Nicholas Tierney

R Package Essentials

Table of contents

About this

This is a book on the essential components of creating an R package. It is aimed at those who want to learn how to make R packages. You probably have written some functions, but if you haven't, we discuss how to do that. I care a lot about writing functions, and have a lot of thoughts and ideas on how to do it.

It was initially developed as a full-day hour workshop, "R package essentials". It is a developed into a resource that will grow and change over time as a **living book**.

This book aims to teach the following:

- Installation and setup of dependencies
  - git + github
  - R, RStudio
  - package dependencies
- Function essentials
  - DRY;DRY (Don't Repeat Yourself; Don't Reread Yourself)
  - Expression
  - Finding the inputs
- Moving a script to a series of functions
- Create package barebones with `create_package()`
- How to add dependencies with `use_package()` - DESCRIPTION file
- How to add documentation with `roxygen2`
- Why you should use R CMD Check
- How to add data to a package
- How to add a README
- How to put your package on github
- How to add vignettes
- Writing tests
- Using a NEWS file
- Adding a website
- Using Continuous Integration to check and test
- Publishing your software on R universe

Getting course materials

Course materials are in the github repository njtierney/learned. These can be downloaded by using the following command from the `usethis` package:

```
usethis::use_course("njtierney/learned")
```

Licence

License

1 Philosophy

I first learnt to write an R package from [Hilary Parker's famous blog post, "Writing an R package from scratch". Then I consulted Hadley Wickham's "R packages" book (1st edition). I consider the "R packages" book (now in its second edition, by Hadley Wickham and Jenny Bryan), to be the authority on best practices for package development, alongside the rOpenSci guide, "rOpenSci Packages: Development, Maintenance, and Peer Review".

These are excellent pieces of reference test, however I think there is a need for a resource that sits somewhere between a blog post on making an R package, and resource. I want something that contains **just enough** information to get you started on the right path to making an R package. This is what that book represents to me. Along the way I'll include breadcrumbs to other resources to look into when you want to learn more.

This book also represents my efforts to explain the key parts of what I think people should know about how to write functions, and also to format this in a teachable way that can be covered in a single workshop.

So, why write a book?

Similar to my book, "Quarto for Scientists", writing this as a book provides a nice way to structure the content in the form of a workshop, in a way suitable for learning in a day. It is not to say that there aren't already the resources out there; there are. It is instead adding to the list of other (useful, hopefully!) information out there on the internet. To answer a question with another question: "Why NOT write this as a book?"

2

Installation

In this section, the aim is to have everyone setup with R, RStudio, the tools you need to build an R package, and **git**.

2.1 Overview

- **Duration** 15 minutes

2.2 Questions

- How do I install R?
- How do I install RStudio
  - What about Positron?
- How do I install git?
- How do I install RTools?

2.3 Software Setup

2.3.1 Installing R

2.3.1.1 Windows

https://cloud.r-project.org/bin/windows/

2.3.1.2 MacOS

https://cloud.r-project.org/bin/macosx/

2.3.1.3 Linux

https://cloud.r-project.org/bin/linux/

2.3.2 Installing RStudio

https://posit.co/download/rstudio-desktop/#download

2.3.3 Installing R packages for development

To ensure you are up to date, run the following script to install the packages.

```r
install.packages(c("devtools", "roxygen2", "testthat", "knitr", "pak"))
```

2.3.3.1 Personalising your R Profile

This is really neat, and I think it's actually worthwhile doing, but it does take up some time, and there are some warnings.

As you develop R packages, you'll need to go through a cycle of restarting R, and loading things up to be ready. One of the issues with this is that you'll find yourself writing code like:

```r
library(devtools)
```

A lot. To save you time, we can edit a very special file called "The R profile", which is saved as `.RProfile`. This code is special, and awesome, because it is run *every time you start R*. It is also dangerous, for exactly the same reason.

I recommend running the following code from devtools to help set this up:

```r
use_devtools()
```

Which will bring up the following message:

```
 Include this code in .Rprofile to make devtools
  available in all interactive sessions:
  if (interactive()) {
    suppressMessages(require(devtools))
  }
  [Copied to clipboard]
 Modify /Users/nick/.Rprofile.
 Restart R for changes to take effect.
```

So, copy and paste the above, which I will now explain. There are three parts to this that I will break down:

```r
require(devtools)
```

we usually recommend writing `library(devtools)`, but in this instance, `require` is what we want, because if the package is not installed, `require` will throw a warning, rather than an error:

```
# warn
require(whatevenisthis)
```

```
Loading required package: whatevenisthis

Warning in library(package, lib.loc = lib.loc, character.only = TRUE,
logical.return = TRUE, : there is no package called 'whatevenisthis'
```

```
# error
library(whatevenisthis)
```

```
Error in library(whatevenisthis): there is no package called 'whatevenisthis'
```

We do not want an error when we start R, it is annoying.

```
suppressMessages()
```

This code suppresses any messages that appear from running this code, which again, we want, because we don't (generally) want our R session to announce something upon startup.

```
  if (interactive()) {
    suppressMessages(require(devtools))
  }
```

This means that this code is only run if the R session is interactive. This always felt a bit strange to me - because I had only ever run R interactively. But you don't want to run `require(devtools)` when we aren't using R interactively, because it means we are potentially changing the state of things. Essentially, it's good practice.

Also, here are a couple of times that you might not realise you are using R non-interactively:

- rendering a document using quarto or rmarkdown
- building an R package (which you'll learn about later)

You also use R non-interactively when you are running `Rscript` in the command line.

Finally, another bit of useful code in your R profile is something like this:

```
# usethis options
options(
  usethis.full_name = "Nicholas Tierney",
  usethis.protocol = "https",
  usethis.description = list(
    `Authors@R` = '
    c(
      person(
        given = "Nicholas",
        family = "Tierney",
        role = c("aut", "cre"),
        email = "nicholas.tierney@gmail.com",
        comment = c(ORCID = "https://orcid.org/0000-0003-1460-8722")
      )
    )',
    License = "MIT + file LICENSE",
    Language = "en-GB",
    Version = "0.0.0.9000"
  ),
  # set SI to true
  reprex.session_info = TRUE
)
```

This helps when setting up your R package for the first time, to make sure you set up your DESCRIPTION file. It isn't required, but it is neat, and I think worthwhile.

Because I need to set these things up on different laptops sometimes, I actually write all these files to github. They are typically called "dotfiles" - you can see mine at http://github.com/njtierney/dotfiles.

2.3.4 git and github

Very briefly, `git` is essentially a way of managing versions and changes. You can think of it like a product such as dropbox, but with super powers. You can go back in time, you can make copies for changing, and delicately and precisely mege them back in, or leave them where they are.

Your software needs a home. You'll typically start with your project on your laptop or computer. GitHub is where you can store it online. The benefits to sharing your work on github are many, but my personal top reasons are:

- Build trust in your software. If the community can see your code, they can trust it better.
- Provides a way to log ideas and bugs via issues.
- Provides a way for the community to contribute to your code.

My favourite book on using git and github with R is the book "happy git with R" By Jenny Bryan, Jim Hester, and the Stat 545 TAs. Honestly, it's hard to recommend better installation instructions than their battle tested ones, so I'll point you to this resource in case you run into troubles here.

2.3.4.1 setting up github

Getting set up on github you need an account. It's easy enough to set up - go to https://github.com/ . When picking a username, I recommend the following:

1. Keep it short. `jsmith` is better than `jonathansmith`.
2. Avoid numbers and jokes. `jsmith` is better than `jsmith123`
3. Keep it professional. `jsmithisthebest`
4. Keep it lowercase `DONOTSHOUT`

2.3.4.2 installing git

Installing git can sometimes be a challenge. This is largely because sometimes there are small differences that arise to install windows vs mac vs linux. Or sometimes there are issues with work computers with strong permissions.

Generally, you should install git from: https://git-scm.com/downloads

But, if you encounter issues, I would advise checking out the battle-tested instructions at: https://happygitwithr.com/install-git.

Once you've installed git, I recommend running this:

```r
usethis::git_vaccinate()
```

Which ensures that you ignore specific files (specifically, Rproj.user, .Rhistory, .Rdata, .httr-oauth, .DS_Store, and .quarto). This is important because it decreases your chances of leaking credentials or other important details to GitHub.

2.3.4.3 The "git handshake"

In order for your computer to talk to git and github properly, it needs to know three things:

1. Name
2. Email
3. Credentials

git needs to know your name and email - this should be the name and email you used to set up your github account. Set this up with `use_git_config()`

```r
library(usethis)
use_git_config(
  user.name = "Ned Kelly",
  user.email = "ned@example.org"
)
```

github needs a personal access token - this is so you can talk to github from

R. This becomes really handy, and dare I say it, nearly magical later on. To get this, run:

```r
usethis::create_github_token()
```

This will open up GitHub and create a Personal Access Token. If this doesn't work, go to https://github.com/settings/tokens and click "Generate New Token", and select the (classic)."

Generally speaking you want the following scopes selected:

- "repo"
- "user"
- "workflow".

A token will be created - keep this page open, and copy the token to your clipboard.

Then, go to R, and run:

```r
gitcreds::gitcreds_set()
```

And paste this PAT code in. Then, verify all of this with:

```r
usethis::git_sitrep()
```

2.3.5 Installing RTools

This is actually something that you only need to do if you want to use C or C++ with your R package, which isn't something you need to do for this course. To read more on this, see "The R build toolchain" from the R Packages book.

3

RStudio, What and Why

(This section is also in my other book, "Quarto for Scientists")

3.1 Overview

- **Teaching** 5 minutes
- **Exercises** 2 minutes

3.2 Questions

- What is RStudio?
- Why should I use RStudio?
- What features should I change?

3.3 Objectives

- Get familiarised with RStudio
- Get set up with not storing the RStudio workspace
- Download the course materials for the workshop

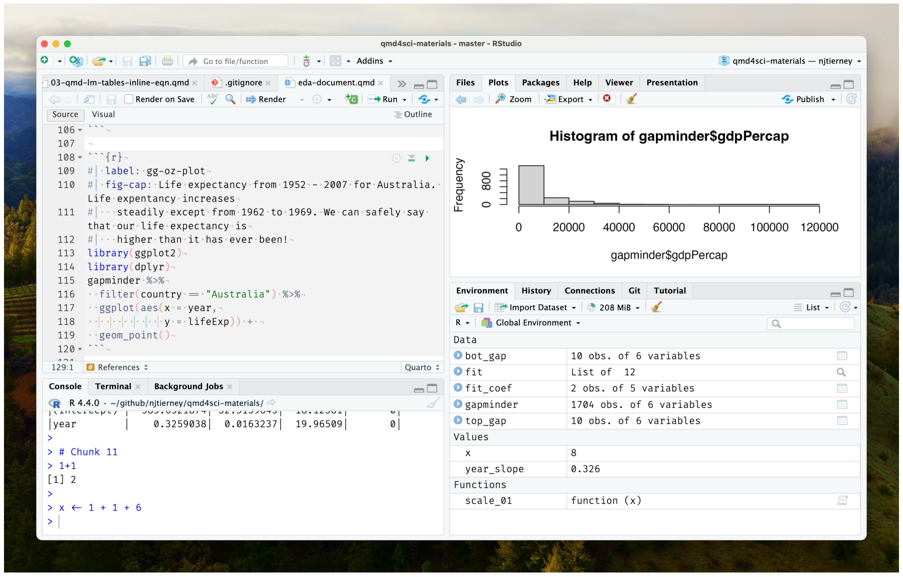
3.4 What is RStudio, and why should I use it?

If R is the engine and bare bones of your car, then RStudio is like *the rest of the car*.

The engine is super critical part of your car. But in order to make things properly functional, you need to have a steering wheel, comfy seats, a radio, rear and side view mirrors, storage, and seatbelts. RStudio is all those niceties

The RStudio layout has the following features:

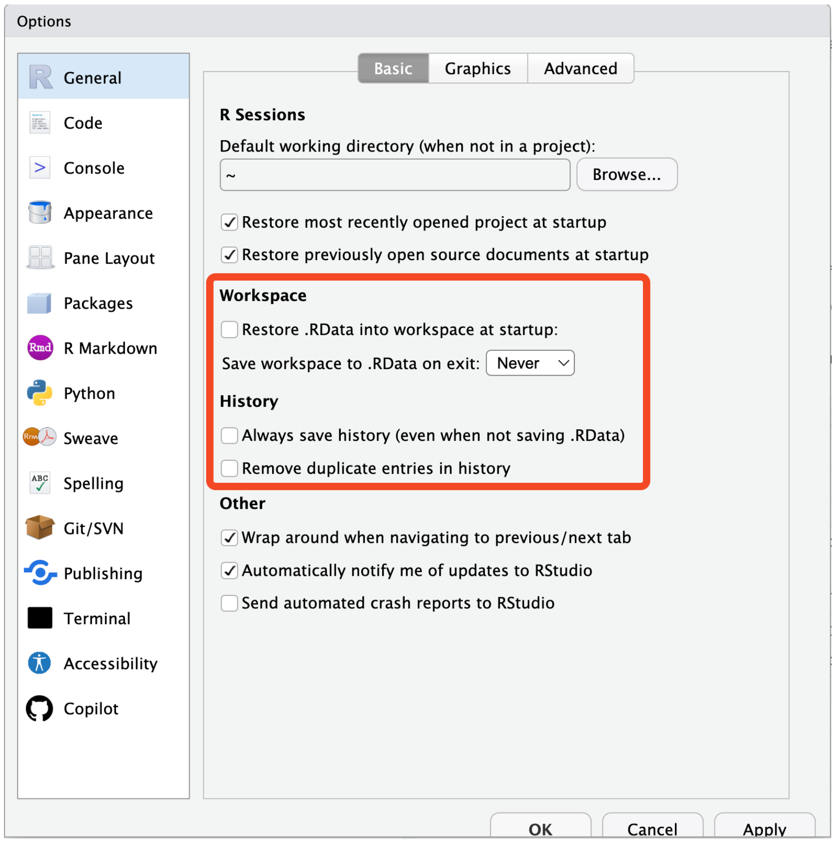
- On the upper left, the Quarto script
- On the lower left, the R console
- On the lower right, the view for files, plots, packages, help, and viewer.
- On the upper right, the environment / history pane



```
```

```{r}
#| label: gg-oz-plot
#| fig-cap: Life expectancy from 1952 - 2007 for Australia. Life expentancy increases
#|   steadily except from 1962 to 1969. We can safely say that our life expectancy is
#|   higher than it has ever been!
library(ggplot2)
library(dplyr)
gapminder %>%
  filter(country == "Australia") %>%
  ggplot(aes(x = year,
             y = lifeExp)) +
  geom_point()
```
```

```
|year        |    0.3259038|  0.0163237|  19.96509|        0|
>
> # Chunk 11
> 1+1
[1] 2
>
> x <- 1 + 1 + 6
>
```

```
Data
bot_gap       10 obs. of 6 variables
fit           List of  12
fit_coef      2 obs. of 5 variables
gapminder     1704 obs. of 6 variables
top_gap       10 obs. of 6 variables
Values
  x           8
  year_slope  0.326
Functions
  scale_01    function (x)
```

Figure 3.1: A screenshot of the RStudio working environment.

We saw a bit of what an Quarto script does.

- The R console is the bit where you can run your code.
- The file/plot/package viewer is a handy browser for your current files, like Finder, or File Explorer.
- Plots are where your plots appear, you can view packages, see the help files.
- The environment / history pane contains the list of things you have created, and the past commands that you have run.

ℹ Your Turn: RStudio default options

To first get set up, I highly recommend changing the following setting Tools > Global Options (or `Cmd + ,` on macOS)
Under the **General** tab:
```

- For **workspace**:
  - Uncheck restore .RData into workspace at startup.
  - Save workspace to .RData on exit : "Never".
- For **History**:
  - Uncheck "Always save history (even when not saving .RData).
  - Uncheck "Remove duplicate entries in history".



R Sessions

Default working directory (when not in a project): ~

☑ Restore most recently opened project at startup

☑ Restore previously open source documents at startup

Workspace

☐ Restore .RData into workspace at startup:

Save workspace to .RData on exit: Never

History

☐ Always save history (even when not saving .RData)

☐ Remove duplicate entries in history

Other

☑ Wrap around when navigating to previous/next tab

☑ Automatically notify me of updates to RStudio

☐ Send automated crash reports to RStudio

Figure 3.2: Setting the options right for RStudio, so you don't restore previous sessions work, and don't save it either.

This means that you won't save the objects and other things that you create in your R session and reload them. This is important for two reasons

1. **Reproducibility**: you don't want to have objects from last week cluttering your session

2. **Privacy**: you don't want to save private data or other things to your session. You only want to read these in.

Your "history" is the commands that you have entered into R. Additionally, not saving your history means that you won't be relying on things that you typed in the last session, which is a good habit to get into!

3.5 Learning more

- RStudio IDE cheatsheet

4 Workflow

(Note that this section is borrowed from my book, Quarto for Scientists: "workflow")

> Before we start with Quarto, we need to make sure that you understand *file storage hygiene*.

We can prevent **unexpected problems** if we can maintain an order to your files, paths, and directories. A common problem that arises is R not knowing where a certain file is. For example, we get the error:

```r
read.csv("my-very-important-data-file-somewhere.csv")
```

```
Warning in file(file, "rt"): cannot open file
'my-very-important-data-file-somewhere.csv': No such file or directory
```

```
Error in file(file, "rt"): cannot open the connection
```

Because R doesn't know where `"my-very-important-data-file-somewhere.csv"` is.

Practicing *good file storage hygiene* will help maintain an order to files, paths, and directories. This will make you more productive in the future, because you'll spend less time fighting against file paths.

Not sure what a file path is? We explain that as well.

4.1 Overview

- **Teaching** 10 minutes
- **Exercises** 10 minutes

4.2 Questions

- Where should I put all my files?
- What is an RStudio project, anyway?
- What is a file path?

4.3 Objectives

- Understand what a file path is
- Set up an RStudio Project to organise your work
- Put some data in your project to set up the next tasks

Your Turn

In groups of 2-4 discuss:

1. What your normal “workflow” is for starting a new project
2. Possible challenges that might arise when maintaining your project

4.4 When you start a new project: Open a new RStudio project

This section is heavily influenced by Jenny Bryan’s great blog post on project based workflows.

Sometimes this is the first line of an R Script or R markdown file.

```r
setwd("c:/really/long/file/path/to/this/directory")
```

Question

What do you think the `setwd` code does?

4.4.1 So what does this do?

This says, “set my working directory to this specific working directory”.

It means that you can read in data and other things like this:

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

Instead of

```
data <- read_csv("c:/really/long/file/path/to/this/directory/data/mydata.csv")
```

So while this has the effect of **making the file paths work in your file**, it is a problem. It is a problem because, among other things, using `setwd()` like this:

- Has 0% chance of working on someone else's machine (**this could include you in 6 months!**)
- Your file is not self-contained and portable. (Think: *"What if this folder moved to /Downloads, or onto another machine?"*)

So, to get this to work, you need to hand edit the file path to your machine.

This is painful.

When you do this all the time, it gets old, fast.

4.5 What is a file path?

This might all be a bit confusing if you don't know what a file path is. A file path is the machine-readable directions to where files on your computer live. So, the file path:

`/Users/njtierney/Desktop/qmd4sci-materials/demo.R`

Describes the location of the file "demo.R". This could be visualised as:

```
users
    njtierney
        Desktop
             qmd4sci-materials
                 demo.R << THIS IS THE FILE HERE
                 exercises
                 exploratory-data-analysis
                     eda-document.qmd
                     eda-script.R
                  data
                       gapminder.csv
```

So, if you want to read in the `gapminder.csv` file, you might need to write code like this:

```
gapminder <- read_csv("/Users/njtierney/Desktop/qmd4sci-materials/data/gapminder.csv")
```

As we now know, this is a problem, because this is not portable code. It is unlikely someone else will have the `gapminder.csv` data stored under the folders, `"Users/njtierney/Desktop"`.

If you have an RStudio project file inside the `qmd4sci-materials` folder, you can instead write the following:

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

Your Turn

- (1-2 minutes) Imagine you see the following directory path: `"/Users/miles/etc1010/week1/data/health.csv"` what are the folders above the file, `health.csv`?
- What would be the result of using the following code in `demo-gapminder.qmd`, and then using the code, and then moving this to another location, say inside your C drive?

```
setwd("Downloads/etc1010/week1/week1.qmd)
```

4.6 Is there an answer to the madness?

This file path situation is a real pain. Is there an answer to the madness?

The answer is **yes**!

I highly recommend when you start on a new idea, new research project, paper. Anything that is new. It should start its life as an **rstudio project**.

An rstudio project helps keep related work together in the same place. Amongst other things, they:

- Keep all your files together.
- Set the working directory to the project directory.
- Starts a new session of R.
- Restore previously edited files into the editor tabs.
- Restore other rstudio settings.
- Allow for multiple R projects open at the same time.

This helps keep you sane, because:

- Your projects are each independent.
- You can work on different projects at the same time.

- Objects and functions you create and run from project idea won't impact one another.

- You can refer to your data and other projects in a consistent way.

And finally, the big one:

RStudio projects help resolve file path problems, because they automatically set the working directory to the location of the rstudio project.

Let's open one together.

Your Turn Use your own rstudio project

1. In RStudio, and run the following code to start a new rstudio project called "qmd4sci-materials".

```r
usethis::use_course("njtierney/qmd4sci-materials")
```

2. Follow the prompts to download this to your desktop and then run the rstudio project. (You can move it later if you like!)
3. You are now in an rstudio project!

Your Turn: open the `demo.R` file

1. Run the code inside the `demo.R` file
2. Why does the `read_csv` code work?
3. Run the code inside the `exploratory-data-analysis` folder - `eda-script.R`.
4. Does the `read_csv` code work?
5. Run the code inside the `exploratory-data-analysis` folder - `eda-document.qmd`, by clicking the "render" button (we'll go into this in more detail soon!)
6. Does it work?

4.7 The "here" package

Although RStudio projects help resolve file path problems, in some cases you might have many folders in your r project. To help navigate them appropriately, you can use the `here` package to provide the full path directory, in a compact way.

```r
here::here("data")
```

returns

```
[1] "/Users/nick/github/njtierney/qmd4sci-materials/data"
```

And

```r
here::here("data", "gapminder.csv")
```

returns

```
[1] "/Users/nick/github/njtierney/qmd4sci-materials/data/gapminder.csv"
```

(Note that these absolute file paths will indeed be different on my computer compared to yours - super neat!)

You can read the above `here` code as:

> In the folder `data`, there is a file called `gapminder.csv`, can you please give me the full path to that file?

This is really handy for a few reasons:

1. It makes things *completely* portable
2. Quarto documents have a special way of looking for files, this helps eliminate file path pain.
3. If you decide to not use RStudio projects, you have code that will work on *any machine*

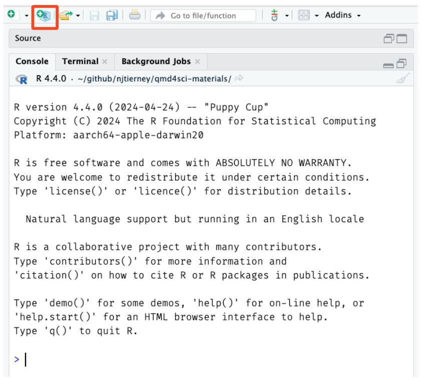
4.8 Remember

> 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 .

– Jenny Bryan

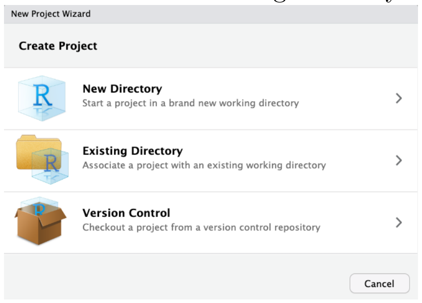
Aside: Creating an RStudio project

You can create an Rstudio project by going to:
file > new project > new directory > new project > name your project > create project.
You can also click on the create project button in the top left corner

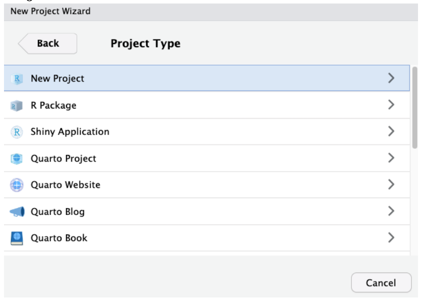


```
R version 4.4.0 (2024-04-24) -- "Puppy Cup"
Copyright (C) 2024 The R Foundation for Statistical Computing
Platform: aarch64-apple-darwin20

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.

> |
```

Then go to new directory, if it is a new folder - otherwise if you have an existing folder you have - click on existing directory.


New Project Wizard
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

Then go to new project

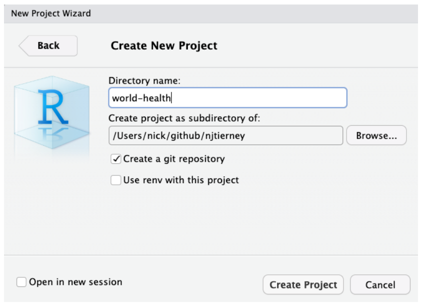


Then write the name of your project. I think it is usually worthwhile spending a bit of time thinking of a name for your project. Even if it is only a few minutes, it can make a difference. You want to think about:

- Keeping it short.
- No spaces.
- Combining words.

For example, I had a project looking at bat calls, so I called it `screech`, because bats make a screech-y noise. But maybe you're doing some global health analysis so you call it "world-health".

And click "create project".



Directory name: world-health

Create project as subdirectory of: /Users/nick/github/njtierney

✓ Create a git repository

☐ Use renv with this project

5

Summary

In this lesson we've:

- Learnt what file paths are
- How to setup an rstudio project
- How to construct full file paths with the `here` package

6 Why functions?

At their core, an R package is a way to share code. The way we share that code is primarily through R functions. There is a lot about the mechanics, and the tools to create and write R packages, but what I want to communicate here is the **what, why, when, and how** of using functions.

6.1 Overview

- **Teaching** 20 minutes
- **Exercises** 15 minutes

6.2 Questions

- What is a function?
- Why should I use a function?
- When should I use a function?
- How do I create a function?

6.3 Objectives

- Understand why functions should be used
- Understand when do use functions
- Understand how to write functions

6.4 Prior Art

There's a lot of work and thought that's gone into writing functions. A lot of my own understanding of this has been informed by others, and I want to make sure I properly acknowledge them:

- **Joe Cheng: You have to be able to reason about it**
- **Hadley Wickham's 'Many Models' talk**
- **Hadley Wickham's 'The design of everyday functions'**
- **Miles Mcbain's 'Our colour of magic'**
- **Jenny Bryan's 'Code Smells and Feels'**
- **Roger Peng's 'From tapply to Tidyverse'**
- **Advanced R: Functions**
- **Tidy Design Principles**
- **Lexical Scope and Statistical Computing**
- **stat545 chapter on functions**

These are all well worth the time reading, or watching these. If I had to pick two of the most influential, I would say:

1. Hadley Wickham's "Many Models" talk, and
2. Jenny Bryan's "Code Smells and Feels"

6.5 Code is for people

If I could have you walk away with one key idea, it would be this:

> Functions are tools to manage complexity that allow us to reason with and understand our code.

In essence, **code is for people**. This stems from a famous (well, I think it's famous), quote:

> [W]e want to establish the idea that a computer language is not just a way of getting a computer to perform operations but rather that it is a novel formal medium for expressing ideas about methodology. Thus, **programs must be written for people to read, and only incidentally for machines to execute**.

— **Structure and Interpretation of Computer Programs**. Abelson, Sussman, and Sussman, 1984.

6.6 OK, but what actually is a function?

Going back to my quote:

> Functions are tools to manage complexity that allow us to reason with and understand our code.

I actually think before we talk about the anatomy, the **what**. We first must discuss **why** functions.

A function is something that helps us manage complexity. You can think about this as something that allows us to repeat certain tasks. Kind of like how a robot, or a manufacturing line can repeat manual tasks.

Let's say we had some data on age groups - the number of contacts these people record on a given day.

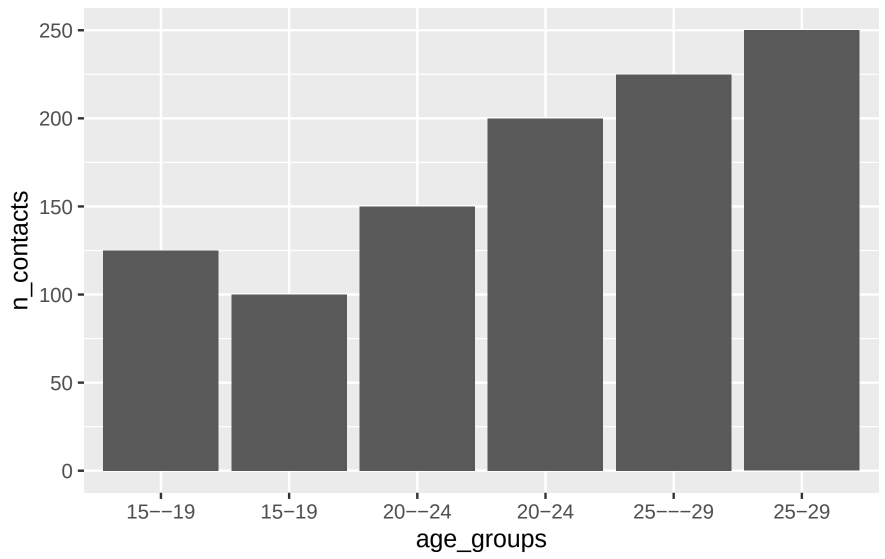
```
options(tidyverse.quiet=TRUE)
library(tidyverse)
contact <- tibble(
  location = rep(c("QLD", "NSW"), 3),
  age_groups = c("15-19", "15--19", "20--24", "20-24", "25---29", "25-29"),
  n_contacts = c(100, 125, 150, 200, 225, 250)
)

contact
```

```
# A tibble: 6 x 3
  location age_groups n_contacts
  <chr>    <chr>           <dbl>
1 QLD      15-19             100
2 NSW      15--19            125
3 QLD      20--24            150
4 NSW      20-24             200
5 QLD      25---29           225
6 NSW      25-29             250
```

We want to produce a plot of age groups and the number of contacts.

But, we can't do this, because there are all these different ways of representing "age_group".

```
ggplot(
  contact,
  aes(x = age_groups,
      y = n_contacts)) +
  geom_col()
```



Well rather, we CAN do this, but we want to get the totals of each age group.

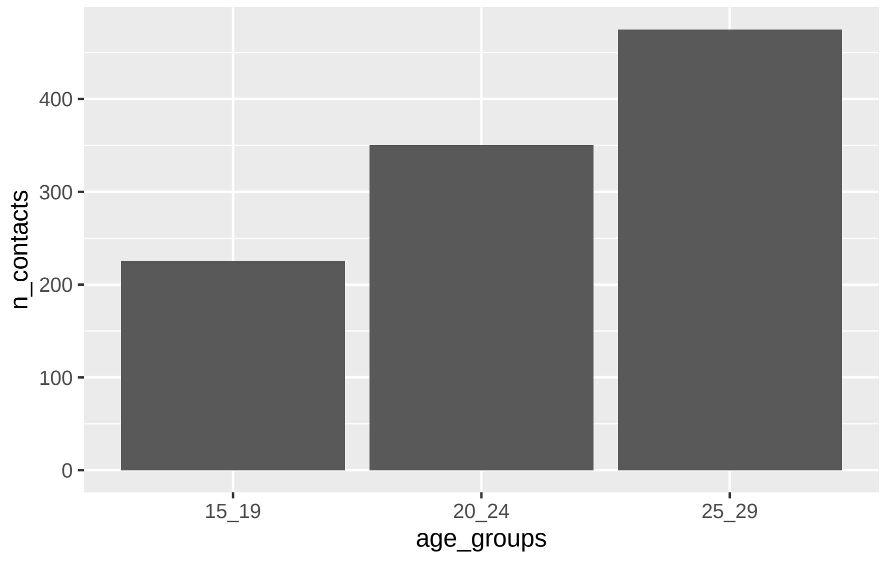
What we want out of this is them all to be separated out by an underscore "_", and turned into a factor:

```r
library(stringr)
tidy_contact <- contact |>
  mutate(
    age_groups = str_replace_all(
      string = age_groups,
      pattern = "---|--|-",
      replacement = "_"
      ),
    age_groups = as.factor(age_groups)
  )

tidy_contact
```

```
# A tibble: 6 x 3
  location age_groups n_contacts
  <chr>    <fct>           <dbl>
1 QLD      15_19             100
2 NSW      15_19             125
3 QLD      20_24             150
4 NSW      20_24             200
5 QLD      25_29             225
6 NSW      25_29             250
```

And then we can plot this:

```
ggplot(
  tidy_contact,
  aes(x = age_groups,
      y = n_contacts)) +
  geom_col()
```



Sure, job done.

But now we have some new data, this one contains similar information, but it has population data that we need to join onto it so we can get proportion information.

```
population <- tibble(
  location = rep(c("QLD", "NSW"), 3),
  age_groups = c("15--19", "15-19", "20---24", "20-24", "25-29", "25--29"),
  population = c(319014, 468550, 338824, 540233, 370468, 607891)
)

population
```

```
# A tibble: 6 x 3
  location age_groups population
  <chr>    <chr>           <dbl>
1 QLD      15--19         319014
2 NSW      15-19          468550
3 QLD      20---24        338824
```

```
4 NSW      20-24          540233
5 QLD      25-29          370468
6 NSW      25--29         607891
```

And **this** is why I think you should write a function.

We want to **encapsulate the idea** of cleaning up age group. That is: “clean age groups”. So let’s write a function that captures this idea.

```
clean_age_groups <- function(age_groups){

  age_underscore <- str_replace_all(
      string = age_groups,
      pattern = "---|--|-",
      replacement = "_"
      )
  as.factor(age_underscore)

}
```

And this is the difference in the worflow, for each of these, script, or function tidying up processes:

6.7 script

```
tidy_contact <- contact |>
  mutate(
    age_groups = str_replace_all(
      string = age_groups,
      pattern = "---|--|-",
      replacement = "_"
      ),
    age_groups = as.factor(age_groups)
  )

tidy_population <- population |>
  mutate(
    age_groups = str_replace_all(
      string = age_groups,
      pattern = "---|--|-",
      replacement = "_"
      ),
    age_groups = as.factor(age_groups)
  )

tidy_proportion <- tidy_contact |>
  left_join(tidy_population,
            by = c("location", "age_groups")) |>
  mutate(proportion = n_contacts / population)


tidy_proportion
```

```
# A tibble: 6 x 5
  location age_groups n_contacts population proportion
  <chr>    <fct>           <dbl>      <dbl>      <dbl>
1 QLD      15_19             100     319014   0.000313
2 NSW      15_19             125     468550   0.000267
3 QLD      20_24             150     338824   0.000443
4 NSW      20_24             200     540233   0.000370
5 QLD      25_29             225     370468   0.000607
6 NSW      25_29             250     607891   0.000411
```

6.8 function

```
clean_age_groups <- function(age_groups){

  age_underscore <- str_replace_all(
      string = age_groups,
      pattern = "---|--|-",
      replacement = "_"
      )
  as.factor(age_underscore)

}

tidy_contact <- contact |>
  mutate(
    age_groups = clean_age_groups(age_groups)
  )

tidy_population <- population |>
  mutate(
    age_groups = clean_age_groups(age_groups)
  )

tidy_proportion <- tidy_contact |>
  left_join(tidy_population,
            by = c("location", "age_groups")) |>
  mutate(proportion = n_contacts / population)
```

6.8.1 Functions give ideas a home

Functions provide a way to **express** the idea of what we want to do. They also provide your ideas a home. What if the data changes? Do you want to go back and change each line of code? No! You can update the function in one place, and then repeat it again.

Once you start writing functions to do things, they will start to be little repositories of knowledge. Little shortcuts that you can use to just remember the most important part.

Now, on to the anatomy of functions

6.9 Anatomy of a function

Now, to speak about the mechanics of writing functions: a function is composed of three parts:

1. Name
2. Arguments
3. Body

To look at our `clean_age_groups` function again, we can see the following:

```
# The name of the function
clean_age_groups <- function(age_groups){ # The argument - age_groups

  # The body of the function
  age_underscore <- str_replace_all(
      string = age_groups,
      pattern = "---|--|-",
      replacement = "_"
      )

  # The last thing you do with the function is what it returns
  as.factor(age_underscore)
}
```

⚠ The last thing you do shouldn't be assignment `<-`

The last thing that a function does is what it returns. If we take our example above and change the last line to assign to some variable, then the function will not return anything!

```
# The name of the function
clean_age_groups <- function(age_groups){ # The argument - age_groups

  # The body of the function
  age_underscore <- str_replace_all(
      string = age_groups,
      pattern = "---|--|-",
      replacement = "_"
      )

  # The last thing you do with the function is what it returns
  factored <- as.factor(age_underscore)

}

clean_age_groups("10--11")
```

This is a pretty common mistake, one I still make–something to be aware of!
The way to fix this is to make sure that the last thing you do isn't assigned. So, our example above should look like so:

```r
# The name of the function
clean_age_groups <- function(age_groups){ # The argument - age_groups

  # The body of the function
  age_underscore <- str_replace_all(
      string = age_groups,
      pattern = "---|--|-",
      replacement = "_"
      )

  # The last thing you do with the function is what it returns
  # NOT THIS
  # factored <- as.factor(age_underscore)
  # THIS
  as.factor(age_underscore)

}

clean_age_groups("10--11")
```

```
[1] 10_11
Levels: 10_11
```

6.10 How to think about writing functions

There are many ways to start writing functions.

Fundamentally, it is about identifying **inputs** and **outputs**.

One useful approach, I think, is to identify the outputs **before** the inputs:

1. The output. What **one thing** do you want this function to return?
2. The input. What (potentially **many**) thing(s) go in to this.

This “gestalt”, or top-down approach isn’t how it always needs to be done. But I think it helps you identify **the thing you need** first, which can help guide you.

6.10.1 Identifying the output - what do we need?

It might feel a bit like putting the cart before the horse, but I think there is a nice advantage to thinking about the output first: you focus on what you want the function to do.

In the case of our `clean_age_groups` function, we want to get values like "15_19" that are factors.

6.10.2 Identifying the input

So now we have a clear idea of what we need - we can now clarify what we have, which in our case earlier, was some contact data

```
contact
```

```
# A tibble: 6 x 3
  location age_groups n_contacts
  <chr>    <chr>           <dbl>
1 QLD      15-19             100
2 NSW      15--19            125
3 QLD      20--24            150
4 NSW      20-24             200
5 QLD      25---29           225
6 NSW      25-29             250
```

Where we want to focus on age groups, and take inputs like

```
c("15-19", "15--19")
```

```
[1] "15-19"  "15--19"
```

And then turn them into:

```
c("15_19", "15_19")
```

```
[1] "15_19" "15_19"
```

Breaking things down like this means we can focus on a really small example of the thing we want, which makes the problem easier to solve.

There are many ways to manage turning strings into other strings, and I like to use the `stringr` package to do this. We can use the `str_replace_all` function. So I'll start by scratching up some inputs like so, and seeing if this works

```
ages <- c("15-19", "15--19")
str_replace_all(
  string = ages,
  pattern = "-",
  replacement = "_"
)
```

```
[1] "15_19"  "15__19"
```

Not quite what I'm after - I've got two underscores when I want just one.

6.10.2.1 Iteration: Writing functions is writing

I didn't get this right the first time - and I rarely do! The point I want to make here is:

> Writing functions is just like writing. It takes iteration.

We have incidentally replaced every `-` with `_`, which means `--` becomes `__`.

Let's change that by using `|` in the "pattern" argument, which allows us to specify `-|--`, which means, `-` OR `--`:

```
ages <- c("15-19", "15--19")
str_replace_all(
  string = ages,
  pattern = "-|--",
  replacement = "_"
)
```

```
[1] "15_19"  "15__19"
```

OK, the same problem. We actually need to flip the order here, so we change `--` first:

```
ages <- c("15-19", "15--19")
str_replace_all(
  string = ages,
  pattern = "--|-",
  replacement = "_"
)
```

```
[1] "15_19" "15_19"
```

Great! Now let's put that into the body of the function, and **give the function a good name**.

```
clean_age_groups <- function(age_groups){
  str_replace_all(
  string = ages,
  pattern = "--|-",
  replacement = "_"
)
}
```

```r
clean_age_groups(ages)
```

```
[1] "15_19" "15_19"
```

It's a useful process to scratch out a function like this. As you get more confident with this, you will start to be able to write the code as a function first, and then iterate in that way.

⚠ beware copying and pasting into functions

The process of writing a function out in scratchings as we've done, is that we can leave some scraps in the code. In this case, I've actually left the `ages` object in the function, but the `argument` is `age_groups`:

```r
clean_age_groups <- function(age_groups){
  str_replace_all(
  string = ages,
  pattern = "--|-",
  replacement = "_"
)
}

clean_age_groups(ages)
```

```
[1] "15_19" "15_19"
```

Notice that this still works! This is because the `ages` object still exists as a variable I've created. But if we try another input, we'll get some strange output:

```r
clean_age_groups(c("10-12", "10--12"))
```

```
[1] "15_19" "15_19"
```

So, make sure to clean up after you've copied and pasted - remember to check the arguments match how they are used in the function.

And on that note, let's redefine `clean_age_groups` correctly so we don't get an error later on (which happened during the development of the book)

```r
clean_age_groups <- function(age_groups){
  str_replace_all(
  string = age_groups,
  pattern = "--|-",
  replacement = "_"
)
}

clean_age_groups(ages)
```

```
[1] "15_19" "15_19"
```

6.10.3 Managing scope - functions are best (generally) when they do one thing

Also, note that we wrote `clean_age_groups` to just focus on converting input like "10–12" into "10_12". We could have instead focussed on cleaning up the data frame, like so:

```r
contact
```

```
# A tibble: 6 x 3
  location age_groups n_contacts
  <chr>    <chr>           <dbl>
1 QLD      15-19             100
2 NSW      15--19            125
3 QLD      20--24            150
4 NSW      20-24             200
5 QLD      25---29           225
6 NSW      25-29             250
```

```r
clean_age_groups_data <- function(data){

  tidy_contact <- data |>
  mutate(
    age_groups = str_replace_all(
      string = age_groups,
      pattern = "---|--|-",
      replacement = "_"
      ),
    age_groups = as.factor(age_groups)
  )

  tidy_contact

}

clean_age_groups_data(contact)
```

```
# A tibble: 6 x 3
  location age_groups n_contacts
  <chr>    <fct>           <dbl>
1 QLD      15_19             100
2 NSW      15_19             125
3 QLD      20_24             150
```

```
4 NSW      20_24               200
5 QLD      25_29               225
6 NSW      25_29               250
```

I think there are a couple of issues with this:

1. We assume the age groups column is always `age_groups`
2. The scope is now larger - we are always working with data and returning data
3. We haven't necessarily made the expression easier.

It is fine to wrap up the existing function into another function that cleans the data - to me this better encapsulates and expresses the ideas:

```
contact
```

```
# A tibble: 6 x 3
  location age_groups n_contacts
  <chr>    <chr>           <dbl>
1 QLD      15-19             100
2 NSW      15--19            125
3 QLD      20--24            150
4 NSW      20-24             200
5 QLD      25---29           225
6 NSW      25-29             250
```

```
clean_contacts <- function(data){

  data |>
  mutate(
    age_groups = clean_age_groups(age_groups)
  )

}

clean_contacts(contact)
```

```
# A tibble: 6 x 3
  location age_groups n_contacts
  <chr>    <chr>           <dbl>
1 QLD      15_19             100
2 NSW      15_19             125
3 QLD      20_24             150
4 NSW      20_24             200
5 QLD      25__29            225
6 NSW      25_29             250
```

Some of the improvements I notice

- We are just focussing on cleaning up the age group column.
- We have given it a name that refers to cleaning up the data, which might also give us some space and room to add more cleaning function here.

6.11 When to function

One of my overall points with functions is:

> functions help you express your intention.

However, there are some generally good heuristics to follow to help guide you towards writing a function. Generally, it is time to write a function if:

1. You've **copied and pasted** the code 3 or more times.
2. You've **re-read** your code more than 3 times.

This first principle is often called **DRY** - "Don't Repeat Yourself.

The second principle has been coined by Miles McBain, also as DRY, or possibly DRRY: Don't ReRead Yourself.

6.12 Naming things is hard

> There are only two hard things in Computer Science: cache invalidation and **naming things**.

– Phil Karlton

What does this function **do**?

```r
myfun <- function(x){
  (x * 9/5) + 32
}
```

Converting temperature?

```r
temperature_conversion <- function(x){
  (x * 9/5) + 32
}
```

Clearly state `input_to_output()`

```r
celcius_to_fahrenheit <- function(x){
  (x * 9/5) + 32
}
```

Name argument and intermediate variables

```
celcius_to_fahrenheit <- function(celcius){
  fahrenheit <- (celcius * 9/5) + 32
  fahrenheit
}
```

What, what does make functions **hard**?

```
celcius_to_fahrenheit <- function(celcius){
  (celcius * 9/5) + 32
}
```

Identifying inputs and outputs is hard.

But what **is** hard it taking code, (like the code in a data analysis) and **finding the parts that need to change**

> There's a level of "I got it to work" and there's a level of "It works, and I can reason about it"

– Joe Cheng **You have to be able to reason about it | Data Science Hangout**

I can **reason** about it

> …how do you take all this complexity and break it down into smaller pieces…each of which you can **reason about**…each of which you can **hold in your head**…each of which you can look at and be like "yup, I can fully ingest this entire function definition, I can read it line by line and prove to myself this is definitely correct…So software engineering… is a lot about this: How do you **break up inherently complicated things** that we are trying to do into small **pieces that are individually easy to reason about**. That's half the battle…The other half of the battle is how do we combine them in ways that can be reliable and also easy to reason about

6.13 The other hard part of writing functions

Practice naming things

Give names to the following functions:

```r
thingy <- function(x){
  x^3
}

bobby <- function(x){
  str_replace_all(
    string = x,
    pattern = """,
    replacement = '"'
  )
}

f <- function(x) {
  ux <- unique(x)
  ux[which.max(tabulate(match(x, ux)))]
}
```

Generally speaking, it is good to following a naming convention of some kind, and also to keep the names descriptive:

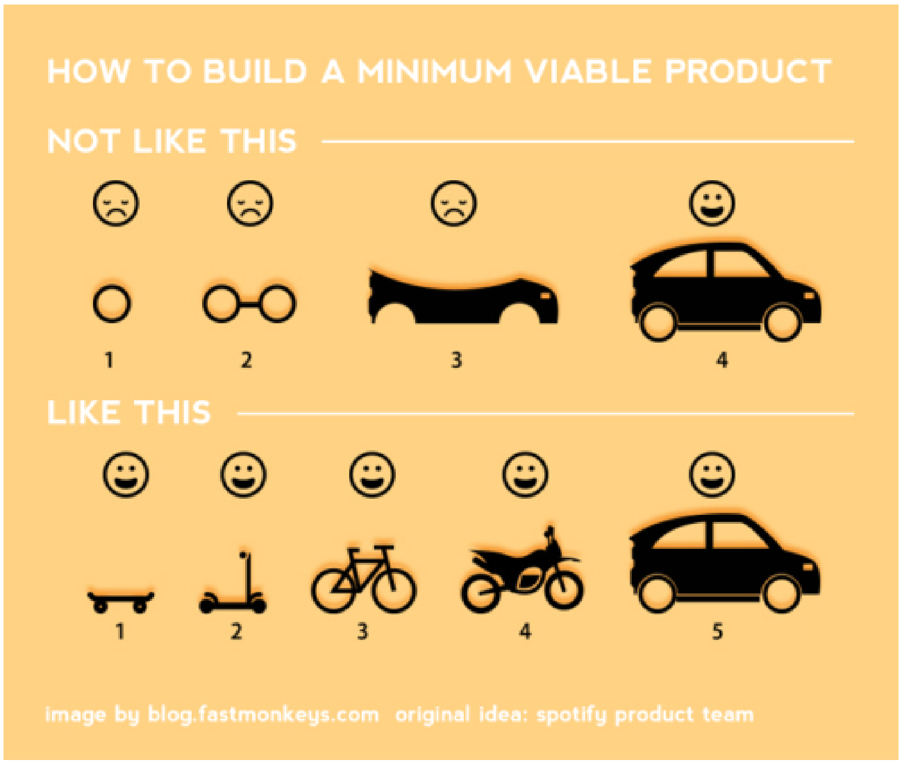
```r
# good
fit_lm()
fit_cart()
fit_glm()

# less good - tab complete isn't as good, unless we have a lot of functions also named `lm
lm_fit()
cart_fit()
glm_fit()

# bad
flm()
fcart()
fglm()
```

6.14 Conclusion

The process of writing a function is:

- Identify outputs and inputs
- Identify the complexity to abstract away
- Writing functions is **iterative**, Just like **regular writing**
- Naming things is hard. Focus on making "slightly better" names.

On a final note, I think it's worthwhile thinking about the iteration - and the idea of moving from a skateboard to a car, rather than building the car:


HOW TO BUILD A MINIMUM VIABLE PRODUCT
NOT LIKE THIS
1
2
3
4
LIKE THIS
1
2
3
4
5
image by blog.fastmonkeys.com original idea: spotify product team

(heard via **Stat545 functions chapter**)

7

Motivation

We've gone through a lot of setup, and now we're going to start building an R package. Soon. But we need to have some motivation, first. It involves a bit of a story, and a bit of imagination.

7.1 Overview

- **Teaching** 30 minutes
- **Exercises** 10 minutes

7.2 Questions

- How to convert code into functions?

7.3 Objectives

- Start to wrangle with a script to turn it into functions

7.4 How this works

One of my big goals with teaching functions, and with teaching R packages, is that I want the examples to be somewhat rooted in the familiar and the real. There are really useful toy examples of writing packages that deliver praise (e.g., ones I've used to teach R packages in the past: https://github.

com/njtierney/praiseme), or do simple conversions between units (celcius to farenheit being a very common example).

These examples are useful because they teach you the tools, and the process. However in this course, I want to focus on a bit more than this and incorporate the process of turning code into functions. I think this is important, because it more closely represents other examples we come across in using R, and presents a bit of a richer learning journey, because in addition to learning about the tools and the process of R package building, you will also learn:

- How to think about converting scripts to functions
- How to write better functions

I have written up some example code, which starts as a quarto document. We are going to take this document, and then eventually turn it into an R package.

The structure of this exercise has taken inspiration from "The package within" chapter from R Packages.

7.5 The example: "learned"

We are going to be looking at a role-play situation where we imagine we are at some fictional workplace, where part of our job is to look at education data that we have acquired from some source. The overall goal of our job here is to produce some **key outputs** from some data.

You can see this example at: https://github.com/njtierney/learned

To download it, run the following code

```
library(usethis)
use_course("njtierney/learned")
```

Your turn

1. Download the repository using the code above
2. Render the document, "analysis2014.qmd"
3. Read over the document, thinking about what we discussed in Why functions.
4. Identify some potential problems with the code
5. Think about what might happen if we want to read in data from 2015 (or later years), how would you like to do this?

7.6 Discussion of potential problems

After you've taken from time to think about some of the potential problems, open the box below

Some of the potential problems

- Copying and pasting a document could lead to errors
- What if the data changes?
- What if other people collaborate on this project? How do they have the source of truth?
- Is there a way to formalise this all?

7.7 Identifying the report outputs

To get us started with some key things, let's think about what the key outputs of this report are.

Your turn

1. Identify the **key outputs** of the report
2. Pick one of those key outputs and start to write out a function for it

Key outputs

They key outputs are related to the "Produce a …" steps of the document:

1. Produce a plot of the **proportion of people educated** in each age group in each state
2. Produce a box plot of **proportion of people educated** for each state.
3. Produce a table of The 5 number summary (min, 1st quantile, median, 3rd quantile, max) of **proportion of people educated** for each state.

So now we know where we are headed - we want to write some functions that produce these plots and tables.

However, the main problem that we encountered was that there was actually a bit of data cleaning that needed to happen before we did this. Let's focus on cleaning up and rearranging the quarto document first to identify the data cleaning steps required.

Your turn

1. Open, "alpha-analysis2014.qmd"
2. Move all the "data quality" checks into a new section called "data quality"
3. Move all of the data cleaning code up to the top, so we just work with one data set, named `tidy_age_state_education_2014`
4. Create two functions to clean the data:
   1. tidy the age groups
   2. remove the missing values
5. Put these two functions into another function that does the data cleaning

7.8 Discussion of data cleaning function implementations

- Discuss "solutions-alpha-analysis2014.qmd"

7.9 Applying functions to give the outputs

We've got some data cleaning functions! Now let's see if we can capture the intention of the plots, and wrap these up into functions, too.

The overall idea here is that we can capture the overall intention of what we are doing in a concise way, that is **easy to reason with**. That is **not** to say that it is **just smaller**, but being **easy to reason with** is key here.

Your turn

1. Open, "bravo-analysis2014.qmd"
2. Write a function for each of the following:
   1. Bar plot of proportion of people educated by state and age

   2. Boxplot of **proportion of people educated** for each state
   3. Table of 5 number summary of **proportion of people educated** for each state
3. Move all these functions to the top of the document into a single code chunk labelled “functions”
4. Move the “data checks” to the end of the document. We will come back to this later
5. Make a “libraries” code chunk and put the library call in there

7.10 Discussion of plot and table, rearranging functions

- What do we notice now?

7.11 Moving towards a quasi-package.

We’re slowly isolating the parts of the code that we care about, and now we’re going to make an incremental change again - to move all of the functions out to an R folder, with one function per R file.

For those who are more familiar with R Packages, this might start to look a lot more like an R package - while this isn’t the “standard” process, the intention here is to demonstrate the key changes:

- Functions: Identifying points of expression / abstraction
- Clearly expressing functions
- Using functions to clearly articulate your work

‘ergonomic’ / interactive helper packages

To do package development, we’re going to be using packages like `devtools` and `usethis`. These packages are things we use in the console - we use them interactively. They help us automate a lot of things, so we can focus on the task at hand.
You can be almost guaranteed that we will use any functions from `devtools` and `usethis` in the console. It’s OK for that to feel strange! It will feel better soon.

Your turn

1. Open "charlie-analysis2014.qmd"
2. Use `use_r()` from `devtools/usethis` to create a separate R file for each of the R functions:
   1. clean_age_groups
   2. clean_education_data
   3. plot_study_age_state
   4. boxplot_study_state
   5. summarise_prop_study
3. load all of these R functions, by calling `source` at the top of the quarto document

7.12 Discussion

- Thoughts on this process?
- Shall we just get to the making the R package?
- See https://github.com/njtierney/learned/pull/1 for solution

Now, let's move to making the R package!

8

Create the package!

It's happening! We're nearly there!

We've lined things up, now it's time to start tearing down some walls, and getting things ready to be an R package.

8.1 `create_package(here))`

Inside of the `learned` repository, run:

```
create_package(here::here(), open = FALSE)
```

This says:

> Create a package in this directory, and don't open a new RStudio session, please

You get a note that it does the following

```
  Setting active project to
   "/Users/nick/github/njtierney/learned".
  Writing DESCRIPTION.
  Writing NAMESPACE.
! Overwrite pre-existing file learned.Rproj?
```

Personally, I recommend No, there's not really a point to this in this instance.

```
  Leaving learned.Rproj unchanged.
  Adding "^learned\\.Rproj$" to .Rbuildignore.
  Adding "^\\.Rproj\\.user$" to .Rbuildignore.
  Setting active project to "<no active project>".
```

There's a bit to unpack here, you can see these new things have been created!

- See changes at this commit

There are three files added, which I'll briefly discuss, and then we can move on to more substantial matters of **cleaning this directory up a little bit**.

8.1.1 DESCRIPTION file

My DESCRIPTION file looks like this, initially

```
Package: learned
Title: What the Package Does (One Line, Title Case)
Version: 0.0.0.9000
Authors@R:
    person("Nicholas", "Tierney", , "nicholas.tierney@gmail.com", role = c("aut", "cre"),
           comment = c(ORCID = "https://orcid.org/0000-0003-1460-8722"))
Description: What the package does (one paragraph).
License: MIT + file LICENSE
Encoding: UTF-8
Language: en-GB
Roxygen: list(markdown = TRUE)
RoxygenNote: 7.3.2
```

Essentially, this is a bunch of metadata about the package. It is a very critical file to an R package, and we'll come back to it later. For the moment, I think there are two things worth noting:

1. It's a bunch of metadata that is really important and specifically formatted
2. The DESCRIPTION details here have my name and a few other things set (like `Language: en-GB`), because of where we set this up in installation.

8.1.2 .Rbuildignore

This lists files that we don't want to package up when we eventually build our R package.

This contains:

```
^learned\.Rproj$
^\.Rproj\.user$
```

Which are specific RStudio files. As you get further along in the package building process, more files will be added to `.Rbuildignore`.

8.1.3 NAMESPACE

Another super critical file. We don't touch it by hand. It gets updated automatically via `devtools` and `usethis`. It looks like this currently:

```
# Generated by roxygen2: do not edit by hand
```

8.2 Some housekeeping

We need to really tidy up this repository to be an R package. There is stuff **everywhere**.

8.2.1 Move the data to data-raw

Let's clean up all the existing data and move it into a special `data-raw` folder. The `data-raw` folder exists in R packages as a place to hold original copies of data that will eventually be cleaned up and shared in the `data` folder. For the time being, we can store our data sets in here, and we'll come back to it later.

Your turn

1. Run `use_data_raw()`
2. Move `raw_education_2014...2019.csv` into the newly created `data-raw` folder

8.2.1.1 Running `use_data_raw()`

When we run `use_data_raw()`, we get a message like the following:

```
Setting active project to
 "/Users/nick/github/njtierney/learned".
Creating data-raw/.
Adding "^data-raw$" to .Rbuildignore.
Writing data-raw/DATASET.R.
Modify data-raw/DATASET.R.
Finish writing the data preparation script in
 data-raw/DATASET.R.
Use `usethis::use_data()` to add prepared data
 to package.
```

This is the `usethis` package's way of telling us a couple of nifty things:

1. DATASET.R is a file it has created, this is where you document changes to the data before it gets saved into the `data` folder
2. the `data` folder gets created with `usethis::use_data()` - we will come back to this.

A takeaway point from this is that the `usethis` package is quite chatty, and quite helpful!

- You can see my initial commit of `use_data_raw()`.

8.2.1.2 Move CSVs into `data-raw/`

There's a few ways to do this, personally, I just use RStudio's file interface and move the files around. - Here's what that commit looks like. Note, that this is effectively a renaming, so there's not much going on in this commit.

8.2.2 Delete old quarto files, move one into vignettes

We don't need the alpha-bravo-charlie of it all anymore, we can just stick with one. We're going to move it to a special folder called a "vignette". We're going to shelve it there for a little bit.

> Your turn
>
> 1. Delete "analysis2014.qmd" and associated HTML/folders
> 2. Delete "alpha-analysis2014.qmd" and associated HTML/folders
> 3. Delete "bravo-analysis2014.qmd" and associated HTML/folders
> 4. (potentially move "solution-charlie-analysis2014.qmd") into root level and delete "charlie-analysis2014.qmd")
> 5. Delete "solutions" folder
> 6. run `use_vignette("analysis-2014")`

8.2.2.1 deleting old files

Here's the commit of deleting and moving files, and deleting solutions folder

- commit deleting and moving files
- commit deleting solutions folder

8.2.2.2 Creating quarto vignettes

Once we run

```
use_vignette("analysis-2014")
```

We get another nice chatty message from `usethis`:

```
Adding knitr to Suggests field in DESCRIPTION.
Adding "inst/doc" to .gitignore.
Adding rmarkdown to Suggests field in
 DESCRIPTION.
Adding "knitr" to VignetteBuilder.
Creating vignettes/.
Adding "*.html" and "*.R" to
 vignettes/.gitignore.
```

```
Writing vignettes/analysis-2014.Rmd.
Modify vignettes/analysis-2014.Rmd.
```

You can see the commit for this

Note that although we started with a Quarto vignette, it opened an Rmarkdown vignette. It is actually possible to use a quarto vignette - as described in the Quarto R package.

And, just because I want this to be more on the side of bleeding-edge erring to timeless, we're going to do that.

There's a few steps, so hold on.

8.2.2.2.1 update VignetteBuilder

Open the DESCRIPTION file, and replace `knitr` with `quarto` in `VignetteBuilder`.

- commit

Then, copy our "solution/charlie-analysis.qmd" file into vignette folder

- commit

Then, add some of the appropriate metadata into the quarto YAML:

- commit

Finally, delete the other Rmd file, and rename ours "analysis-2014.qmd"

- commit

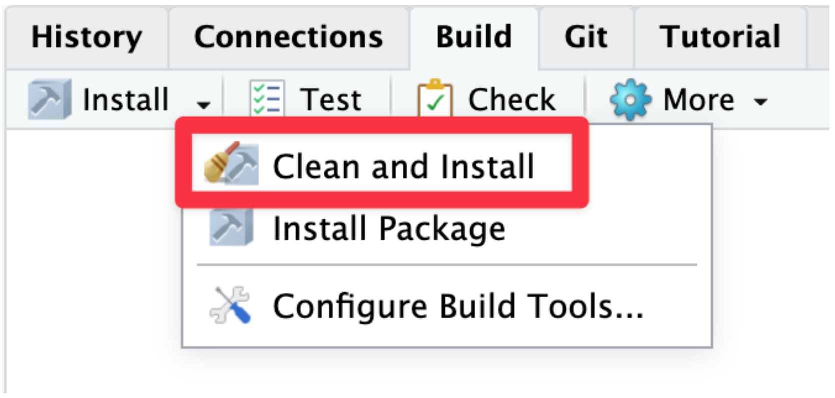
It's worthwhile noting that this vignette in its current state will not build properly, but we will get to it later.

8.3 Build/install the package

Now, we have a package! This will actually build, and install! Although we still need to do a few things to make it useable, it's worthwhile celebrating the small steps!

Your turn

Build the package by navigating to the "build" pane in the top right:



Click this, and see some text like the following appear!

```
==> R CMD INSTALL --preclean --no-multiarch --with-keep.source learned

* installing to library '/Users/nick/Library/R/arm64/4.4/library/_build'
* installing *source* package 'learned' ...
** using staged installation
** R
** byte-compile and prepare package for lazy loading
No man pages found in package  'learned'
** help
*** installing help indices
** building package indices
** installing vignettes
** testing if installed package can be loaded from temporary location
** testing if installed package can be loaded from final location
** testing if installed package keeps a record of temporary installation path
* DONE (learned)
```

Let's celebrate this win, and also wrap up this long section on a slight cliffhanger: let's try using our package.

8.4 The workflow of package development.

One of the major workflows during R package development is this:

1. Edit R functions
2. `load_all()` (or keyboard shortcut Ctrl/Cmd+Shift+L)
3. Edit R functions.

Cheatsheets

There is a great cheatsheet for package development that I had stapled to my cubicle wall during my PhD. The website now is very useful, but make sure to check out the PDF.

Let's try out our R package. To do this, I'm going to recommend you create a special R file, sometimes called a "scratch file". It lives inside a folder called `inst`, which is typically not touched by R package building. This is a useful trick if you want to have a way to play around with some R code, keeping it inside your R package.

Your turn

Create a scratch file:

1. Create a directory `inst`
2. Inside that directory, create an R file called `scratch.R` (note that this can actually be called anything, but `scratch.R` is what I use)
3. Write `library(learned)` inside `scratch.R`
4. Read in the raw education 2014 data from data-raw, and use one of the functions from the R package, like `clean_education_data()`. What happens when you use this?

See the commit for this

If you're like me, I got this error:

```
> clean_education_data(raw_education_2014)
Error in mutate(data, age_group = clean_age_groups(age_group), prop_studying = na_if(prop_
  could not find function "mutate"
```

Let's pick this up in the next section, "How to use extra packages"

This is a non-standard way of making an R package

This whole approach we have taken in this course, is not what I would describe as "the standard way" to make an R package. This represents a bit of a funny situation, where we have taken an existing workflow, and then morphed things into an R package. The benefits to this are that it is useful for teaching. And sometimes, this is how things happen.

9 How to use extra packages

One of the major shifts from writing R code and analyses to writing R packages is how you interact with other R packages you want to use.

Normally, in an R script, when you want to use a function from, say, `dplyr`, you use `library(dplyr)`.

However, we **do not ever** want to call `library(dplyr)` inside a function in an R package. The reason is to do with NAMESPACE conflicts. A popular usecase of this is in the `tidyverse` R package - where we get this message when we call `library(tidyverse)`.

```r
library(tidyverse)
```

```
-- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
v dplyr     1.1.4     v readr     2.1.5
v forcats   1.0.0     v stringr   1.5.1
v ggplot2   3.5.2     v tibble    3.2.1
v lubridate 1.9.4     v tidyr     1.3.1
v purrr     1.0.4
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to be
```

The conflict message at the bottom tells us that `dplyr::filter()` masks `stats::filter()`.

This is a key issue with package development - masking. If all R packages called `library(<package>)` on every R package that they depended upon, then we'd have SO MUCH masking. It's almost considered rude or overbearing.

> How do you NOT use library??

You say? The solution is to use what is called the "namespaced" form: `pkg::fun()`.

For example, if I want to use `filter` from `dplyr`, I do so with:

```r
cars |>
  dplyr::filter(speed <= 4)
  speed dist
1     4    2
2     4   10
```

So for every package we want to use a function from in an R package, we use the “namespaced form”, e.g., `dplyr::filter()`. We also have to declare the dependencies formally, which I’ll discuss now.

9.1 `use_package(<pkg>)` andpkg::fun()‘

So now we need to identify the packages that we use in our R package, and then namespace them. We then formally add the dependency with: `use_package(<pkg>)`.

For example, looking at

```r
boxplot_study_state <- function(data) {
  ggplot(
    data,
    aes(
      x = prop_studying,
      y = state_territory
    )
  ) +
    geom_boxplot()
}
```

We turn it into its **namespaced** form like so:

```r
boxplot_study_state <- function(data) {
  ggplot2::ggplot(
    data,
    ggplot2::aes(
      x = prop_studying,
      y = state_territory
    )
  ) +
    ggplot2::geom_boxplot()
}
```

And then call `use_package()`:

```r
use_package("ggplot2")
```

Which gives us a nice chatty response from `usethis`:

```
Setting active project to
 "/Users/nick/github/njtierney/learned".
Adding ggplot2 to Imports field in DESCRIPTION.
Refer to functions with `ggplot2::fun()`.
```

You can see this at this commit

Your turn

1. Identify all the R packages used in `learned`
2. call `use_package()` on each of these packages
3. Namespace all the functions

See this commit for what you should end up with.

How many dependencies should you have?

My opinion is that you should depend on as many R packages as you like! It's far faster, I think, to depend on packages that get the job done, and then maybe later trim back some dependencies and rewrite code.
My reasoning is that it is (generally) really cheap to add dependencies, but more expensive (for my brain), to write them from scratch.
So, be greedy, add dependencies, then prune back.

9.2 Demo in `scratch`

Now that we've done this, let's install the package, and then go to `scratch.R`

I then get this error:

```
> clean_education_data(raw_education_2014)
Error in clean_education_data(raw_education_2014) :
  could not find function "clean_education_data"
```

However, if we do load all, with:

```r
load_all(".")
```

Then it works!

But what gives with the function not being available? Let's have a look inside `learned` by using `:::`

There's nothing in there! Now let's focus on getting that working, which will involve learning about **documenting our code!**

The `tidyverse` package is not a dependency in packages

Because the purpose of the R package `tidyverse` is only really to load other R packages, it does not contain functions. Don't put `tidyverse` in imports.

Managing conflicts in scripts with `conflicted`

Outside of the R package development world, it's a good idea to proactively manage function conflicts. Lest you use `stats::filter()` instead of `dplyr::filter()`
See the `conflicted` R package for more on this idea.

Some other ideas

- Imports vs Depends. We only really use Imports. Don't use Depends unless you're building an extension package, e.g., something that works with ggplot2, where it doesn't make sense to have the package without ggplot2. Depends is like having `library(pkg)`. Generally, don't do it.
- It's usually a good idea to state package versions after their name in Imports. Find out the package name with `packageVersion("pkg")`. Generally onlt do `>=`, and not `==` and never `<`

10 Add documentation

So we noticed that there was **nothing** in `learned` when we looked at it - the reason is that we need to export the functions. To do this, we need to document our functions using `roxygen2`.

In short, `roxygen2` provides us with nifty syntax to give documentation to our functions, which also allows them to be exported.

Let's demonstrate with `boxplot_study_state.R`:

It starts like this:

```r
boxplot_study_state <- function(data) {
  ggplot2::ggplot(
    data,
    ggplot2::aes(
      x = prop_studying,
      y = state_territory
    )
  ) +
    ggplot2::geom_boxplot()
}
```

And we can add a "roxygen skeleton" by going to `code > insert roxygen skeleton` (or with Alt/Option + Ctrl/Cmd + Shift + R) whilst our cursor is in the function, and we get this:

```r
#' Title
#'
#' @param data
#'
#' @returns
#' @export
#'
#' @examples
boxplot_study_state <- function(data) {
  ggplot2::ggplot(
    data,
    ggplot2::aes(
      x = prop_studying,
      y = state_territory
    )
  ) +
    ggplot2::geom_boxplot()
}
```

Which we can then populate:

```
#' Provide a boxplot of study data
#'
#' @param data data from education 2014
#'
#' @returns a ggplot object
#' @export
#'
#' @examples
#' # no example data yet
boxplot_study_state <- function(data) {
  ggplot2::ggplot(
    data,
    ggplot2::aes(
      x = prop_studying,
      y = state_territory
    )
  ) +
    ggplot2::geom_boxplot()
}
```

We then call `document()`, which gives us the output:

```
  Updating learned documentation
  Loading learned
Writing NAMESPACE
Writing boxplot_study_state.Rd
```

This allows us to look at the documented function with `?boxplot_study_state`.

This function also gets added to the NAMESPACE file.

Here's the commit of that

Note three things:

1. NAMESPACE now has `boxplot_study_state.R` in it
2. We have added some roxygen code

3. There's some LaTeX looking code in `man/boxplot_study_state.Rd`

Your Turn

1. Repeat this process for all the remaining functions in `learned`
    1. add the roxygen skeleton
    2. population the roxygen skeleton
    3. run `document()`

See this commit to see what this looks like when done.

So now we have some functions in our package! Woo!

For fun, try looking at the documentation with `?clean_age_groups`

What's missing from these functions? Examples! For examples, we need our data! Let's add that to the package.

11

Adding data to an R package

overall goals - add raw 2014 data - add tidy 2014 data

- demonstrate using data-raw/dataset.R
    - briefly discuss compression = "xz"
- show using set.seed(yyyy-mm-dd) / providing a date
- demo now by having the data be used without read_csv in scratch.R

But now just use library, not load all - see that there's no functions!

So now go on to having documentation and **?**

12

Passing Checks

- explain this isn't something to run all the time, but as the quote goes, frequency reduces difficulty
- unpack the process of R CMD Check

13 Adding a vignette

We have already covered this earlier, but recap this, and also check that the vignette is now building properly

Your turn

1. Open up “analysis-2014.qmd”
2. Remove “Your Tasks”
3. Remove “Functions”
4. Update the Data cleaning so you can it interactively to read in the “raw_education_2014.csv” data (NB: we will eventually move this data into the R package itself)

14 Adding a README file

- Take the vignette and make it small.
- This is your README
- `use_readme_rmd()`

15 Pushing it to GitHub

We can pretty much achieve this with:

`use_git() use_github()`

Discuss the benefits and tools of issues, releases, milestones, etc

16

Using tests

- initially these tests will make sure that they run on the example data
- demonstrate testing dimensions/names
- demonstrate snapshot tests
- demonstrate vdiffr tests
- circle back to the data quality checks
  - maybe these can become our tests?
- touch on the idea that R packages can be used for other things than CRAN - although this does maybe go against some ideas in this course, the R package structure can be exploited to do some cool things - like perhaps data validation?
- reference Miles's https://milesmcbain.xyz/posts/an-okay-idea/

17 Adding a website

- pkgdown
- use_pkgdown_github_pages()
- look at other packages for inspiration on structure
- try to use a different default theme

18

Using Continuous Integration

- `use_github_actions()`
  - follow prompts
- This is all much easier than it used to be
- Explain that there is a game of "watching the green lights"
- Explain test coverage

19

Publishing your R package

- Places to publish packages
  - github
  - CRAN
  - R Universe
- pros and cons
- work through getting it on R Universe
- Discuss risks of building/installing from github

20

Next Steps

- R packages book
- official R docs
- rOpenSci docs

Submitting to CRAN

Submitting to JOSS

Submitting to rOpenSci

21

- R packages book
- Hilary Parker's blog post
- rOpenSci dev guide

22

- Hadley Wickham
- Jenny Bryan
- Miles McBain
- Joe Cheng
- Di Cook
- Rob Hyndman
- Kerrie Mengersen
- Damjan Vukcevic
- Matt Moores

A

B

Bibliography