust = -0.1) +  # horizontally align the label so its outside the bar
  grattan_colour_manual(2)   # define colour as two grattan colours
```



To just label *one* of the plots – ie the first one, ACT in this case – we need to tell `grattan_label`. The easiest way to do this is by **creating a label dataset beforehand**, like `label_gender` below. This just includes the observations you want to label:

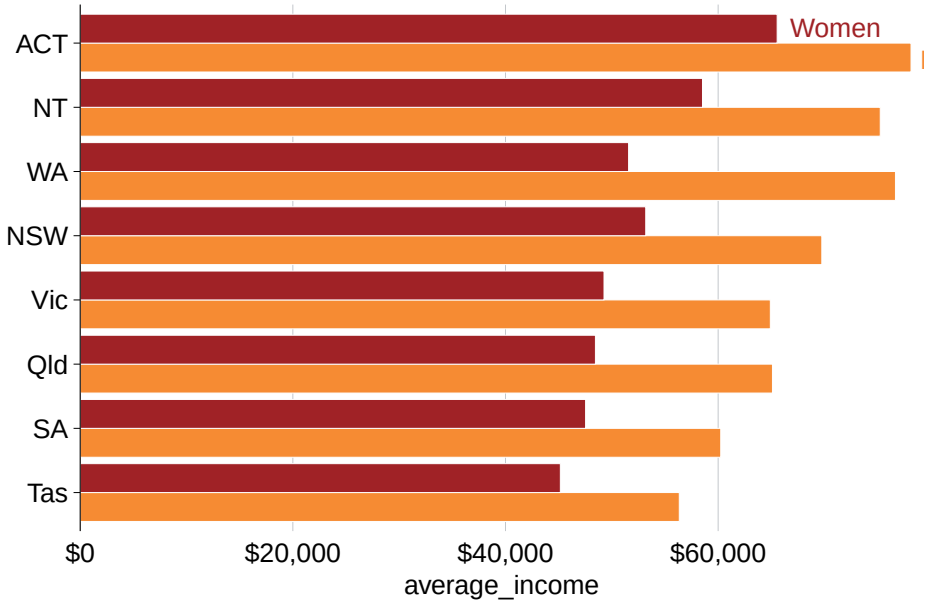
```
label_gender <- data %>%
  filter(state == "ACT")  # just want Tasmania observations

label_gender
```

```
## # A tibble: 2 x 3
## # Groups:   state [1]
##   state gender average_income
##   <chr> <chr>           <dbl>
## 1 ACT   Men            78141.
## 2 ACT   Women          65548.
```

So you can pass that `label_gender` dataset to `grattan_label`:

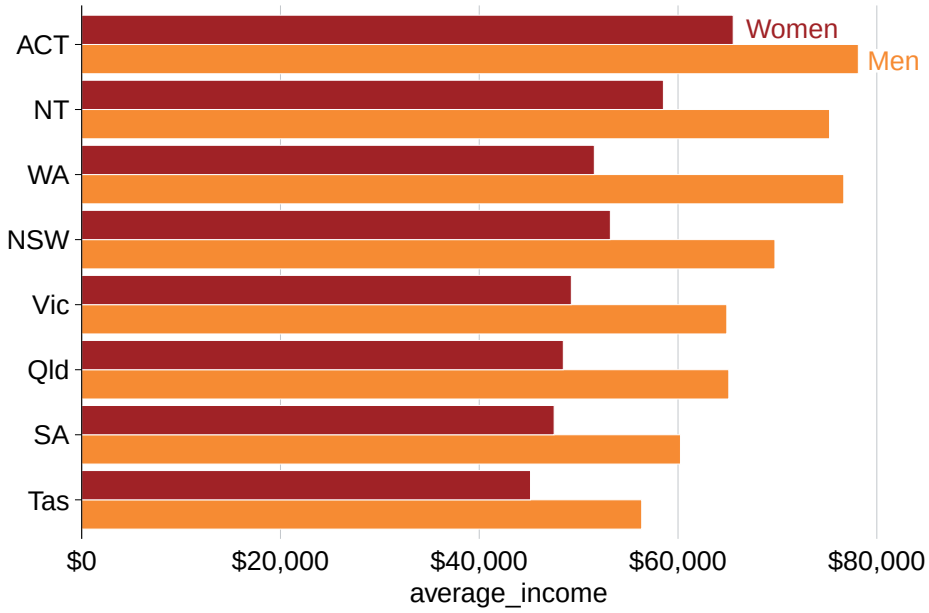
```
data %>%
  ggplot(aes(x = reorder(state, average_income),
             y = average_income,
             fill = gender)) +
  geom_col(position = "dodge") +
  coord_flip() +
  theme_grattan(flipped = TRUE) +
  grattan_y_continuous(labels = dollar) +
  grattan_fill_manual(2) +
```

```
  grattan_label(data = label_gender,  # supply the new dataset
                aes(colour = gender,
                    label = gender),
                position = position_dodge(width = 1),
                hjust = -0.1) +
  grattan_colour_manual(2)
```



Almost there! The labels go out of range a little bit, and we can fix this by expanding the plot:

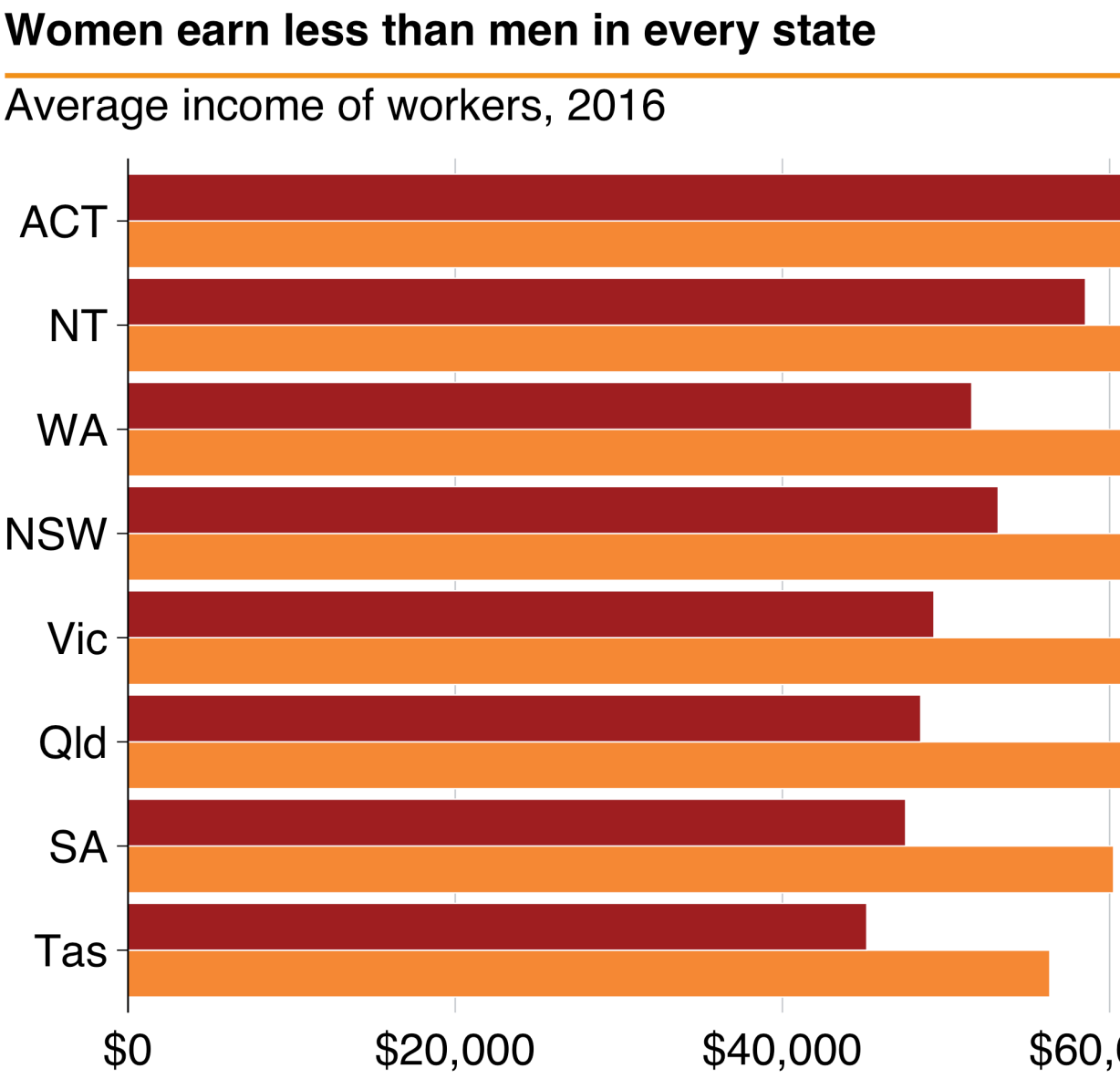
```
data %>%
  ggplot(aes(x = reorder(state, average_income),
             y = average_income,
             fill = gender)) +
  geom_col(position = "dodge") +
  coord_flip() +
  theme_grattan(flipped = TRUE) +
  grattan_y_continuous(labels = dollar,
                       expand_top = .1) + # expand the plot
  grattan_fill_manual(2) +
  grattan_label(data = label_gender,
                aes(colour = gender,
                    label = gender),
                position = position_dodge(width = 1),
                hjust = -0.1) +
  grattan_colour_manual(2)
```



Looks tip-top! Now you can add titles and a caption, and save using `grattan_save`:

```
multiple_bar <- data %>%
  ggplot(aes(x = reorder(state, average_income),
             y = average_income,
             fill = gender)) +
  geom_col(position = "dodge") +
  coord_flip() +
  theme_grattan(flipped = TRUE) +
  grattan_y_continuous(labels = dollar,
                       expand_top = .1) +
  grattan_fill_manual(2) +
  grattan_label(data = label_gender,
                aes(colour = gender,
                    label = gender),
                position = position_dodge(width = 1),
                hjust = -0.1) +
  grattan_colour_manual(2) +
  labs(title = "Women earn less than men in every state",
       subtitle = "Average income of workers, 2016",
       x = "",
       y = "",
       caption = "Notes: Only includes people who submitted a tax return in 2016-16. Source: ABS
```

```
grattan_save("atlas/multiple_bar.pdf", multiple_bar, type = "fullslide")
```



Notes: Only includes people who submitted a tax return in 2016-16.
Source: ABS (2018)

13.2.3 Facetted bar charts

'Facetting' a chart means you create a separate plot for each group. It's particularly useful in showing differences between more than one group. The chart you'll make in this section will show annual income by gender and state, *and* by professional and non-professional workers:

Start by creating the dataset you want to plot:

```
data <- sa3_income %>%
  group_by(state, gender, prof) %>%
  summarise(average_income = sum(total_income) / sum(workers))

data
```

```
## # A tibble: 32 x 4
## # Groups:   state, gender [16]
##    state gender prof             average_income
##    <chr> <chr>  <chr>                     <dbl>
##  1 ACT   Men    Non-professional         52545.
##  2 ACT   Men    Professional             96488.
##  3 ACT   Women  Non-professional         46151.
##  4 ACT   Women  Professional             79828.
##  5 NSW   Men    Non-professional         49182.
##  6 NSW   Men    Professional             91624.
##  7 NSW   Women  Non-professional         36772.
##  8 NSW   Women  Professional             68445.
##  9 NT    Men    Non-professional         58844.
## 10 NT    Men    Professional             87666.
## # ... with 22 more rows
```

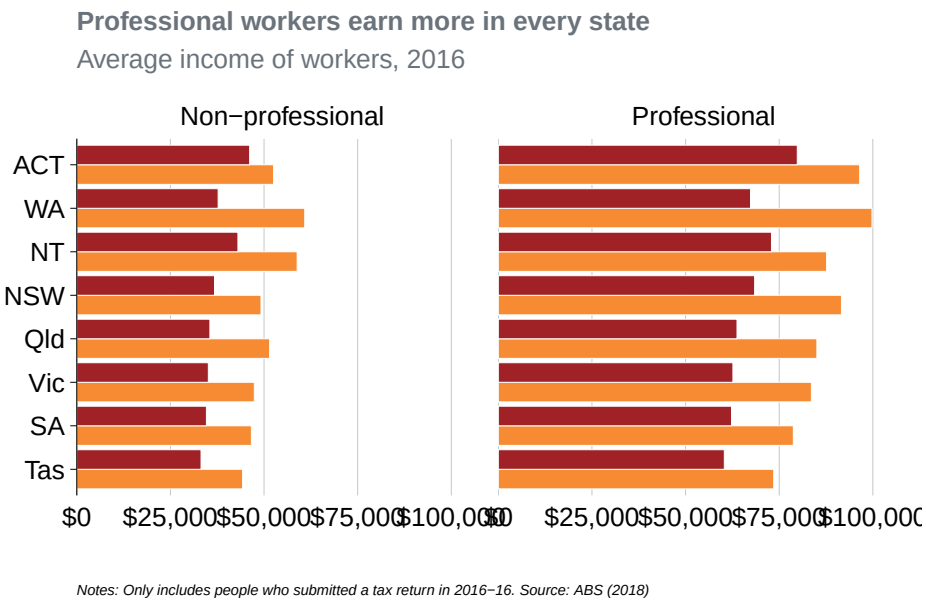
Then plot a bar chart with `geom_col` and `theme_grattan` elements, using a similar chain to the final plot of 13.2.2 (without the labelling). We'll build on this chart:

```
facet_bar <- data %>%
  ggplot(aes(x = reorder(state, average_income),
             y = average_income,
             fill = gender)) +
  geom_col(position = "dodge") +
  coord_flip() +
  theme_grattan(flipped = TRUE) +
  grattan_y_continuous(labels = dollar,
                       expand_top = .1) +
  grattan_fill_manual(2) +
  grattan_colour_manual(2) +
  labs(title = "Professional workers earn more in every state",
       subtitle = "Average income of workers, 2016",
```

```
       x = "",
       y = "",
       caption = "Notes: Only includes people who submitted a tax return in 2016-16. Source: ABS
```

You can 'facet' bar charts – and any other chart type – with the `facet_grid` or `facet_wrap` commands. The latter tends to give you more control over label placement, so let's start with that. `fadcet_wrap` asks the questions: "what variables should I create separete charts for", and "how should I place them on the page"? Tell it to use the `prof` variable with the `vars()` function.[5]

```
facet_bar +
  facet_wrap(vars(prof))
```



That's good! It does what it should. Now you just need to tidy it up a little bit by adding labels and avoiding clashes along the bottom axis.

Create labels in the same way you have done before: you only want to label one 'women' and 'men' series, so create a dataset that contains only that information:

```
label_data <- data %>%
  filter(state == "ACT",
         prof == "Non-professional")

label_data
```

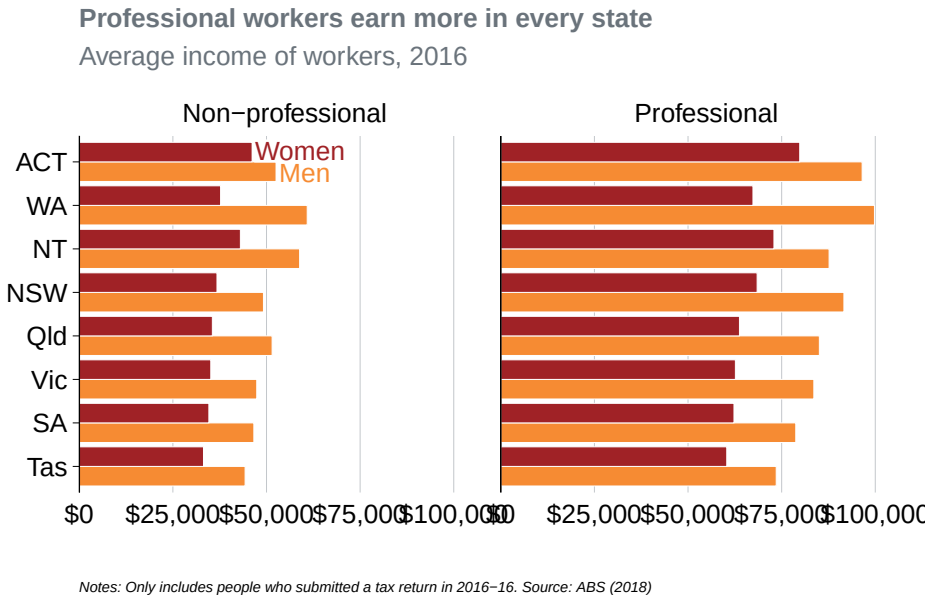
[5]The `vars()` function is sometimes used in the `tidyverse` to specifically say "I am using a variable name here". You can't use variable names directly because of legacy issues. You can learn more about it in the official documentation.

```
## # A tibble: 2 x 4
## # Groups:   state, gender [2]
##   state gender prof             average_income
##   <chr> <chr>  <chr>                     <dbl>
## 1 ACT   Men    Non-professional         52545.
## 2 ACT   Women  Non-professional         46151.
```

Good – now add that to the plot with `grattan_label`, supplying the required aesthetics and position. And use `hjust = 0` to tell the labels to be left-aligned.

To give each plot a black base axis, you can add `geom_hline()` with `yintercept = 0`.

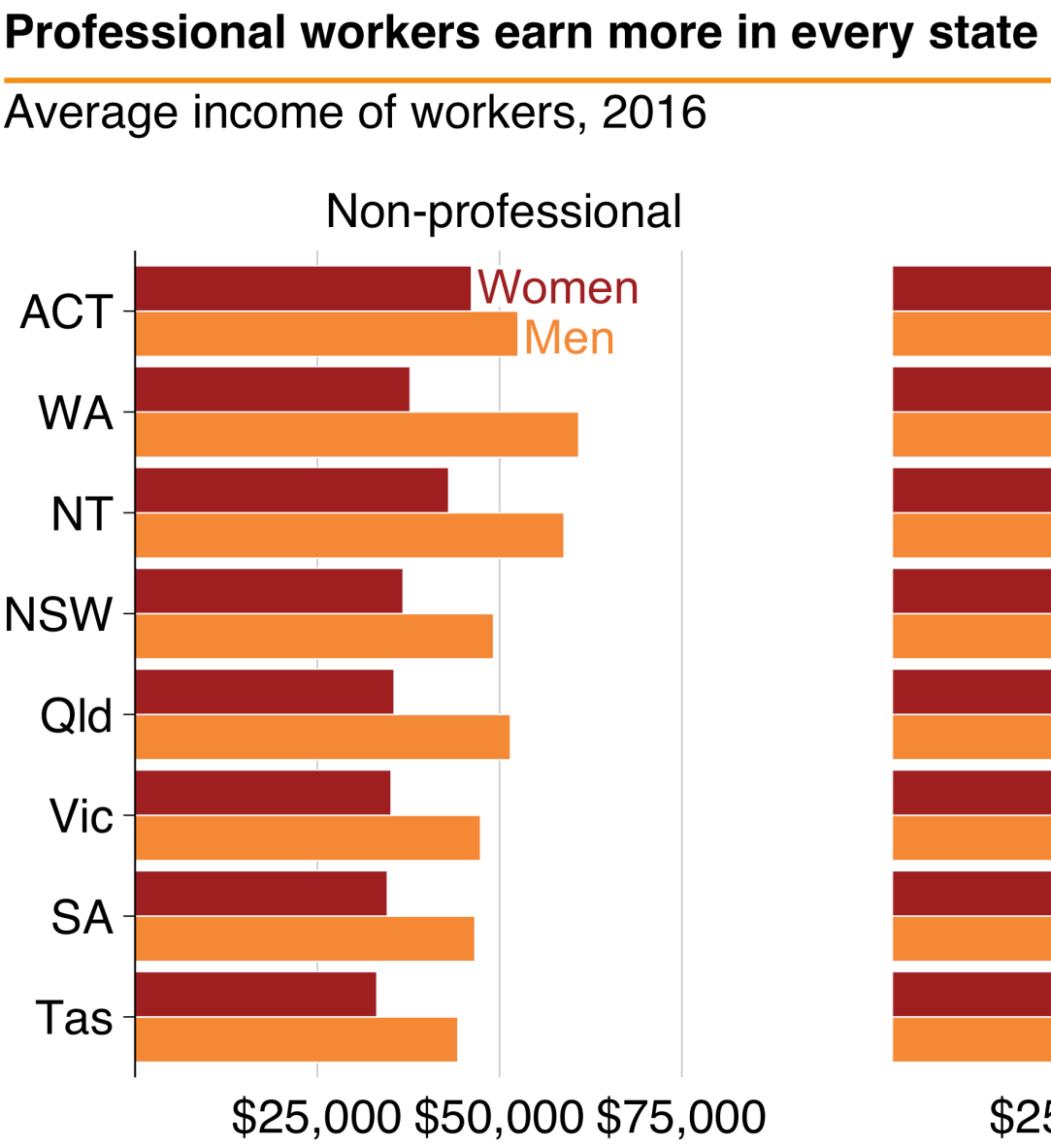
```
facet_bar +
  facet_wrap(vars(prof)) +
  geom_hline(yintercept = 0) +  # add black line
  grattan_label(data = label_data, # supply label data
                aes(label = gender,
                    colour = gender),
                position = position_dodge(width = 1),
                hjust = 0)
```



Sublime! But the "$0" and "$100,000" labels are clashing along the horizontal axis. To tidy these up, we redefine the `breaks` – the points that will be labelled – to 25,000, 50,000 and 75,000 inside `grattan_y_continuous`. Putting everything together and saving the plot as a fullslide chart with `grattan_save`:

```
# Create label data
label_data <- data %>%
  filter(state == "ACT",
         prof == "Non-professional")

# Create plot
facet_bar <- data %>%
  ggplot(aes(x = reorder(state, average_income),
             y = average_income,
             fill = gender)) +
  geom_col(position = "dodge") +
  coord_flip() +
  theme_grattan(flipped = TRUE) +
  grattan_y_continuous(labels = dollar,
                       breaks = c(25e3, 50e3, 75e3)) +  # change breaks
  grattan_fill_manual(2) +
  grattan_colour_manual(2) +
  labs(title = "Professional workers earn more in every state",
       subtitle = "Average income of workers, 2016",
       x = "",
       y = "",
       caption = "Notes: Only includes people who submitted a tax return in 2016-16. Source: ABS
  facet_wrap(vars(prof)) +
  grattan_label(data = label_data,
                aes(label = gender,
                    colour = gender),
                position = position_dodge(width = 1),
                hjust = 0)
```

```
grattan_save("atlas/facet_bar.pdf", facet_bar, type = "fullslide")
```



Notes: Only includes people who submitted a tax return in 2016-16.
Source: ABS (2018)

13.3 Line charts

A line chart has one key aesthetic: `group`. This tells `ggplot` how to connect individual lines.

13.3.1 Simple line chart

The first line chart shows the number of workers in Australia between 2011 and 2016:

13.3.2 Line chart with multiple series

This line chart will show how **real** average income has changed for each state over the past five years:

First, take the `sa3_income` dataset and create a summary table average income by year and state. Ignore the territories for this chart.

```
data <- sa3_income %>%
  filter(!state %in% c("ACT", "NT")) %>%
  group_by(year, state) %>%
  summarise(average_income = sum(total_income) / sum(workers))

head(data)
```

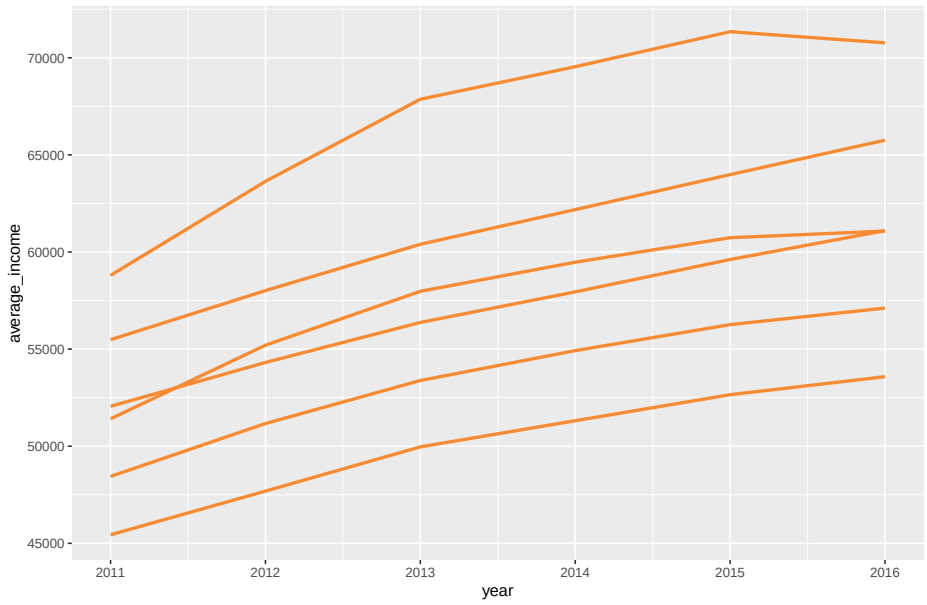
```
## # A tibble: 6 x 3
## # Groups:   year [1]
##    year state average_income
##   <dbl> <chr>          <dbl>
## 1  2011 NSW           55483.
## 2  2011 Qld           51408.
## 3  2011 SA            48443.
## 4  2011 Tas           45439.
## 5  2011 Vic           52053.
## 6  2011 WA            58795.
```

The income data presented is nominal, so you'll need to inflate to 'real' dollars using the 'cpi_inflate

Plot a line chart by taking the `data`, passing it to `ggplot` with *aes*thetics, then using `geom_line`:

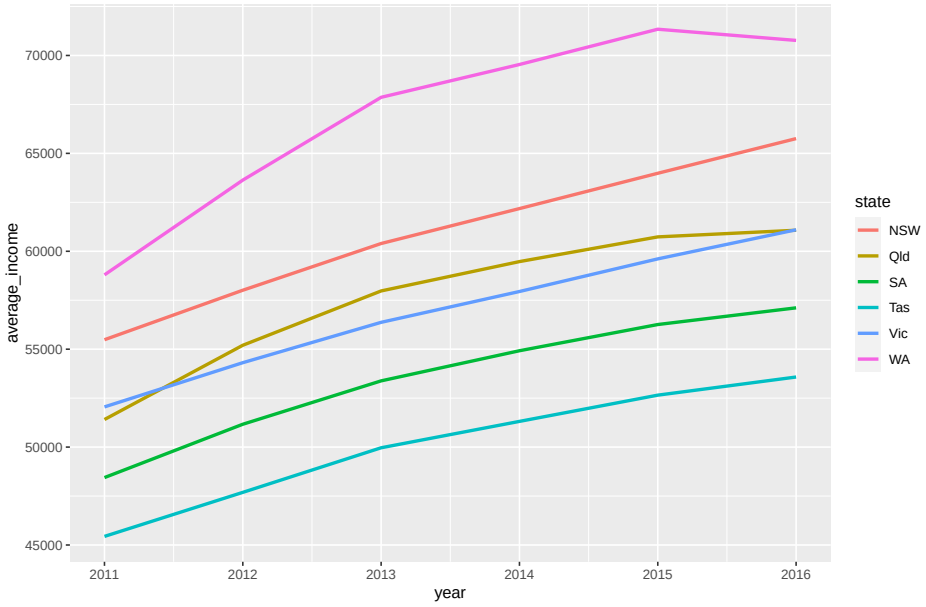
```
data %>%
  ggplot(aes(x = year,
             y = average_income,
```

```
            group = state)) +
  geom_line()
```



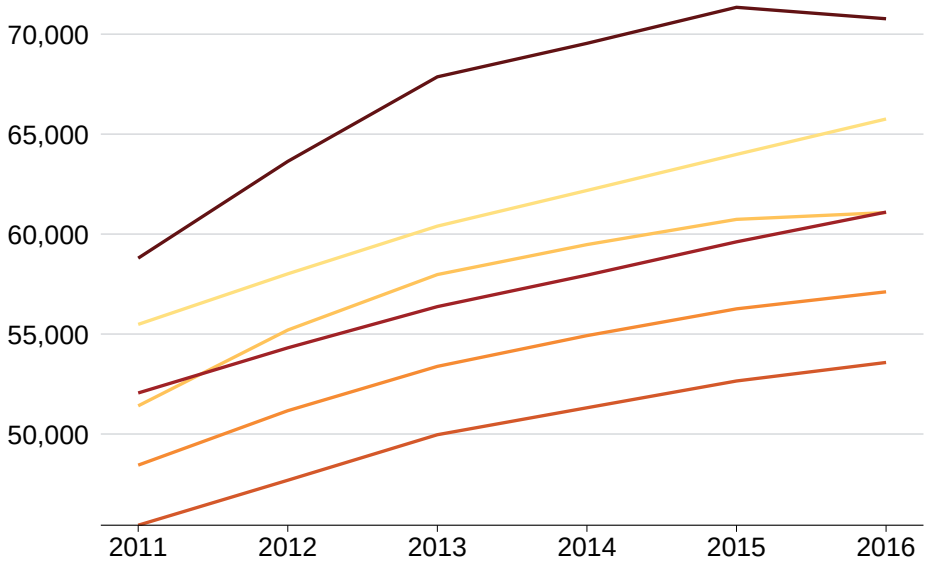
Now you can represent each `state` by colour:

```
data %>%
  ggplot(aes(x = year,
             y = average_income,
             group = state,
             colour = state)) +
  geom_line()
```



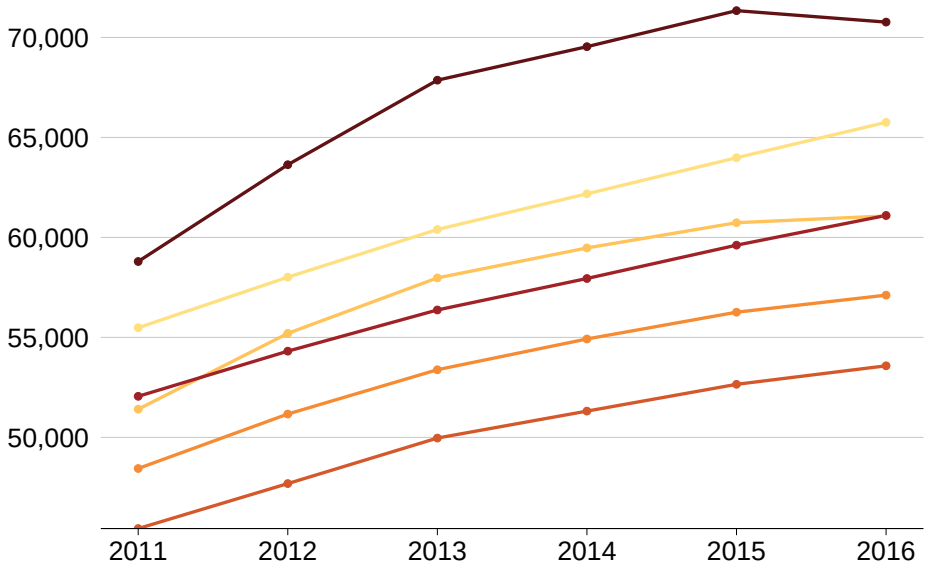
Cooler! Adding some Grattan formatting to it and define it as our 'base chart':

```
base_chart <-data %>%
  ggplot(aes(x = year,
             y = average_income,
             group = state,
             colour = state)) +
  geom_line() +
  theme_grattan() +
  grattan_y_continuous(labels = comma) +
  grattan_colour_manual(6) +
  labs(x = "",
       y = "")

base_chart
```



You can add 'dots' for each year by layering `geom_point` on top of `geom_line`:

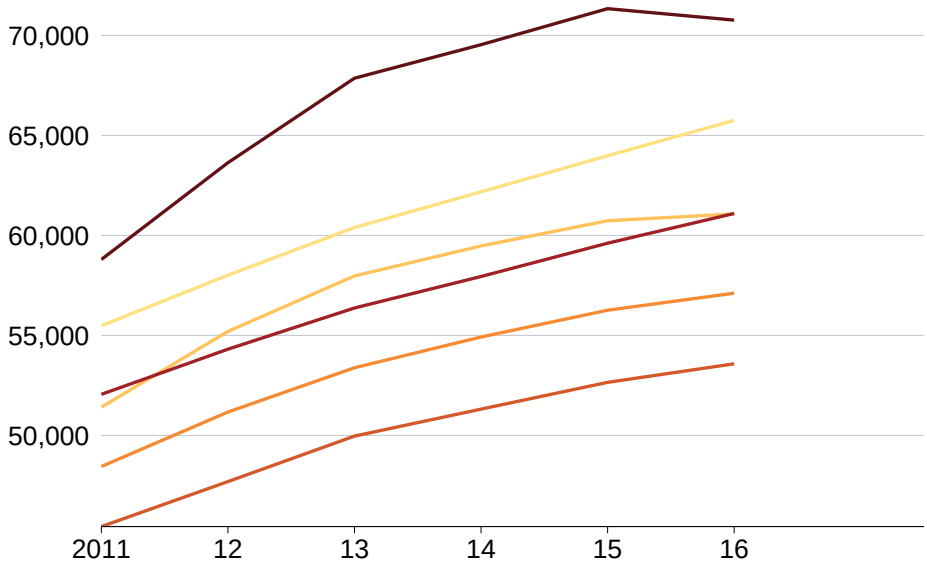
```
base_chart +
  geom_point()
```



To add labels to the end of each line, you would expand the x-axis to make room for labels and add reasonable breaks:

```
base_chart +
  grattan_x_continuous(expand_right = .3,
                       breaks = seq(2011, 2016, 1),
```

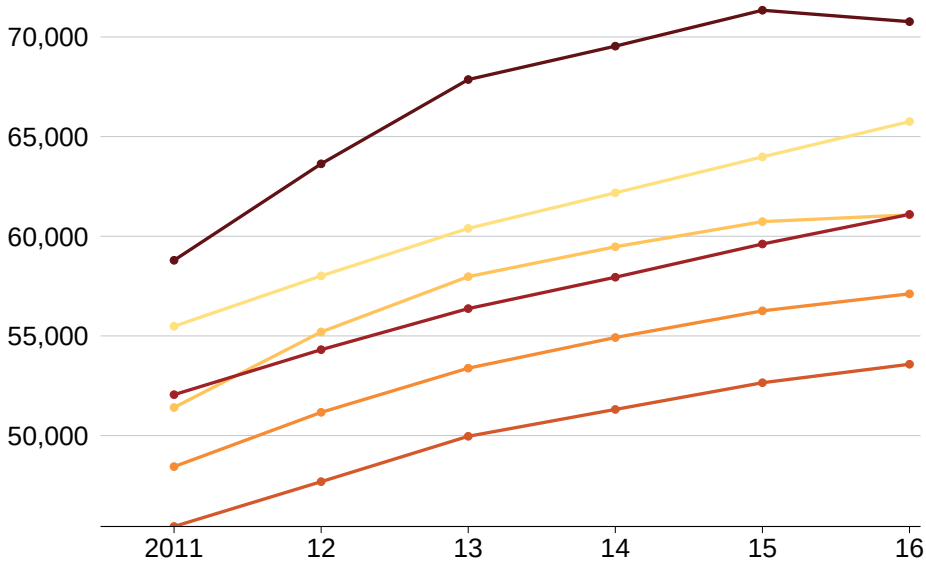
```
                        labels = c("2011", "12", "13", "14", "15", "16"))
```



Then add labels, using

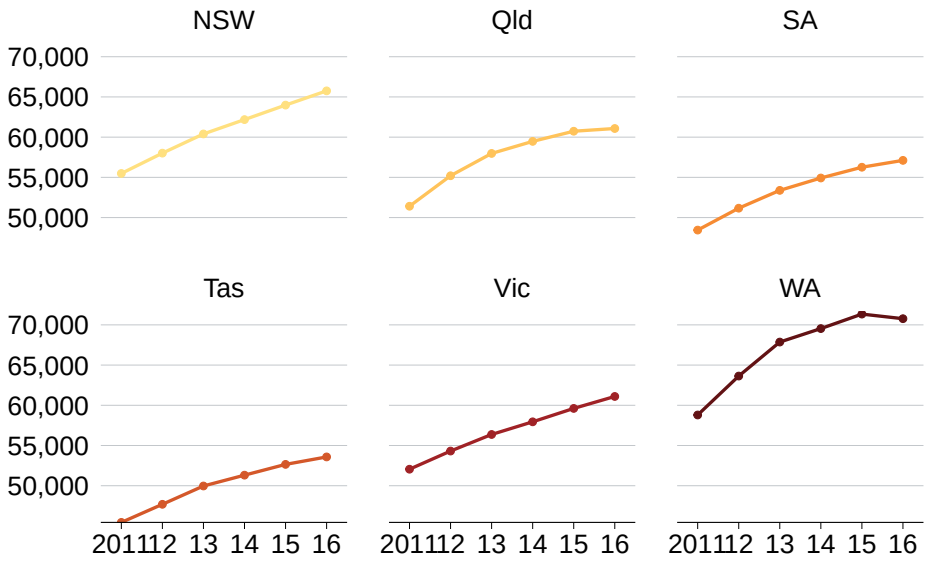
```
label_line <- data %>%
  filter(year == 2010)

base_chart +
  geom_point() +
  grattan_x_continuous(expand_left = .1,
                       breaks = seq(2011, 2016, 1),
                       labels = c("2011", "12", "13", "14", "15", "16")) +
  grattan_label(data = label_line,
                aes(label = state),
                nudge_x = -Inf,
                segment.colour = NA)
```

```
## Warning: Ignoring unknown parameters: segment.colour
```



If you wanted to show each state individually, you could **facet** your chart so that a separate plot was produced for each state:

```
base_chart +
  geom_point() +
    grattan_x_continuous(expand_left = .1,
                         expand_right = .1,
                         breaks = seq(2011, 2016, 1),
                         labels = c("2011", "12", "13", "14", "15", "16")) +
  theme(panel.spacing.x = unit(10, "mm")) +
  facet_wrap(state ~ .)
```



13.4 Scatter plots

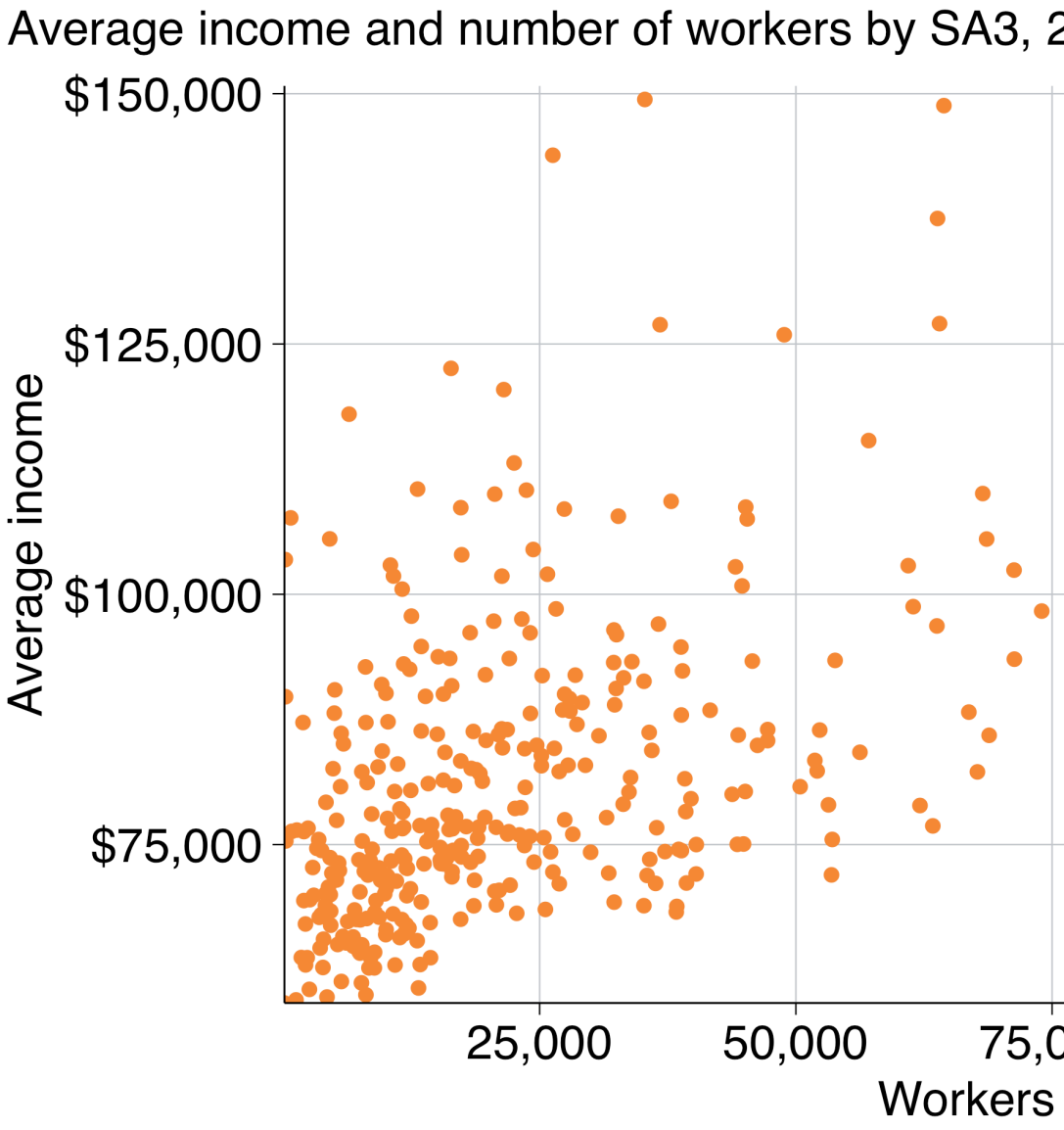
Scatter plots require `x` and `y` aesthetics. These can then be coloured and faceted.

13.4.1 Simple scatter plot

The first simple scatter plot will show the relationship between average incomes of professionals and the number of professional workers by area in 2016:

```
include_graphics("atlas/simple_scatter.png")
```

More workers, more income



Notes: Only includes people who submitted a tax return in 2016-16.
Source: ABS (2018)

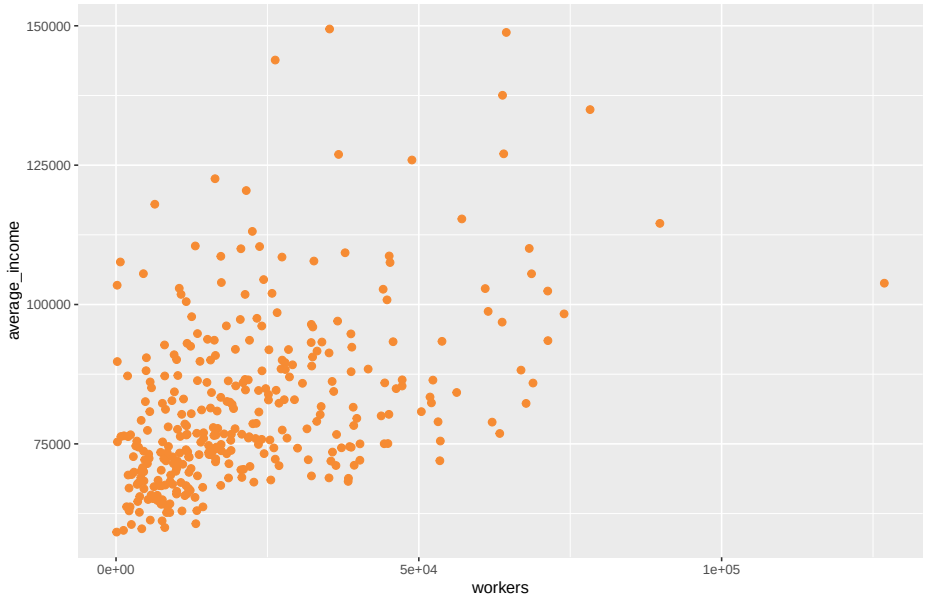
Create the dataset you want to plot:

```
data <- sa3_income %>%
  filter(year == 2016,
         prof == "Professional") %>%
  group_by(sa3_name) %>%
  summarise(workers = sum(workers),
            average_income = sum(total_income) / workers)

head(data)
```

```
## # A tibble: 6 x 3
##   sa3_name       workers average_income
##   <chr>            <dbl>          <dbl>
## 1 Adelaide City    10005         90115.
## 2 Adelaide Hills   24715         84921.
## 3 Albany           12390         70581.
## 4 Albury           16465         72305.
## 5 Alice Springs     9640         84340.
## 6 Armadale         19771         85407.
```

The dataset has one observation per SA3, and the two variables you want to plot: workers and average income. Pass the data to `ggplot`, set the aesthetics and plot with `geom_point`:

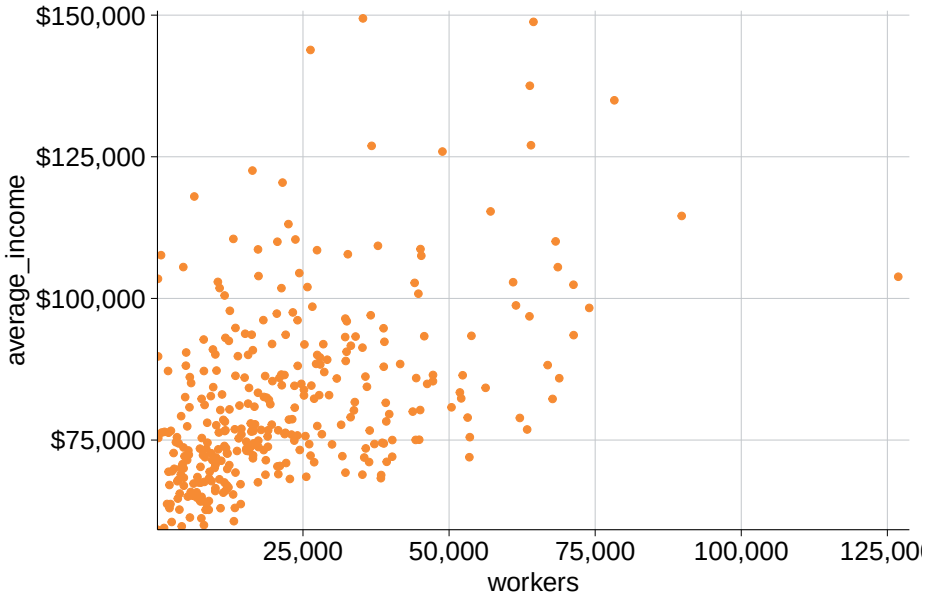
```
data %>%
  ggplot(aes(x = workers,
             y = average_income)) +
  geom_point()
```



Then add Grattan theme elements:

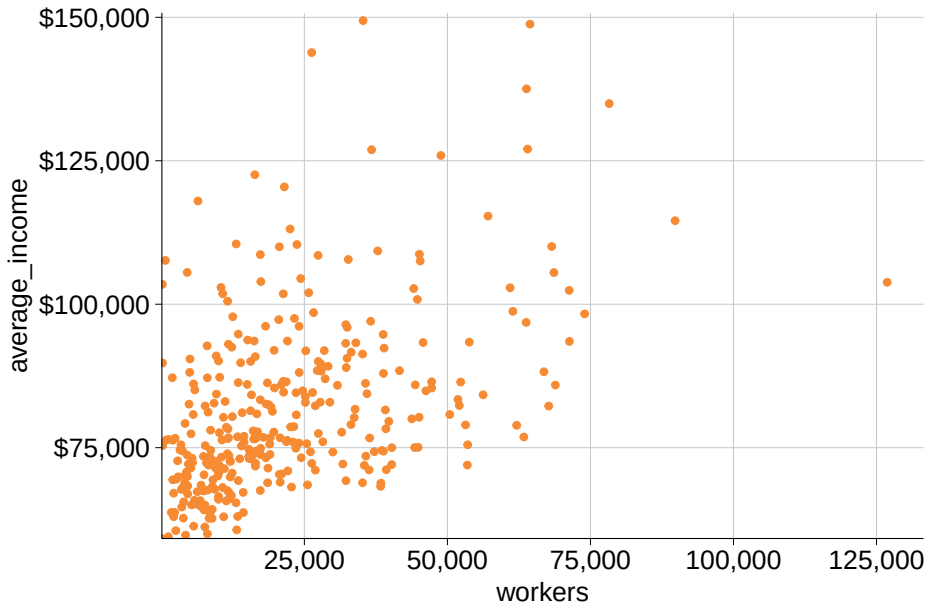
- `theme_grattan()`, telling it that the `chart_type` is a scatter plot.
- `grattan_y_continuous()`, setting the label style to `dollar`.
- `grattan_x_continuous()`, setting the label style to `comma`.

```
data %>%
  ggplot(aes(x = workers,
             y = average_income)) +
  geom_point()  +
  theme_grattan(chart_type = "scatter") +
  grattan_y_continuous(labels = dollar) +
  grattan_x_continuous(labels = comma)
```



Looks champion. The last label on the x-axis goes off the page a bit so you can expand the plot to the right in the `grattan_x_continuous` element:

```
data %>%
  ggplot(aes(x = workers,
             y = average_income)) +
  geom_point()  +
  theme_grattan(chart_type = "scatter") +
  grattan_y_continuous(labels = dollar) +
  grattan_x_continuous(labels = comma,
                       expand_right = .05) # expand the right by 5%
```

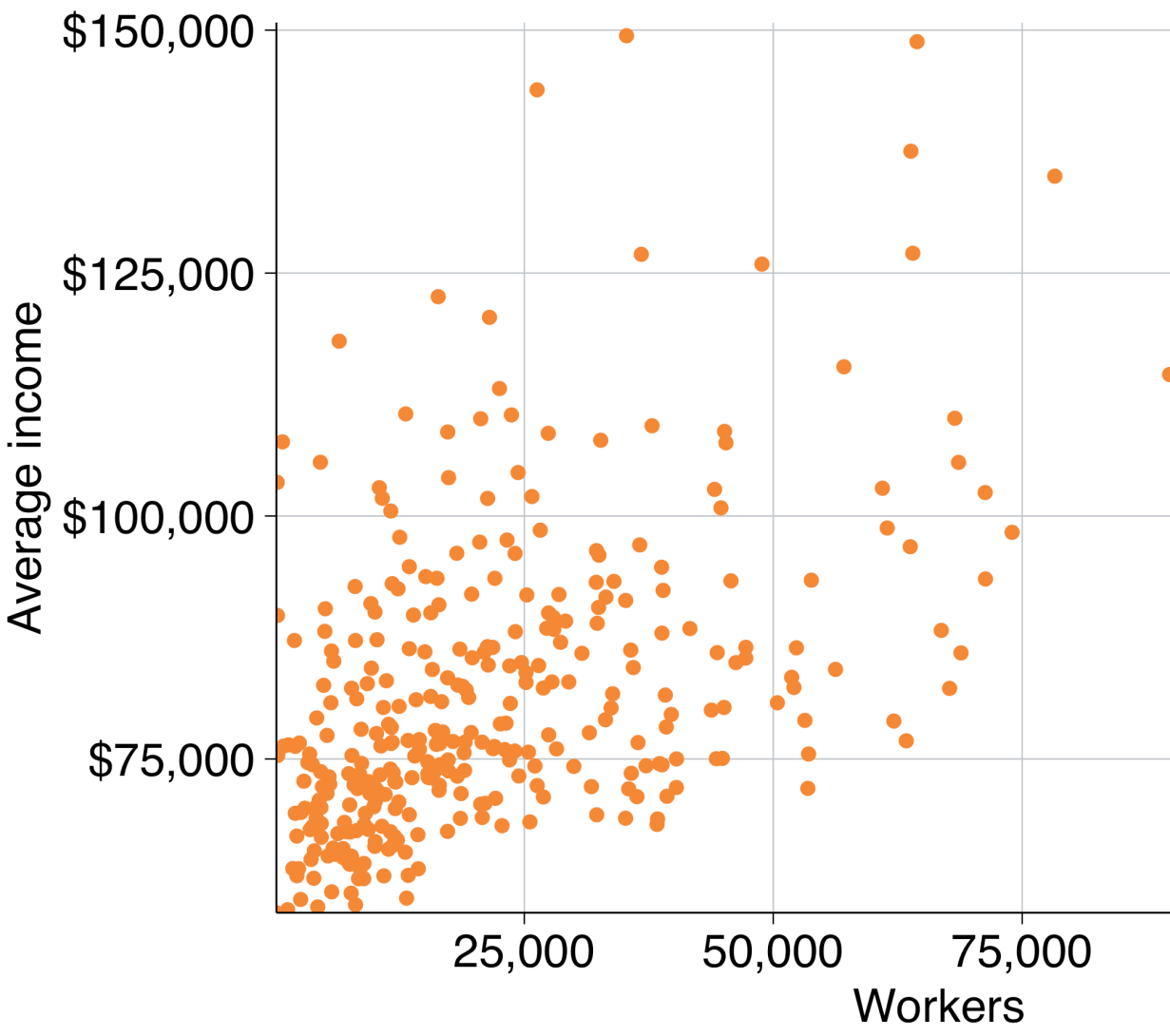


Finally, add titles and save the plot:

```
simple_scatter <- data %>%
  ggplot(aes(x = workers,
             y = average_income)) +
  geom_point()  +
  theme_grattan(chart_type = "scatter") +
  grattan_y_continuous(labels = dollar) +
  grattan_x_continuous(labels = comma,
                       expand_right = .05) +
  labs(title = "More workers, more income",
       subtitle = "Average income and number of workers by SA3, 2016",
       y = "Average income",
       x = "Workers",
       caption = "Notes: Only includes people who submitted a tax return in 2016-16. So
```

```
grattan_save("atlas/simple_scatter.pdf", simple_scatter, type = "fullslide")
```

More workers, more income



Notes: Only includes people who submitted a tax return in 2016-16.
Source: ABS (2018)

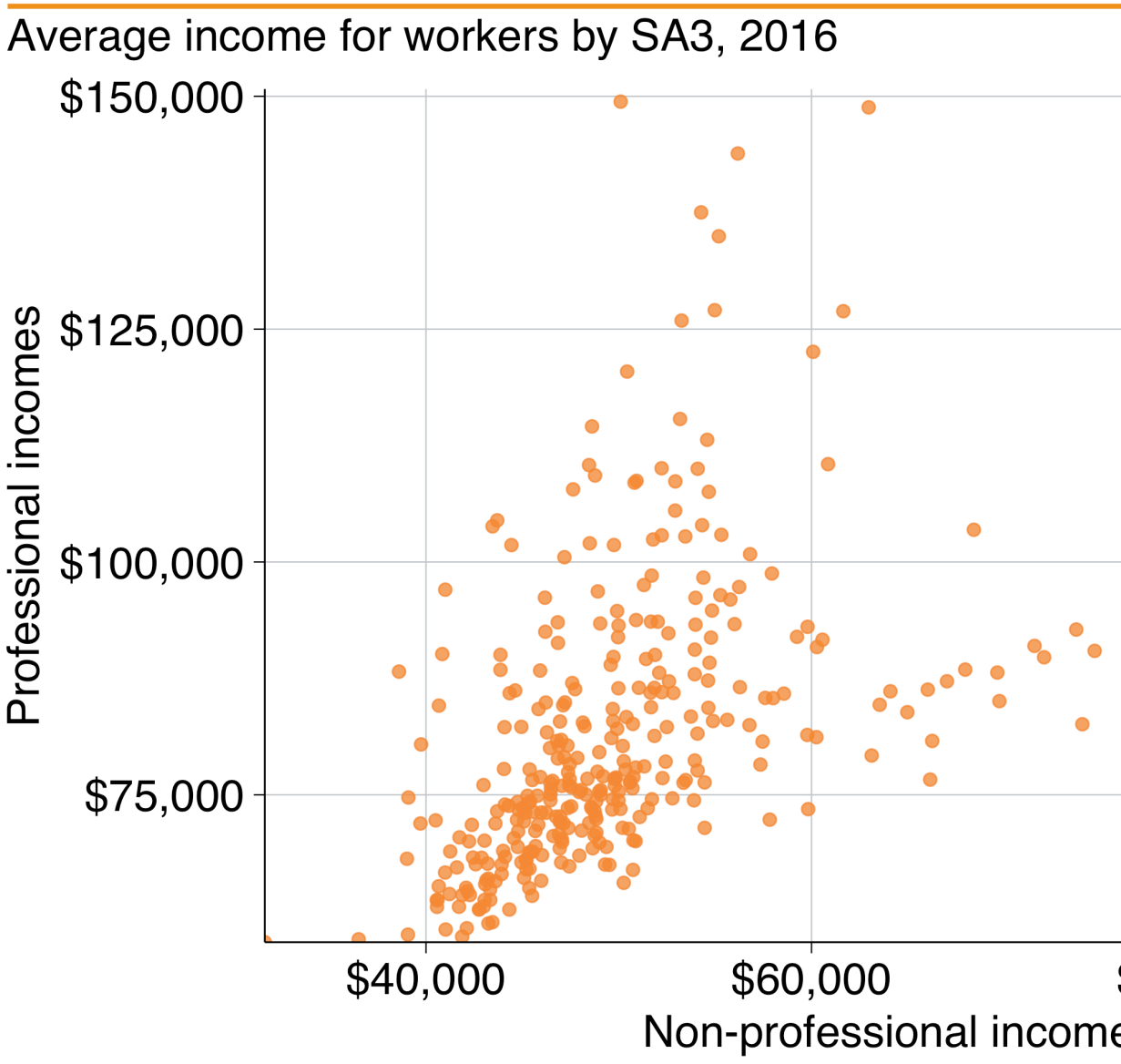
13.4.2 Scatter plot with reshaped data

The next scatter plot involves the same basic plotting principles of the chart above, but requires a bit more data manipulation before plotting.

The chart will show the wages of professional workers and non-professional workers in 2016:

```
include_graphics("atlas/scatter_reshape.png")
```

Non-professionals tend to earn more when professio do



Notes: Only includes people who submitted a tax return in 2016-16.
Source: ABS (2018)

First prepare your data. You want to find the average incomes of all professional and non-professional workers in 2016:

```
data_prep <- sa3_income %>%
  filter(year == 2016) %>%
  group_by(sa3_name, prof) %>%
  summarise(average_income = sum(total_income) / sum(workers))

head(data_prep)
```

```
## # A tibble: 6 x 3
## # Groups:   sa3_name [3]
##   sa3_name       prof             average_income
##   <chr>          <chr>                     <dbl>
## 1 Adelaide City  Non-professional         40843.
## 2 Adelaide City  Professional             90115.
## 3 Adelaide Hills Non-professional         47208.
## 4 Adelaide Hills Professional             84921.
## 5 Albany         Non-professional         46609.
## 6 Albany         Professional             70581.
```

That's good – you have the numbers you need. But think about how you're going to *plot* them using `x` and `y` aesthetics. You'll need one variable for `x = professional_income` and one variable for `y = non_professional_income`. At the moment, these are represented by different *rows*.

You can fix this by reshaping the data with the `pivot_wider` function. The three arguments you provide here are:

- `id_cols = sa3_name`: the variable `sa3_name` uniquely identifies each row in your data.
- `names_from = prof`: the variable `prof` contains the variables names for the *new* variables you are creating.
- `values_from = average_income`: the variable `average_income` contains the *values* that will fill the new variables.

After the `pivot_wider` step is complete, use `janitor::clean_names()` to convert the new `Professional` and `Non-Professional` names to `snake_case` to make them easier to use down the track:

```
data <- data_prep %>%
  pivot_wider(id_cols = sa3_name,  # variables that will stay the same
              names_from = prof,   # variables that will dictate the new names
              values_from = average_income) %>%  # these will be the values
  janitor::clean_names() # tidy up the new variable names

head(data)
```

```
## # A tibble: 6 x 3
```

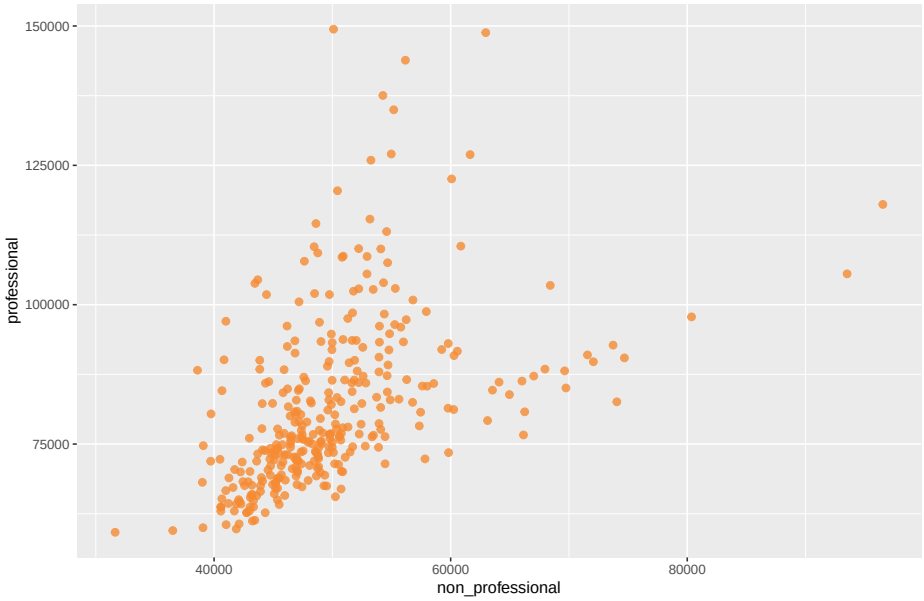
```
## # Groups:   sa3_name [6]
##   sa3_name       non_professional professional
##   <chr>                     <dbl>        <dbl>
## 1 Adelaide City            40843.       90115.
## 2 Adelaide Hills           47208.       84921.
## 3 Albany                   46609.       70581.
## 4 Albury                   44718.       72305.
## 5 Alice Springs            54647.       84340.
## 6 Armadale                 57599.       85407.
```

Getting the data in the right format for your plot – rather than 'hacking' your plot to fit your data – will save you time and effort down the line.

Now you have a dataset in the format you want to plot, you can pass it to `ggplot` and add aesthetics like you normally would.

```
data %>%
  ggplot(aes(x = non_professional,
             y = professional)) +
  geom_point(alpha = 0.8) # make the points a little transparent
```

```
## Warning: Removed 1 rows containing missing values (geom_point).
```



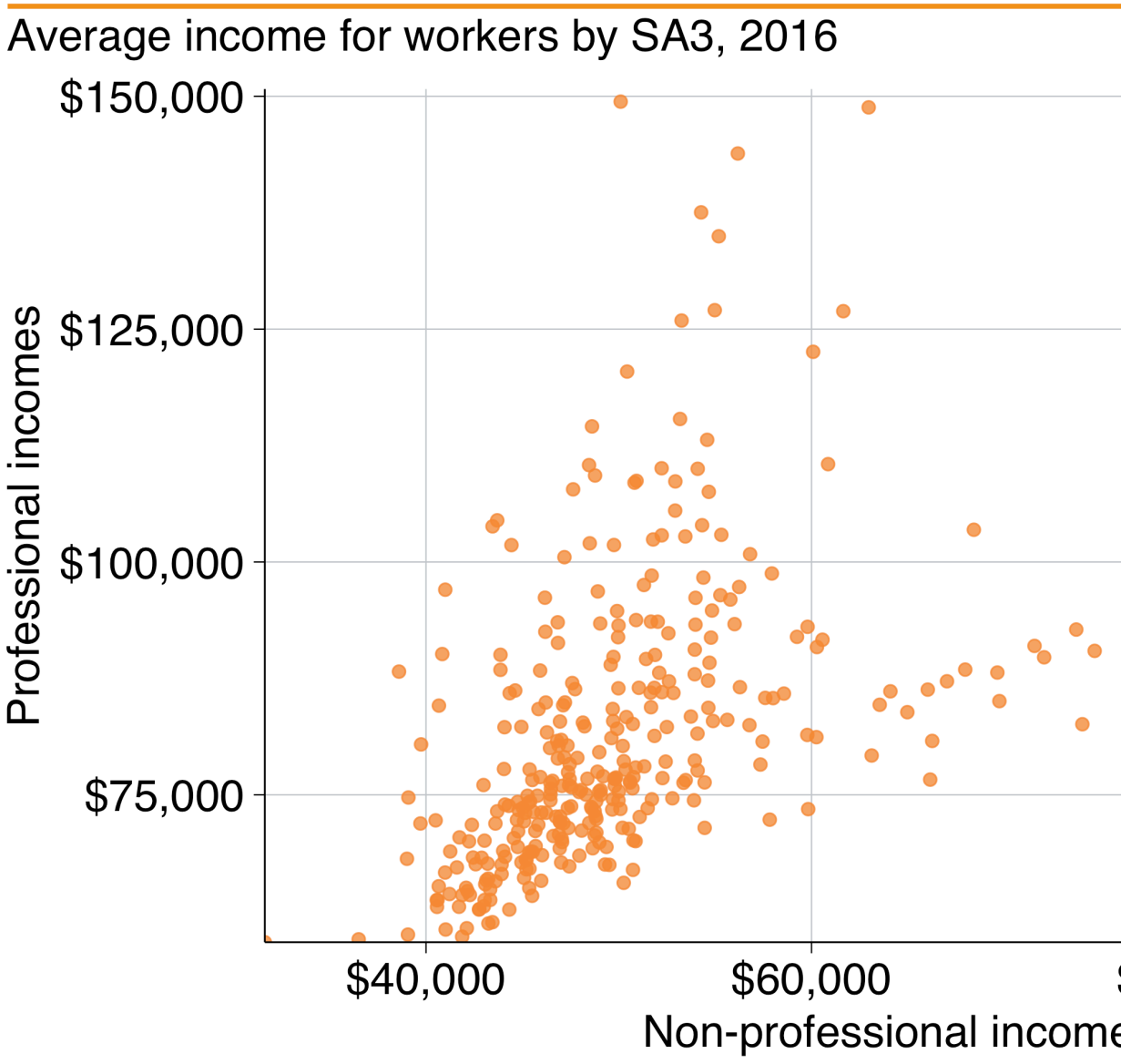
Then, like you've done before, add Grattan theme elements and titles, and save with `grattan_save`:

```
scatter_reshape <- data %>%
  ggplot(aes(x = non_professional,
             y = professional)) +
```

```
  geom_point(alpha = 0.8) +
  theme_grattan(chart_type = "scatter") +
  grattan_y_continuous(labels = dollar) +
  grattan_x_continuous(labels = dollar) +
  labs(title = "Non-professionals tend to earn more when professionals do",
       subtitle = "Average income for workers by SA3, 2016",
       y = "Professional incomes",
       x = "Non-professional incomes",
       caption = "Notes: Only includes people who submitted a tax return in 2016-16. So
```

```
grattan_save("atlas/scatter_reshape.pdf", scatter_reshape, type = "fullslide")
```

Non-professionals tend to earn more when professio do

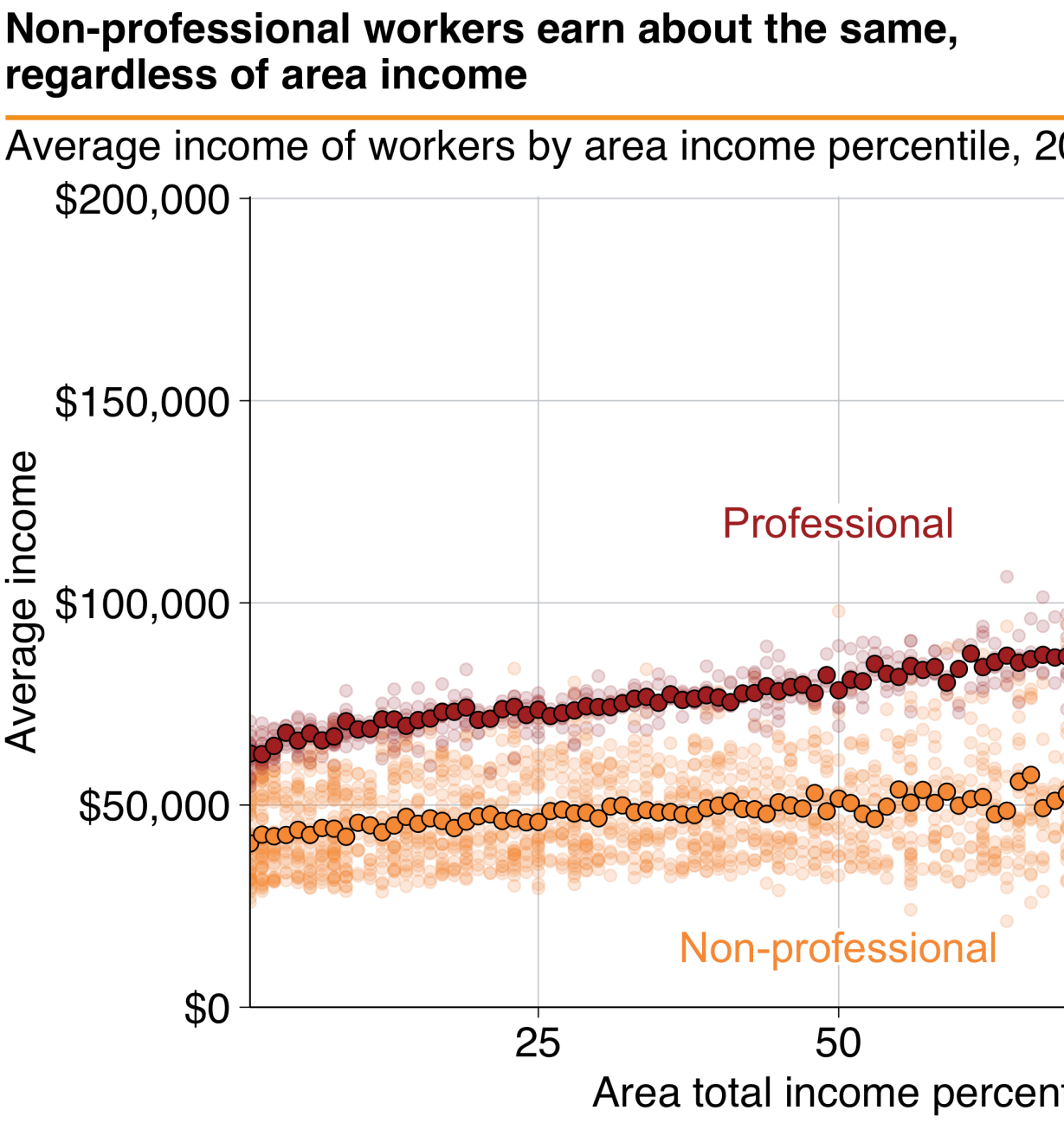


Notes: Only includes people who submitted a tax return in 2016-16.
Source: ABS (2018)

13.4.3 Layered scatter plot

For the third plot, look at the incomes of non-professional workers by their area's total income percentile:

```
include_graphics("atlas/scatter_layer.png")
```



Notes: Only includes people who submitted a tax return in 2016-16.
Source: ABS (2018)

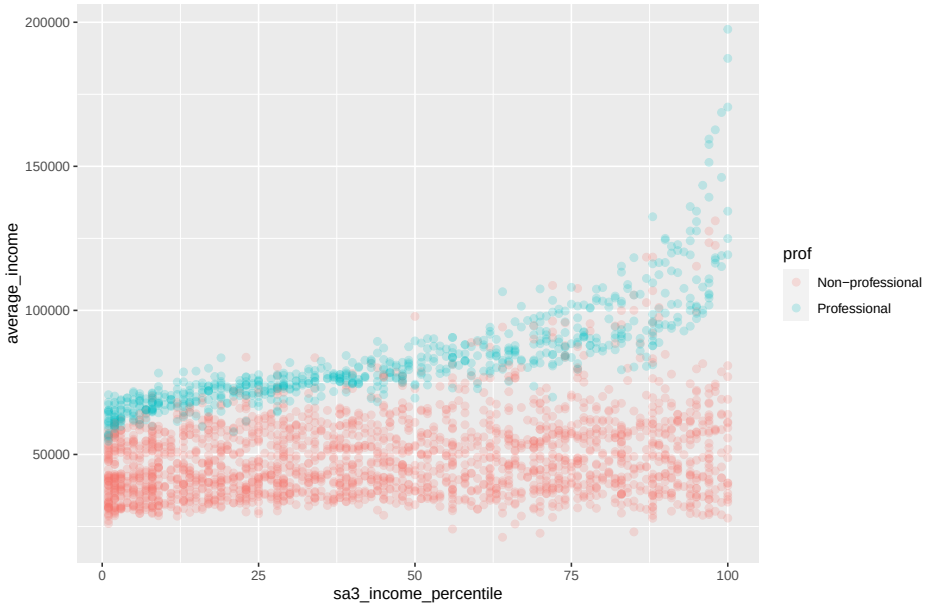
Get the data you want to plot:

```
data <- sa3_income %>%
  filter(year == 2016) %>%
  mutate(total_income = average_income * workers) %>%
  group_by(sa3_name, sa3_income_percentile, prof, occ_short) %>%
  summarise(income = sum(total_income),
            workers = sum(workers),
            average_income = income / workers)

head(data)
```

```
## # A tibble: 6 x 7
## # Groups:   sa3_name, sa3_income_percentile, prof [1]
##   sa3_name   sa3_income_percen~ prof     occ_short income workers average_income
##   <chr>                   <dbl> <chr>    <chr>      <dbl>   <dbl>          <dbl>
## 1 Adelaide ~                 66 Non-pro~ Admin     1.44e8    2674         53979.
## 2 Adelaide ~                 66 Non-pro~ Driver    1.85e7     396         46762.
## 3 Adelaide ~                 66 Non-pro~ Labourer  3.92e7    1516         25868.
## 4 Adelaide ~                 66 Non-pro~ Sales     5.05e7    1546         32680.
## 5 Adelaide ~                 66 Non-pro~ Service   7.75e7    2346         33034.
## 6 Adelaide ~                 66 Non-pro~ Trades    7.85e7    1525         51448.
```

To make a scatter plot with `average_income` against `sa3_income_percentile`, pass the `income` dataset to `ggplot`, add `x = sa3_income_percentile`, `y = average_income` and `colour = gender` aesthetics, then plot it with `geom_point`. Tell `geom_point` to reduce the opacity with `alpha = 0.2`, as these individual points are more of the 'background' to the plot:

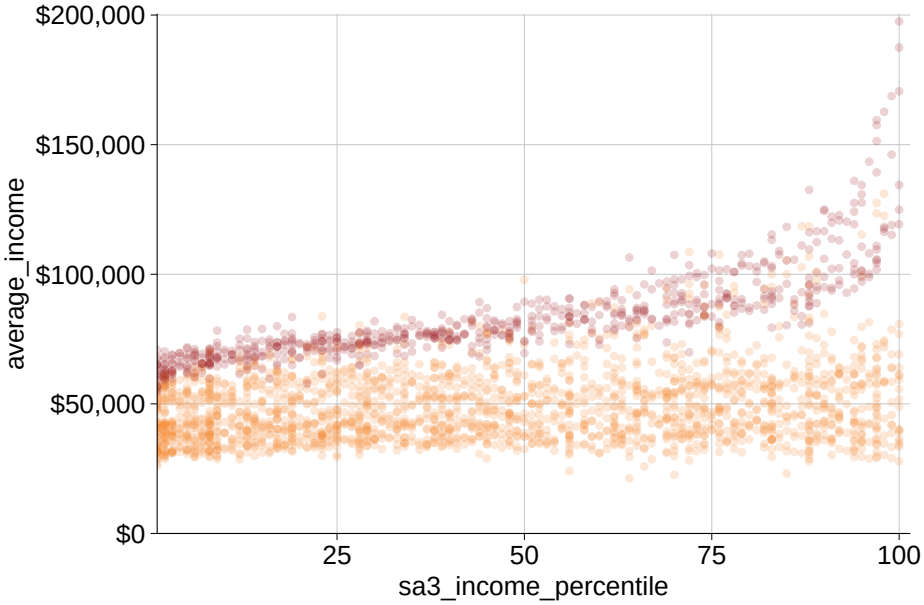
```
data %>%
  ggplot(aes(x = sa3_income_percentile,
             y = average_income,
             colour = prof)) +
  geom_point(alpha = 0.2)
```



Now add your Grattan theme elements:

- `theme_grattan()`, telling it that the `chart_type` is a scatter plot.
- `grattan_colour_manual()` with 2 colours.
- `grattan_y_continuous()`, setting the label style to `dollar`. Also tell the plot to start at zero by setting `limits = c(0, NA)` (lower, upper limits, with `NA` representing 'choose automatically'). Note that starting at zero isn't a requirement for scatter plots, but here it will give you some breathing space for your labels.
- `grattan_x_continuous()`.

```
base_chart <- data %>%
  ggplot(aes(x = sa3_income_percentile,
             y = average_income,
             colour = prof)) +
  geom_point(alpha = 0.2) +
  theme_grattan(chart_type = "scatter") +
  grattan_colour_manual(2) +
  grattan_y_continuous(labels = dollar,
                       limits = c(0, NA)) +
  grattan_x_continuous()

base_chart
```



Looks of the highest quality! To make the point a little clearer, we can overlay a point for average income each percentile. Create a dataset that has the average income for each area and professional work category:

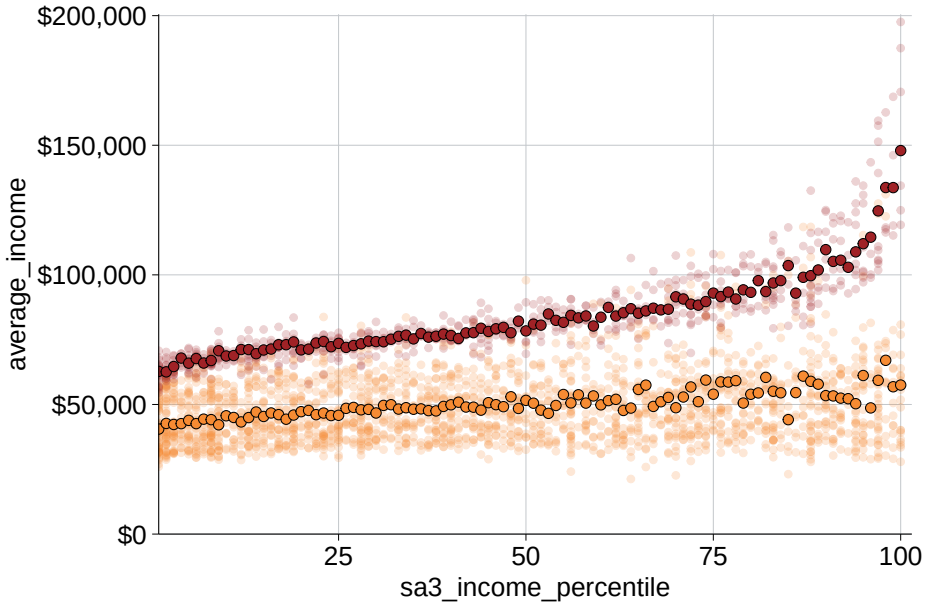
```
perc_average <- data %>%
  group_by(prof, sa3_income_percentile) %>%
  summarise(average_income = sum(income) / sum(workers))

head(perc_average)
```

```
## # A tibble: 6 x 3
## # Groups:   prof [1]
##   prof             sa3_income_percentile average_income
##   <chr>                            <dbl>          <dbl>
## 1 Non-professional                     1         40515.
## 2 Non-professional                     2         42689.
## 3 Non-professional                     3         42280.
## 4 Non-professional                     4         42600.
## 5 Non-professional                     5         43868.
## 6 Non-professional                     6         42615.
```

Then layer this on your plot by adding another `geom_point` and providing the `perc_average` data. Add a `fill` aesthetic and change the shape to `21`: a circle with a border (controlled by `colour`) and fill colour (controlled by `fill`).[6] Make the outline of the circle black with `colour` and make the `size` a little bigger:

[6] See the full list of shapes here.

```
base_chart +
  geom_point(data = perc_average,
             aes(fill = prof),
             shape = 21,
             size = 3,
             colour = "black") +
  grattan_fill_manual(2)
```



To add labels, first decide where they should go. Try positioning the "Professional" above its averages, and "Non-professional" at the bottom.

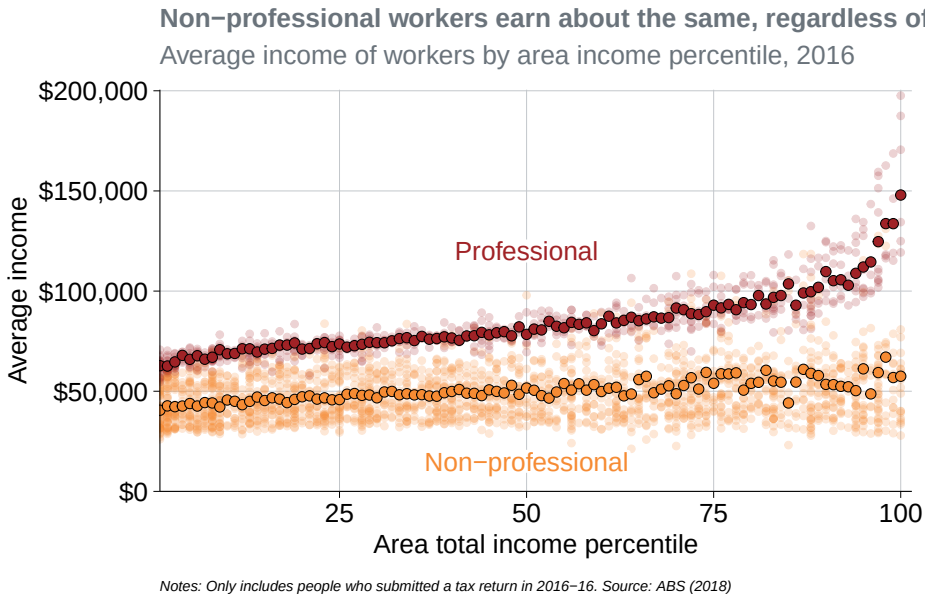
Like labelling before, you should create a new dataset with your label information, and pass that label dataset to the `grattan_label` function:

```
label_data <- tibble(
  sa3_income_percentile = c(50, 50),
  average_income = c(15e3, 120e3),
  prof =  c("Non-professional", "Professional"))
```

Finally, add the labels to the plot and give some titles:

```
base_chart +
  geom_point(data = perc_average,
             aes(fill = prof),
             shape = 21,
             size = 3,
             colour = "black") +
  grattan_fill_manual(2) +
```

```
  grattan_label(data = label_data,
                aes(label = prof)) +
  labs(title = "Non-professional workers earn about the same, regardless of area income
       subtitle = "Average income of workers by area income percentile, 2016",
       x = "Area total income percentile",
       y = "Average income",
       caption = "Notes: Only includes people who submitted a tax return in 2016-16. So
```



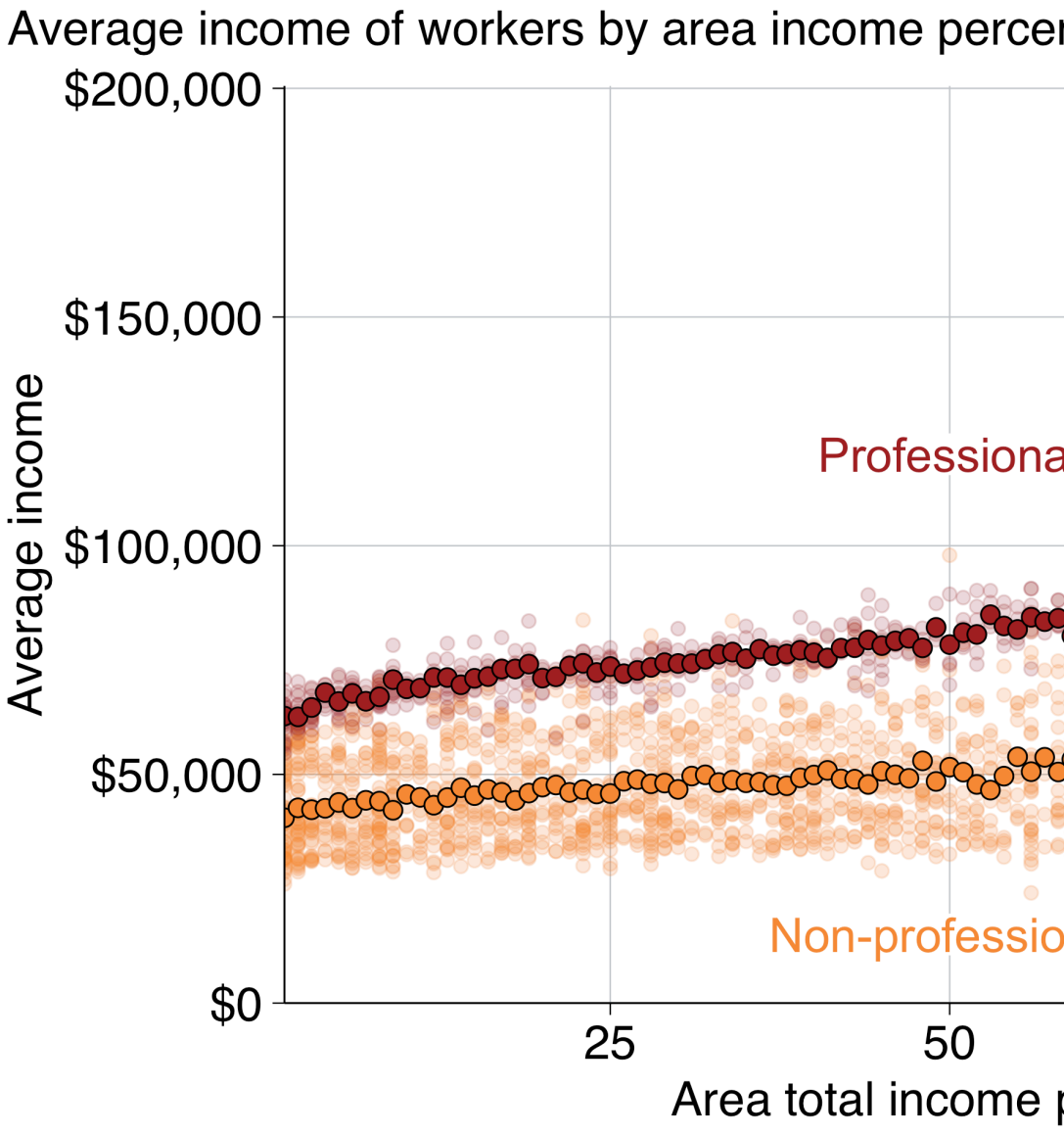
Putting that all together, your code will look something like this:

```
# Create percentage data
perc_average <- data %>%
  group_by(prof, sa3_income_percentile) %>%
  summarise(average_income = sum(income) / sum(workers))

# Create label data
label_data <- tibble(
  sa3_income_percentile = c(50, 50),
  average_income = c(15e3, 120e3),
  prof =  c("Non-professional", "Professional"))


# Plot
scatter_layer <- data %>%
  ggplot(aes(x = sa3_income_percentile,
             y = average_income,
             colour = prof)) +
```

```
  geom_point(alpha = 0.2) +
  theme_grattan(chart_type = "scatter") +
  grattan_colour_manual(2) +
  grattan_y_continuous(labels = dollar,
                       limits = c(0, NA)) +
  grattan_x_continuous() +
  geom_point(data = perc_average,
             aes(fill = prof),
             shape = 21,
             size = 3,
             colour = "black") +
  grattan_fill_manual(2) +
  grattan_label(data = label_data,
                aes(label = prof)) +
  labs(title = "Non-professional workers earn about the same, regardless of area income",
       subtitle = "Average income of workers by area income percentile, 2016",
       x = "Area total income percentile",
       y = "Average income",
       caption = "Notes: Only includes people who submitted a tax return in 2016-16. Source: ABS
```

```
grattan_save("atlas/scatter_layer.pdf", scatter_layer, type = "fullslide")
```

Non-professional workers earn about the same regardless of area income



Notes: Only includes people who submitted a tax return in 2016-16.
Source: ABS (2018)

13.4.4 Scatter plots with trendlines

13.4.5 Facetted scatter plots

13.5 Distributions

`geom_histogram` `geom_density`

`ggridges::`

13.6 Maps

13.6.1 `sf` objects

[what is]

13.6.2 Using `absmapsdata`

The `absmapsdata` contains compressed, and tidied `sf` objects containing geometric information about ABS data structures. The included objects are:

- Statistical Area 1 2011 and 2016: `sa12011` or `sa12016`
- Statistical Area 2 2011 and 2016: `sa22011` or `sa22016`
- Statistical Area 3 2011 and 2016: `sa32011` or `sa32016`
- Statistical Area 4 2011 and 2016: `sa42011` or `sa42016`
- Greater Capital Cities 2011 and 2016: `gcc2011` or `gcc2016`
- Remoteness Areas 2011 and 2016: `ra2011` or `ra2016`
- State 2011 and 2016: `state2011` or `state2016`
- Commonwealth Electoral Divisions 2018: `ced2018`
- State Electoral Divisions 2018:`sed2018`
- Local Government Areas 2016 and 2018: `lga2016` or `lga2018`
- Postcodes 2016: `postcodes2016`

The package is hosted on Github and can be installed with `remotes::install_github()`

```
remotes::install_github("wfmackey/absmapsdata")
library(absmapsdata)
```

You will also need the `sf` package installed to handle the `sf` objects:

```
install.packages("sf")
library(sf)
```

Now you can view `sf` objects stored in `absmapsdata`:

```
glimpse(sa32016)
```

```
## Observations: 358
## Variables: 12
## $ sa3_code_2016   <chr> "10102", "10103", "10104", "10105", "10106", "10201...
## $ sa3_name_2016   <chr> "Queanbeyan", "Snowy Mountains", "South Coast", "Go...
## $ sa4_code_2016   <chr> "101", "101", "101", "101", "101", "102", "102", "1...
## $ sa4_name_2016   <chr> "Capital Region", "Capital Region", "Capital Region...
## $ gcc_code_2016   <chr> "1RNSW", "1RNSW", "1RNSW", "1RNSW", "1RNSW", "1GSYD...
## $ gcc_name_2016   <chr> "Rest of NSW", "Rest of NSW", "Rest of NSW", "Rest ...
## $ state_code_2016 <chr> "1", "1", "1", "1", "1", "1", "1", "1", "1", "1", "...
## $ state_name_2016 <chr> "New South Wales", "New South Wales", "New South Wa...
## $ areasqkm_2016   <dbl> 6511.1906, 14283.4221, 9864.8680, 9099.9086, 12136....
## $ cent_long       <dbl> 149.6013, 148.9415, 149.8063, 149.6054, 148.6799, 1...
## $ cent_lat        <dbl> -35.44939, -36.43952, -36.49933, -34.51814, -34.580...
## $ geometry        <MULTIPOLYGON [°]> MULTIPOLYGON (((149.979 -35..., MULTIP...
```

13.6.3 Making choropleth maps

Choropleth maps break an area into 'bits', and colours each 'bit' according to a variable.

You can join the `sf` objects from `absmapsdata` to your dataset using `left_join`. The variable names might be different – eg `sa3_name` compared to `sa3_name_2016` – so use the `by` argument to match them.

First, take the `sa3_income` dataset and join the `sf` object `sa32016` from `absmapsdata`:

```
map_data <- sa3_income %>%
  left_join(sa32016, by = c("sa3_name" = "sa3_name_2016"))
```

You then plot a map like you would any other `ggplot`: provide your data, then choose your `aes` and your `geom`. For maps with `sf` objects, the **key aesthetic** is `geometry = geometry`, and the **key geom** is `geom_sf`.

The argument `lwd` controls the line width of area borders.

Note that RStudio takes a long time to render a map in the

Showing all of Australia on a single map is difficult: there are enormous areas that are home to few people which dominate the space. Showing individual states or capital city areas can sometimes be useful.

To do this, filter the `map_data` object:

13.6.3.1 Adding labels to maps

You can add labels to choropleth maps with the standard `geom_text` or `geom_label`. Because it is likely that some labels will overlap, `ggrepel::geom_text_repel` or `ggrepel::geom_label_repel` is usually the better option.

To use `geom_(text|label)_repel`, you need to tell `ggrepel` where in

```
map <- map_data %>%
        filter(state == "Vic") %>%
        ggplot(aes(geometry = geometry)) +
        geom_sf(aes(fill = pop_change),
                lwd = .1,
                colour = "black") +
        theme_void() +
        grattan_fill_manual(discrete = FALSE,
                            palette = "diverging",
                            limits = c(-20, 20),
                            breaks = seq(-20, 20, 10)) +
  geom_label_repel(aes(label = sa3_name),
                   stat = "sf_coordinates", nudge_x = 1000, segment.alpha = .5,
                   size = 4,
                   direction = "y",
                   label.size = 0,
                   label.padding = unit(0.1, "lines"),
                   colour = "grey50",
                   segment.color = "grey50") +
  scale_y_continuous(expand = expand_scale(mult = c(0, .2))) +
  theme(legend.position = "top") +
  labs(fill = "Population \nchange")

map
```

13.7 Creating simple interactive graphs with `plotly`

```
plotly::ggplotly()
```

Part IV

Advanced topics

Chapter 14

Creating functions

14.1 It can be useful to make your own function

Why on earth would you create your own function?

14.2 Defining simple functions

14.3 More complex functions

14.4 Sets of functions

14.5 Using `purrr::map`

14.6 Sharing your useful functions with Grattan

Chapter 15

Version control

15.1 Version control is important and intimidating

Version control is great!

15.2 Github

We use Github to version-control and share reports in LaTeX, so you're already a bit set-up.

15.3 Git

Using Git within R Studio...