Using R at Grattan Institute

Will Mackey and Matt Cowgill

2019-09-15

Contents

Welcome **7**

1 Introduction to R **9**
- 1.1 What is R? 9
- 1.2 What is RStudio? 12
- 1.3 Installing R and RStudio 14
- 1.4 Packages 20

2 Why use R? **23**
- 2.1 Why use script-based software? 23
- 2.2 Why use R specifically? 24

3 Organising an R project at Grattan **25**
- 3.1 Use RStudio projects, not `setwd()` 25
- 3.2 Keep your stuff together 26
- 3.3 Include a README file 27
- 3.4 Use relative filepaths 27
- 3.5 Keep your scripts manageable 28

4 Grattan coding style **29**
- 4.1 Use meaningful filenames 29
- 4.2 Script preamble 30
- 4.3 Comments 30
- 4.4 Breaking your script into parts 31

5 Naming objects and variables **33**

6 Spacing **35**
- 6.1 Assign and equals 35
- 6.2 Commas 36
- 6.3 Parentheses 36
- 6.4 Short lines and line indentation 36

7 Blocks of code **39**

8 The tidyverse **41**
8.1 What is the tidyverse and why do we use it? 41
8.2 An introduction to RMarkdown 41

9 Data Visualisation **43**
9.1 Set-up and packages 43
9.2 Concepts 44
9.3 Making Grattan-y charts 47
9.4 Chart cookbook 57
9.5 Creating simple interactive graphs with `plotly` 70
9.6 bin: generate data used (before prior sections are constructed) . 71

10 Reading data **73**
10.1 Importing data 73
10.2 Reading common files: 74
10.3 Appropriately renaming variables 74
10.4 Getting to tidy data 74

11 Different data types **75**
11.1 Tidy data 75
11.2 Dates with `lubridate::` 75
11.3 Strings with `stringr::` 75
11.4 Factors with `forcats::` 75

12 Data transformation **77**
12.1 The pipe 77
12.2 Key `dplyr` functions: 77
12.3 Filter with `filter()` 78
12.4 Arrange with `arrange()` 78
12.5 Select variables with `select()` 78
12.6 Group data with `group_by()` 78
12.7 Edit and add new variables with `mutate()` 78
12.8 Summarise data with `summarise()` 78
12.9 Joining datasets with `*_join()` 78

13 Analysis **79**

14 Creating functions **81**
14.1 It can be useful to make your own function 81
14.2 Defining simple functions 81
14.3 More complex functions 81
14.4 Sets of functions 81
14.5 Using `purrr::map` 81
14.6 Sharing your useful functions with Grattan 81

15 Version control **83**
15.1 Version control is important and intimidating 83

15.2 Github 83
15.3 Git 83

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.

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

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.

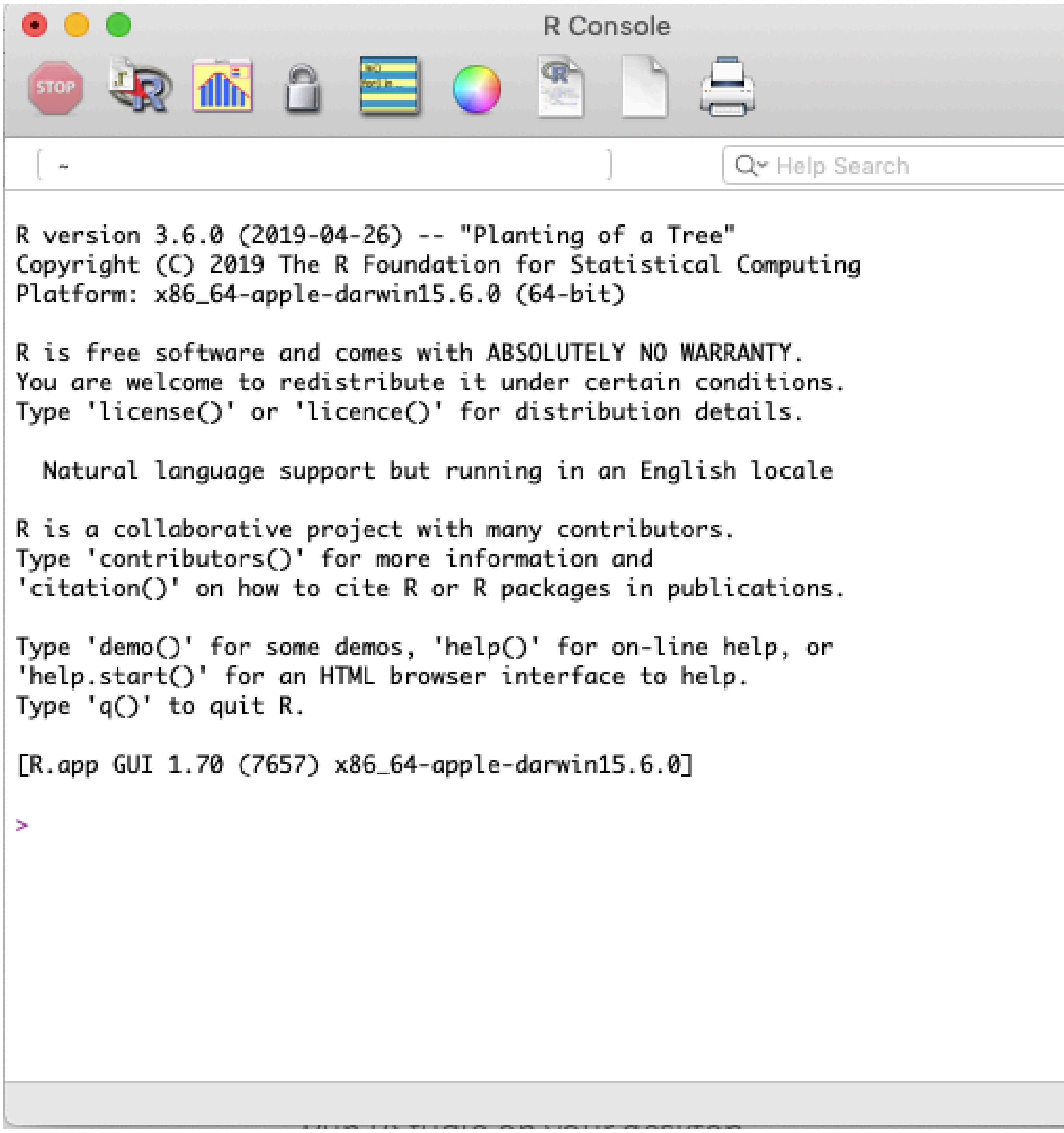
R has a lot in common with other statistical software like SAS, Stata, SPSS or Eviews. You can use those software packages to read data, manipulate it, generate summary statistics, estimate models, and so on. You can use R for all those things and more.

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 a fair bit of 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]

>
```

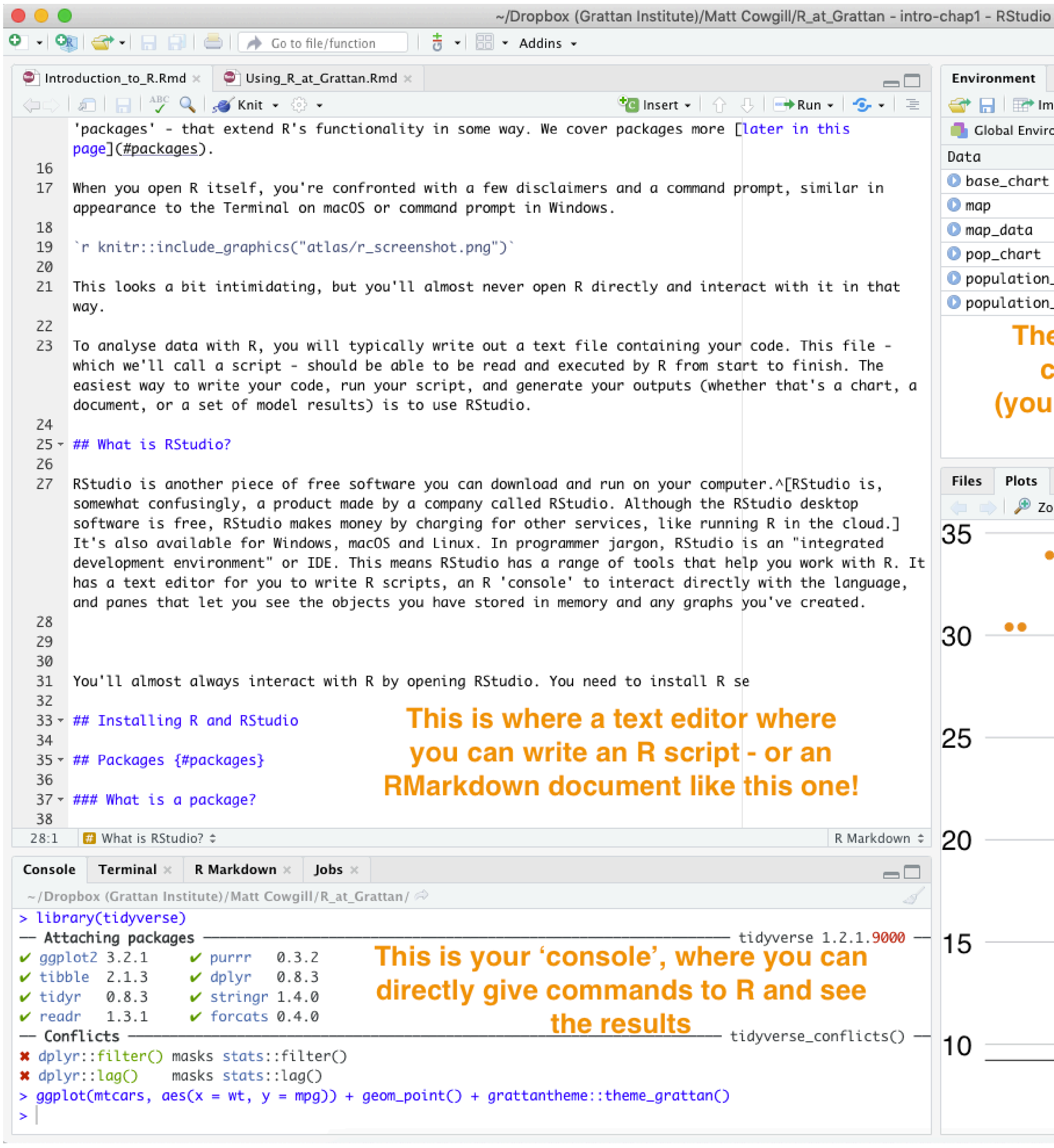
This might look a bit intimidating, but you'll almost never open R directly and interact with it in that way.

To analyse data with R, you will typically write out a text file containing 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.[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.



```
'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!

```
> 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
── 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()
>
```

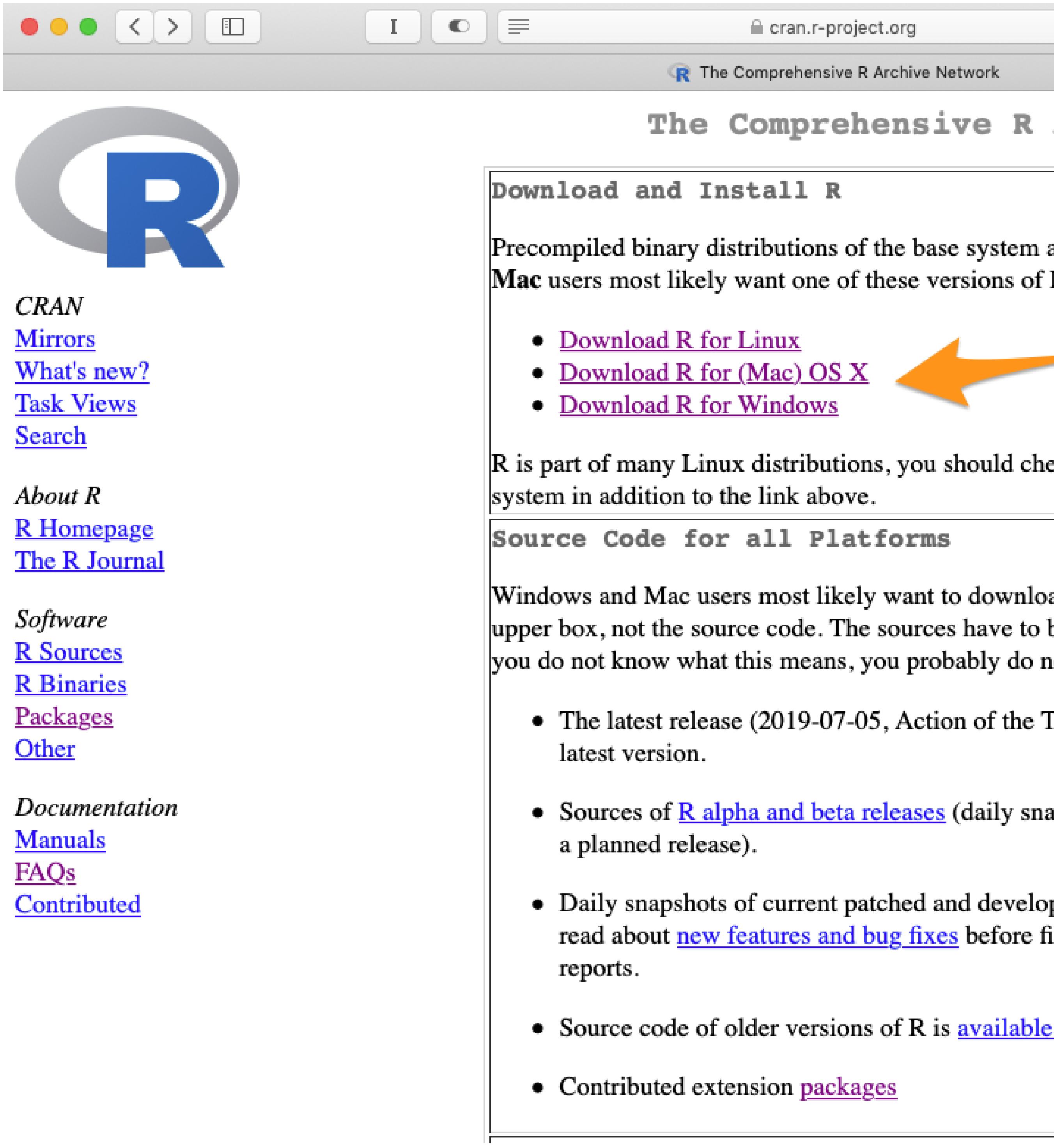
This is your ‘console’, where you can directly give commands to R and see the results

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.



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.

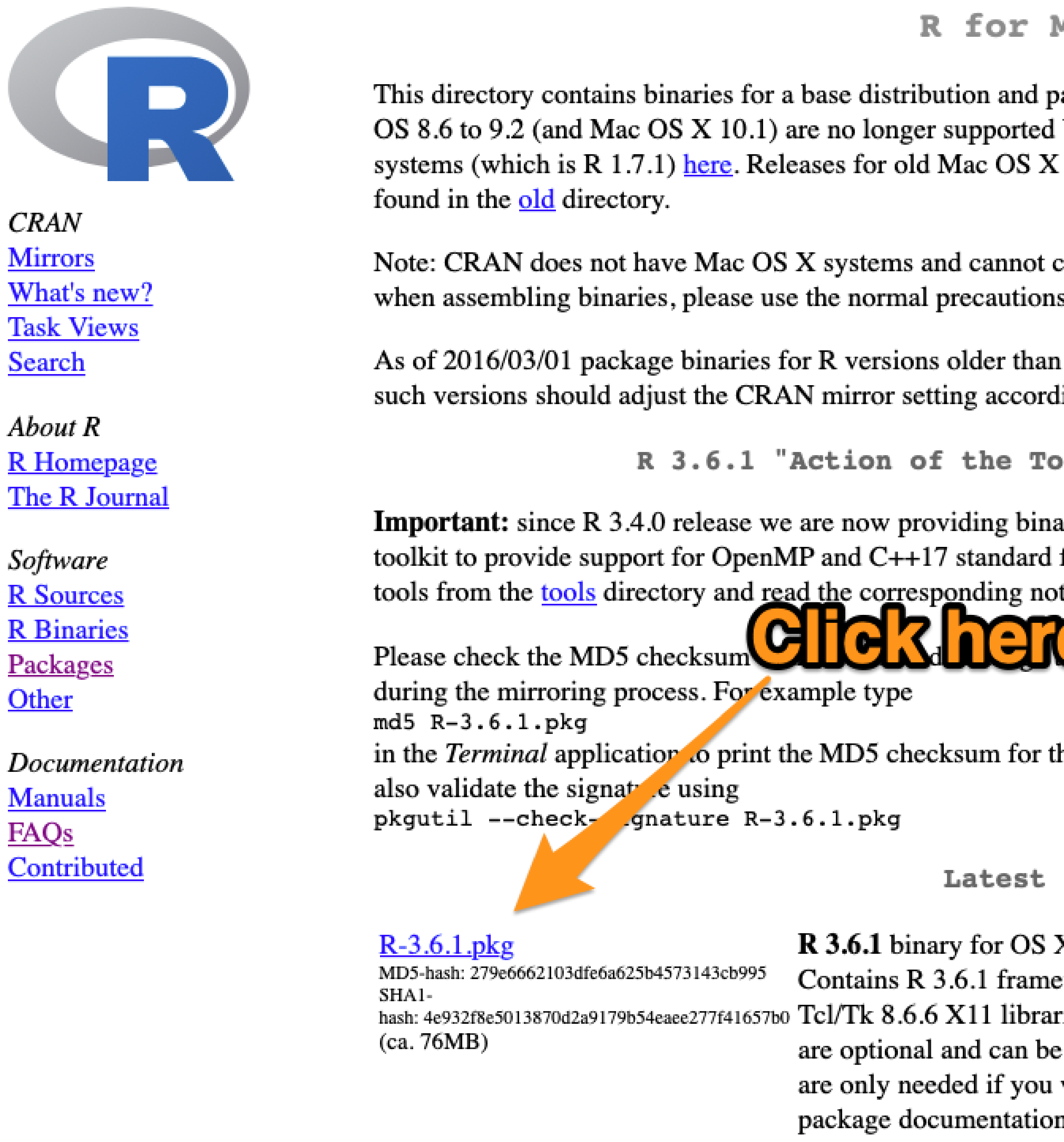
Source Code for all Platforms

Windows and Mac users most likely want to downlo
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 develo
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:



R for M

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

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

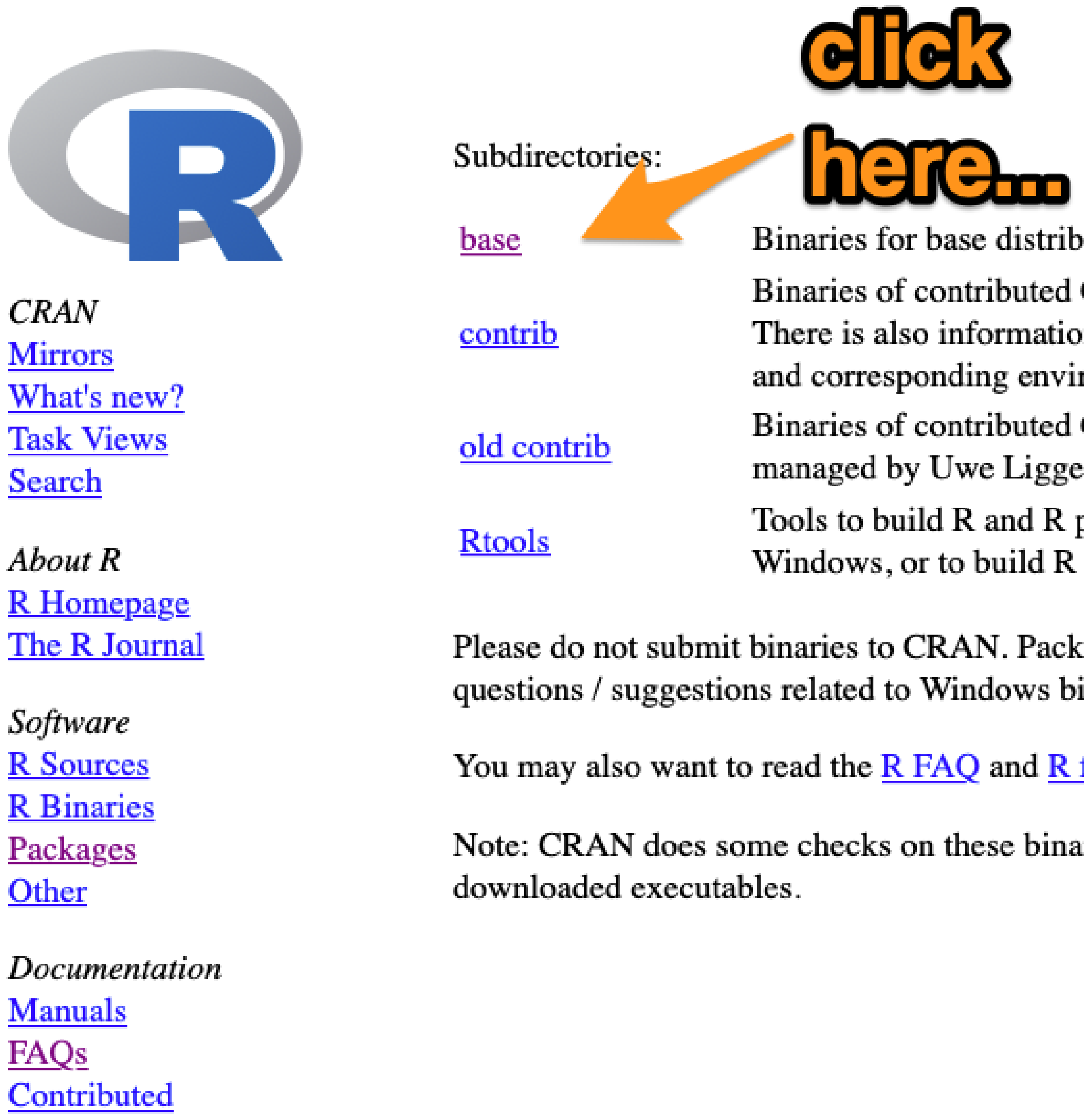
Please check the MD5 checksum d
during the mirroring process. For example type
`md5 R-3.6.1.pkg`
in the *Terminal* application to print the MD5 checksum for th
also validate the signature using
`pkgutil --check-signature R-3.6.1.pkg`

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

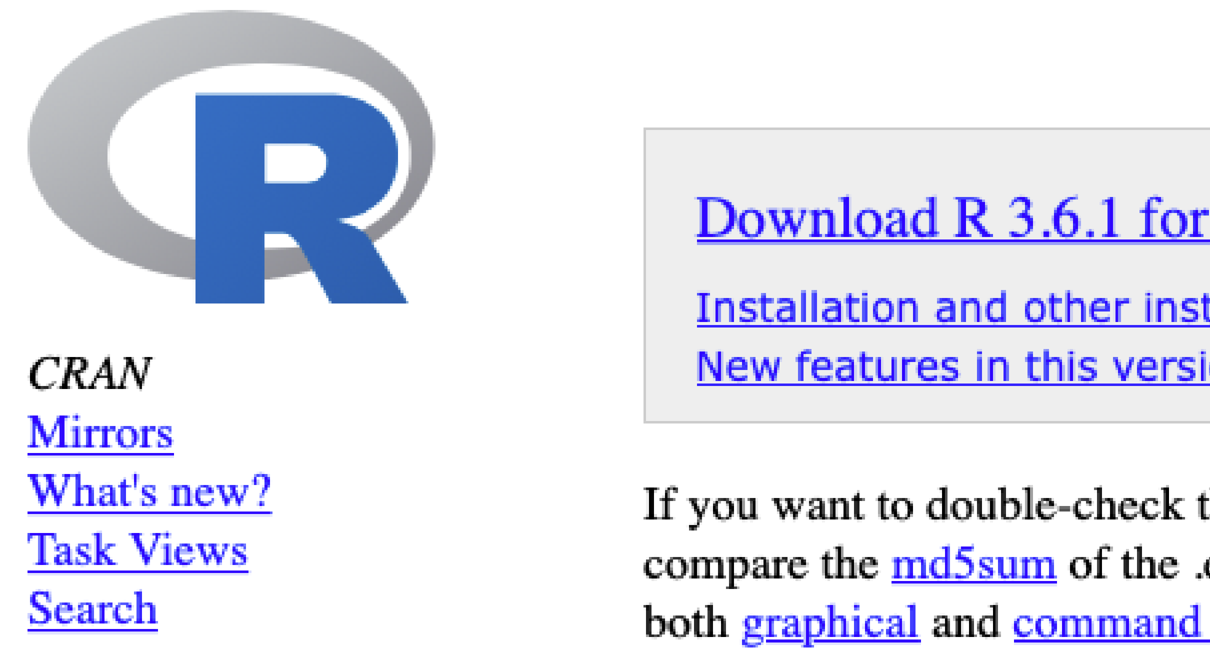
Subdirectories:

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.



CRAN
Mirrors
What's new?
Task Views
Search

Download R 3.6.1 for
Installation and other inst
New features in this versi

If you want to double-check t
compare the md5sum of the .
both graphical and command

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 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'[2].

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.

[2]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 this 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:

```r
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:

```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:

```r
remotes::install_github("mattcowgill/grattantheme", dependencies = TRUE, upgrade = "always")
```

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. We typically do this at the top of a script.

```r
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:

```r
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.

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 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.

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 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

[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.

of public debt on economic growth. It's still possible to make errors in a script-based analysis, but those errors are easier to find when the analysis is more transparent.

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 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 harder in SPSS, where data is primarily 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. * It has an active community extending its capabilities all the time and providing support online. * It is becoming the norm in academic research and common in the corporate world.

Everything you can do in Excel can be done in R.

Chapter 3

Organising an R project at Grattan

All our work, in R or otherwise, 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
- Naming your scripts and keeping them manageable
- Coding style at Grattan

3.1 Use RStudio projects, not `setwd()`

You'll almost always be reading and/or writing files to disk as part of your analysis in R. To do this, R needs to know where to read files from and save files to. By default, it uses your working directory.

One way to tell R which folder to use as your working directory is using the command `setwd()`, like `setwd("~/Desktop/some random folder")` or `setwd("C:\Users\mcowgill\Documents\Somerandomfolder")`. **This is a**

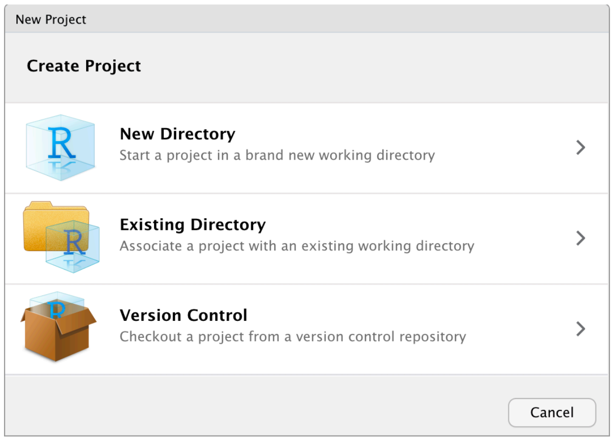
bad idea! If anyone - including you - tries to run your script on a different machine, with a different folder structure, it probably won't work. If people can't get past the first line when they're trying to run your script, there's an annoying and unnecessary hurdle to reproducing and checking your 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. 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!


New Project
Create Project
New Directory
Start a project in a brand new working directory
Existing Directory
Associate a project with an existing working directory
Version Control
Checkout a project from a version control repository
Cancel

RStudio will then create a file with an .Rproj extension in the folder you've chosen. When you want to work in this 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 Keep your stuff together

Your script(s), data, and output should generally all live in the same place. [1] That place should be the folder that contains the .Rproj file that was created when you created an RStudio project, and subfolders of that folder.

[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 your work, you should do it.

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.[2]

3.3 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.

3.4 Use relative filepaths

When you read or write files with R, don't use filepaths that are specific to your machine. For one thing, these machine-specific filepaths will fail when someone else tries to run your script. For another, they're super annoying! Who wants to type out a full filepath everytime you load or save a file?

Bad

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

[2]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.

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

```
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.

3.5 Keep your scripts manageable

Unless your project is very simple, it's probably 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

It should be clear what each script is trying to do. Use comments to explain why you're doing things!

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.

Don't include interactive work (like `View(mydf)`, `str(mydf)`, and commands like that) in your saved script. These type of commands should usually be entered straight into the R console, not in a script.

Chapter 4

Grattan coding style

This Grattan style guide is a condensed and lightly adapted version of the Hadley Wickham's `tidyverse` style guide, found at https://style.tidyverse.org/.

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

> 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** code-style elements, 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 projects.

4.1 Use meaningful filenames

You already know that you should break your analysis into manageable pieces, with one script per piece (remember?).

Your filenames should be meaningful, describing the 'point' of the script in a word or two:

Good

```
combine_data.R
run_regressions.R
```

Bad

```
analysis.r
stuff.r
```

If files need to be run in sequence, prefix them with numbers:

```
0_read_data.R
1_combine_data.R
2_explore.R
3_regression.R
4_visualisation.R
```

Don't create multiple versions of the same script (like `analysis_FINAL_002_MC.R` and `analysis_FINALFINAL_003_MC_WM.R`.) If you do end up with multiple versions, put everything other than the latest version in a subfolder, called "archive".

We set out more rules and guidelines for organising R analyses at Grattan in the `Organising_Grattan_R_projects` document that is found together with this guide.

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
```

Bad

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

If it's hard to concisely describe what your script does in a comment, that might be a sign that your script does too many things. Consider breaking your analysis into a series of scripts. See `Organising_Grattan_R_projects` for more on project organisation.

4.3 Comments

Comments are necessary where the code *alone* doesn't tell the full story. This is important when groups are coded with numbers rather than character strings:

Necessary to comment

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

Not necessary (but okay if included)

```
data %>%
  filter(gender == "Female",
         age >= 35 & age <= 39)
```

Comments can either go next to code, as in the example above, or they can precede the code, like this:

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

You should also include comments where your code is more complex and may not be easily understood by the reader.

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.

4.4 Breaking your script into parts

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

Good

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


# Read file B -----

# Merge files A and B ----
```

This helps you, and others, navigate your code better, using the navigation tool built in to RStudio. It also makes your code easier to read.

Chapter 5

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`.

As with filenames, Grattan primarily uses *words separated by underscores* `_` (aka 'snake_case') to name objects and variables. This is considered good 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.

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.

Chapter 6

Spacing

Giving you code room to breathe greatly helps readability for future-you and others who will have to read your code.

6.1 Assign and equals

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

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()
```

6.2 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)
```

6.3 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.

6.4 Short lines and line indentation

Tedious – yes – but short lines and consistent line indentation 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 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. 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))
```

Less good: 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))
```

Bad: long lines

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

War-crime bad: long lines without pipes

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

Chapter 7

Blocks of code

As shown above, 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.

Chapter 8

The tidyverse

8.1 What is the tidyverse and why do we use it?

8.2 An introduction to RMarkdown

Chapter 9

Data Visualisation

[intro]

9.1 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.

```r
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:

```r
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
```

9.2 Concepts

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

The main ingredients to a `ggplot` chart:

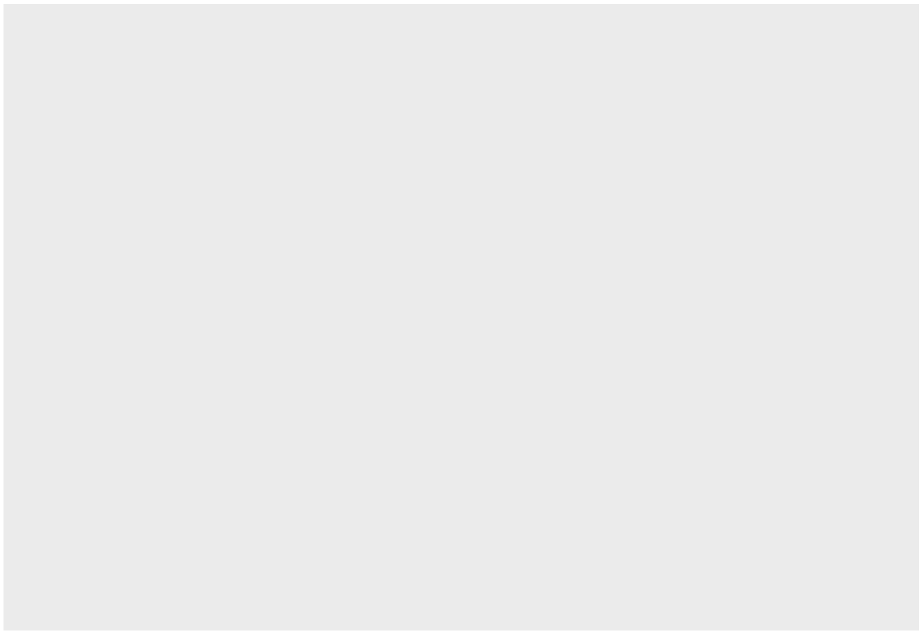
- **Data**: what data should be plotted. e.g. `data`
- **Aesthetics**: what variables should be linked to what chart elements. e.g. `aes(x = population, y = age)` to connect the `population` variable to the `x` axis, and the `age` variable to the `y` axis.
- **Geoms**: how the data should be plotted. 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 wth this data"). This produces an empty plot:

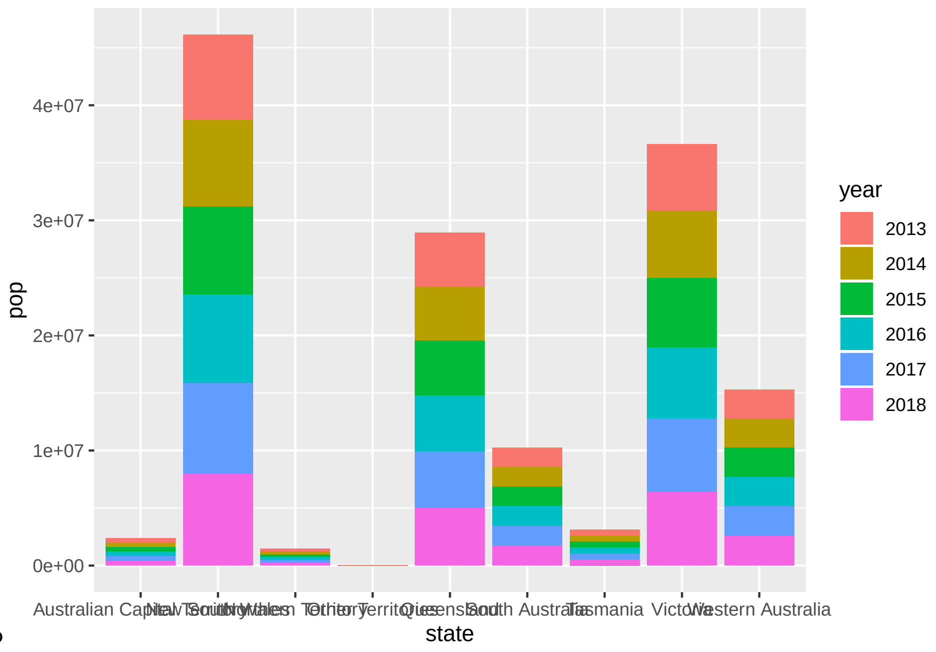
```
population_table %>%
        ggplot()
```

plot-1.bb 

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`:

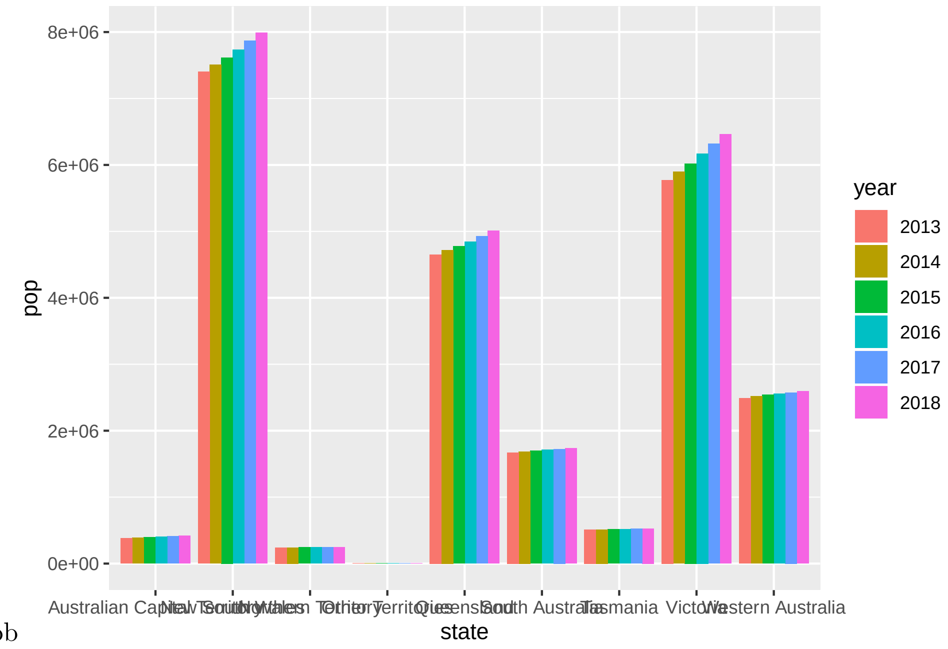
```
population_table %>%
        ggplot(aes(x = state,
                   y = pop,
                   fill = year)) +
        geom_col()
```



plot-1.bb

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 ontop of each other using `position = "dodge"`:

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



dodge-1.bb

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

9.3 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 palete for `colour` aesthetics.

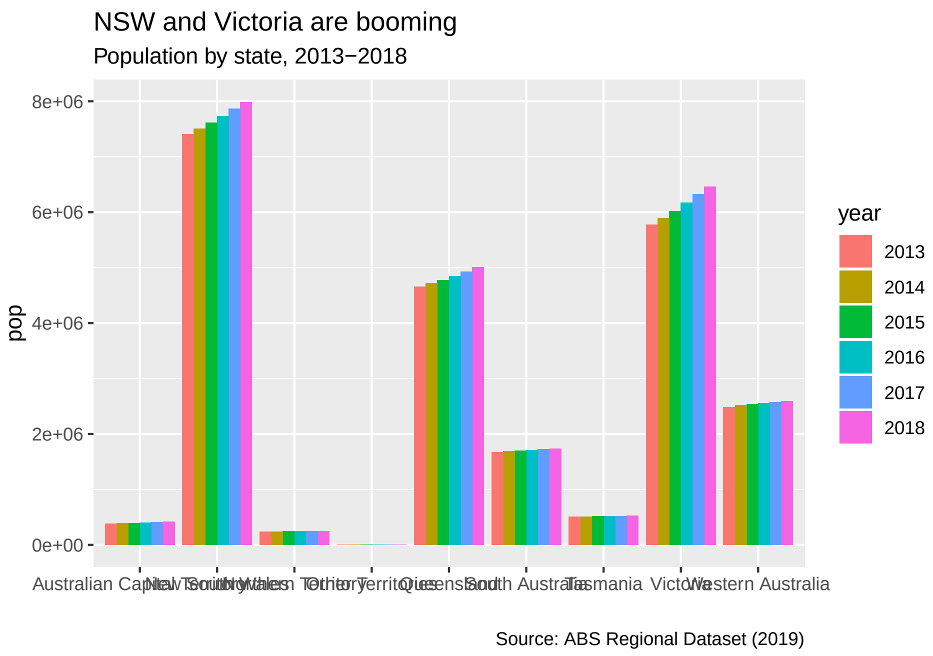
- `grattan_fill_continuous`: pulls colours from the Grattan colour palete for `fill` aesthetics.
- `grattan_save`: a save function that exports charts in correct report or presentation dimensions.

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

9.3.1 Making Grattan charts

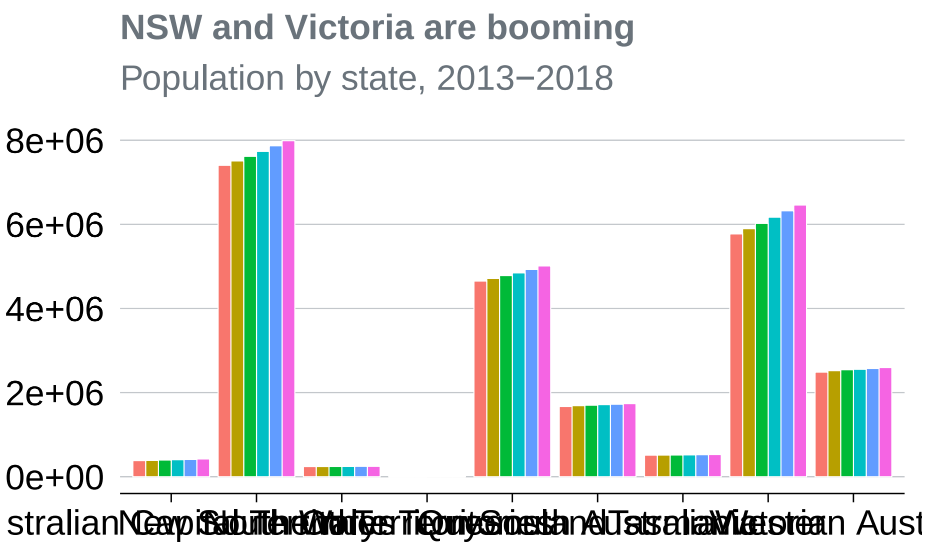
Start with a column chart, similar to the one made above:

```
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:

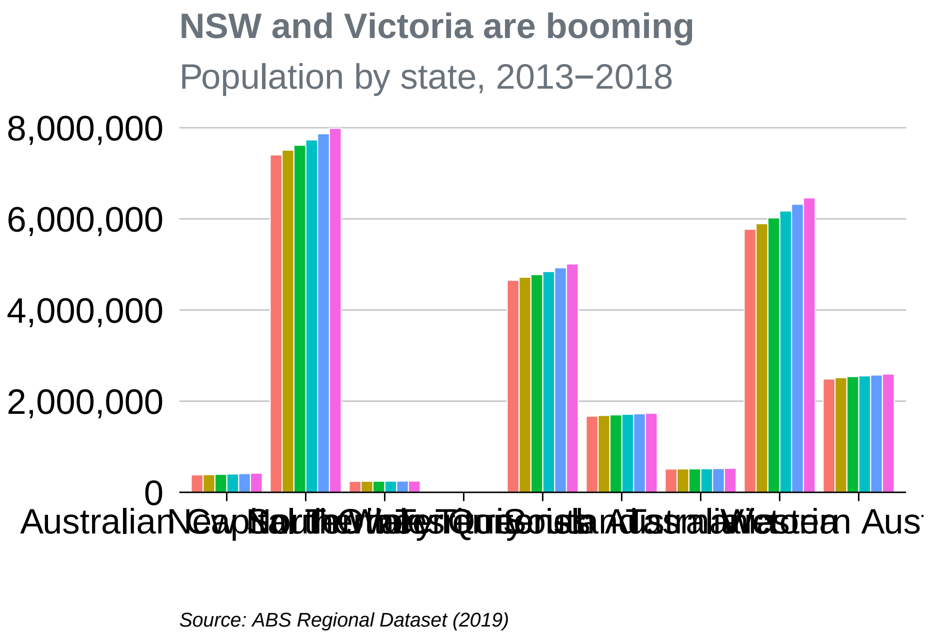
```
base_chart +
        theme_grattan()
```



Source: ABS Regional Dataset (2019)

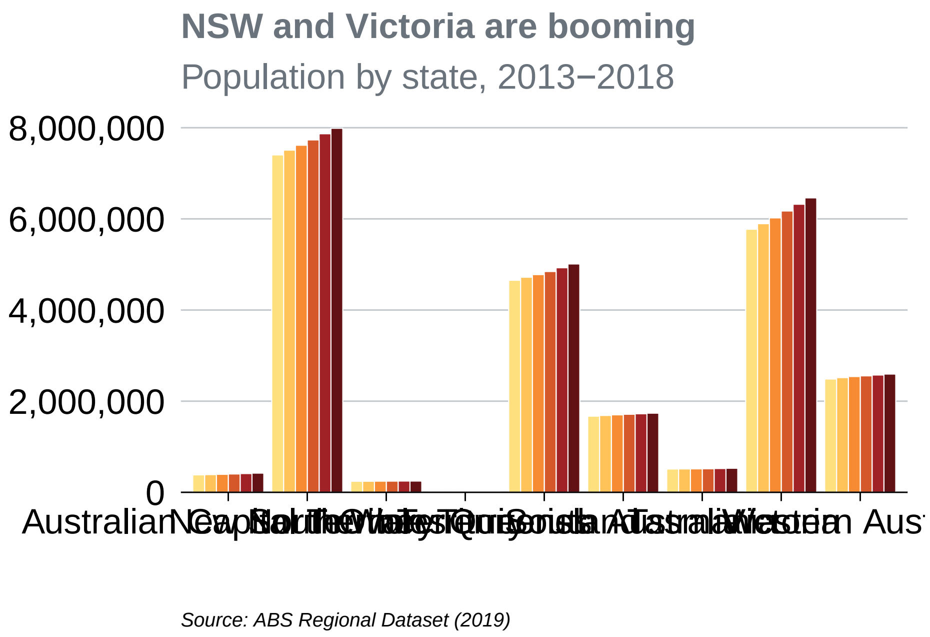
Then `grattan_y_continuous` to align the x-axis with zero. This function takes the same arguments as `scale_y_continuous`, so you can add `labels = comma()` to reformat the y-axis labels:

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



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

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

pop_chart
```



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

9.3.2 Saving Grattan charts

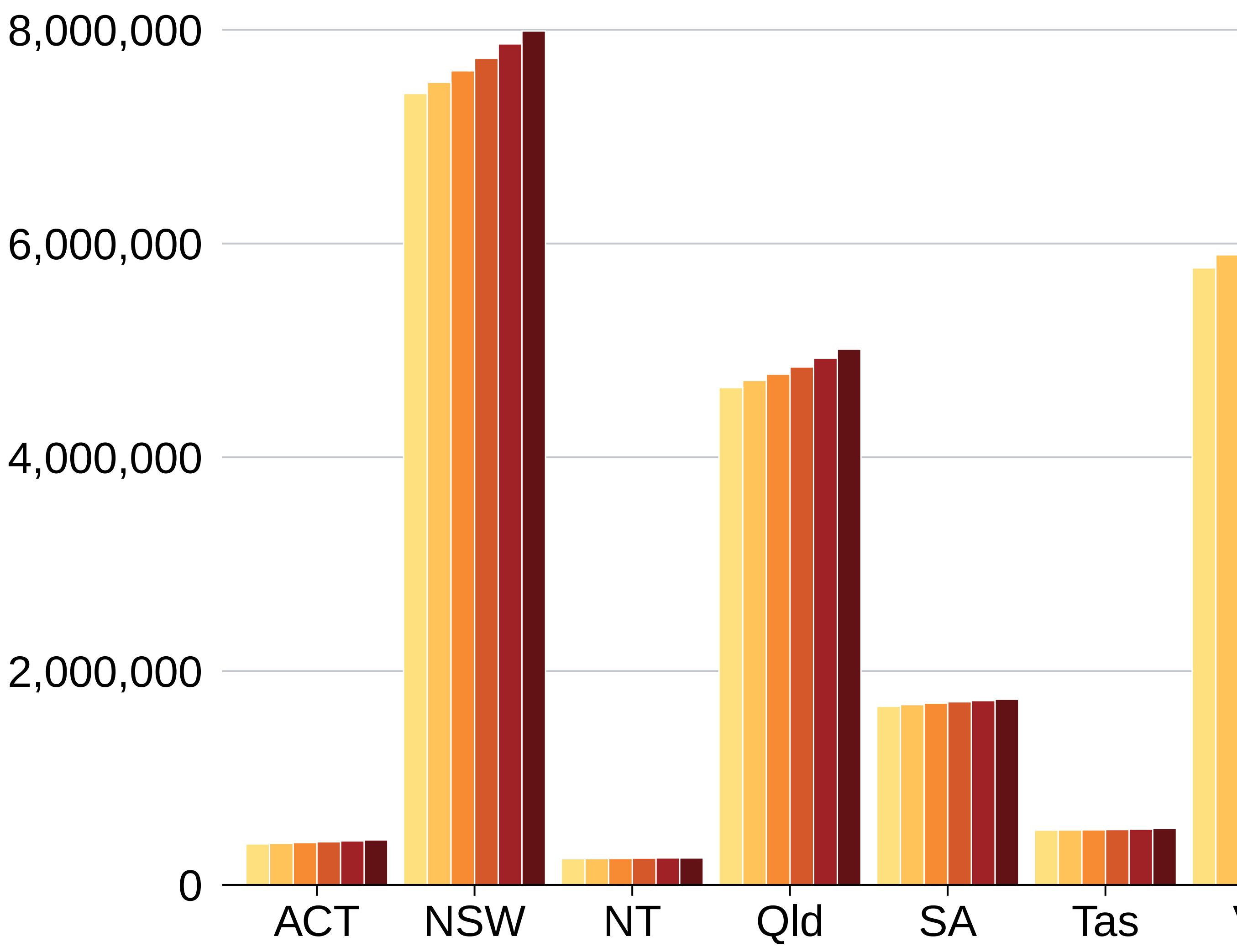
The `grattan_save` function saves your charts according to Grattan templates. It takes these arguments:

- `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.

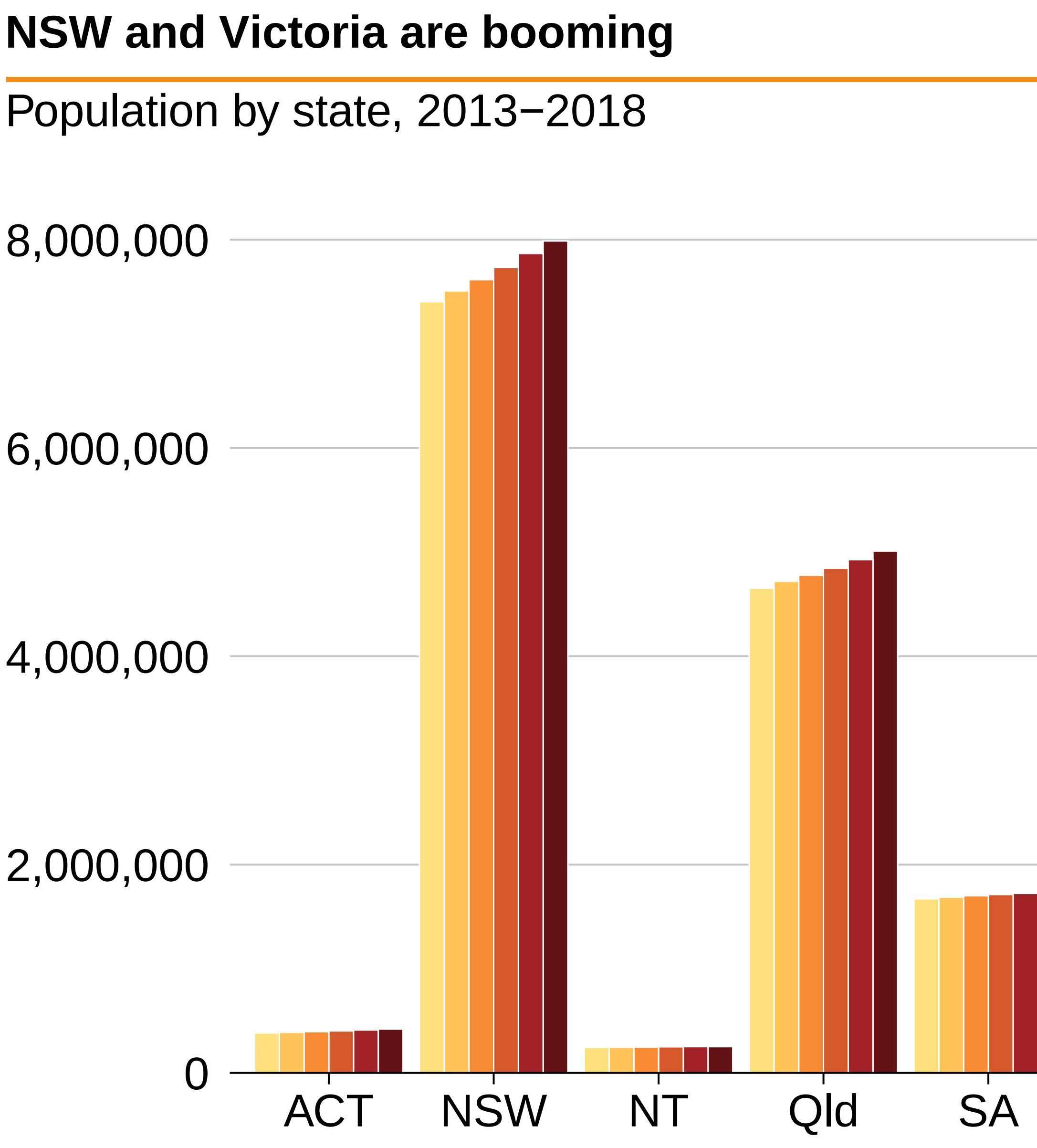
To save the `pop_chart` plot created above as a whole-column chart for a **report**:

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



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

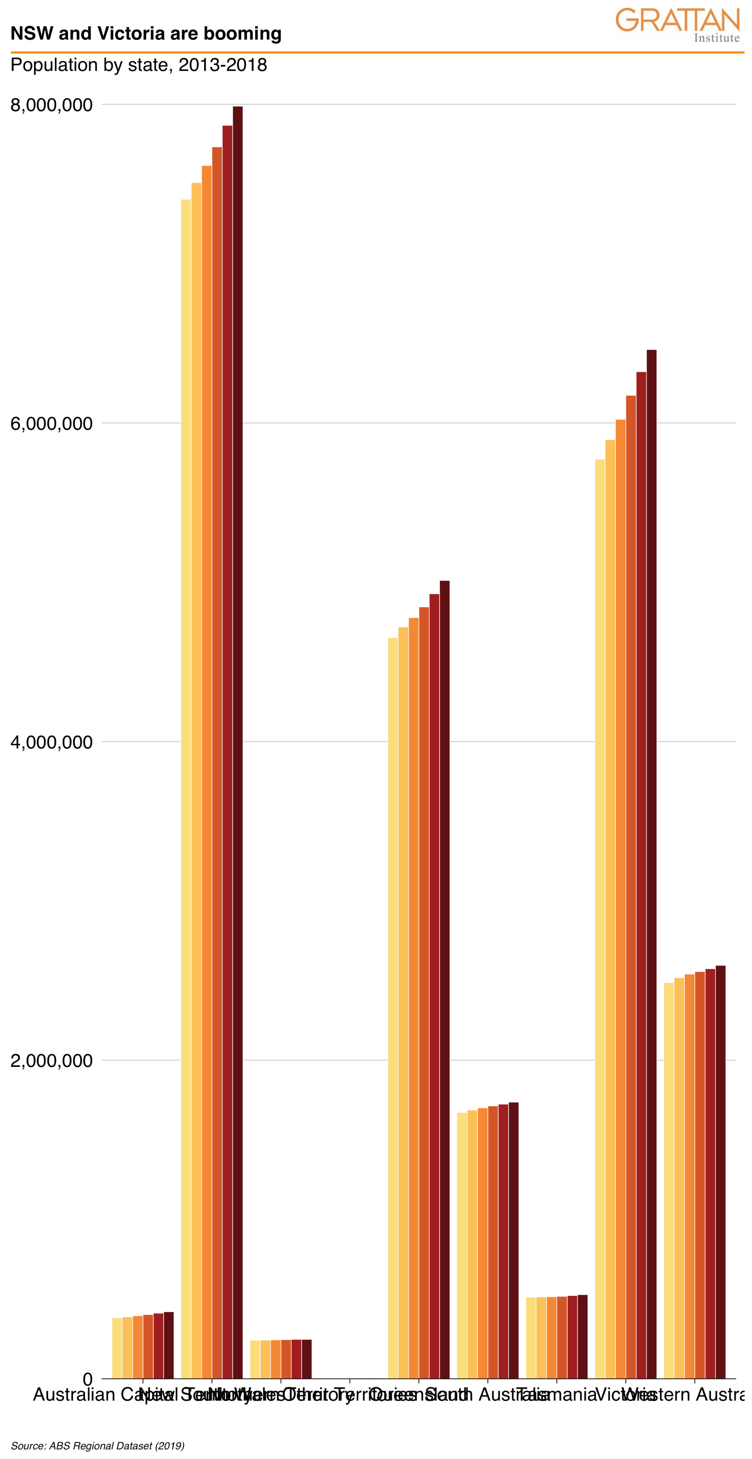
```
grattan_save("atlas/population_chart_presentation.pdf", pop_chart, type = "fullslide")
```



Source: ABS Regional Dataset (2019)

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)
```


NSW and Victoria are booming
GRATTAN Institute
Population by state, 2013-2018
8,000,000
6,000,000
4,000,000
2,000,000
0
Australian Capital Territory
New South Wales
Northern Territory
Other Territories
Queensland
South Australia
Tasmania
Victoria
Western Austra
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.

9.4 Chart cookbook

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

9.4.1 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:

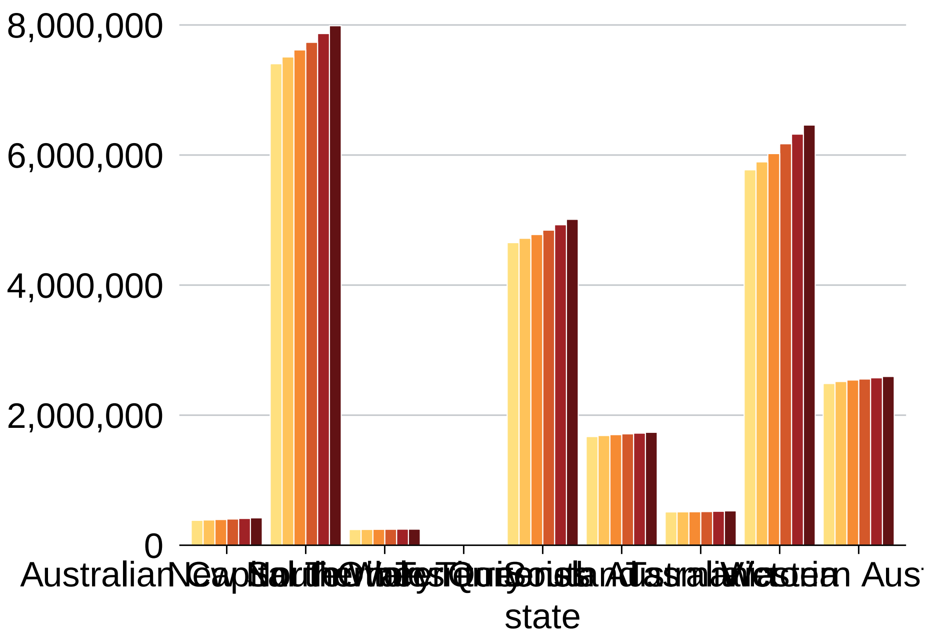
- `"count"`, the default: count the number of observations in a particular group, and plot that number. This is useful when you're using microdata. When this is the case, there is no need for a `y` aesthetic.
- `"sum"`: sum the values of the `y` aesthetic.
- `"identity"`: directly report the values of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

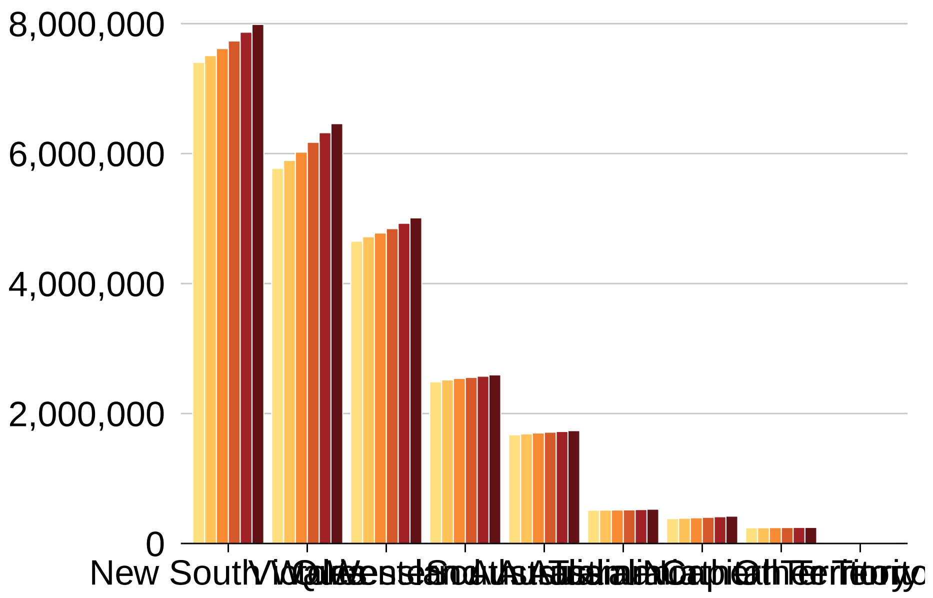
- `"stack"`, the default: bars in the same group are stacked atop one another.
- `"dodge"`: bars in the same group are positioned next to one another.
- `"fill"`: bars in the same group are stacked and all fill to 100 per cent.

```
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)
```



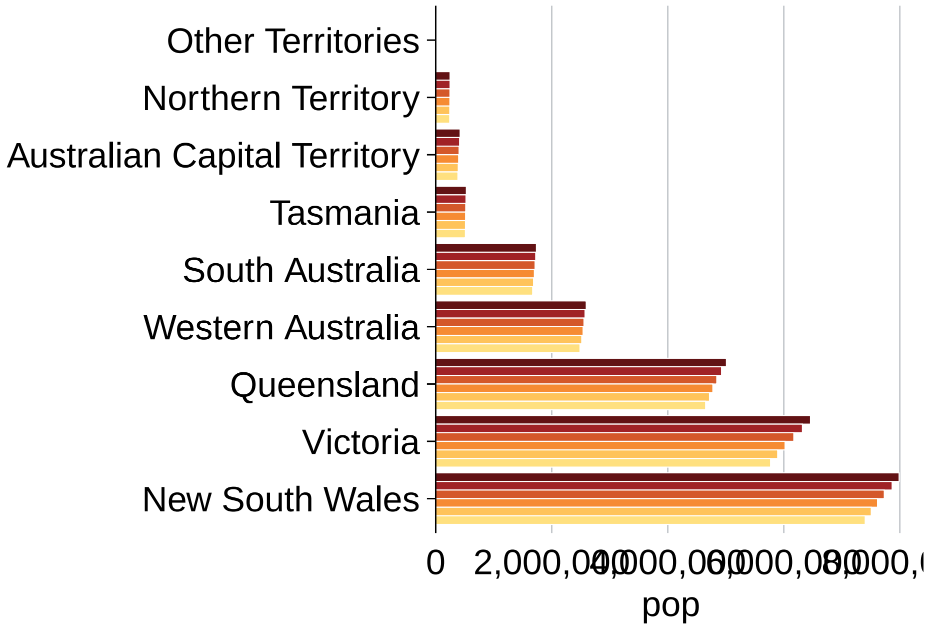
You can also **order** the groups in your chart by a variable. If you want to order states by population, use `reorder` inside `aes`:

```
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 = "")
```

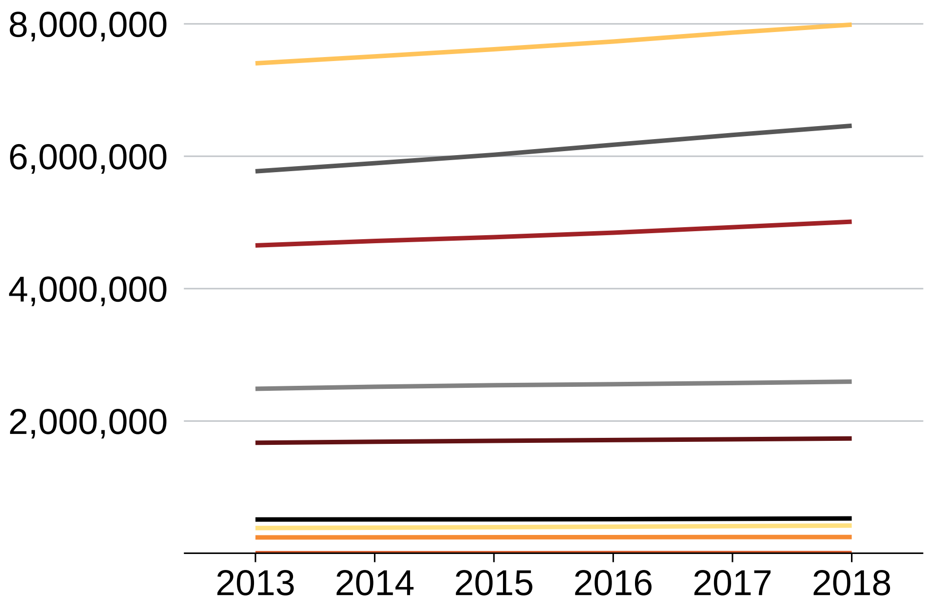


9.4.2 Line charts

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 = "")
```

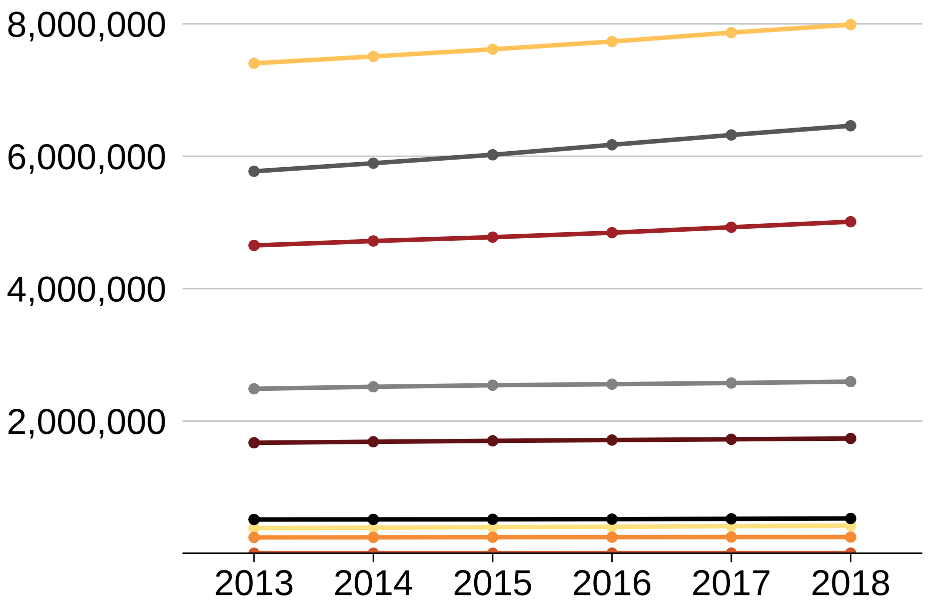
```
## Warning in grattantheme::grattan_pal(n = n, reverse = reverse, faded =
## faded): Using more than six colours is not recommended.
```



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

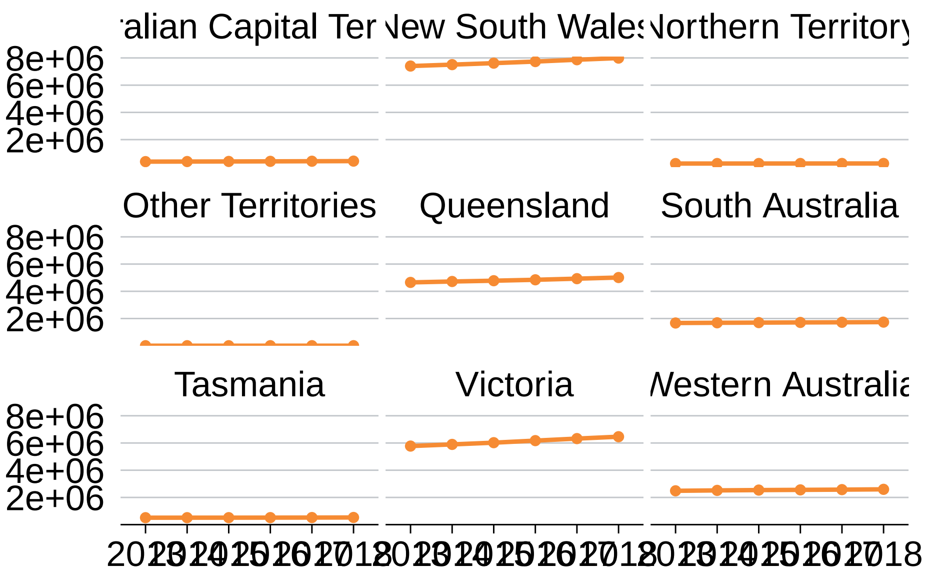
```
population_table %>%
        ggplot(aes(x = year,
                   y = pop,
                   colour = state,
                   group = state)) +
        geom_line() +
        geom_point(size = 2) +
        theme_grattan() +
        grattan_y_continuous(labels = comma) +
        grattan_colour_manual(9)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

```
## Warning in grattantheme::grattan_pal(n = n, reverse = reverse, faded =
## faded): 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:

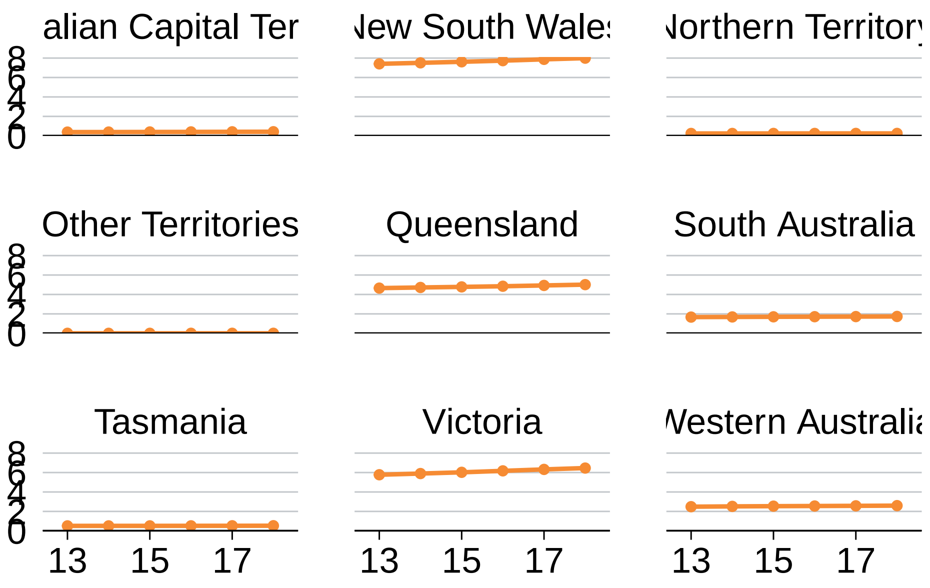
```
population_table %>%
        filter(state != "ACT",
               state != "NT") %>%
        ggplot(aes(x = year,
                   y = pop,
                   group = state)) +
        geom_line() +
        geom_point(size = 2) +
        theme_grattan() +
        grattan_y_continuous() +
    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



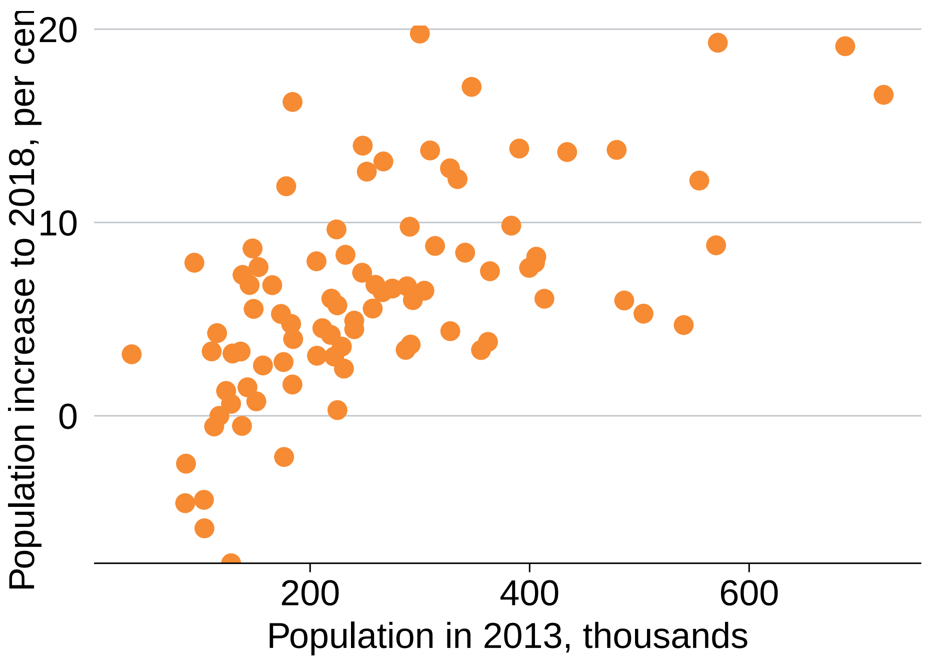
9.4.3 Scatter plots

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

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:

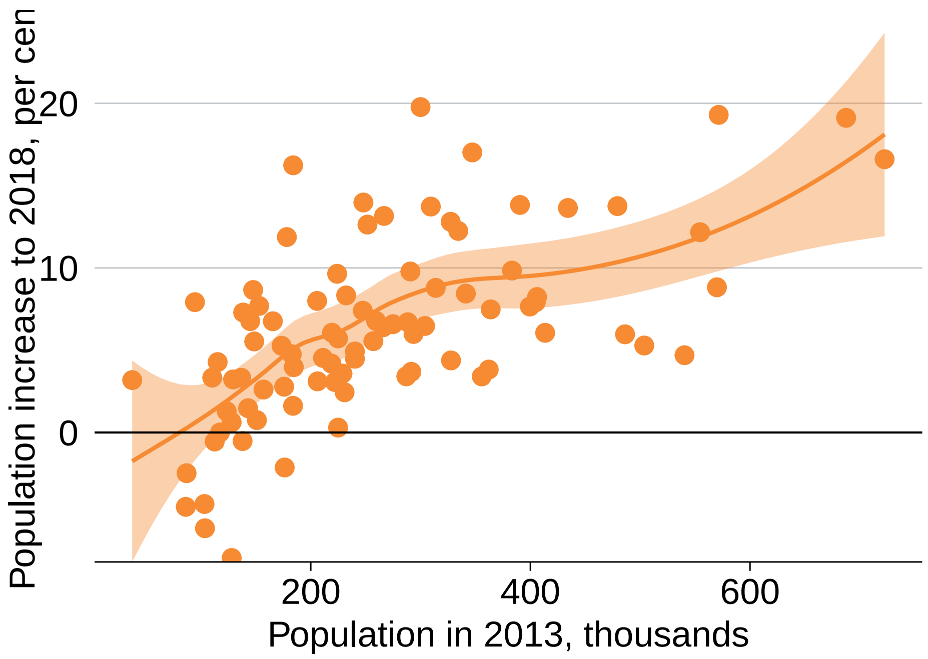
```
population_diff <- read_csv("data/population_sa4.csv") %>%
        mutate(state_long = state,
               state = strayr::strayr(state_long),
               pop = as.numeric(value),
               year = as.factor(glue::glue("y{year}"))) %>%
        filter(year %in% c("y2013", "y2018"),
               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")
```



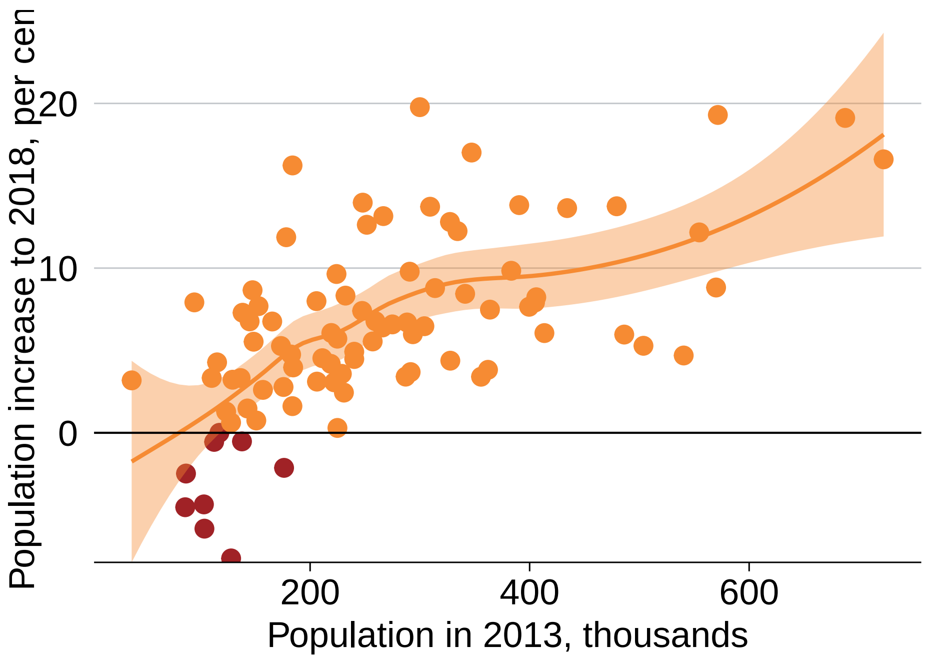
It looks like the areas with the largest population grew the most between 2013 and 2018. To explore the relationship further, you can add a line-of-best-fit with `geom_smooth`:

```
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")
```



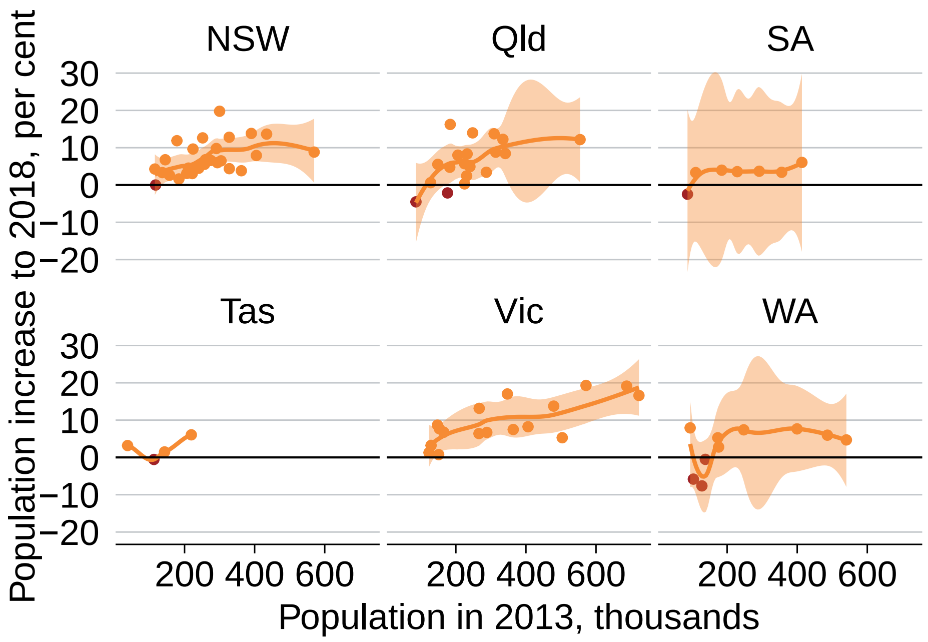
You could colour-code positive and negative changes from within the `geom_point` aesthetic. Making a change there won't pass through to the `geom_smooth` aesthetic, so your line-of-best-fit will apply to all data points.

```
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 ~ .)
```



9.4.4 Distributions

`geom_histogram` `geom_density`

`ggridges::`

9.4.5 Maps

9.4.5.1 `sf` objects

[what is]

9.4.5.2 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: `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/absmapsdata")
library(absmapsdata)

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

9.4.5.3 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.

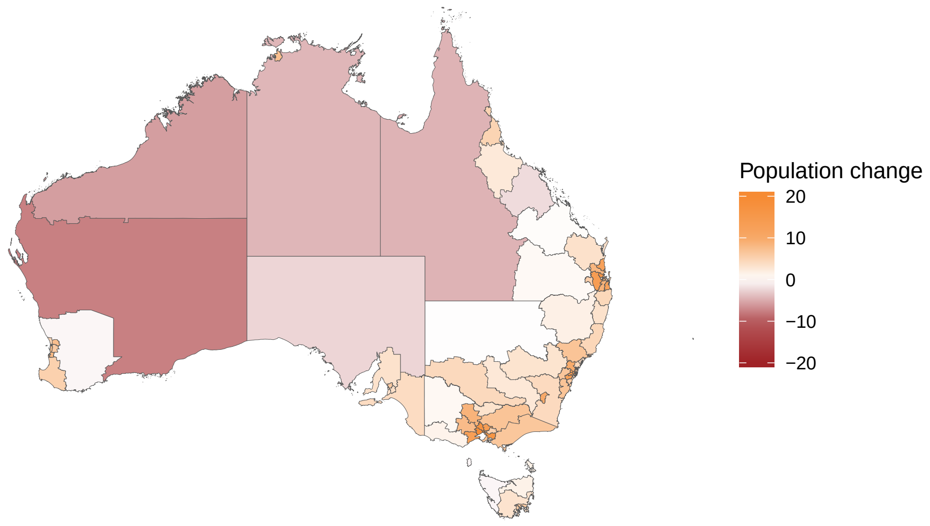
You can join the `sf` objects from `absmapsdata` to your dataset using `left_join`. The variable names might be different – eg `sa4_name` compared to `sa4_name_2016` – so use the `by` function to match them.

```
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`.

```r
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
```



9.5 Creating simple interactive graphs with `plotly`

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

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

```r
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 10

Reading data

10.1 Importing data

10.1.1 Reading CSV files

10.1.1.1 `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.

10.1.1.2 `data.table::fread()`

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

10.1.2 `readxl::read_excel()`

10.1.3 `rio`

10.1.4 `readabs`

10.2 Reading common files:

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

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

10.3 Appropriately renaming variables

As shown in the style guide

Add `rename_abs` function to a common Grattan package?

10.4 Getting to tidy data

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

Chapter 11

Different data types

11.1 Tidy data

Other data structures

11.2 Dates with `lubridate::`

The `lubridate::` package

11.3 Strings with `stringr::`

- Replacing values
- Matching values
- Separating columns

11.4 Factors with `forcats::`

- Dangers with factors

Chapter 12

Data transformation

12.1 The pipe

12.2 Key `dplyr` functions:

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

12.3 Filter with `filter()`

12.4 Arrange with `arrange()`

12.5 Select variables with `select()`

12.6 Group data with `group_by()`

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

12.7.1 Cases when you should use `case_when()`

12.8 Summarise data with `summarise()`

12.9 Joining datasets with `*_join()`

Chapter 13

Analysis

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...