

# Using R at Grattan Institute

*Will Mackey and Matt Cowgill*

*2019-08-19*



# Contents

<b>Welcome</b>	<b>5</b>
<b>1 Introduction to R</b>	<b>7</b>
1.1 What is R? . . . . .	7
1.2 What is RStudio? . . . . .	9
1.3 Installing R and RStudio . . . . .	9
1.4 Packages . . . . .	11
<b>2 Why use R?</b>	<b>15</b>
2.1 Why use script-based software? . . . . .	15
2.2 Why use R specifically? . . . . .	15
<b>3 Using R at Grattan</b>	<b>17</b>
3.1 Using R projects for a fully reproducible workflow. . . . .	17
3.2 Grattan coding style guide . . . . .	18
3.3 What is the tidyverse and why do we use it? . . . . .	18
3.4 An introduction to RMarkdown . . . . .	18
3.5 Resources in this package . . . . .	18
<b>4 Data Visualisation</b>	<b>19</b>
4.1 Set-up and packages . . . . .	19
4.2 Concepts . . . . .	20
4.3 Making Grattan-y charts . . . . .	23
4.4 Chart cookbook . . . . .	32
4.5 Creating simple interactive graphs with <b>plotly</b> . . . . .	45
4.6 bin: generate data used (before prior sections are constructed) . . . . .	46
<b>5 Reading data</b>	<b>47</b>
5.1 Importing data . . . . .	47
5.2 Reading common files: . . . . .	48
5.3 Appropriately renaming variables . . . . .	48
5.4 Getting to tidy data . . . . .	48
<b>6 Different data types</b>	<b>49</b>

6.1	Tidy data . . . . .	49
6.2	Dates with <code>lubridate::</code> . . . . .	49
6.3	Strings with <code>stringr::</code> . . . . .	49
6.4	Factors with <code>forcats::</code> . . . . .	49
<b>7</b>	<b>Data transformation</b>	<b>51</b>
7.1	The pipe . . . . .	51
7.2	Key <code>dplyr</code> functions: . . . . .	51
7.3	Filter with <code>filter()</code> . . . . .	52
7.4	Arrange with <code>arrange()</code> . . . . .	52
7.5	Select variables with <code>select()</code> . . . . .	52
7.6	Group data with <code>group_by()</code> . . . . .	52
7.7	Edit and add new variables with <code>mutate()</code> . . . . .	52
7.8	Summarise data with <code>summarise()</code> . . . . .	52
7.9	Joining datasets with <code>*_join()</code> . . . . .	52
<b>8</b>	<b>Analysis</b>	<b>53</b>
<b>9</b>	<b>Creating functions</b>	<b>55</b>
9.1	It can be useful to make your own function . . . . .	55
9.2	Defining simple functions . . . . .	55
9.3	More complex functions . . . . .	55
9.4	Sets of functions . . . . .	55
9.5	Using <code>purrr::map</code> . . . . .	55
9.6	Sharing your useful functions with Grattan . . . . .	55
<b>10</b>	<b>Version control</b>	<b>57</b>
10.1	Version control is important and intimidating . . . . .	57
10.2	Github . . . . .	57
10.3	Git . . . . .	57

# Welcome

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

It does two main things: 1. Shows you how to use R to complete common analytical tasks you'll face at Grattan. 2. Sets out some guidelines and good practices when using R 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.

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



# 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.

### 1.1 What is R?

R is a programming language that is designed by and for statisticians, data scientists, and other people who work with data. It's free - you can download R at no charge. It's also open source - you can view and (if you're game) modify the code that underlies the R language. R is available for all major computing platforms including Windows, macOS, and Linux.

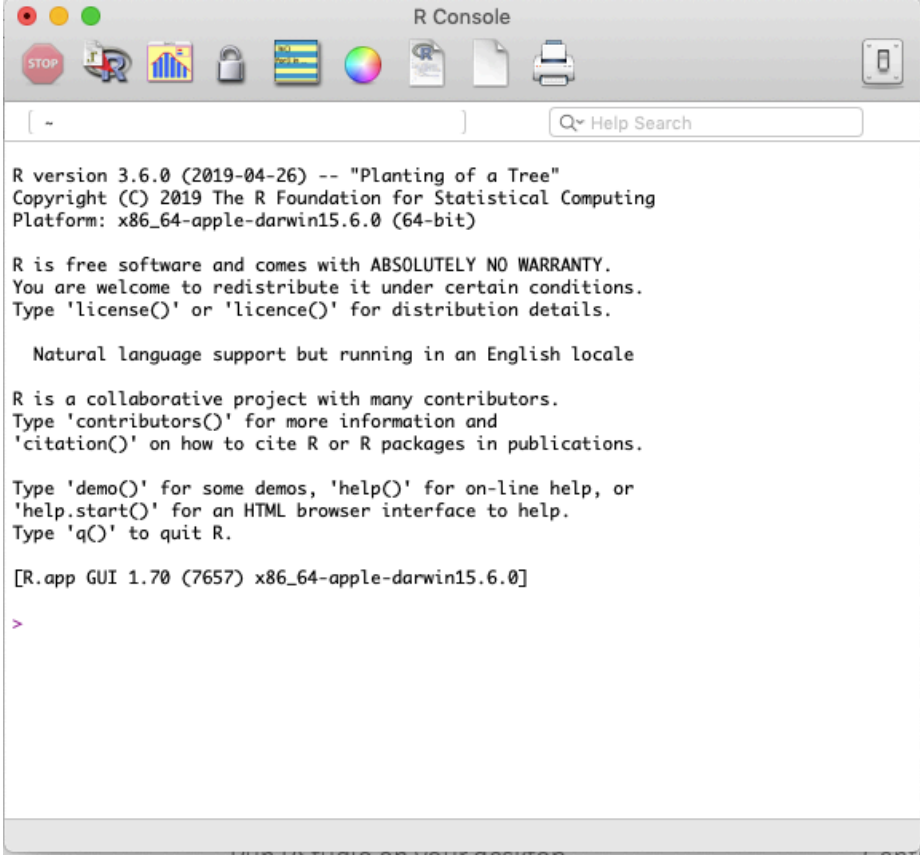
R has a lot in common with other statistical software like SAS, Stata, SPSS or Eviews. You can use those software packages to read data, manipulate it, generate summary statistics, estimate models, and so on. You can use R for all those things and more. You interact with R by writing code. This is a little different to Stata or SPSS, 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 also has some overlap with general purpose programming languages like Python. But R is more focused on the sort of tasks that statisticians, data scientists, and academic researchers do.

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.

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 version 3.6.0 (2019-04-26) -- "Planting of a Tree"
Copyright (C) 2019 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin15.6.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[R.app GUI 1.70 (7657) x86_64-apple-darwin15.6.0]
>
```

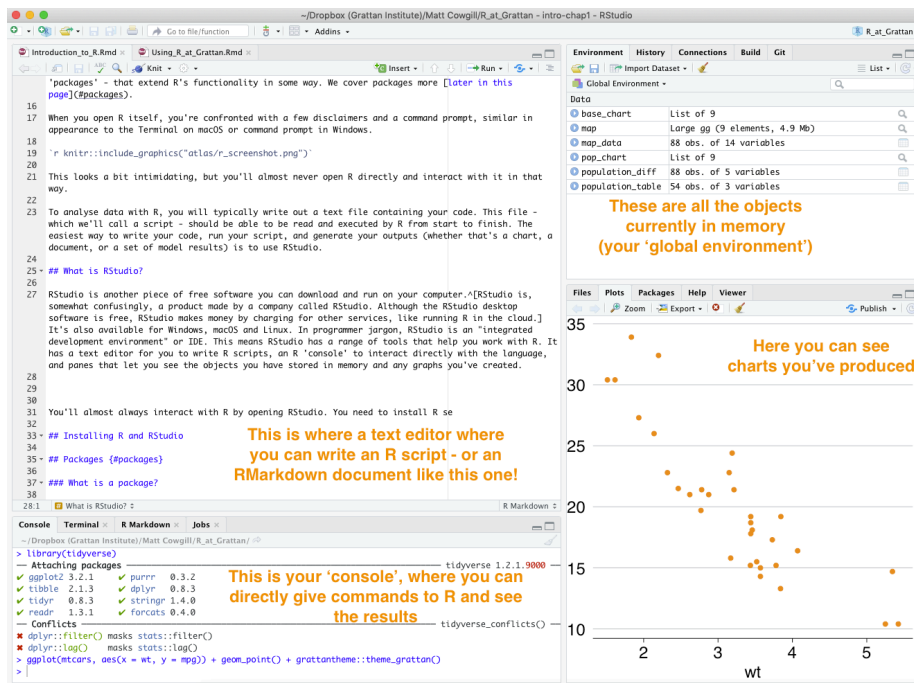
This looks a bit intimidating, but you’ll almost never open R directly and interact with it in that way.

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.



## 1.2 What is RStudio?

RStudio is another piece of free software you can download and run on your computer.<sup>1</sup> 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.

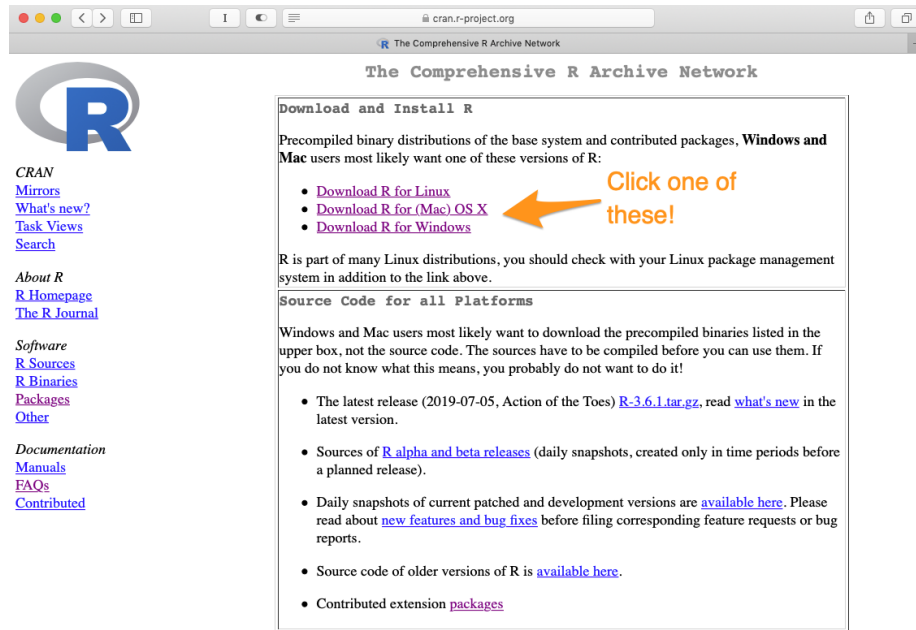
## 1.3 Installing R and RStudio

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

<sup>1</sup>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.

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:

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](#)

**R for Windows**

Subdirectories:

- [base](#): Binaries for base distribution. This is what you want to [install R for the first time](#).
- [contrib](#): Binaries of contributed CRAN packages (for R >= 2.13.x; managed by Uwe Ligges). There is also information on [third party software](#) available for CRAN Windows services and corresponding environment and make variables.
- [old contrib](#): Binaries of contributed CRAN packages for outdated versions of R (for R < 2.13.x; managed by Uwe Ligges).
- [Rtools](#): Tools to build R and R packages. This is what you want to build your own packages on Windows, or to build R itself.

Please do not submit binaries to CRAN. Package developers might want to contact Uwe Ligges directly in case of questions / suggestions related to Windows binaries.

You may also want to read the [R FAQ](#) and [R for Windows FAQ](#).

Note: CRAN does some checks on these binaries for viruses, but cannot give guarantees. Use the normal precautions with downloaded executables.

**R-3.6.1 for Windows (32/64 bit)**

[Download R 3.6.1 for Windows](#) (81 megabytes, 32/64 bit)  
[Installation and other instructions](#)  
[New features in this version](#)

If you want to double-check that the package you have downloaded matches the package distributed by CRAN, you can compare the [md5sum](#) of the .exe to the [fingerprint](#) on the master server. You will need a version of md5sum for windows: both [graphical](#) and [command line versions](#) are available.

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 Packages

R comes with a lot of functions - commands - built in to do a broad range of data tasks. You could, if you really wanted, import a dataset, clean it up, estimate a model, and make a plot all using the functions that come with R - known as 'base R'<sup>2</sup>.

But a lot of our work at Grattan uses add-on software to base R, known as 'packages'. Some packages, like the popular 'dplyr', make it quicker and/or easier to do tasks that you could otherwise do in base R. Other packages expand the possibilities of what R can do - like fitting a machine learning model, for example.

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

<sup>2</sup>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.

At Grattan, we make heavy use of a set of related packages known collectively as the **tidyverse**. We'll cover these more in a later chapter.

### 1.4.1 Installing 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: 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")
```

To install a package from Github, we use the function `install_github()`. Unfortunately, this package doesn't come with R - it's part of the `devtools` package. First, we install `devtools` from CRAN:

```
install.packages("devtools")
```

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

```
devtools::install_github("mattcowgill/grattantheme", dependencies = TRUE)
```

### 1.4.2 Using packages

Before using a function that comes from a package, as opposed to base R, 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:

```
library(devtools)
```

*# Now that the `devtools` 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:

```
devtools::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.



## 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?

1. Make your analysis reproducible by setting out the complete series of steps taken from raw data to final output.
2. Work with large data sets.

### 2.2 Why use R specifically?

```
library(tidyverse)
```





## Chapter 3

# Using R at Grattan

### 3.1 Using R projects for a fully reproducible workflow.

*Finally adhering to the ‘hit by a bus’ rule.*

Having a clear, consistent structure for our analyses means that our work is more easily checked and revised, including by ourselves in the future. A small investment of time up front to set up your analysis will save time (your own and others’) down the track.

Cover: 1. `setwd()` and machine-specific filepaths are bad 2. relative file paths are good 3. RStudio projects are an easy, reproducible way to set your wd

#### 3.1.1 Filepaths

Filepaths should be relative to the working directory, and the working directory should be set by the project.

##### Good

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

##### Bad

```
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")
```

### 3.1.2 Keep your scripts manageable

As a general rule of thumb, use one script per output. It should be clear what your script is trying to do (use comments!).

Consider breaking your analysis into pieces. For example:

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

**Don't** include interactive work (like `View(mydf)`, `str(mydf)`, `mean(mydf$variable)`, etc.) in your saved script.

### 3.1.3 Use subfolders of your project folder

Remember the hit-by-a-bus rule. It should be easy for any Grattan colleague to open your project folder and get up to speed with what it does. Putting all your files - raw data, scripts, output - in the one folder makes it harder to understand how your work fits together.

Use subfolders to clearly separate your code, raw data, and output.

## 3.2 Grattan coding style guide

Short summary of why

[Link to style guide](#)

## 3.3 What is the tidyverse and why do we use it?

Introduce following chapters

## 3.4 An introduction to RMarkdown

## 3.5 Resources in this package

- Starting a piece of analysis 'cheat sheet'.
- Updated style guide.
- Written guide/slides.

## Chapter 4

# Data Visualisation

[intro]

### 4.1 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)
library(absmapsdata)
library(sf)
library(scales)
```

For most charts in this chapter, we'll use the `population_table` data summarised here. It contains the population in each state between 2013 and 2018:

```
population_table <- read_csv("data/population_sa4.csv") %>%
  filter(data_item == "Persons - Total (no.)") %>%
  mutate(pop = as.numeric(value),
         year = as.factor(year)) %>%
  group_by(year, state) %>%
  summarise(pop = sum(pop))

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

```
## # A tibble: 6 x 3
## # Groups:   year [1]
##   year state                pop
##   <fct> <chr>                <dbl>
## 1 2013 Australian Capital Territory 383257
## 2 2013 New South Wales           7404032
## 3 2013 Northern Territory        241722
## 4 2013 Other Territories          2962
## 5 2013 Queensland               4652824
## 6 2013 South Australia           1671488
```

## 4.2 Concepts

The `ggplot2` package is based on the grammar of graphics. ...

The main ingredients to a `ggplot` chart:

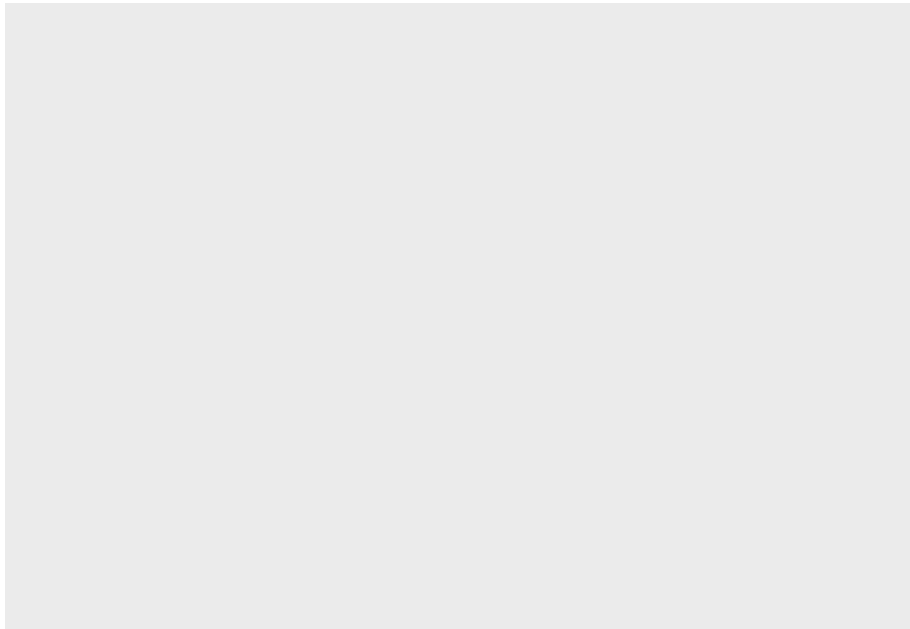
- **Data:** what data should be plotted. e.g. `data`
- **Aesthetics:** what variables should be linked to what chart elements. e.g. `aes(x = population, y = age)` to connect the `population` variable to the x axis, and the `age` variable to the y axis.
- **Geoms:** how the data should be plotted. e.g.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

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(data = <DATA>) +
  <GEOM_FUNCTION> (
    mapping = aes(<MAPPINGS>),
    stat = <STAT>,
    position = <POSITION>
  ) +
  <COORDINATE_FUNCTION> +
  <FACET_FUNCTION>
```

For example, you can plot a column chart by passing the `population_table` dataset into `ggplot()` (“make a chart with this data”). This produces an empty plot:

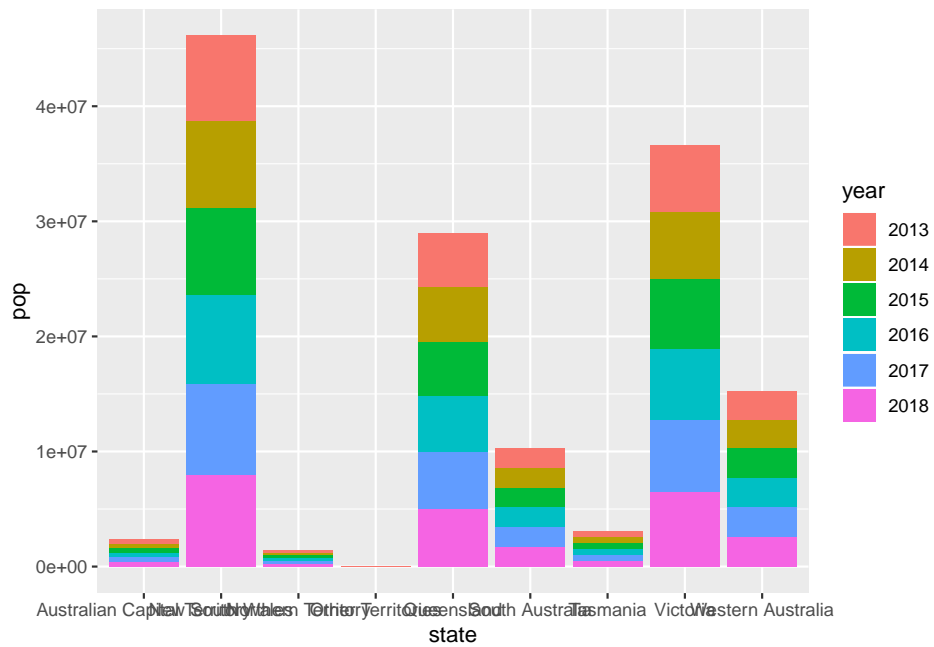
```
population_table %>%
  ggplot()
```



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: {r empty with aes} `population_table %>% ggplot(aes(x = state, y = pop, fill = year))`

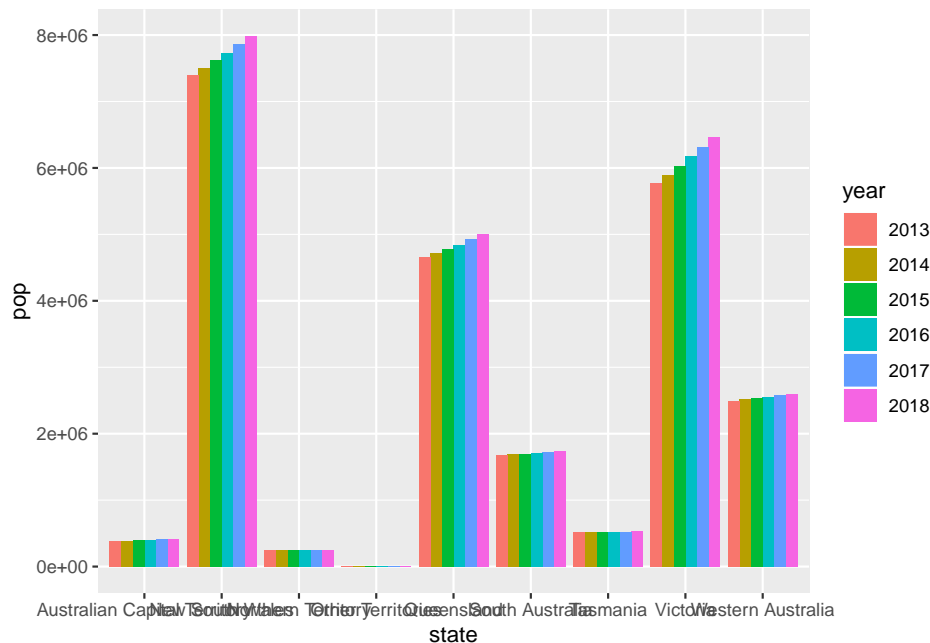
Now that `ggplot` knows where things should go, it needs to *how* to plot them. For this we use `geoms`. Tell it to plot a column chart by using `geom_col`:

```
population_table %>%  
  ggplot(aes(x = state,  
             y = pop,  
             fill = year)) +  
  geom_col()
```



Great! Although stacking populations is a bit silly. You can adjust the way `geoms` work with arguments. In this case, tell it to place the different categories next to each other rather than on top of each other using `position = "dodge"`:

```
population_table %>%
  ggplot(aes(x = state,
             y = pop,
             fill = year)) +
  geom_col(position = "dodge")
```



That's nicer. The following sections in this chapter will build on this chart. The rest of the chapter will explore:

- 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

## 4.3 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 `devtools::install_github` from the package:

```
install.packages("devtools")
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.

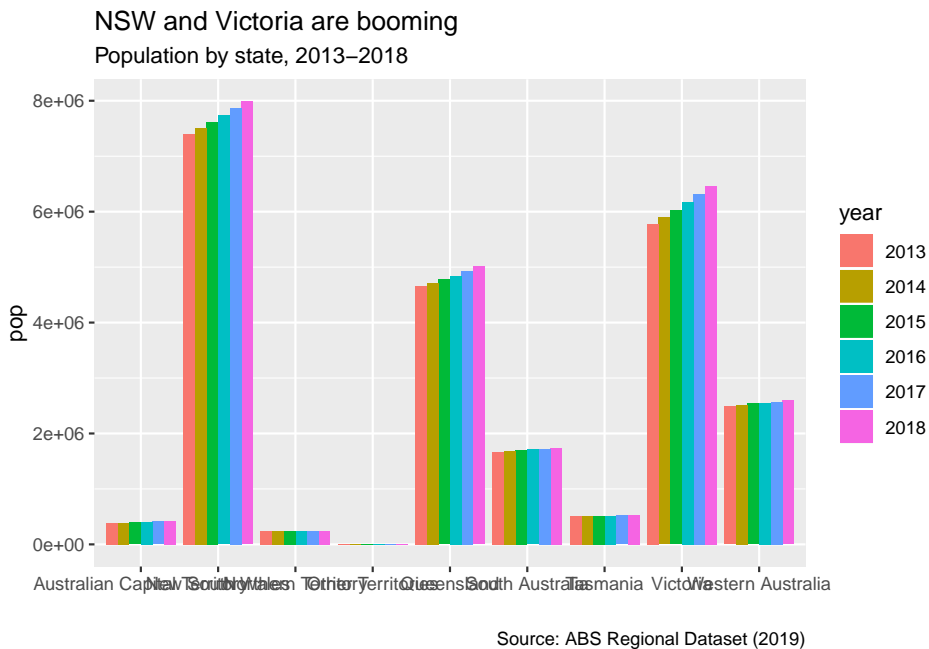
This section will run through some examples of *Grattanising* charts. The `ggplot` functions are explored in more detail in the next section.

### 4.3.1 Making Grattan charts

Start with a column chart, similar to the one made above:

```
base_chart <- population_table %>%
  ggplot(aes(x = state,
             y = pop,
             fill = year)) +
  geom_col(position = "dodge") +
  labs(x = "",
       title = "NSW and Victoria are booming",
       subtitle = "Population by state, 2013-2018",
       caption = "Source: ABS Regional Dataset (2019)")
```

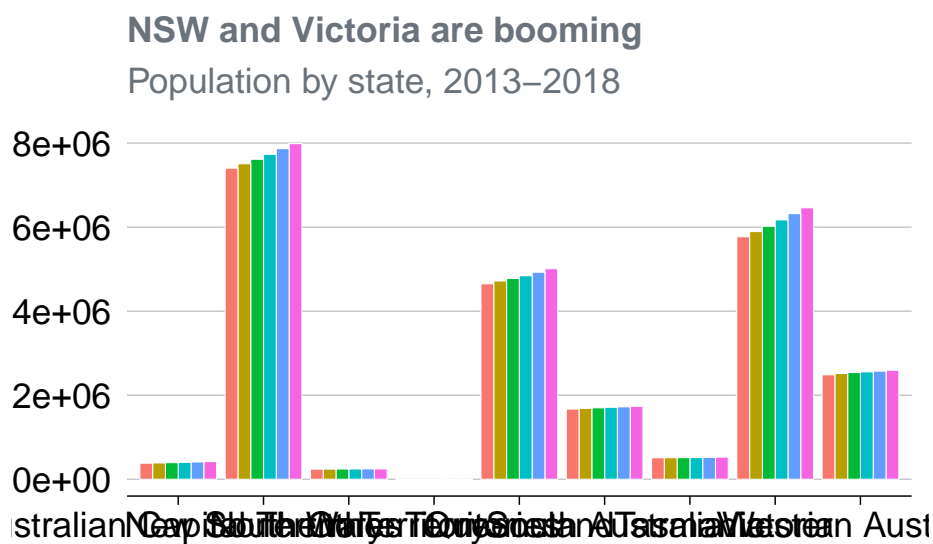
base\_chart



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



```
base_chart +
  theme_grattan()
```



Source: ABS Regional Dataset (2019)

Then `grattan_y_continuous` to align the x-axis with zero. This function takes the same arguments as `scale_y_continuous`, so you can add `labels = comma()` to reformat the y-axis labels:

```
base_chart +
  theme_grattan() +
  grattan_y_continuous(labels = comma)
```



Source: ABS Regional Dataset (2019)

To define fill colours, use `grattan_fill_manual` with the number of colours you need (six, in this case):

```
pop_chart <- base_chart +
  theme_grattan() +
  grattan_y_continuous(labels = comma) +
  grattan_fill_manual(6)

pop_chart
```



*Source: ABS Regional Dataset (2019)*

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

### 4.3.2 Saving Grattan charts

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

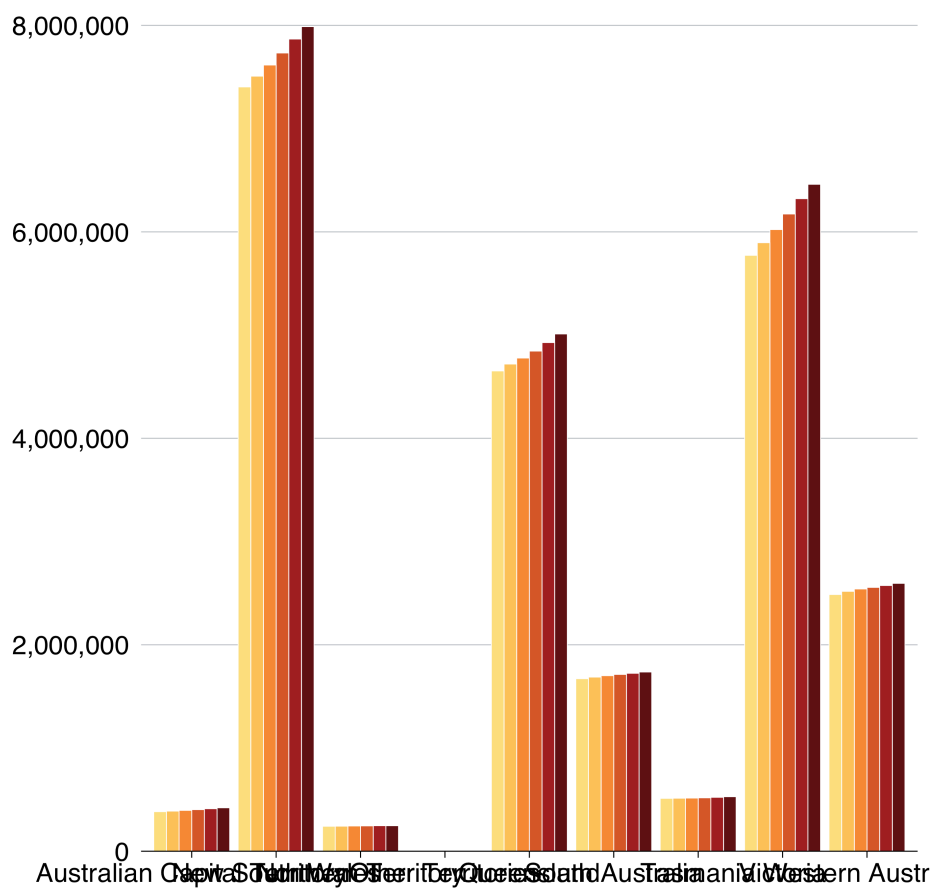
- **filename:** the path, name and file-type of your saved chart. eg: "atlas/population\_chart.pdf".
- **object:** the R object that you want to save. eg: `pop_chart`. If left blank, it grabs the last chart that was displayed.
- **type:** the Grattan template to be used. This is one of:
  - "normal" 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.

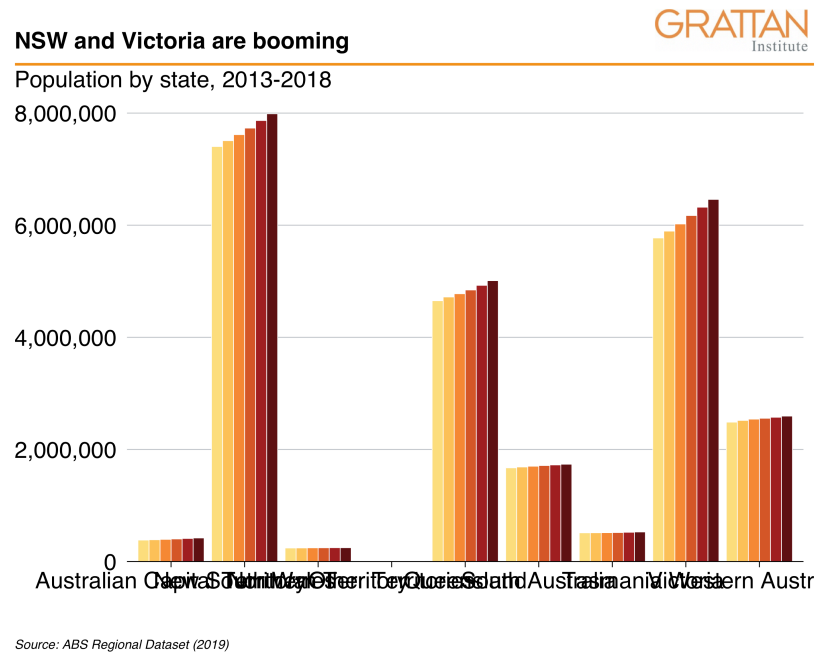
To save the `pop_chart` plot created above as a whole-column chart for a **report**:

```
grattan_save("atlas/population_chart_report.pdf", pop_chart, type = "wholecolumn")
```



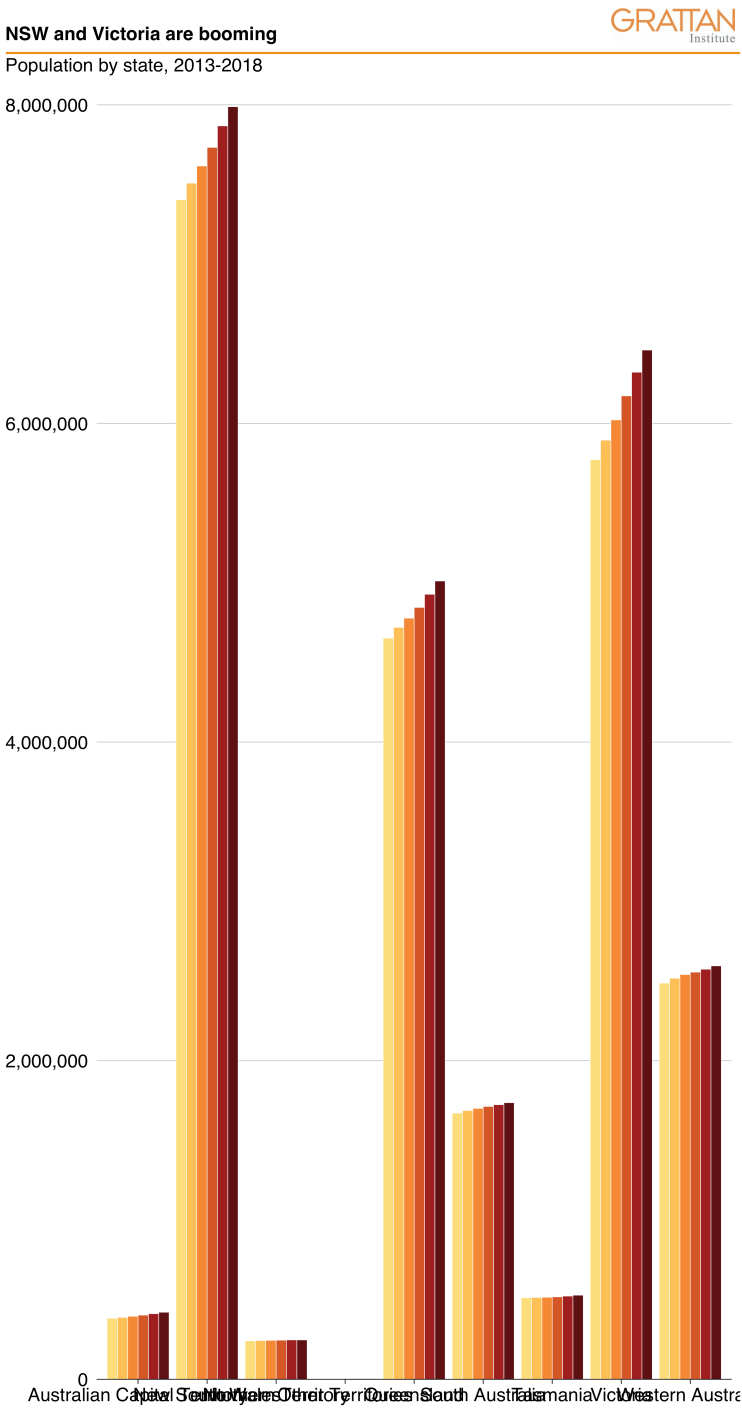
To save it as a **presentation** slide instead, use `type = "fullslide"`:

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



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/population_chart_blog.png", pop_chart,
             type = "blog", height = 50)
```



Source: ABS Regional Dataset (2019)

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.

## 4.4 Chart cookbook

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

### 4.4.1 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")`.

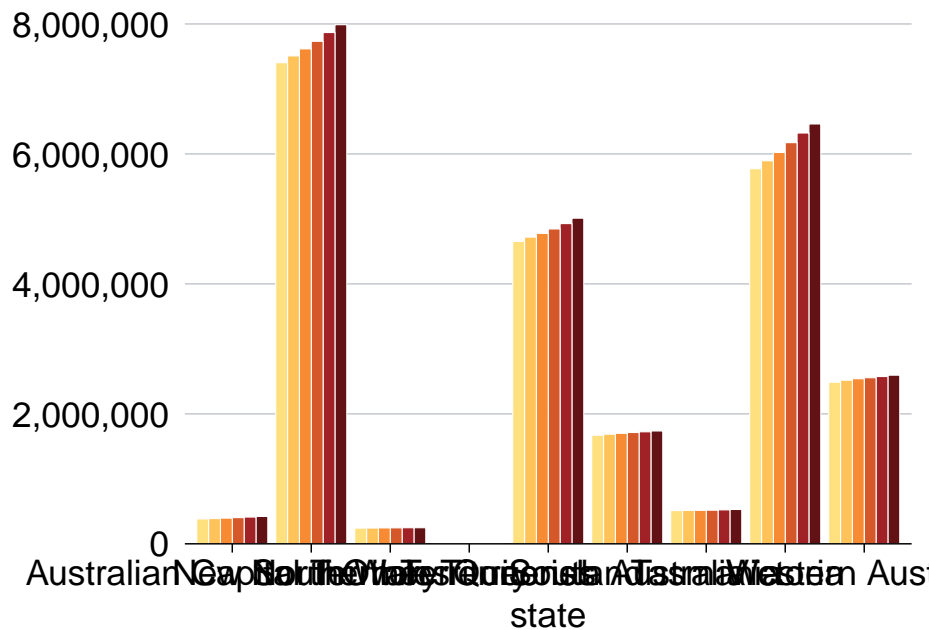
Second, `position`, dictates how multiple bars occupying the same x-axis position will be positioned. The options are:

- `"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.

```
population_table %>%
  ggplot(aes(x = state,
    y = pop,
    fill = year)) +
  geom_bar(stat = "identity",
    position = "dodge") +
  theme_grattan() +
```

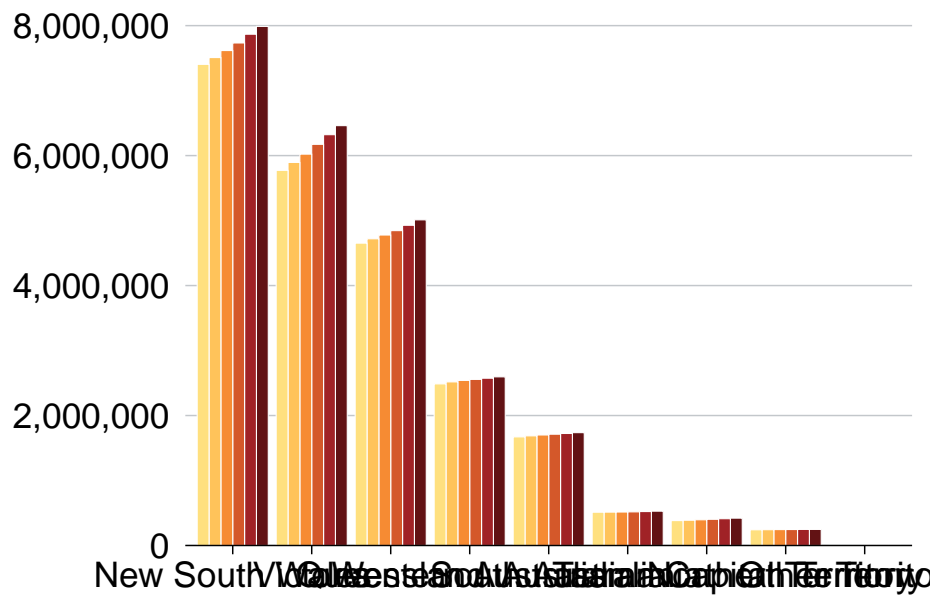


```
grattan_y_continuous(labels = comma) +
grattan_fill_manual(6)
```



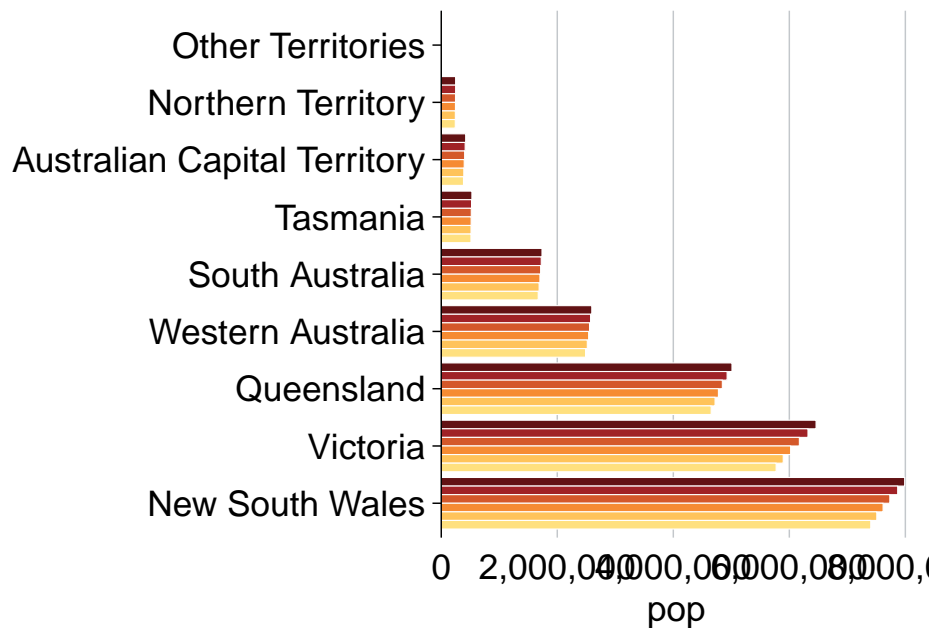
You can also **order** the groups in your chart by a variable. If you want to order states by population, use `reorder` inside `aes`:

```
population_table %>%
  ggplot(aes(x = reorder(state, -pop), # reorder state by negative population
             y = pop,
             fill = year)) +
  geom_bar(stat = "identity",
           position = "dodge") +
  theme_grattan() +
  grattan_y_continuous(labels = comma) +
  grattan_fill_manual(6) +
  labs(x = "")
```



To flip the chart – a useful move when you have long labels – add `coord_flipped` (ie ‘flip coordinates’) and tell `theme_grattan` that the plot is flipped using `flipped = TRUE`.

```
population_table %>%
  ggplot(aes(x = reorder(state, -pop),
              y = pop,
              fill = year)) +
  geom_bar(stat = "identity",
           position = "dodge") +
  coord_flip() + # flip the coordinates
  theme_grattan(flipped = TRUE) + # tell theme_grattan
  grattan_y_continuous(labels = comma) +
  grattan_fill_manual(6) +
  labs(x = "")
```

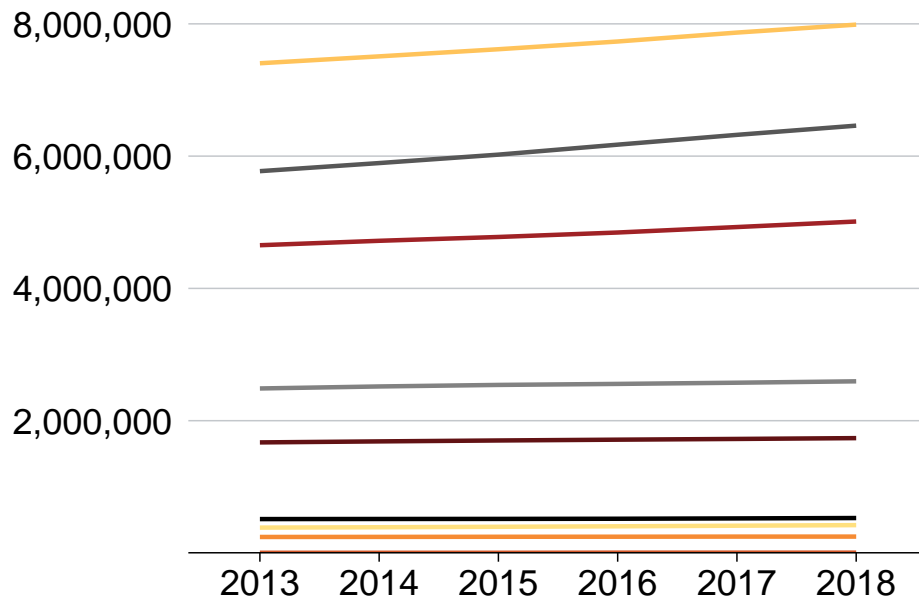


#### 4.4.2 Line charts

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

```
population_table %>%
  ggplot(aes(x = year,
             y = pop,
             colour = state,
             group = state)) +
  geom_line() +
  theme_grattan() +
  grattan_y_continuous(labels = comma) +
  grattan_colour_manual(9) +
  labs(x = "")
```

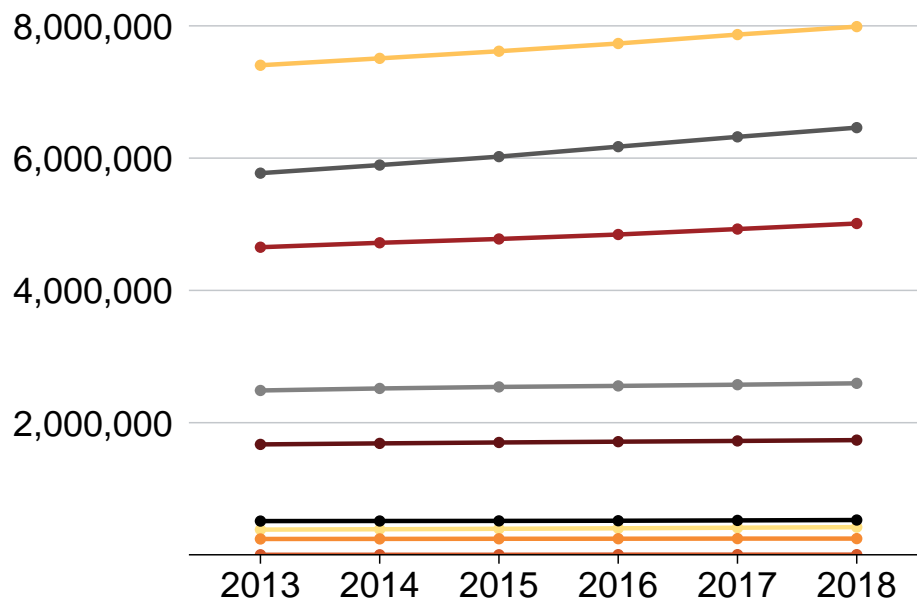
```
## Warning in grattantheme::grattan_pal(n = n, reverse = reverse): Using more
## than six colours is not recommended.
```



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

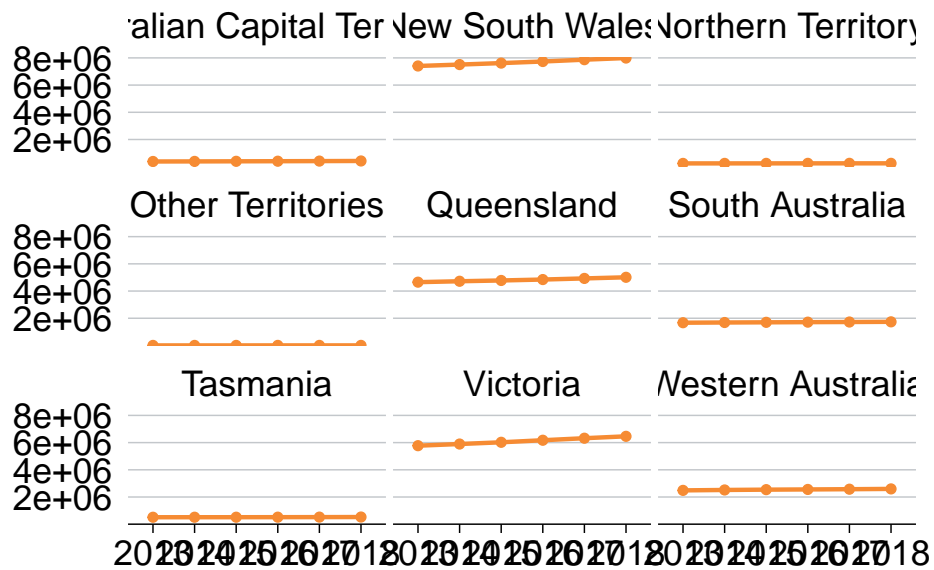
```
population_table %>%
  ggplot(aes(x = year,
             y = pop,
             colour = state,
             group = state)) +
  geom_line() +
  geom_point(size = 2) +
  theme_grattan() +
  grattan_y_continuous(labels = comma) +
  grattan_colour_manual(9) +
  labs(x = "")
```

```
## Warning in grattantheme::grattan_pal(n = n, reverse = reverse): Using more
## than six colours is not recommended.
```



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

```
population_table %>%
  filter(state != "ACT",
         state != "NT") %>%
  ggplot(aes(x = year,
             y = pop,
             group = state)) +
  geom_line() +
  geom_point(size = 2) +
  theme_grattan() +
  grattan_y_continuous() +
  facet_wrap(state ~ .) +
  labs(x = "")
```



To tidy this up, we can:

1. shorten the years to be “13”, “14”, etc instead of “2013”, “2014”, etc (via the x aesthetic)
2. shorten the y-axis labels to “millions” (via the y aesthetic)
3. add a black horizontal line at the bottom of each facet
4. give the facets a bit of room by adjusting `panel.spacing`
5. define our own x-axis label breaks to just show 13, 15 and 17

```
population_table %>%
  filter(state != "ACT",
         state != "NT") %>%
  ggplot(aes(x = substr(year, 3, 4), # 1: just take the last two characters
            y = pop / 1e6, # 2: divide population by one million
            group = state)) +
  geom_line() +
  geom_point(size = 2) +
  geom_hline(yintercept = 0) + # 3: add horizontal line at the bottom
  theme_grattan() +
  theme(panel.spacing = unit(10, "mm")) + # 4: add panel spacing
  grattan_y_continuous(labels = comma) +
  scale_x_discrete(breaks = c("13", "15", "17")) + # 5: define our own label breaks
  facet_wrap(state ~ .) +
  labs(x = "")
```



#### 4.4.3 Scatter plots

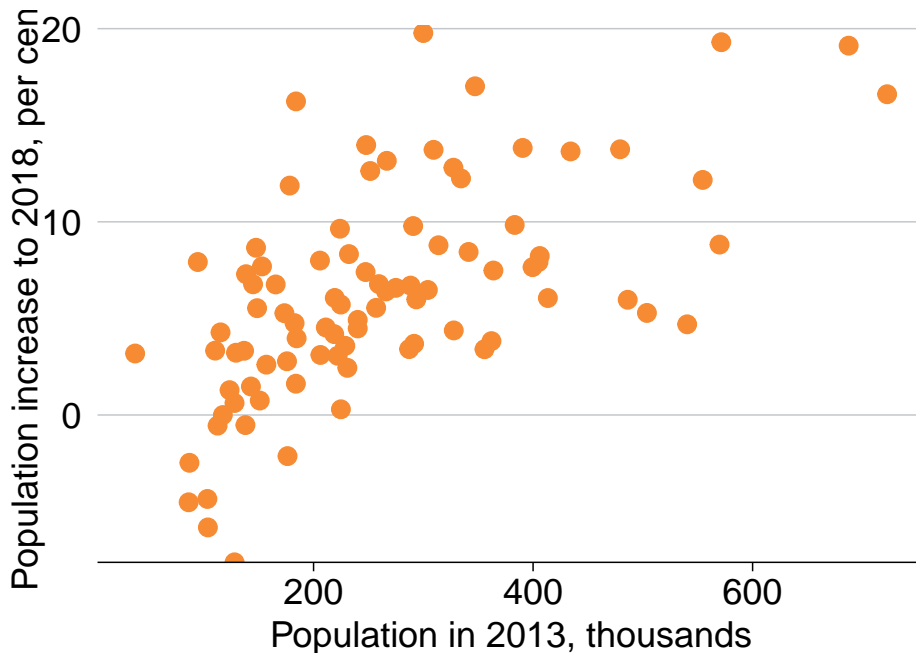
Scatter plots require `x` and `y` aesthetics. These can then be coloured and faceted.

First, create a dataset that we'll use for scatter plots. Take the `population_table` dataset and transform it to have one variable for population in 2013, and another for population in 2018:

```
population_diff <- read_csv("data/population_sa4.csv") %>%
  mutate(state_long = state,
         state = strayr::strayr(state_long),
         pop = as.numeric(value),
         year = as.factor(glue::glue("y{year}"))) %>%
  filter(year %in% c("y2013", "y2018"),
         data_item == "Persons - Total (no.)",
         sa4_name != "Other Territories") %>%
  group_by(year, state, sa4_name) %>%
  summarise(pop = sum(pop)) %>%
  spread(year, pop) %>%
  mutate(pop_change = 100 * (y2018 / y2013 - 1))
```

```
population_diff %>%
  ggplot(aes(x = y2013/1000,
            y = pop_change)) +
  geom_point(size = 4) +
  theme_grattan() +
```

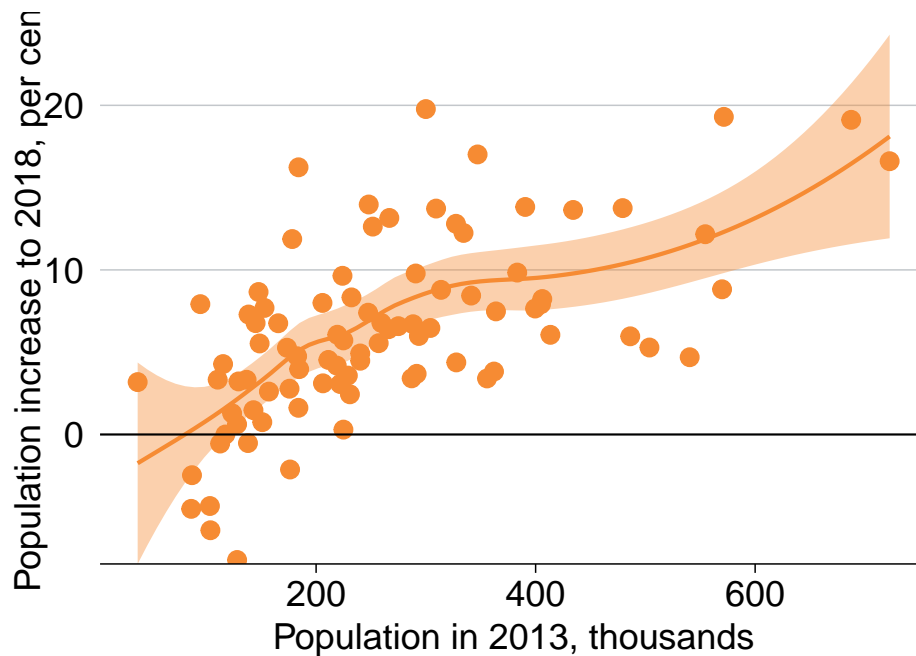
```
theme(axis.title.y = element_text(angle = 90)) +
grattan_y_continuous() +
labs(y = "Population increase to 2018, per cent",
     x = "Population in 2013, thousands")
```



It looks like the areas with the largest population grew the most between 2013 and 2018. To explore the relationship further, you can add a line-of-best-fit with `geom_smooth`:

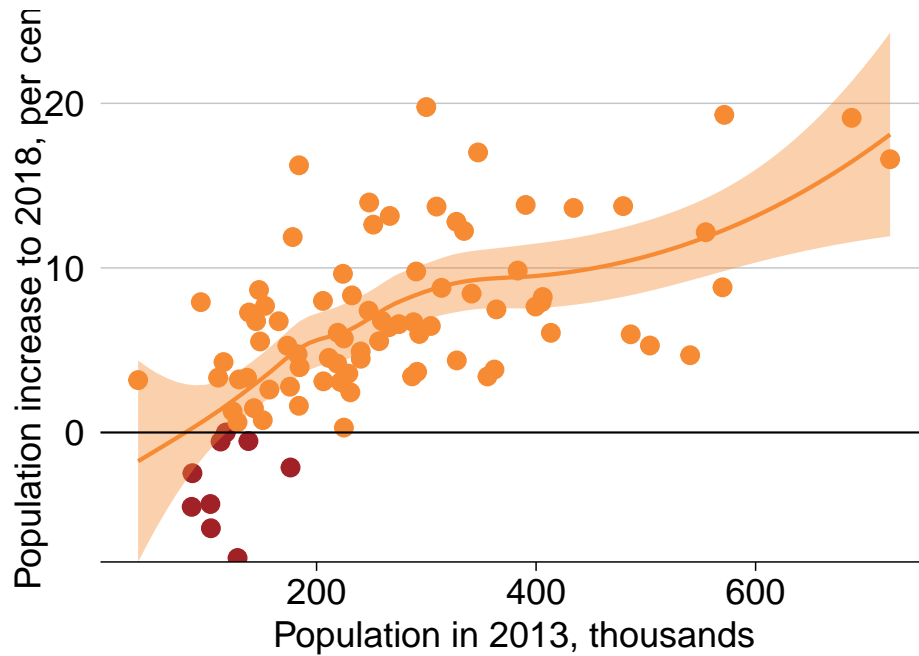
```
population_diff %>%
  ggplot(aes(x = y2013/1000, # display the x-axis as thousands
             y = pop_change)) +
  geom_point(size = 4) +
  geom_smooth() +
  geom_hline(yintercept = 0) +
  theme_grattan() +
  theme(axis.title.y = element_text(angle = 90)) +
  grattan_y_continuous() +
  labs(y = "Population increase to 2018, per cent",
       x = "Population in 2013, thousands")
```





You could colour-code positive and negative changes from within the `geom_point` aesthetic. Making a change there won't pass through to the `geom_smooth` aesthetic, so your line-of-best-fit will apply to all data points.

```
population_diff %>%
  ggplot(aes(x = y2013/1000, # display the x-axis as thousands
             y = pop_change)) +
  geom_point(aes(colour = pop_change < 0),
             size = 4) +
  geom_smooth() +
  geom_hline(yintercept = 0) +
  theme_grattan() +
  theme(axis.title.y = element_text(angle = 90)) +
  grattan_y_continuous() +
  grattan_colour_manual(2) +
  labs(y = "Population increase to 2018, per cent",
       x = "Population in 2013, thousands")
```



Like the charts above, you could facet this by state to see if there were any interesting patterns. We'll filter out ACT and NT because they only have one and two data points (SA4s) in them, respectively.

```
population_diff %>%
  filter(state != "ACT",
         state != "NT") %>%
  ggplot(aes(x = y2013/1000, # display the x-axis as thousands
            y = pop_change)) +
  geom_point(aes(colour = pop_change < 0),
            size = 2) +
  geom_smooth() +
  geom_hline(yintercept = 0) +
  theme_grattan() +
  theme(axis.title.y = element_text(angle = 90)) +
  grattan_y_continuous() +
  grattan_colour_manual(2) +
  labs(y = "Population increase to 2018, per cent",
       x = "Population in 2013, thousands") +
  facet_wrap(state ~ .)
```



#### 4.4.4 Distributions

```
geom_histogram geom_density
ggribes::
```

#### 4.4.5 Maps

##### 4.4.5.1 sf objects

[what is]

##### 4.4.5.2 Using absmappedata

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

- Statistical Area 1 2011: `sa12011`
- Statistical Area 1 2016: `sa12016`
- Statistical Area 2 2011: `sa22011`
- Statistical Area 2 2016: `sa22016`
- Statistical Area 3 2011: `sa32011`
- Statistical Area 3 2016: `sa32016`

- Statistical Area 4 2011: `sa42011`
- Statistical Area 4 2016: `sa42016`
- Greater Capital Cities 2011: `gcc2011`
- Greater Capital Cities 2016: `gcc2016`
- Remoteness Areas 2011: `ra2011`
- Remoteness Areas 2016: `ra2016`
- State 2011: `state2011`
- State 2016: `state2016`
- Commonwealth Electoral Divisions 2018: `ced2018`
- State Electoral Divisions 2018: `sed2018`
- Local Government Areas 2016: `lga2016`
- Local Government Areas 2018: `lga2018`

You can install the package from Github. You will also need the `sf` package installed to handle the `sf` objects.

```
devtools::install_github("wfmackey/absmappedata")
library(absmappedata)

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

#### 4.4.5.3 Making choropleth maps

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

SA4 is the largest non-state statistical area in the ABS ASGS standard.

You can join the `sf` objects from `absmappedata` to your dataset using `left_join`. The variable names might be different – eg `sa4_name` compared to `sa4_name_2016` – so use the `by` function to match them.

```
map_data <- population_diff %>%
  left_join(sa42016, by = c("sa4_name" = "sa4_name_2016"))

head(map_data %>%
  select(sa4_name, geometry))

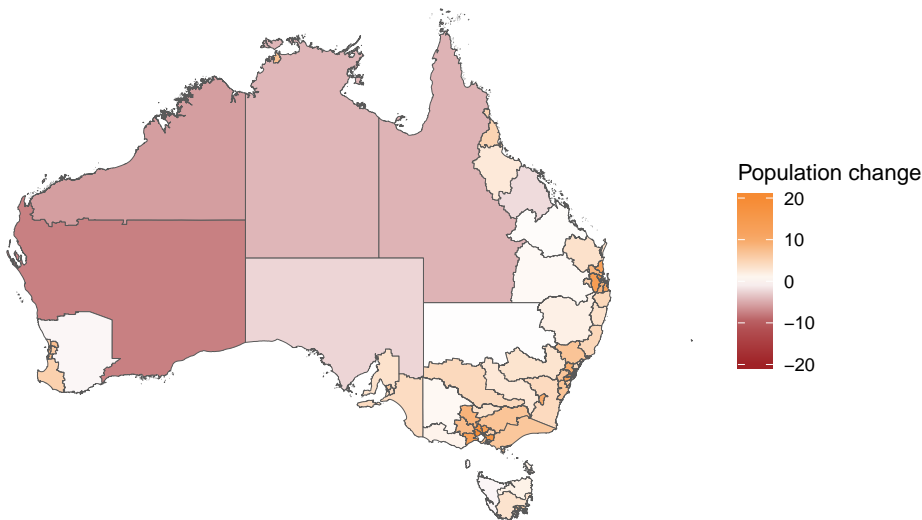
## # A tibble: 6 x 3
## # Groups:   state [2]
##   state sa4_name geometry
##   <chr> <chr>      <MULTIPOLYGON [°]>
## 1 ACT   Australian Capita~ (((148.8041 -35.71402, 148.8018 -35.7121, 148.7~
## 2 NSW   Capital Region    (((150.3113 -35.66588, 150.3126 -35.66814, 150.~
## 3 NSW   Central Coast      (((151.315 -33.55582, 151.3159 -33.55503, 151.3~
## 4 NSW   Central West        (((150.6107 -33.06614, 150.6117 -33.07051, 150.~
```

```
## 5 NSW Coffs Harbour - G~ (((153.2785 -29.91874, 153.2773 -29.92067, 153.~
## 6 NSW Far West and Orana (((150.1106 -31.74613, 150.1103 -31.74892, 150.~
```

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

```
map <- map_data %>%
  ggplot(aes(geometry = geometry,
             fill = pop_change)) +
  geom_sf(lwd = 0) +
  theme_void() +
  grattan_fill_manual(discrete = FALSE,
                     palette = "diverging",
                     limits = c(-20, 20),
                     breaks = seq(-20, 20, 10)) +
  labs(fill = "Population change")

map
```



## 4.5 Creating simple interactive graphs with plotly

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

## 4.6 bin: generate data used (before prior sections are constructed)

```
library(tidyverse)
library(janitor)
library(absmapsdata)

data <- read_csv("data/ABS_REGIONAL_ASGS2016_02082019164509969.csv") %>%
  clean_names() %>%
  select(data_code = measure,
         data_item,
         asgs = regiontype,
         sa4_code_2016 = asgs_2016,
         sa4_name_2016 = region,
         year = time,
         value) %>%
  mutate(sa4_code_2016 = as.character(sa4_code_2016)) %>%
  left_join(sa42016 %>% select(sa4_code_2016, state_name_2016)) %>%
  rename(state = state_name_2016,
         sa4_code = sa4_code_2016,
         sa4_name = sa4_name_2016) %>%
  mutate(state_long = state,
         state = strayr::strayr(state_long))

write_csv(data, "data/population_sa4.csv")
```

## Chapter 5

# Reading data

### 5.1 Importing data

#### 5.1.1 Reading CSV files

##### 5.1.1.1 `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.

##### 5.1.1.2 `data.table::fread()`

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

### 5.1.2 `readxl::read_excel()`

### 5.1.3 `rio`

### 5.1.4 `readabs`

## 5.2 Reading common files:

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

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

## 5.3 Appropriately renaming variables

As shown in the style guide

Add `rename_abs` function to a common Grattan package?

## 5.4 Getting to tidy data

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



## Chapter 6

# Different data types

### 6.1 Tidy data

Other data structures

### 6.2 Dates with `lubridate::`

The `lubridate::` package

### 6.3 Strings with `stringr::`

- Replacing values
- Matching values
- Separating columns

### 6.4 Factors with `forcats::`

- Dangers with factors



## Chapter 7

# Data transformation

### 7.1 The pipe

### 7.2 Key `dplyr` functions:

All have the same syntax structure, which enable pipe-chains.

### 7.3 Filter with `filter()`

### 7.4 Arrange with `arrange()`

### 7.5 Select variables with `select()`

### 7.6 Group data with `group_by()`

### 7.7 Edit and add new variables with `mutate()`

#### 7.7.1 Cases when you should use `case_when()`

### 7.8 Summarise data with `summarise()`

### 7.9 Joining datasets with `*_join()`

## Chapter 8

# Analysis



## Chapter 9

# Creating functions

### 9.1 It can be useful to make your own function

Why on earth would you create your own function?

### 9.2 Defining simple functions

### 9.3 More complex functions

### 9.4 Sets of functions

### 9.5 Using `purrr::map`

### 9.6 Sharing your useful functions with Grattan





## Chapter 10

# Version control

### 10.1 Version control is important and intimidating

Version control is great!

### 10.2 Github

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

### 10.3 Git

Using Git within R Studio...