R Best Practices

Nicholas Tierney

2026-02-17

Table of contents

Welcome

Welcome to R Best Practices! This course covers essential best practices for writing clear, maintainable, and reproducible R code.

Course materials

Materials will be added here as we progress through the course.

https://r-best-practices.njtierney.com

Prerequisites

- Basic R programming experience
- Familiarity with writing R scripts
- Experience working on data analysis projects

Learning outcomes

- How to name things effectively
- Using a style guide
- How to refactor your code
- How to review your code and others'
- How to lay out a project so others know how to run your code
- How to make a reproducible example (reprex)

Schedule

Project Organisation

- Understanding file paths
- Pitfalls of organisation
- Common project structures
- Naming files
- "Good Enough"/Common Sense principles of project organisation
- Always have a README
- RStudio and positron set up

Code style and readability

- Style is like grammar
- Layout, naming, and correctness are three different jobs
- Why we care about consistent names, spaces, and indentation
- Using the {air} formatter
- Using linters
- Bonus: typing skills and keyboard shortcuts

Writing readable code

- Where we are up to: what the tools fixed, and what they didn't
- Good vs bad variable names
- De-chunking code
- What to look for: code smells, inputs and outputs, complexity, refactoring
- A process for reviewing code
- Reviewing someone else's code
- Clean as you go

Writing Functions

- The problem functions solve
- Anatomy of a function
- Good (simple) function design
- Using {fnmate} to speed up creating functions
- How to use a debugger

Writing a reprex / getting unstuck

- How to share problems
- Using {reprex}
- Practicing reprex
- Practical tips on debugging
    - Keep solving vs cleaning up

Putting It All Together

- Take an existing project and provide code review
- Apply code linting, code styling, functions
- Discussion and Q & A

Setup

Two things before we start:

1. How this book is put together, and
2. What to install.

The first part takes two minutes to read and will make the rest of the course easier to follow.

How this book works

The book is longer than the course

The course is six hours. The book has more in it than six hours can cover, and that is on purpose.

We follow one path through it live, and the rest stays here for when you want it. So if we skip a section, or if you notice a box we never opened, nothing has gone wrong - it is there for you to come back to. I would rather write the complete version and choose what to teach on the day than write only what fits and leave you with notes that stop where the session did.

How we will work

Mostly by typing. I will write code, you write it too, and we will break things together and then fix them. You will get more out of this by typing the examples than by watching me type them, even when it feels slower.

You do not need to keep up with every keystroke. Everything on screen is in the book, with a copy button on every code block.

The boxes

Four kinds of box turn up throughout, each with its own colour and icon:

- **Your Turn** - something for you to do. Most of these we do together in the session, and a few are there for afterwards. The ones that ask what you think have no wrong answers; I am genuinely curious what has gone wrong for you before.

- **Some history** - where something came from. Why a linter is called a linter, where the backslash in `\(x)` comes from. Entirely skippable, and quietly my favourite part of the book.
- **Going deeper** - optional extra detail, a longer example, or a list you might want later. Boxes titled **Read more** point at the blog post or talk a section is adapted from, so you can follow the original if you want it.
- - the things that will actually bite you. There are not many, and that is deliberate.

Rather than describe them, here is one of each.

> **Your Turn**
>
> Exercises look like this. Most of them give you something to run:
>
> ```
> R.version.string
> ```
>
> Some of them just ask you something. Here is the one I actually care about: what is the worst R code you have ever had to work with? Your own counts, and mine definitely does.
> There is no wrong answer. I am asking because the answer usually tells me which parts of the day to slow down on.

> **Some history: where the name R comes from**
>
> Two things at once, which I think is lovely.
> R was written by Ross Ihaka and Robert Gentleman at the University of Auckland in the early 1990s, and they share a first initial. It is also a play on S, the language it was modelled on, in the same way that C came after B.
> So the name is a pun and a signature at the same time. You can read their own account of it in R: A Language for Data Analysis and Graphics.

> **Going deeper: how these boxes are made**
>
> You just clicked a title, so that is the collapsing demonstrated.
> These are Quarto callouts, and there is nothing clever going on. Each one is a fenced div with a class on it:
>
> ```
> ::: {.callout-note title="Your Turn"}
>
> Something for you to do.
>
> :::
> ```
>
> The colours and the icons come from a stylesheet in the book's repo, not from Quarto. Quarto ships five callout types with its own icons, and I have repainted all five.
> If you want them in your own work, the docs are at Quarto: callout

blocks.

> **!** Restart R after a big install
>
> If you install a package that is already loaded in your session, R can end up half on the old version and half on the new one. You then get errors that make no sense and do not survive a restart, which is a miserable way to spend twenty minutes.
> So once you have worked through the installs below, restart R. In RStudio that is `Cmd / Ctrl + Shift + F10`.

Some boxes are collapsed, showing only their title, like the **Going deeper** one above. Click the title to open one. If you are reading the PDF they are all open already, which is one reason the PDF is longer than it looks online.

The code

Code blocks look like this:

```
library(tidyverse)

airquality |>
  count(Month)
#>   Month  n
#> 1     5 31
#> 2     6 30
#> 3     7 31
#> 4     8 31
#> 5     9 30
```

Lines starting with `#>` are **output**, not code. You do not type those - they are what R prints back, shown so you can check you got the same thing.

Longer blocks have line numbers in the margin so that I can say "look at line 4" and we are both looking at the same line. Short blocks do not, because there is nothing to point at.

Most examples use data that comes with R, like `airquality` above, so you can run anything here without downloading a dataset first.

We use the base pipe, `|>`, rather than `%>%` throughout. If you are used to `%>%`, everything here works the same way.

Where everything lives

- **This book** - https://r-best-practices.njtierney.com, and there is a PDF download in the sidebar if you would rather print it or annotate it.

- **The book's source** - https://github.com/njtierney/rbestpractices. Every page has an "Edit this page" link, which goes straight to the file that made it.
- **The books exercises** - https://github.com/njtierney/rbp-exercises. We will discuss how to "automagically" open this in the course

If you find a mistake, please tell me. Every page has a link to open an issue, or you can email me. Errors in teaching material are worth more than errors in almost anything else I write, because thirty people hit them at once. Finding one is a favour, not a complaint.

What to install

If you can, it is worthwhile to install the below before the first session.

We have time on the day, and I would much rather spend ten minutes helping you get set up than have you sit out an exercise. So work through what you can, and bring the rest along.

R

You need **R 4.5.0 or later**, from https://cran.r-project.org/. Check what you have with:

```
R.version.string
#> [1] "R version 4.6.1 (2026-06-24)"
```

The reason for 4.5.0 specifically is the penguins. From that version, R ships a `penguins` dataset with base R, so a few of the examples work without installing anything at all. Before 4.5.0, you had to get it from a package.

You also need an editor - the thing you use to edit your R code.

Personally, I recommend RStudio.

If you like, you can use Positron - for this course I mostly focus on RStudio, but am able to discuss some of the features with positron, which is posit(the company that made RStudio)'s latest generation tool.

If you already have one and you like it, stay where you are.

> If you are on an older R
>
> Everything in the course still works, but you will need the palmerpenguins package for the handful of examples that use `penguins`:
>
> ```
> install.packages("palmerpenguins")
> library(palmerpenguins)
> ```

> One catch: **the column names are different.** When the data moved into base R the names were shortened, so you will need to translate:
>
> | In this book | In palmerpenguins |
> |---|---|
> | `bill_len` | `bill_length_mm` |
> | `bill_dep` | `bill_depth_mm` |
> | `flipper_len` | `flipper_length_mm` |
> | `body_mass` | `body_mass_g` |
>
> The units left the names but not the data - `body_mass` is still in grams.
> I wrote about this and the rest of what landed in 4.5.0 here: R version 4.5.0 is out.
> You can use Ella Kaye's `basepenguins` R package to help you move your existing scripts across if you want to use base R penguins.
> Separately, some examples read a `penguins.csv` file rather than the dataset. Those use the longer names, because that is what is in the file - not because of your R version.

The R packages

It might will take a few minutes:

```
install.packages(c(
  "tidyverse",
  "here",
  "lintr",
  "styler",
  "spelling",
  "usethis",
  "reprex",
  "goodpractice",
  "sf",
  "pak",
  "sp"
))

install.packages('fnmate', repos =
 ↪  c('https://milesmcbain.r-universe.dev',
 ↪  'https://cloud.r-project.org'))
```

`sf` and `sp` are only needed for one optional exercises. They are the heaviest things on the list, so if they give you trouble, leave them and let me know.

Code formatters

Air is an R formatter from Posit. It is not an R package, so it installs

from the terminal rather than from R.

See their installation instruction on their website:

But briefly, here are the most popular ways to install:

On macOS and Linux:

```
curl -LsSf https://github.com/posit-dev/air/releases/latest/d
↪  ownload/air-installer.sh | sh
```

On Windows:

```
powershell -ExecutionPolicy Bypass -c "irm https://github.com
↪  /posit-dev/air/releases/latest/download/air-installer.ps1
↪  | iex"
```

Check it worked:

```
air --version
```

You should get a version number back. If you get "command not found", the install did not finish, so try it again.

We set up format-on-save together in chapter 2, so there is nothing else to do with air before we start.

If you have trouble installing {air}, `{styler}` is in the package list above and does the same job, just not as fast.

Jarl, the linter

Jarl is a fast linter by Etienne Bacher, built on top of Air. Installation instructions are on its site.

Jarl is optional. Like air, it is a command line tool rather than an R package, so the install can go wrong in all the same ways, and I do not want you fighting it before we start.

If it gives you any trouble, do not worry about it. `{flir}` is in the package list above - it is by Etienne as well, it installs the way every other R package installs, and it is the one that will still fix things for you. `{lintr}` is in the list too. Everything in chapter 2 works with any of the three.

Checking it all worked

Run this in R. It should print `TRUE` for everything.

```
pkgs <- c(
  "tidyverse", "here", "lintr", "styler",
  "spelling", "usethis", "reprex", "fnmate",
  "flir", "goodpractice", "sf", "sp"
)

sapply(pkgs, requireNamespace, quietly = TRUE)
```

Then, in the terminal:

```
air --version
```

If both of those look right, you are set.

If something will not install

Please do send me an email before the session rather than fighting with it on your own.

Installation problems are almost always specific to one machine, and they are much faster to solve with two people looking at them. They are also, genuinely, not a reflection on you. Every one of us has lost an afternoon to a package that would not build. Ask me for some horror stories.

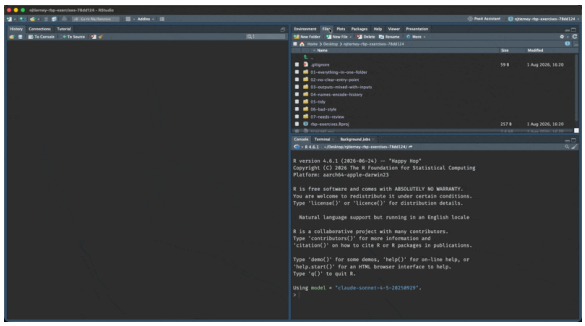
Get course exercise materials on your machine

Your Turn

Let's download the course exercise - try running this code:

```
use_course("njtierney/rbp-exercises")
```

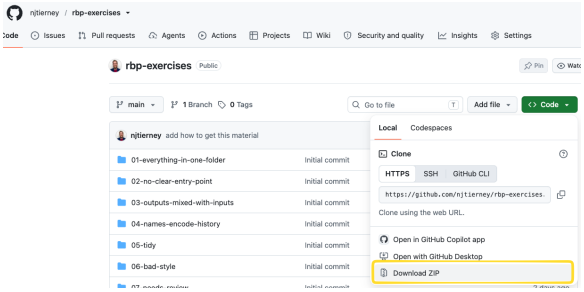
Which will prompt you to download the repository, https://github.com/njtierney/rbp-exercises, and then open it in an RStudio project.



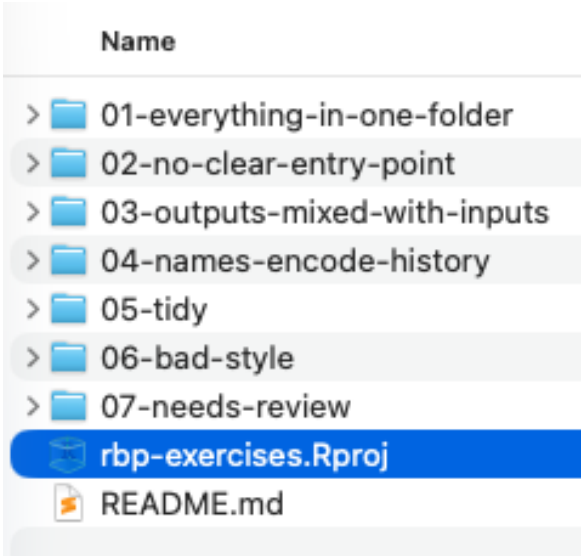
Where you end up. This one is a screen recording in the web version of the book.

If you have troubles with this (sometimes firewalls can stop you)

you can download it from the github page https://github.com/njtierney/rbp-exercises like so:



Then, unzip that, and click on the rbp-exercises.Rproj file:



1. Project organisation

> "Remember, you are always collaborating with your future self."

This course is all about best practices in R. At its foundation, I think this begins with a well organised project that is clean and tidy. We start by talking about basic "hygiene" for your projects and files, before moving on to how to organise your project, and some potential pitfalls.

Remember: your project doesn't have to be perfectly organised. Instead, we focus on making the next project a little better, each time.

It is worth noting that this section draws from "Workflow" and RStudio, what and why from my course: "quarto for scientists".

Overview

Duration 60 minutes
Questions

- How should I set up RStudio or Positron?
- What is a file path, and why does `setwd()` break things?
- What is a project, anyway?
- How does `{here}` find my files?
- Why should every project have a README?
- What goes wrong when a project is not organised?
- What does "good enough" project organisation look like?
- How should I name my files?

1.1. Set up

Go to course exercises and follow instructions to download course materials.

Your Turn

When you start a **new** project:

- What is your normal workflow?
- What helps you, what hinders you?
- What do you think you should do?

When you come back to an **old** project:

- What has made it easier to work on it?
- What makes it difficult?

- Anything you wish you did?

There are no wrong answers! I want to have a discussion, and learn what wins, what makes it hard, what makes it OK.

Let's talk about some easy-wins in how you set up RStudio and Positron, to make it easier to make things reproducible.

1.1.1. RStudio

We want to go to global options, like so:

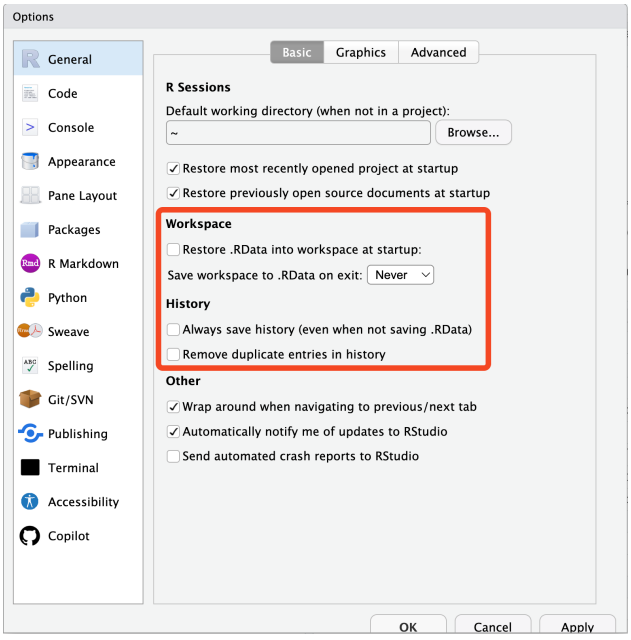
Tools → Global Options (or `Cmd + ,` on macOS) → General.

Under **Workspace**:

- Uncheck **Restore .RData into workspace at startup**
- Set **Save workspace to .RData on exit** to **Never**

Under **History**:

- Uncheck **Always save history (even when not saving .RData)**
- Uncheck **Remove duplicate entries in history**



Setting the options so you neither restore nor save your previous session's work.

This matters for two reasons:

1. **Reproducibility.** You don't want old objects from your last analysis cluttering your session - and you also don't want to depend on them, accidentally.

2. **Privacy.** You really do not want private data written into a `.RData` file on disk. You want to read data in, deliberately, every time.

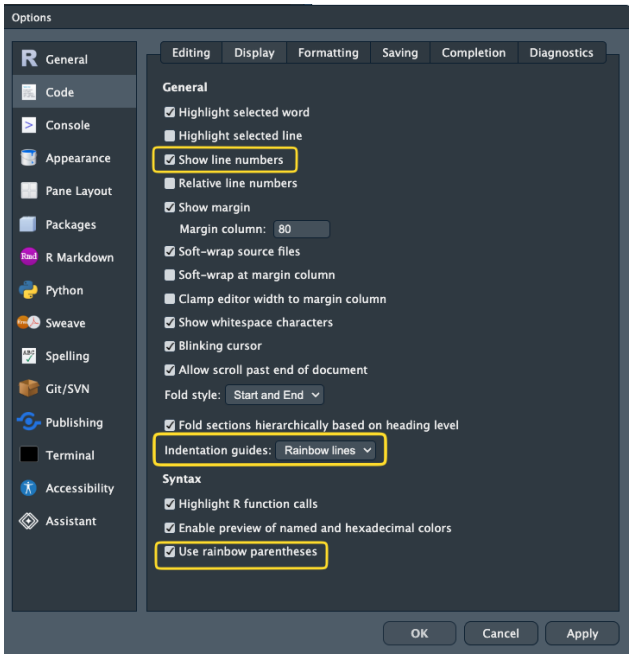
Your history is the list of commands you have typed. Not saving it means you won't come to rely on something you typed in a previous session - which is a good habit to build.

You can also do establish these settings via the `{usethis}` package:

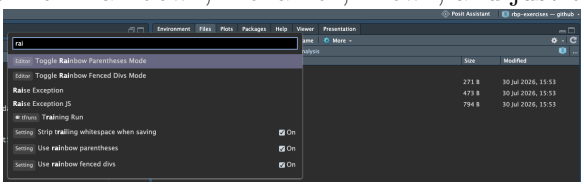
```
usethis::use_blank_slate()
```

While you are in the settings, two more worth turning on:

- **Show line numbers** - Easier to say: "The code on line 54"
- **Rainbow parentheses** - Catch those missing brackets
- **Rainbow indentations** - Fun indentation markers



> **Make a habit**
>
> You can restart R with a keyboard shortcut:
>
> - **Restart R**: `Cmd/Ctrl + Shift + F10`
> - Your R session will always start from scratch now.
> - Make a habit of restarting R and starting from scratch.
> - If your code works from a fresh session, that is a great standard
> - If it errors on a fresh session, you've got a problem to solve (which is overall a good thing to identify!)

> **Your Turn**
>
> There is a really neat feature in RStudio called the "Command Pallete" - it's kind of like "spotlight" or "search" for any setting in RStudio.
> Try opening it using the keyboard shortcut:
>
> - Windows: `Ctrl + Shift + P`
> - Mac: `Cmd + Shift + P`
>
> And search for "rainbow", "rename", "new", and just explore!
>
> 
>
>
> Read more at Posit's RStudio guide to the Command Palette

1.1.2. Positron

The same settings live under Settings → Extensions → R. The equivalent behaviour is off by default, so there is usually nothing to change - but it is worth looking, so you know where it is.

Positron also has the command palette! The same command as RStudio.

- Windows: `Ctrl + Shift + P`
- Mac: `Cmd + Shift + P`

> **Your Turn**
>
> 1. Turn off workspace saving using the settings above, or run `usethis::use_blank_slate()`.
> 2. Restart R with `Cmd/Ctrl + Shift + F10`.
> 3. Open your most recent analysis script and run it from the top in the clean session.
> 4. Discuss why your work did/did not work

> **RStudio vs Positron?**
>
> Worth saying upfront - RStudio isn't going away.
>
> - RStudio
>   - created in 2011.
>   - Focus on R, but allows you to write in Python.
>   - Supports other languages like C, C++.
>   - It has 15 years of design for use in data analysis. It just works.
> - Positron

  - Created in 2024.
  - A fork of "Visual Studio Code - Open Source"
  - Comes with many (most?) of the features of VS Code
  - Rich marketplace of extensions
  - "polyglot" - works with many languages, not just R

Which to choose? Here are my opinions:

- New to R –> RStudio
- Just using R –> RStudio
- Mostly using R, occasional other language –> RStudio
- Sometimes R, mostly other language –> Positron
- R + frequently other other languages (Python, Rust, C) –> Positron
- Experienced in R and want full control of every setting –> Positron

If you want to read more about the features of Positron and RStudio, see Posit's article on the topic.

1.2. A common problem: file paths

Before we get into projects, and organising them, I really think it is worthwhile spending a bit of time talking about **file paths**.

I can almost guarantee that you have run into this problem:

```
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()`:
! cannot open the connection
```

This error is R saying:

> I do not know where "my-very-important-data-file-somewhere.csv" is.

There are lots of reasons that R doesn't know where that file could be!

Chances are **you** know exactly where the file is, and now you need to get into the mind of R, to understand **why it cannot find the file**.

We can resolve these errors if we keep our **files**, **paths**, and **directories** ordered. This goes together with a **workflow** for clearly, and portably letting R know where these are. This practice is sometimes referred to as **good file path hygiene**.

Let's define a few of these terms: files, directories, and paths.

1.2.1. Files, directories, paths

Below we show some examples of a folder structure, of directories, and files

```
Users/                                ← Directory
└── njtierney/                        ← Directory
    └── Desktop/                      ← Directory
        └── analysis-2026/            ← Directory
            ├── analysis-2026.Rproj   ← File
            ├── data/                 ← Directory
            │   └── penguins.csv      ← File
            ├── exploratory-analysis/ ← Directory
            │   ├── explore.R         ← File
            │   ├── eda-document.qmd  ← File
            │   ├── eda-document.docx ← File
            │   └── graph.png         ← File
            └── README.md             ← File
```

We can refer to a given location with a "path" - which we refer to as a "file path" or a "directory path" depending on which we are navigating to. Here are some examples of paths for two directories, and two files.

- `Users/njtierney/Desktop/analysis-2026/`
  - **Directory** path of "analysis-2026"
- `Users/njtierney/Desktop/analysis-2026/README.md`
  - **File** path of "README.md"
- `Users/njtierney/Desktop/analysis-2026/data`
  - **Directory** path of "data"
- `Users/njtierney/Desktop/analysis-2026/data/penguins.csv`
  - **File** path of "penguins.csv"

A path is just the folders you walk through to get there, joined up with `/`. Here is that last one again, with every step of it marked:

```
Users/                          ●
└── njtierney/                  ●
    └── Desktop/                ●
        └── analysis-2026/      ●
            ├── analysis-2026.Rproj
            ├── data/           ●
            │   └── penguins.csv●
            ├── exploratory-analysis/
            │   ├── explore.R
            │   ├── eda-document.qmd
            │   ├── eda-document.docx
            │   └── graph.png
            └── README.md
```

Read the marked lines top to bottom and you have written the path:

```
/Users/njtierney/Desktop/analysis-2026/data/penguins.csv
```

Nothing else in the tree is part of it. That is the whole idea - a path names one route down through the folders, and ignores everything it walks past.

Let's talk a bit more about files, directories and paths.

Files

- Files are something I can open.
- These are all files: "explore.R", "eda-document.docx", "graph.png", "README.md", "penguins.csv"
- Files have "file extensions", which are the last characters after a `.`, and they tell us what kind of file they are:
  - "explore.R" ends with **.R**, it is an R script
  - "eda-document.docx" is a Microsoft Word Document
  - "graph.png" is an image (specifically, a "PNG")
  - "README.md" is a **m**ark**d**own text file.
  - "penguins.csv" is a "comma separated values" file

Directories

- Also known as folders (files inside a folder)
- A directory contains files
- A directory can also contain directories

Paths

A path, or "file path", or "directory path" is the machine-readable directions to where files on your computer live. Kind of like a street address. "Machine readable" just means your computer can read it easily:

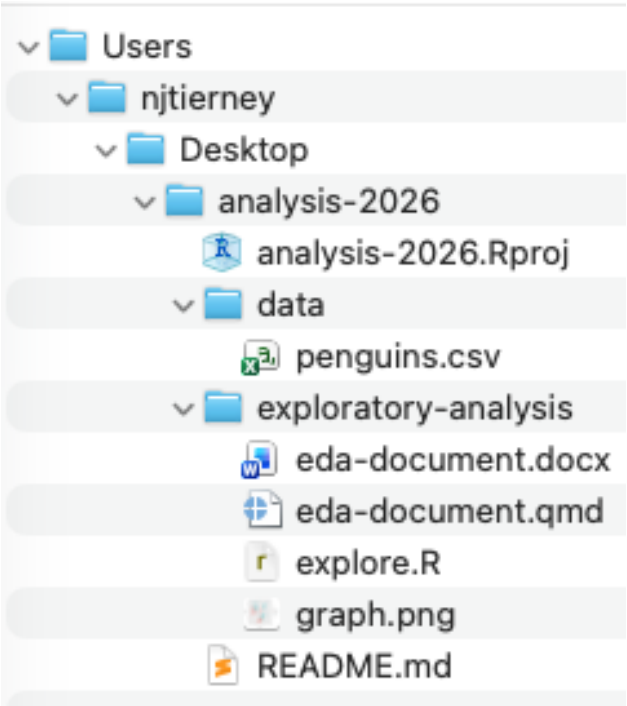
- Machine readable:
  - `/Users/njtierney/Desktop/analysis-2026/data/penguins.csv`
- Not-quite machine readable:
  - "The folder on my desktop named analysis 2026"

So, the **file path**:

```
/Users/njtierney/Desktop/analysis-2026/data/penguins.csv
```

Describes the location of `penguins.csv`

It might be easier to see this as a tree, or how this file might look in a file explorer on your computer:



1.2.2. Reading files

So to read CSV in R, you can put the path of penguins.csv in `read_csv()`:

```
penguins <- read_csv("/Users/njtierney/Desktop/analysis-2026/
↪  data/penguins.csv")
```

In our diagram, that is this file:

```
Users/
└── njtierney/
    └── Desktop/
        └── analysis-2026/
            ├── analysis-2026.Rproj
            ├── data/
            │   └── penguins.csv        ← this is the file being read in
            ├── exploratory-analysis/
            │   ├── explore.R
            │   ├── eda-document.qmd
            │   ├── eda-document.docx
            │   └── graph.png
            └── README.md
```

You might also note that sometimes in your work, you don't write down this full path - it is very long! You might do something like:

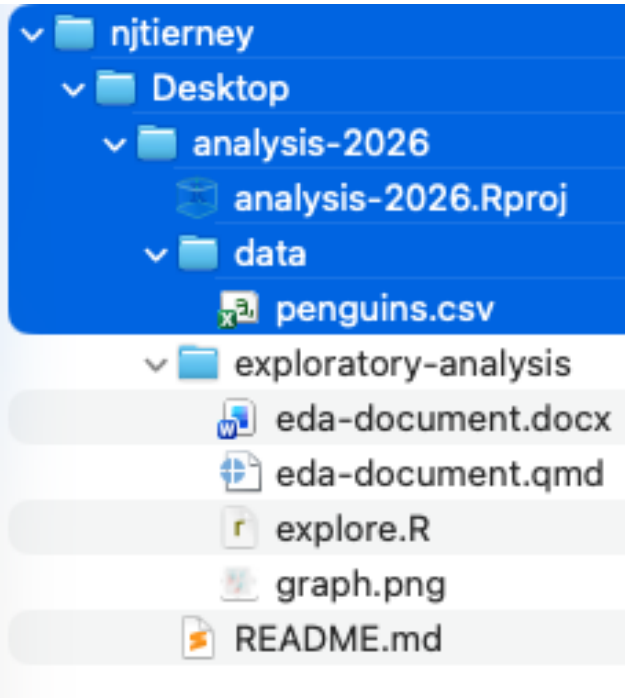
```
penguins <- read_csv("data/penguins.csv")
```

And this brings us to an important concept - these are two kinds of path:

- **absolute paths** starts from the root of your computer.
- **relative paths** starts from wherever you currently are.

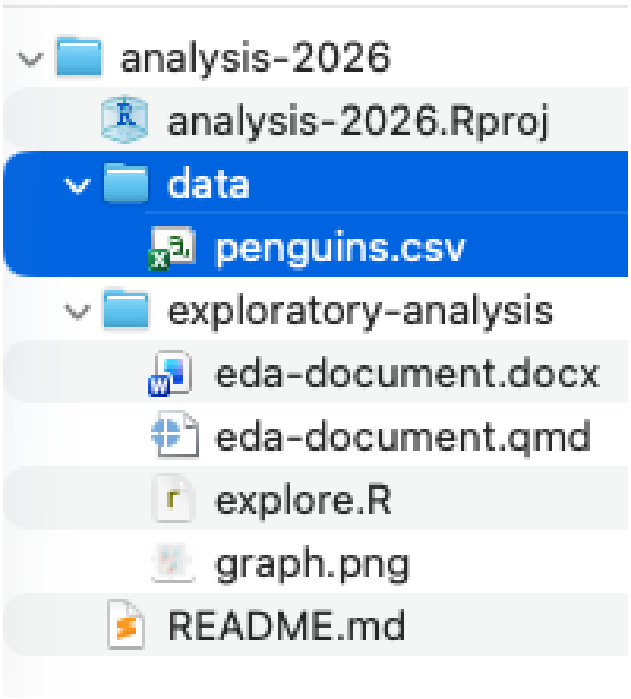
So, the code below is using an **absolute** path:

```
penguins <- read_csv("/Users/njtierney/Desktop/analysis-2026/
↪  data/penguins.csv")
```



The code below is using a **relative** path:

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



- The relative path looks nice and shorter. But it requires context - it needs to know from where you are reading.
- The absolute path will work from anywhere on your machine. It reads from the lowest folder on your machine.[1]

But, will the absolute work on your colleagues machine? On your machine in two weeks, next month, next year? No. My absolute path will not work on your machine. Your computer (almost certainly) doesn't have a `/Users/njtierney/` folder. It won't work on your own machine after you reorganise your folders when you spring clean. It won't work on the cloud.

> **Your Turn**
>
> 1. Imagine you see the path `/Users/miles/etc1010/week1/data/health.csv`. What are the folders above the file `health.csv`?
> 2. Run `getwd()`. Where are you?
> 3. Look at your most recent analysis script. Find every file path in it. How many are absolute? How many are relative?

A common answer to this is that people will often use relative paths, along with a function `setwd()`:

```
setwd("/Users/njtierney/Desktop/analysis-2026")
penguins <- read_csv("data/penguins.csv")
```

[1] That lowest folder is called the **root** - the one folder that contains everything else. On macOS and Linux it is written `/`, so an absolute path always starts with a `/`. On Windows it is usually `C:\`. Your own home folder sits just inside it, at `/Users/yourname` on macOS or `/home/yourname` on Linux, and has a shorthand: `~`. So `~/Desktop/analysis-2026` and `/Users/njtierney/Desktop/analysis-2026` are the same place, on my machine only.

Let's talk about why this is **not a good idea**.

1.2.3. What does `setwd()` do?

Sometimes the first line of a script looks like this:

```
setwd("/Users/njtierney/Desktop/analysis-2026")
```

This says:

> Set my working directory to this specific directory, "analysis-2026", which is in the folders, Users/njtierney/Desktop.

It means you can then read data using the **relative paths** we discussed above:

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

instead of using the **absolute path**:

```
penguins <- read_csv("/Users/njtierney/Desktop/analysis-2026/
 ↪  data/penguins.csv")
```

So it does have the effect of **making the file paths work in your file**.

But, it is still a problem, because using `setwd()` like this creates the same problem you had when you used the **absolute path**:

- 0% chance of working on someone else's machine - **and this could include you, in six months**.
- Means your file is not self-contained or portable. What happens if this folder moves to `/Downloads`, or onto another machine?

I used to send files to people like this, and you might have come across this yourself - where an R file comes to you and your colleague says:

> Yup, all you need to do is **change the `setwd("/Users/njtierney/Desktop/analysis-2026")` to point at where you are, then run it**

So, to get it working, you have to hand-edit the file path for every machine it lands on.

This is painful. And when you do it all the time, it gets old, fast. There is a better way!

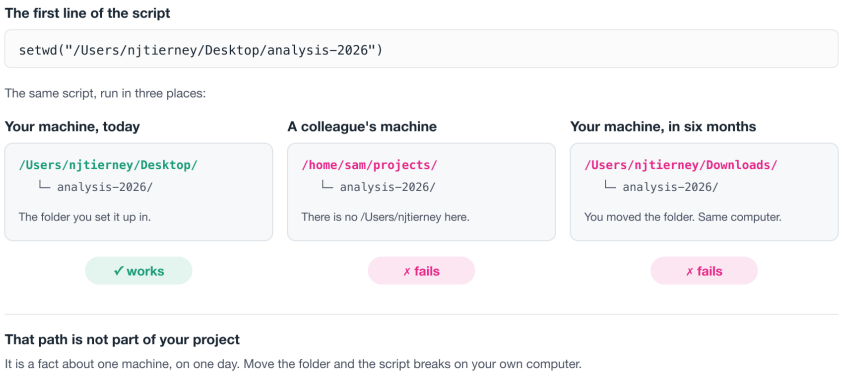


Figure 1.1.: The path inside `setwd()` names a folder that only one machine has, and will not work in future places.

This is the heart of the famous line from Jenny Bryan, from her brilliant article, "Project-oriented workflow"- you should read it:

> 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 .

Let's talk about using projects

1.3. Project oriented workflow

When you start on a new idea, a new project, a new paper - anything new - it should start its life as a **project**. In RStudio that is an `.Rproj` file; in Positron it is a **workspace folder**.

A project keeps related work together in the same place. They also do the following:

- Set the working directory to the project directory.
- Start a new session of R.
- Restore previously edited files into your editor tabs.
- Let you have several projects open at once.

This helps keep you sane, because:

- Your projects are independent.
- You can work on different projects at the same time.
- Objects and functions you create in one project won't affect another.

Projects resolve file path problems, because they automatically set the working directory to the location of the project.

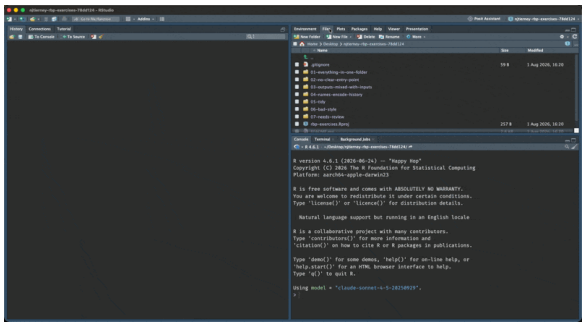
1.4. Get course exercise materials on your machine

Your Turn

Let's download the course exercise - try running this code:

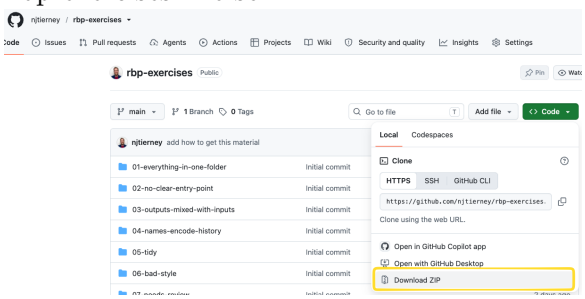
```
usethis::use_course("njtierney/rbp-exercises")
```

Which will prompt you to download the repository, https://github.com/njtierney/rbp-exercises, and then open it in an RStudio project.

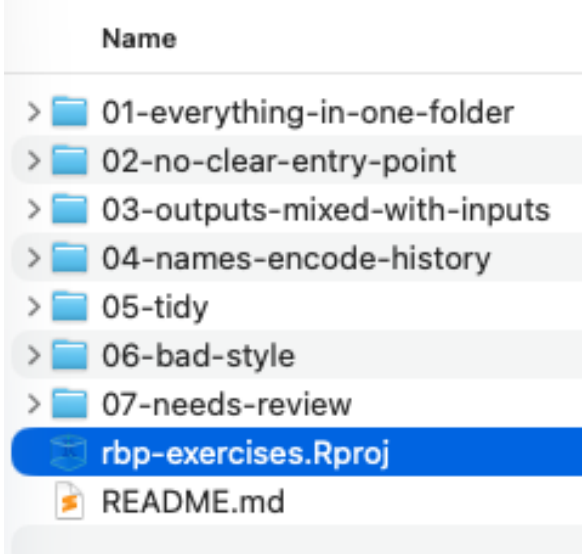


Where you end up. This one is a screen recording in the web version of the book.

If you have troubles with this (sometimes firewalls can stop you) you can download it from the github page https://github.com/njtierney/rbp-exercises like so:



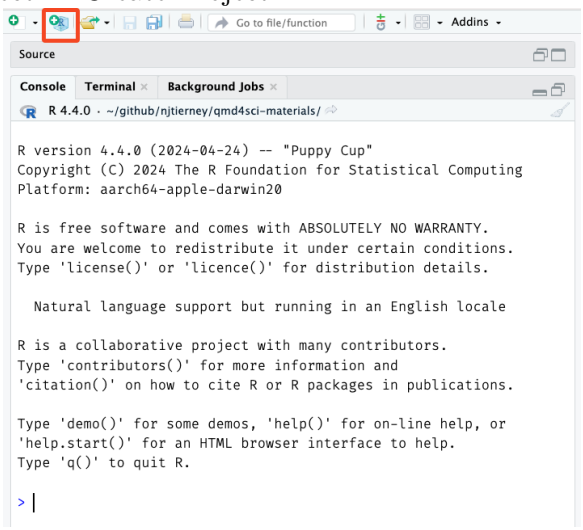
Then, unzip that, and click on the rbp-exercises.Rproj file:



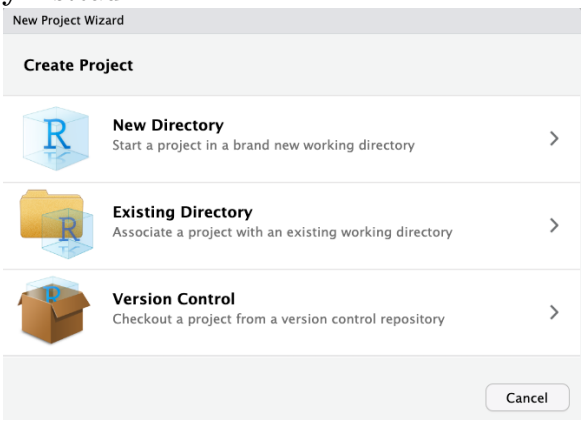
Aside: creating a project

1.5. RStudio

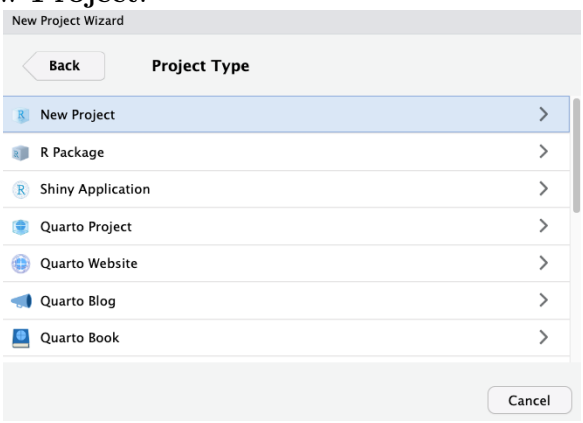
Either use the button in the top right corner, or go:
File → New Project → New Directory → New Project → name your project → Create Project



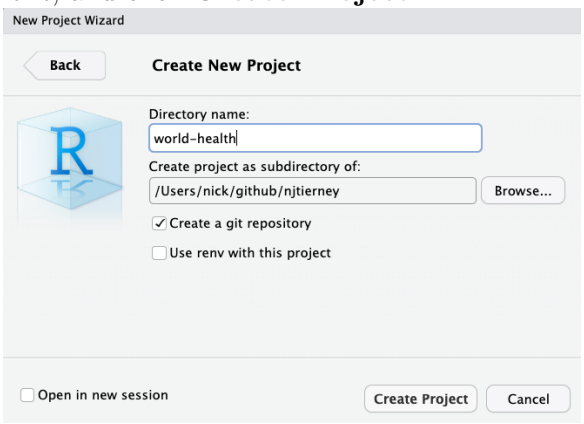
Starting a new project in RStudio.
Then choose **New Directory** if this is a new folder. If you already have a folder you want to turn into a project, choose **Existing Directory** instead.



Choosing a new or existing directory.
Then **New Project**.



Choosing the project type.
Then name it, and click **Create Project**.



Naming the project.

1.6. Positron

Positron doesn't use .Rproj files - that is an RStudio specific file. The posit team talk about this in their docs: "The R Proj File". The gist of this is that Positron does not have Rproj files - you essentially just open a folder and Positron will treat it as a "workspace". You can open a folder as a workspace: File -> Open Folder.
If you also want the project to work in RStudio, add an `.Rproj` file:

```
usethis::use_rstudio()
```

Either way, it is worth spending a couple of minutes on the name. Even a few minutes can make a difference. You want to:

- Keep it short.
- Use no spaces.
- Combine words.

For example, I had a project looking at bat calls, so I called it `screech`, because bats make a screech-y noise. If you are doing global health analysis, maybe it is `world-health`.

1.7. The {here} package

Although projects resolve most file path problems, in some cases you might have many nested folders. To navigate them reliably you can use the {here} package, which builds the full path from the project root, compactly.

```
here::here("data")
#> [1] "/Users/njtierney/github/njtierney/analysis-2026/data"
```

and

```
here::here("data", "penguins.csv")
#> [1] "/Users/njtierney/github/njtierney/analysis-2026/data/penguins.csv"
```

Note that these absolute paths will be different on my computer compared to yours - which is exactly what I want you to see.

You can read that `here` code as:

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

This is handy for a few reasons:

1. It makes things *completely* portable.
2. Quarto documents have their own way of looking for files, and this eliminates a whole class of file path pain.
3. If you decide not to use projects, you still have code that works on *any* machine.

So in practice:

```
library(here)
penguins <- read_csv(here("data", "penguins.csv"))
```

works whether the code is called from the top of the project, from inside `analysis/`, or from a Quarto document being rendered in a subfolder - all places where a plain relative path can quietly mean something different.

> **Your Turn**
>
> 1. Run `here::here()`. Is it where you expected?
> 2. Open another rstudio project. Take one absolute path from your most recent script and rewrite it with `here()`.
> 3. What would have to be true about your working directory for the old version to work?

1.8. Organise your project with a README

When you are working on a project, you are almost certainly going to be returning to it, at some point. When you do, you might look at the code, and wonder:

> Who wrote this, and are they insane?

And then you realise that **you wrote this**.

I think it is worthwhile internalising this saying:

> You are always collaborating with your future self.[2]

I think a really easy win here is to have a README file, which is a plain text file that helps answer:

- **Why** you did it: what was the goal
- **When** you did it: Last year, last month?
- **What** you did: What was run
- **How** you did it: What was the order you ran things? Which file should I look at first?
- **Who** did it: Was it just you? Were there many people?
- **Where** you did it: On your computer? The cloud?

If you can't answer these questions about a project, how much harder would it be for you to return to it?

I like to think of this as some self-care for your future self.

What I am getting at here is:

> **A project that is organised is a project that's more likely to re-run.**

A well organised project does not guarantee reproducibility. But, if you can't tell me which script to run first, then we cannot reproduce an analysis.

A well organised project makes it easier to check, run later, and understand.

A README does not need to be long. Mine are often ten lines. It just has to answer those six questions.

Here is a complete, entirely adequate README, with the question each part is answering marked in the margin:

```
# Penguin body mass analysis                          ← WHAT

Fits a model of penguin body mass against flipper
length and species, for the 2026 report to the
department.                                           ← WHY

Run by Nick Tierney. Last run 2026-03-14, on my
laptop, with R 4.5.0.                                 ← WHO,
 ↪  WHEN, WHERE

## How to run                                         ← HOW

Open `penguins.Rproj`, then run the scripts in
`analysis/` in order:

1. `01-clean-data.R`
2. `02-fit-model.R`
3. `03-make-figures.R`
```

[2] I swear I heard it from someone else, but I cannot find the source

```
Figures are written to `output/figures/`.

## Data

`data/penguins.csv` is from the {palmerpenguins}
package, saved on 2026-03-14. Do not edit this file.
```

The arrows are not part of the file. They are there to show you that a short README is not a summary of the project - it is six answers, in whatever order suits you.

If you are staring at an empty `README.md` and do not know where to start, write the six words down the page and fill them in.

Write the README first, if you can, and return to it often.

You can have **more than one README**. A short one inside `data-raw/` explaining where those files came from is often the most valuable file in the whole project.

And it's worth recording **when you obtained the data** - just noting the data, and also **what versions of software you used**, so that someone later, including you, can reconstruct similar conditions.

You can get the dependencies in an R project with:

```
renv::dependencies("path/to/directory")
```

Your Turn

Write a README for the project you sketched above. Answer the questions. Start it with:

```
usethis::use_readme_md()
```

Do it now, before you write any code - you will find it clarifies what you are actually about to do.

This pays off when the project becomes a paper

Everything in this lesson is framed around your future self. But the same structure is what makes your work usable by *anyone* else, and that matters most at publication.
Karthik Ram and I wrote about this in Common sense approaches to sharing tabular data alongside publication (*Patterns*, 2021). The argument, in short, is that depositing data doesn't make it reusable. The reusable part is mundane and structural:

- A README that says who, what, when, where and how.
- **Both** the raw data and the analysis-ready data, each in its own folder.

- The cleaning script that turns one into the other, kept with the raw data.
- Metadata describing each variable: its name, what it means, and its type. Enough to stop someone reading a gene sequence as a date.
- A licence. We recommend CC0.

If you organise your project this way from the start, you don't have to do anything special at submission. You're already most of the way there.
That's the real payoff, and it's why I'd rather you set this up now than tidy it up later.

1.9. Pitfalls of organisation

There are so many ways to organise a project. While there isn't a single **right** way, there are definitely **better** and **worse** ways. A key takeaway from today I want you to remember is:

> Your project might not be perfectly organised, but it should be possible for someone to understand how it is structured, and what to run first.

There are a few patterns that make picking up a new project hard. I am guilty of all of these. I am not judging you if these are how you work now. What I am saying is there are some ways you can improve.

You can see these patterns in the course exercises you downloaded earlier: https://github.com/njtierney/rbp-exercises

We will quickly move through these now.

1.9.1. Everything in one folder

```
01-everything-in-one-folder/
├── Rplot.png
├── Screen Shot 2026-03-14 at 10.23.45 am.png
├── Untitled1.R
├── analysis.R
├── figure1.png
├── notes.txt
├── penguins.csv
├── plot_stuff.R
├── report draft.docx
└── results.csv
```

Notice the R scripts, the data, the figures, a Word document, a screenshot, and `Untitled1.R`, are all *flat* - they are in the same folder. This works until there are more than about fifteen files, at which point finding anything becomes a search problem.

I think about this as like a grocery list: If my shopping list is only 5 items long, it might not impact me when I go to the shops:

- Milk
- Yoghurt
- Muesli
- Bananas
- Muesli bars

But if I'm buying a bunch of food for the week, some meal prep, and a few other bits and pieces, if I don't order the shopping list, and group the similar parts together, I am creating work for myself.

The two shopping lists

1.10. Unordered

- Milk
- Onion powder
- Red capsicum
- Yoghurt
- Muesli
- Bananas
- Muesli bars
- Tomato paste
- Ginger
- Coriander
- Garlic
- Lemongrass
- Coconut milk
- Chicken thighs
- Limes
- Eggplants
- Snow peas
- Thai basil
- 500g beef mince
- Basmati rice

1.11. Ordered

- **Produce**
  - Bananas
  - Ginger
  - Garlic
  - Lemongrass
  - Limes
  - Thai basil
  - Coriander
  - Eggplants
  - Red capsicum
  - Snow peas

- **Dairy**
  - Milk
  - Yoghurt
- **Meat**
  - Chicken thighs
  - 500g beef mince
- **Pantry**
  - Muesli
  - Muesli bars
  - Basmati rice
  - Coconut milk
  - Tomato paste
  - Onion powder

In this instance, I would recommend grouping the files into folders

1.12. Problem

Flat directory structure can make it hard to group things to understand how they are related.

```
01-everything-in-one-folder/
├── Rplot.png
├── Screen Shot 2026-03-14 at 10.23.45 am.png
├── Untitled1.R
├── analysis.R
├── figure1.png
├── notes.txt
├── penguins.csv
├── plot_stuff.R
├── report draft.docx
└── results.csv
```

1.13. Better

Put similar things into folders (directories) that give them structure.

```
01-everything-in-one-folder/
├── internals/
│   └── notes.txt
├── plots/
│   ├── Rplot.png
│   ├── Screen Shot 2026-03-14 at 10.23.45 am.png
│   └── figure1.png
```

```
├── scripts/
│   ├── Untitled1.R
│   ├── analysis.R
│   └── plot_stuff.R
├── data/
│   └── penguins.csv
└── outputs/
    ├── report draft.docx
    └── results.csv
```

1.13.1. No clear entry point

```
02-no-clear-entry-point/
├── analysis.R
├── clean.R
├── data.csv
├── do_it.R
├── final.R
├── helpers.R
├── main.R
├── model.R
├── plots.R
├── run.R
├── script.R
└── tests.R
```

There are eleven `.R` files and no way to tell which one runs first. If the only way to know the order they are run in, is only in your head, the project is not reproducible. It is a performance that only you can give.

1.14. Problem

When task order is required, but it is not written down, you cannot reproduce the task.

showing the above file preview

```
02-no-clear-entry-point/
├── analysis.R
├── clean.R
├── data.csv
├── do_it.R
├── final.R
├── helpers.R
├── main.R
├── model.R
├── plots.R
├── run.R
```

```
├── script.R
└── tests.R
```

1.15. Improvement

Encode the task order in another file, such as "00-run-first.R" - adding numbers to the start of the file will mean they sort the files in order. If files aren't useful anymore, you can delete them, or you can put them in an "attic" folder, if you aren't ready to let them go.

```
02-no-clear-entry-point/
├── 00-run-first.R
├── 01-helpers.R
├── 02-clean.R
├── 03-analysis.R
├── 04-model.R
├── 05-plots.R
├── 06-tests.R
├── 07-final.R
└── data/
    └── data.csv
└── attic/
    ├── xx-main.R
    ├── xx-do_it.R
    └── xx-script.R
```

1.15.1. File names encode history, not content

```
04-names-encode-history/
├── analysis.R
├── analysis2.R
├── analysis_final.R
├── analysis_final_USE_THIS_ONE.R
├── analysis_final_USE_THIS_ONE_v2 (nick edits).R
├── analysis_final_USE_THIS_ONE_v2.R
├── analysis_final_v2.R
└── penguins.csv
```

The *names* of these files tell a story: the analysis progressed, things changed, and there were concerns about knowing which version worked, and being able to go back and change things. You can think of this as using "track changes" in a word document. This is a kind of version control. But naming the files in order to track versions creates problems, and it is better solved with specific version control software, such as git. I cover this in another course, "Introduction to git and github".

1.16. Problem

There are multiples of the same file with names that tell you their version, rather than their task.

showing the above file preview

```
04-names-encode-history/
├── analysis.R
├── analysis2.R
├── analysis_final.R
├── analysis_final_USE_THIS_ONE.R
├── analysis_final_USE_THIS_ONE_v2 (nick edits).R
├── analysis_final_USE_THIS_ONE_v2.R
├── analysis_final_v2.R
└── penguins.csv
```

1.17. Solution

Name the files for what they do, not when you made them[3]. If you aren't ready to let them go, use the "attic" folder.

```
04-names-encode-history/
├── analysis.R
└── data/
    └── penguins.csv
└── attic
    ├── analysis2.R
    ├── analysis_final.R
    ├── analysis_final_USE_THIS_ONE.R
    ├── analysis_final_USE_THIS_ONE_v2 (nick edits).R
    ├── analysis_final_USE_THIS_ONE_v2.R
    └── analysis_final_v2.R
```

1.17.1. Editing raw data in place

You might have some data provided for analysis and there might be instructions to go through by hand to fix typos, or add new data. You should not make hand edits to the file, as this changes the history and increases the chances of a non-repeatable change. You can make a critical error by accidentally hitting the wrong key. From experience, having accidentally sorted a single column in an excel sheet and not realising this, I essentially turned a meaningful data source into random noise.

- The problem: Hand editing raw data

[3]Ideally, you would use a version control system like git. Out of scope for this course.

- The solution: Raw data should be read-only, and changes that happen to it happen with code, and are saved to separate outputs.

The way you actually enforce that is with two folders. `data-raw/` holds the file exactly as it arrived and nothing ever writes to it. `data/` holds the version your analysis reads, and it is produced by a script.

1.18. Problem

One file, edited by hand, and no way to tell what you changed or when.

```
04-editing-raw-data/
├── analysis.R
└── penguin-survey.xlsx     ← opened in Excel, typos fixed by hand
```

1.19. Better

```
04-editing-raw-data/
├── data-raw/
│   ├── penguin-survey.xlsx  ← exactly as it arrived. Read only.
│   ├── clean-penguins.R     ← every fix you would have made by hand
│   └── README.md            ← where it came from, and when
├── data/
│   └── penguins.csv         ← written by clean-penguins.R
└── analysis.R               ← reads data/, never the xlsx
```

Every correction is now a line of code. You can read it, re-run it, and disagree with it in six months. The typo fix that used to be a keystroke nobody saw is now `mutate(species = if_else(species == "Adelei", "Adelie", species))`, sitting in a script with the raw file beside it.

And it survives the thing that hand editing cannot: the data arriving again next year, slightly different. Re-run the script.

1.19.1. Outputs mixed in with inputs

```
03-outputs-mixed-with-inputs/
├── 01-clean.R
├── 02-model.R
├── cleaned_data.csv
├── fig-mass-flipper.png
├── fig-species.png
├── model_fit.rds
├── penguins.csv
└── raw survey data.xlsx
```

In this case, figures, model outputs, and cleaned data sit in the same folder as your raw data and scripts. You cannot tell at a glance what is necessary for your analysis, and what is produced as output by your analysis. This means you don't have a clear sense of how to reproduce files, or where edits can be safely made. A good test: *could I delete this folder entirely and regenerate it by running my code?* If yes, it is an output, and it should live somewhere that says so.

- The problem: Not knowing what is inputs vs outputs impacts reproducibility.
- The solution: Clearly label inputs and outputs - potentially put your outputs in an "outputs" folder. Or have a file that describes the inputs and outputs, such as a README, or an analysis script.

> **Your Turn**
>
> Open a project you worked on more than six months ago.
>
> 1. Can you tell, within thirty seconds, which script to run first?
> 2. Which files could you delete and regenerate from code?
> 3. Which files could you *not* regenerate? Those are the ones that actually matter - are they backed up?

1.20. "Good enough" project organisation

There is no single correct structure, but there is a common shape that I think gives you a great starting place:

```
my-project/
|-- my-project.Rproj
|-- README.md
|-- data-raw/
|   |-- penguins-raw.csv
|   `-- clean-penguins.R
|   `-- README.md
|-- data/
|   `-- penguins.csv
|-- R/
|   `-- functions.R
|-- analysis/
|   |-- 01-clean-data.R
|   |-- 02-fit-model.R
|   `-- 03-make-figures.R
`-- output/
    |-- figures/
    `-- models/
```

- **README** - as discussed at the start - this guides the reader through just what this is.

- **`data-raw/` holds the raw data, exactly as you received it**, along with the script that cleans it. Raw data is read-only, so nothing ever writes here, and you never edit it by hand. Keeping the cleaning script next to the raw data means the two never drift apart. Note that there is a README.md in data-raw/ to tell you a bit more about the data. You can have more than one README.
- **`data/` holds the analysis-ready data** that the cleaning script produces. This is the version your analysis actually reads.
- **`R/` holds functions.** Code that gets *used*, not code that gets *run*. We will spend a whole lesson on this.
- **`analysis/` holds scripts that get run**, in order. The numbering is doing real work: it tells your future self the entry point without needing a README.
- **`output/` holds things you can delete.** Anything in here can be regenerated by re-running the analysis.

If your project is small, collapse this. Three files in a flat folder with a README is a perfectly good project structure. You are not aiming for folders. You are aiming for *a place where each kind of thing goes*, and for being consistent about it.

> **Your Turn**
>
> Sketch the folder structure for a project you are working on right now. Don't create it yet - just write it out.
>
> 1. Where does raw data go?
> 2. Where do the things you can regenerate go?
> 3. Where does someone start reading?

1.21. How to name files

Naming is hard. We will come back to this throughout the course.

Having this "good enough" folder structure tells you **where** things go, but it does not tell you what to **name** them. So let's talk about that.

Some especially good pieces of advice:

- **Machine readable.** No spaces, no punctuation beyond `-` and `_`, and stick to one case. `01-clean-data.R`, not `01 Clean Data (final).R`.
- **Human readable.** The name should say what the file does. `clean-penguins.R` tells you something. `script2.R` does not.
- **Sorts sensibly.** Numbers at the front, zero padded, so `01`, `02`, `10` stay in order. Dates as `YYYY-MM-DD`, which sorts correctly for free.

That last one is the sneaky one. If your file names sort properly, your file explorer does your project organisation for you, and you get the running order without needing a README.

If you want to read more about this, I suggest:

- The files chapter of the Tidyverse style guide. It is short, and it is worth your time reading.
- Jenny Bryan's talk, "How to name files like a normie"

> **Your Turn**
>
> Look at the file names in your most recent project.
>
> 1. How many have spaces in them?
> 2. If you sorted them alphabetically, would the order mean anything?
> 3. Pick the worst named file and give it a better name.

1.22. Make your project "good enough"

I want to be clear that the goal here is *good enough*, not perfect. This phrasing comes from a paper I recommend to everyone, "Good enough practices in scientific computing" by Wilson et al.

The principles I would hold you to:

1. **Put each project in its own folder, with its own `.Rproj` file.**
2. **Treat raw data as read-only.** Never edit it by hand.
3. **Treat generated output as disposable.** If you can't delete it and get it back, it isn't really output.
4. **Use relative paths.** Never `setwd()`.
5. **Name files so a stranger can guess what they do.**
6. **Make the running order obvious**, with numbered scripts or a README.
7. **Write a README.**

That is the whole list. If you do those seven things, you are organised enough, and the remaining effort is better spent on the analysis itself.

One more thing:

> Don't reorganise everything tonight.

Apply this to your *next* project. Retrofitting a large existing project is a big job, and can be a very effective way to talk yourself out of the whole idea. Start clean, once, and let the habit build.

> **Your Turn**
>
> Go through the seven principles above and score your most recent project out of seven.
> Pick the single one that would have saved you the most time, and write down what you would have to change to fix it.

Summary

- You are always collaborating with your future self.
- A file path is the address of a file; absolute paths only work on one machine.
- Use one folder and one `.Rproj` per project, and relative paths within it.
- `here::here()` builds paths from the project root, wherever the code is called from.
- Raw data is read-only; output is disposable.
- Make the running order obvious.
- Write a README that says what, how, and where from.
- Start R with a blank slate, and restart often.

Next, we will look at what happens *inside* those files: how code is styled, and why consistency makes code readable.

2. Code style and readability

> Good coding style is like correct punctuation: you can manage without it, butitsuremakesthingseasiertoread.
>
> – Hadley Wickham:

There is more to your code than having it run. We are aiming for your code to be "running" AND "easy to read, and easy to understand".

There are some really useful tools we can apply to get quick wins on improving the readability of your code. In this section we will discuss concepts of **code style**, talking about how to easily improve code **layout (whitespace, newlines, punctuation standards)**, and improving code with **code linting** - where a program looks at how your code is written and identifies common errors, style inconsistencies, and other known problem patterns.

Overview

Duration 45 minutes
Questions

- Why does style matter, if the code runs either way?
- What counts as style: layout, naming, or correctness?
- How should I lay out a pipeline, and what should I call things?
- What is a code formatter, and how do I get one to run on save?
- What is a linter, and what does it catch that a formatter does not?

2.1. Reading things that are hard to read is hard

Which of these is easier to read?

2.2. quote 1

> i dont understan an ingle werd sumtimes of wot i say, but it is becoz i am so cleva

– Oscar Wilde, The Happy Prince and Other Stories

2.3. quote 2

> I am so clever that sometimes I don't understand a single word of what I am saying

– Oscar Wilde, The Happy Prince and Other Stories

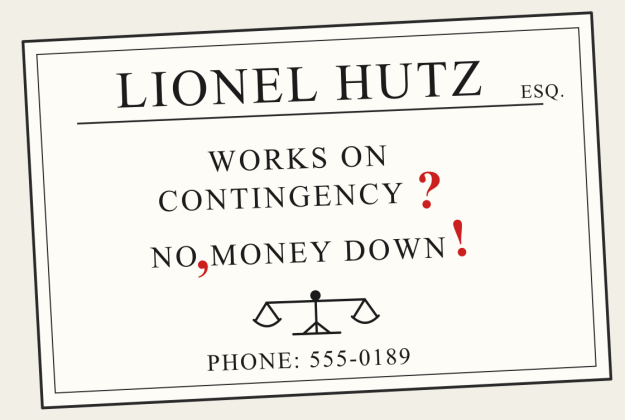
One of these is harder to read than the other. It is easier to read the text that has correct spelling and grammar.

Punctuation is an example of grammar - it plays a role in deciding what the words mean.

Here is Lionel Hutz, from The Simpsons, attorney at law.

2.4. The card

2.5. Marked up



The punctuation changed the meaning!

You can still work out what it says. But you spend energy doing it. Our precious mental energy.

We want our code to be easier to understand, when we're reading a big lump of text:

2.6. As it arrived

> we ran the model on the new data and it's results where different to what we expected. the team thought maybe the cleaning script had changed But after checking, changed it had not. Their was one column read in as character not a number, which meant the mean was computed over text and returned NA this is the kind of thing thats easy to miss when your reading quickly

2.7. Tidied up

> We ran the model on the new data, and its results were different to what we expected. The team thought maybe the cleaning script had changed, but after checking, it had not. There was one column read in as a character rather than a number, which meant the mean was computed over text and returned `NA`. This is the kind of thing that is easy to miss when you are reading quickly.

Both say the same thing. The second one you read once.

Going through the first, you were doing unpaid work the whole way - and these little bumps interrupt your thinking:

- Silently correcting "their" to "there".
- Has a sentence ended? There was a capital letter.
- Re-reading the sentence, it sounds backwards.

Our energy and our attention are finite resources, and if we spend them on these smaller things, then we will spend less energy on the goal: understanding the meaning.

2.7.1. CPU cycles vs mental cycles

A computer has CPU cycles, and we spend real effort saving them - avoiding a copy of a big object, shaving milliseconds off something that runs a thousand times.

You have **mental cycles** too, and yours are far more expensive.

So: terse code saves the machine a tenth of a second, and then you spend an extra minute reading it every time you come back. Nothing got faster. The work moved off the machine, which has cycles to spare, and onto you, who does not.

Spend the computer's freely. Guard your own.

Code is the same, and R has its own version of the Hutz card:

```
x <- 1
x < -1
```

```
[1] FALSE
```

The first line assigns `1` to `x`. The second asks whether `x` is less than negative one. One space, and a completely different meaning.

R accepted both without complaint, which is the whole problem.

Both of these take the chicks at day 21, and ask which diets came out above the average weight.

```
library(tidyverse)
d<-ChickWeight
M<-mean(d$weight)
d<-d▷filter(Time==21)
r<-d▷group_by(Diet)▷summarise(m=mean(weight),n=n())▷arrang
↪  e(desc(m))▷mutate(f=m>M)
r
```

```
# A tibble: 4 x 4
  Diet      m     n f
  <fct> <dbl> <int> <lgl>
1 3      270.    10 TRUE
2 4      239.     9 TRUE
3 2      215.    10 TRUE
4 1      178.    16 TRUE
```

```
library(tidyverse)

overall_mean <- mean(ChickWeight$weight)

chicks_day_21 <- ChickWeight ▷
  filter(Time == 21)

diet_summary <- chicks_day_21 ▷
  group_by(Diet) ▷
  summarise(
    weight_mean = mean(weight),
    n_chicks = n()
  ) ▷
  arrange(desc(weight_mean)) ▷
  mutate(above_overall_mean = weight_mean > overall_mean)

diet_summary
```

```
# A tibble: 4 x 4
  Diet  weight_mean n_chicks above_overall_mean
  <fct>       <dbl>    <int> <lgl>
1 3            270.       10 TRUE
2 4            239.        9 TRUE
3 2            215.       10 TRUE
4 1            178.       16 TRUE
```

These both run. R does not care where you put your spaces. R does not care that your variables are short, that your code is very long-winded. Or short-winded.

But you care about these things - every stray space, and every inconsistent name is one more small correction you make before you can get at what the code is trying to do.

You are spending your mental cycles on this problem. It would be nice if you didn't have to do that.

2.7.2. Both of those are wrong

Let's look closely at that last column. Every diet is above the overall mean. Weird? This happens because the overall mean is taken from `ChickWeight` **before** the filter. It is the average weight across every day of the study - day 0 chicks included, when they weighed about 40 grams. Every day 21 chick clears that easily.

```
mean(ChickWeight$weight) # every day
```

```
[1] 121.8183
```

```
mean(ChickWeight$weight[ChickWeight$Time == 21]) # day 21 only
```

```
[1] 218.6889
```

Both versions have the bug. Good layout doesn't fix the bug, but it makes it (in my opinion!) easier to notice. `M` and `m` tell you nothing, so there is nothing to be suspicious about. `overall_mean` and `weight_mean` sitting on the same line, makes us ask: the overall mean of *what*?

The style helps buy back some of your attention.

And that is part of what this lesson is about - there are ways to automate a lot of this! And there are ways to turn the rest into a habit, so it stops being a decision you make every time you write a line.

Nearly all of that surface level noise is systematic, which means we can get rid of it systematically.

And once it's gone, your attention goes to the part that actually needs you: whether the code does the right thing.

2.8. Style is like grammar

Style guides define a set of rules that keep code easy to read. To quote Hadley Wickham:

> Good coding style is like correct punctuation: you can manage without it, butitsuremakesthingseasiertoread.

2.8.1. Style: layout, naming, correctness

It helps to split "style" into two parts, because they need very different things from you.

Layout. Where the spaces go, where the line breaks go, how deep the indentation is, how a long function call gets split across lines.

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

This part is completely mechanical, and a tool can do all of it.

Naming. What you call your variables, your functions, and your files.

```
# naming
d2 <- d[d$Time == 21, ]
chicks_day_21 <- ChickWeight[ChickWeight$Time == 21, ]
```

This one is yours, because a name is a decision about meaning.

There is a third thing that is not style at all, but often gets lumped in with it.

Correctness. Whether your code actually does what you think it does.

This one is worth doing slowly, with two examples, because the failures are quiet.

Comparing two numbers. You would think this is safe:

```
0.1 + 0.2 == 0.3
```

```
[1] FALSE
```

No! Those numbers are stored as binary approximations and the answer is a hair off:

```
0.1 + 0.2 - 0.3
```

```
[1] 5.551115e-17
```

So you reach for `all.equal()`, which allows a small tolerance:

```
all.equal(0.1 + 0.2, 0.3)
```

```
[1] TRUE
```

Good. Now watch what it does when the two things are genuinely different:

```
all.equal(1, 2)
```

```
[1] "Mean relative difference: 1"
```

It does not return `FALSE`. It returns **a sentence**. And `if()` cannot do anything with a sentence:

```
if (all.equal(1, 2)) "same" else "different"
```

```
Error in `if (all.equal(1, 2)) ...`:
! argument is not interpretable as logical
```

The fix is `isTRUE()`, which asks the narrower question "is this exactly `TRUE`":

```
if (isTRUE(all.equal(1, 2))) "same" else "different"
```

```
[1] "different"
```

What that means in an analysis. While the two values agree, `all.equal()` returns `TRUE`, your `if` works, and you never learn there is a problem. It only breaks on the day the values differ - which is precisely the day you wrote the check for.

Recoding a factor. Here is one that does not error at all:

```
grade <- factor(c("low", "high", "low"))
ifelse(grade == "low", grade, "other")
```

```
[1] "2"     "other" "2"
```

You asked for `"low"` and got `"2"`. A factor is stored as integers with labels attached:

```
as.integer(grade)
```

```
[1] 2 1 2
```

```
levels(grade)
```

```
[1] "high" "low"
```

`ifelse()` drops the labels and hands back the bare codes. So `"low"` became `"2"`, its position in the alphabetical list of levels.

What that means in an analysis. Nothing warns you. You now have a column where a category has been replaced by its alphabetical rank, and every join, plot and table built on it downstream is quietly wrong. Add a new level next year and the numbers all shift.

`dplyr::if_else()` is stricter about types and keeps the factor:

```
dplyr::if_else(grade == "low", grade, factor("other"))
```

```
[1] low   other low
Levels: high low other
```

Both of those problems are laid out perfectly well. Every space is in the right place. A formatter has nothing to say about either of them, and neither does a careful read, most days. That is a *linter's* job, and we come back to it later in this chapter.

So.

- Layout is automatable
- Naming is yours
- Correctness is a different tool

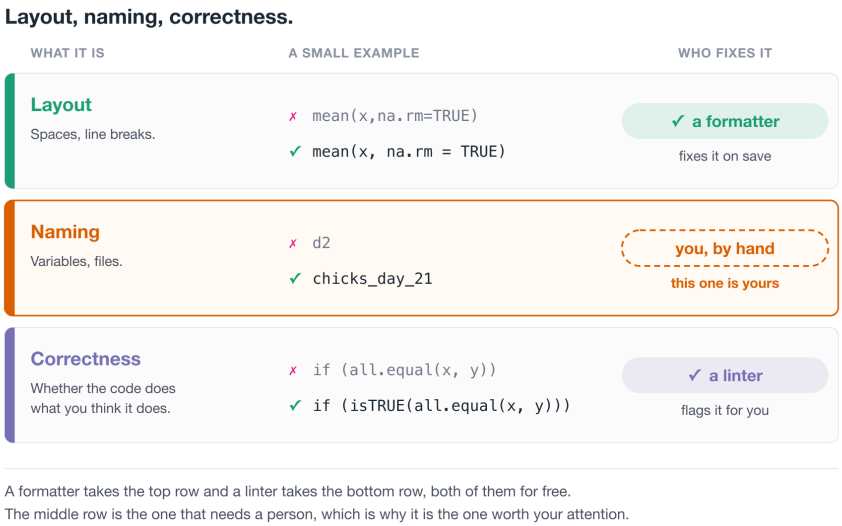


Figure 2.1.: A formatter takes the top row and a linter takes the bottom row. The naming in the middle is the row that needs a person.

Let's get into it.

2.9. Layout

Lines are free. Use them.

> **Your Turn: a quick check on pipes**
>
> Everything from here on uses the pipe, `▷`, so let's make sure we are all in the same place before we start.
>
> - Have you used `▷`, or `%>%`, before?
> - Could you say out loud what this does?
>
> ```
> penguins ▷ filter(species == "Adelie")
> ```
>
> I like to read the pipe as **then**:
>
> > Take the penguins data, **then** filter it to the rows where species is "Adelie".
>
> The pipe gets a proper treatment in Workflow: code style in *R for Data Science*. It's a great book!

Let's look at the same code twice.

2.10. Hard work

```
penguin_summary<-penguins▷filter(!is.na(bill_len))▷group_by
↪ (species,island)▷
summarise(bill_mean=mean(bill_len),bill_sd=sd(bill_len),n=n())
```

2.11. Easy work

```
penguin_summary <- penguins ▷
  filter(!is.na(bill_len)) ▷
  group_by(species, island) ▷
  summarise(
    bill_mean = mean(bill_len),
    bill_sd = sd(bill_len),
    n = n()
  )
```

2.12. No pipe at all

```
penguin_summary <- summarise(
  group_by(
    filter(penguins, !is.na(bill_len)),
    species,
    island
  ),
```

```
  bill_mean = mean(bill_len),
  bill_sd = sd(bill_len),
  n = n()
)
```

All three give the same answer. R sees no difference at all.

But in the second one you can see the shape of the analysis:

- filter()
- group_by()
- summarise()

Three steps, one per line, each indented to show it belongs to the pipeline.

In the first one you have to parse it yourself, character by character, before you can even start thinking about whether the analysis is right.

The third one is the interesting case, because it is the one that looks fine. Every space is in the right place, the indentation is correct, and a formatter would leave it exactly as it is.

It is still harder to read, because the steps now run inside out. `filter()` happens first and sits deepest. `summarise()` happens last and sits at the top. To read it you have to find the innermost bracket, then work your way out, holding each layer in your head as you go. To read the pipe version you start at the top and go down.

That is a different problem from layout, and no formatter will fix it for you. It is also where rainbow parentheses start earning their keep - once you are three or four brackets deep, matching them by eye becomes its own small job, on top of the actual work.

Your Turn

Take this and work out, by reading alone, what it produces:

```
x<-lm(body_mass~flipper_len+species,data=penguins[pengui
↪  ns$year==2008&!is.na(penguins$sex),])
```

1. How many things are being filtered out?
2. What is the response variable?
3. How long did that take you?
4. How would you rewrite this?

2.12.1. The one line special

I once saw someone writing code where the goal, as far as I could tell, was to have as few lines as possible.

Not fewer *steps*. Fewer *lines*.

It was an entire analysis. Read the data in, reshape it, fit a model, draw a plot, save it. Four lines.

Have a look at both of these, and see which one you would rather open on a Monday morning.

2.13. Four lines

```
penguins <- read_csv("data/penguins.csv") ▷
 ↪  filter(!is.na(bill_length_mm), !is.na(body_mass_g)) ▷
 ↪  mutate(mass_kg = body_mass_g / 1000, bill_ratio =
 ↪  bill_length_mm / bill_depth_mm) ▷ group_by(species) ▷
 ↪  mutate(mass_z = (mass_kg - mean(mass_kg)) / sd(mass_kg))
 ↪  ▷ ungroup()
fit <- lm(mass_kg ~ bill_ratio + species + island, data =
 ↪  penguins)
p <- ggplot(penguins, aes(x = bill_ratio, y = mass_kg, colour
 ↪  = species)) + geom_point(alpha = 0.6) + geom_smooth(method
 ↪  = "lm", se = FALSE) + facet_wrap(~island) + labs(x = "Bill
 ↪  length / bill depth", y = "Body mass (kg)") +
 ↪  theme_minimal(base_size = 12)
ggsave("output/figures/mass-bill-ratio.png", p, width = 8,
 ↪  height = 5, dpi = 300)
```

2.14. The same thing, laid out

```
penguins <- read_csv("data/penguins.csv") ▷
  filter(
    !is.na(bill_length_mm),
    !is.na(body_mass_g)
  ) ▷
  mutate(
    mass_kg = body_mass_g / 1000,
    bill_ratio = bill_length_mm / bill_depth_mm
  ) ▷
  group_by(species) ▷
  mutate(mass_z = (mass_kg - mean(mass_kg)) / sd(mass_kg)) ▷
  ungroup()

fit <- lm(mass_kg ~ bill_ratio + species + island, data =
 ↪  penguins)

p <- ggplot(penguins, aes(x = bill_ratio, y = mass_kg, colour
 ↪  = species)) +
  geom_point(alpha = 0.6) +
  geom_smooth(method = "lm", se = FALSE) +
  facet_wrap(~island) +
  labs(
```

```
    x = "Bill length / bill depth",
    y = "Body mass (kg)"
  ) +
  theme_minimal(base_size = 12)

ggsave(
  "output/figures/mass-bill-ratio.png",
  p,
  width = 8,
  height = 5,
  dpi = 300
)
```

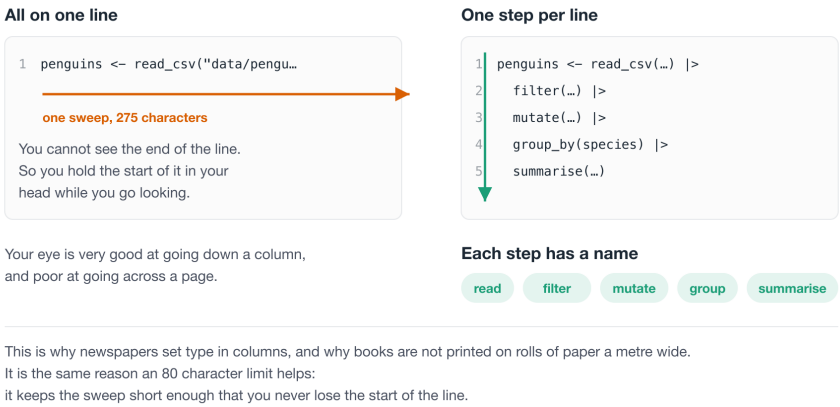


Figure 2.2.: Why code reads better down a column than across a page.

Same code. Same results.

But look at what your eye can do with the second one. You can run down the left hand edge and read the steps off like a list: filter, mutate, group by, mutate, ungroup.

Your eye is very good at scanning down a column. It is terrible at scanning across a page, which is why newspapers use columns and why books are not printed on rolls of paper a metre wide.

The first version makes you do the horizontal thing. On a laptop you cannot even see the end of those lines without scrolling sideways, so you are holding the start of the line in your head while you go looking for the end of it.

Those are mental cycles, and you are burning them on the shape of the code rather than on the analysis.

There is a second thing that shows up the moment something breaks. R tells you which line the error is on. In the first version that is not much help, because a quarter of your analysis is on that line.

Lines are free. Use them.

2.15. Naming

This is the half of style that a formatter cannot help you with.

Pick a convention, `snake_case`, `camelCase`, or `CamelCase`, and stick to it. I prefer `snake_case` because I find it easier to read, but the choice matters far less than the consistency.

Consistency matters for a reason more concrete than tidiness. Compare:

```
weight_mean <- mean(ChickWeight$weight)
sd_weight <- sd(ChickWeight$weight)
```

with:

```
weight_mean <- mean(ChickWeight$weight)
weight_sd <- sd(ChickWeight$weight)
```

The first pair changes the order of the naming. The second sticks to one pattern, `variable_operation`, where the first word says what we measured and the second says what we did to it.

Why does that help? **Tab complete.**

If I am consistent, I can type `weight_` and press tab, and R tells me every single thing I have done to `weight`.

If I had chosen the other order, `sd_` would tell me everything I have taken a standard deviation of:

```
sd_weight <- sd(ChickWeight$weight)
sd_time <- sd(ChickWeight$Time)
```

Either order is fine. What you can't do is mix them.

Because then you have to *remember* which way round you named each thing, and that's exactly the kind of small tax that makes code tiring to work with.

2.16. Inconsistent

```
sd_Weight
SD_weight
Long_variable_Name
timeSD
```

2.17. Consistent

```
weight_mean
weight_sd
time_mean
time_sd
```

2.18. Read a style guide, once

It's worth your time to read through a style guide once. Properly, start to finish, one time.

You'll notice things in your own code that you didn't see before, and you'll see other people's code differently. It's a bit like learning about kerning.

Once you see it, you notice it everywhere.

> **Your Turn**
>
> Read the style chapter of Advanced R, 1st edition. It is short, and you can get through it in about ten minutes.
> As you read, keep a list of the rules you are currently breaking. Don't fix anything yet.

> **Going deeper**
>
> When you have more time, read sections 1 to 5 of the Tidyverse style guide: files, syntax, functions, pipes, and ggplot2.
> It is the more complete treatment, and it explains the reasoning behind each rule rather than just stating it.
> The files chapter is worth reading alongside the project organisation lesson, because it is really about project structure rather than code.

2.19. Do I have to do all this by hand?

No. And you shouldn't.

If you try to apply a style guide by hand you will spend your time putting spaces after commas, lining up arguments, and re-indenting things you just moved. It is tedious, and you will do it inconsistently, because it is exactly the kind of task humans are bad at and get bored by.

Worse, you'll do it *instead of* the analysis.

The good news is that layout is mechanical, and mechanical things can be automated.

2.20. Using the {air} formatter

Air is an R formatter from Posit. You point it at your code, and it lays it out properly. It is basically magic.

Once you have it wired into your editor, you stop thinking about layout. You type whatever comes out of your head, hit save, and it comes out tidy. You don't need to spend another moment going over your code tweaking spaces.

2.20.1. Checking air is installed

You installed air in the setup chapter - Code formatters. We just need to check it is there, and find out where it lives.

> **Your Turn: check air is installed**
>
> 1. Open a terminal. In RStudio that's the Terminal tab, next to the Console.
> 2. Run `air --version` and check you get a number back.
> 3. Run `which air` on macOS or Linux, or `where air` on Windows, and **write the path down**. You need it in a minute.
>
> If you get "command not found", the install didn't finish - go back to Code formatters and run it again.
> If it won't install, don't spend the whole session fighting it. Use `{styler}` for today, and come back to air later.

2.20.2. If you can't install air, use {styler}

If you can't install {air} because of network issues, use `{styler}`. This is an R package that pre dates {air}. Install it with:

```
install.packages("styler")
```

It follows the same tidyverse style guide air does, so the output is very close.

There are a few differences between air and styler, basically: air is faster, and it is less configurable.

2.21. Format on save (aka magic)

This changes how you work!

Once this is set up, you write code however it falls out of your head. Wonky spacing, arguments strewn across the line, indentation all over the place. Do not fix any of it.

Hit `Cmd / Ctrl + S`, and it is all just correct.

I cannot really oversell this.

2.22. RStudio

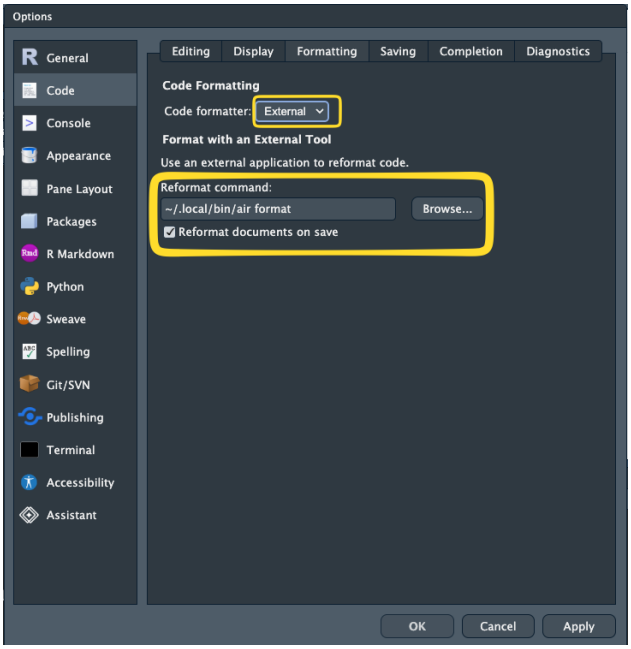
You need RStudio 2024.12.0 or later.

You want the path you wrote down a moment ago, from `which air` or `where air`.

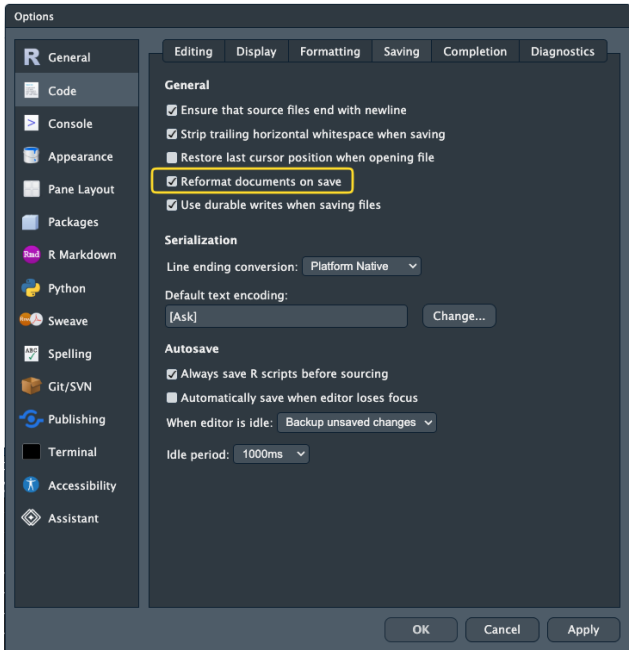
Go: Tools → Global Options → Code → Formatting.

- For {air} Set **Code formatter** to **External**.
- For {styler} Set **Code formatter** to **Styler**.

Set **Reformat command** to your air path followed by `format`, so something like `/Users/nick/.local/bin/air format`. If your path has spaces in it, wrap it in double quotes.



Now turn on format on save. Go: Tools → Global Options → Code → Saving, and tick **Reformat documents on save**.



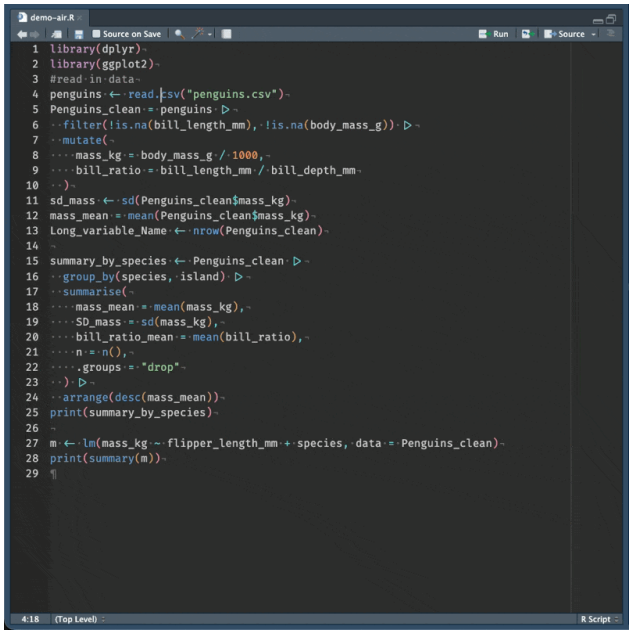


Figure 2.3.: The file after saving. This one is a screen recording in the web version of the book.

2.23. Positron

Air comes with Positron, so there is nothing to install.

Open the command palette (`Cmd/Ctrl + Shift + P`) and run **Air: Initialize Workspace Folder**. This writes the recommended settings into your project.

Figure 2.4.: What the command writes for you. This one is a screen recording in the web version of the book.

If you'd rather do it by hand, add this to your `settings.json`:

```
{
  "[r]": {
    "editor.formatOnSave": true,
    "editor.defaultFormatter": "Posit.air-vscode"
  }
}
```

For R chunks inside Quarto documents, add the Quarto formatter too:

```
{
  "[quarto]": {
    "editor.formatOnSave": true,
    "editor.defaultFormatter": "quarto.quarto"
  }
}
```

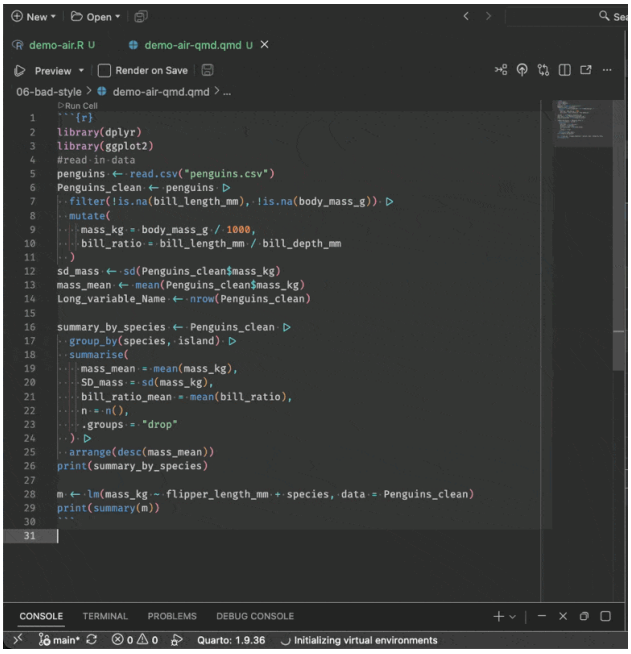


Figure 2.5.: The R chunk inside a Quarto document, after saving. This one is a screen recording in the web version of the book.

Setting it up from R

If you're inside a project already, this does most of the work for you:

```
usethis::use_air()
```

It sets up the editor configuration and adds an `air.toml` to your project, which gives you some options to customise air.
This isn't mandatory, but can be useful.

Your Turn: the good bit

1. Set up format on save in your editor, using the tabs above.
2. Open any R script. Make a mess of it on purpose. Delete the spaces around a ←, jam some arguments together, break the indentation.
3. Hit `Cmd / Ctrl + S`.
4. Do that three or four more times, with a different kind of mess each time,

I want you to do it repeatedly, because you don't have to tidy your own code, and that's awesome.

2.23.1. Formatting a file, or a whole project

You don't need format on save, or even an editor. Both tools will format one file, or everything at once.

2.24. air

Air will format a single file:

```
air format analysis/01-clean-data.R
```

or an entire project:

```
air format .
```

That second one is the command I run most. It is also the one worth putting in front of any code review, so that nobody spends their attention on spacing.

2.25. styler

Styler does the same two jobs from inside R, so there is no terminal involved:

```
styler::style_file("analysis/01-clean-data.R")
styler::style_dir(".")
```

> **Your Turn**
>
> There is a deliberately awful script in the course repository at `06-bad-style/messy-analysis.R`. It breaks essentially every rule in this chapter.
>
> 1. Open `messy-analysis.R` and read it. Note how long it takes you to work out what it does.
> 2. Format it with air or styler.
> 3. Read it again. What can you see now that you couldn't before?
> 4. Look at the variable names in the formatted version. Air has not touched a single one of them, and they're still a mess. Write down three you would rename.

> **Notes on using air**
>
> A few things that will save you some confusion.
> **There is very little to configure, and that's on purpose.** Air's settings live in an `air.toml` file, and there are about ten of them. The ones you might actually change are `line-width` (default 80) and `indent-width` (default 2). There is no way to spend an afternoon tuning it, which is exactly why I like it.
> **It won't work on an untitled tab.** Air needs a file that exists on disk, with an `.R` extension, so that it knows it's looking at R code. Save the file first.
> **In RStudio it won't work inside R chunks in Quarto or R Markdown.** This is an RStudio limitation rather than an air one, and it may well change. Positron handles chunks fine.
> **If your code doesn't reformat, you probably have a syntax error.** Air will not format code it cannot parse. So a file that stubbornly refuses to tidy up is telling you something useful. Go and find your missing bracket.
> That last one turns out to be a small free bonus. Format on save quietly becomes a syntax check on save.

2.26. Using linters

Air has now taken care of how your code *looks*. Here is a piece of code where that isn't the problem.

```r
sites <- character(0)

for (i in 1:length(sites)) {
  message("Processing site ", sites[i])
}
```

Air is perfectly happy with that. Every space is in the right place, the indentation is correct, and running `air format` on it changes nothing

at all.

It is also broken.

When `sites` is empty, `length(sites)` is `0`, and `1:0` does not give you nothing. It gives you this:

```
sites <- character(0)
1:length(sites)
```

```
[1] 1 0
```

So the loop runs twice, on elements `1` and `0`, neither of which exists. No error, no warning, and a message about a site that isn't there.

This is a different kind of problem, and it needs a different kind of tool.

- A **formatter** changes how your code *looks*. `{air}` doesn't care what your code does.
- A **linter** tells you what your code might be doing *wrong*. It reads your code without running it, which is called static analysis, and flags errors, bugs, and suspicious patterns.

You want both. Formatting settles arguments about layout. Linting catches the class of mistake that runs perfectly happily and gives you the wrong answer.

> Some history: why is it called a linter?
>
> The name comes from a program called `lint`, written by Stephen C. Johnson at Bell Labs in 1978 to check C code for bugs and portability problems. You can read more on the Wikipedia page for lint.
> Lint is the fluff that clothes shed in the wash, and a clothes dryer has a lint trap to catch it. Johnson's idea was that his program would work the same way, catching the waste fibres while leaving the fabric intact.
> Your code is the fabric. The linter is the trap.
> I like this because it sets the right expectation. A lint trap does not tell you the jumper was a bad idea. It picks off the fluff, and it does it every single load, without being asked.

2.26.1. jarl

Jarl, which stands for Just Another R Linter, is a fast linter by Etienne Bacher. It is built on top of Air, which is why the two feel like the same tool.

Here is a script of the kind most of us have written. Nothing exotic. Read some data in, flag a few things, print a message.

```r
penguins <- read.csv("data/penguins.csv")

missing_mass <- penguins$body_mass_g == NA

penguins$heavy <- ifelse(penguins$body_mass_g > 5000, T, F)

has_data <- nrow(penguins) > 0

if (has_data == TRUE) {
  message("Loaded ", nrow(penguins), " penguins")
}
```

Air has no complaints. It runs without an error.

Point a linter at it, though, and you get three warnings. Use whichever of these you have:

2.27. jarl

```
$ jarl check penguin-check.R
```

2.28. flir

```r
flir::lint("penguin-check.R")
```

```
warning: equals_na
 --> penguin-check.R:3:17
  |
3 | missing_mass <- penguins$body_mass_g == NA
  |                 ------------------------- Comparing to NA with `==`,
  |
  = help: Use `is.na()` instead.

warning: true_false_symbol
 --> penguin-check.R:5:55
  |
5 | penguins$heavy <- ifelse(penguins$body_mass_g > 5000, T, F)
  |                                                       - `T` and `F` ca
  |

warning: redundant_equals
 --> penguin-check.R:9:5
  |
9 | if (has_data == TRUE) {
  |     ---------------- Using == on a logical vector is redundant.
  |
```

Let's take those one at a time, because each one teaches something.

`missing_mass ← penguins$body_mass_g == NA`

This is the one I want you to remember. It looks completely reasonable, and it is always wrong.

`NA` means "I don't know". So asking "is this value equal to a value I don't know?" can only be answered with "I don't know":

```
x <- c(1, NA, 3)
x == NA
```

```
[1] NA NA NA
```

Every answer is `NA`, including for the values that are obviously not missing. Your `missing_mass` vector contains no information at all! Note that jarl even told you want to do!

```
is.na(x)
```

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

That's the function you wanted.

`ifelse( … , T, F)`

`TRUE` and `FALSE` are reserved words in R. `T` and `F` are just ordinary variables that happen to start out holding `TRUE` and `FALSE`, which means anyone can reassign them.

```
T <- FALSE
mean(c(1, NA, 3), na.rm = T)
```

```
[1] NA
```

```
rm(T)
```

`na.rm = T` quietly became `na.rm = FALSE`, and the mean is now `NA`. Spell them out and this cannot happen to you.

`if (has_data == TRUE)`

`has_data` is already `TRUE` or `FALSE`. Comparing it to `TRUE` gets you back exactly what you started with, so `if (has_data)` says the same thing with less to read.

This one is not a bug. It is a mental cycle.

! Two nastier ones, for later

These are much harder to spot by eye.
`class(dat) == "data.frame"` looks fine until you meet an object with more than one class:

```
class(data.frame(x = 1))
```

```
[1] "data.frame"
```

```
class(datasets::ChickWeight)
```

```
[1] "nfnGroupedData" "nfGroupedData"  "groupedData"    "data.frame"
```

One class for a plain data frame, four for that one. So the comparison hands `if()` four answers where it wanted one, and errors. `inherits(dat, "data.frame")` asks the question you actually meant.
`if (all.equal(x, y))` is worse. `all.equal()` returns `TRUE` when things match, and a *description of the difference* when they don't:

```
all.equal(1, 1)
```

```
[1] TRUE
```

```
all.equal(1, 1.1)
```

```
[1] "Mean relative difference: 0.1"
```

`if()` can't do anything sensible with it. So the code works perfectly while your values match, and falls over the first time they don't, which is precisely the case you wrote the check for. `isTRUE(all.equal(x, y))` fixes it. (demo).

Notice that these fail in two different ways.

`== NA` and `na.rm = T` are **silent**. They run, there is no error and no warning, and the answer is wrong.

That is what a linter is for. Careful reading **could** catch all of these. But a tool catches them **every** time.

Where a fix is clear and safe, jarl or flir can apply it for you:

2.29. jarl

```
$ jarl check penguin-check.R --fix
```

2.30. flir

```
flir::fix("penguin-check.R")
```

which turns the `T` and `F` into `TRUE` and `FALSE`. It won't touch the `== NA`, because deciding what you actually meant there is your job, not the tool's.

It ships 55+ rules, integrates with VS Code, Positron and Zed, and runs in CI. It's also very fast. On the `dplyr` source, roughly 25,000 lines of R, the documented benchmark is 0.131 seconds against 18.5 seconds for `{lintr}`.

> What about lintr?
>
> `{lintr}` is the long established R linter, it's pure R, and it's what you'll meet in most existing projects and CI setups. It's a perfectly good tool and knowing it is worthwhile.
> Jarl is newer and much faster, which matters more than it sounds. A linter you run on every save changes your habits.
> Use something! Either one of them.

2.30.1. If you can't install jarl, use {flir}

Same story as air. Jarl is a command line tool, so the install can go wrong in all the same ways.

Your backup is `{flir}`, which is an ordinary R package:

```
install.packages("flir")
```

It's by Etienne as well, and it's fast for the same sorts of reasons. The documented benchmark is 172 milliseconds against 3.44 seconds for `{lintr}` on the same code.

Two functions do the work:

```
flir::lint("06-bad-style/dodgy-logic.R")
flir::fix("06-bad-style/dodgy-logic.R")
```

`lint()` tells you what it found. `fix()` applies the changes it's confident about, the same way `jarl check --fix` does.

Call them with `flir::` in front, as above. If you run `library(flir)` instead, its `fix()` will mask the `fix()` that comes with R, and you'll get a warning.

One thing you should know. Etienne has moved on to jarl, and says on the flir site that he won't be adding new rules to it. So flir is finished rather than growing.

For our purposes that's fine. It catches everything in this chapter, it installs anywhere R installs, and it fixes what it can.

> **Your Turn**
>
> Back to `06-bad-style/`. There's a second file there, `dodgy-logic.R`, which is formatted perfectly well and still wrong.
>
> 1. Read it, and see if you can spot the bug by eye. Give yourself two minutes.
> 2. Run `jarl check 06-bad-style/dodgy-logic.R`. If you don't have jarl, use `flir::lint("06-bad-style/dodgy-logic.R")` or `lintr::lint("06-bad-style/dodgy-logic.R")`.
> 3. Now run `jarl check` over your own most recent project. What comes up?
>
> Nothing on that last one is a judgement. I run this over my own code and it finds things every time.

> **Read more**
>
> The style and naming material in this chapter is adapted from my blog post How to get good with R.

Summary

- Badly laid out code makes you spend mental cycles on the surface before you get to the ideas, in the same way that bad spelling does.
- Mental cycles are worth more than computer cycles. Spend the machine's, guard your own.
- Lines are free. Use them, because your eye reads down far better than across.
- Pick a naming convention and stick to it, because consistency is what makes tab complete work for you.
- Read a style guide once, properly, and you'll see code differently afterwards.
- Set up `{air}` and format on save, then stop thinking about layout.
- A linter is a different tool again, and it catches the bugs that run perfectly happily.

Next, we look at readability beyond layout: names that carry meaning, breaking a wall of code into pieces, and how to review code.

3. Writing readable code

> "Programs must be written for people to read, and only incidentally for machines to execute."
>
> – Harold Abelson and Gerald Jay Sussman, Structure and Interpretation of Computer Programs

So far we have focussed on structural things I would class as "easy wins":

- File paths
- Reproducibility fixes
- Quality of life improvements to your editor
- Creating projects
- Project structure (Adding a README!)
- Automatic styling
- Automatic linting

I really appreciate is how tools like {air}/{styler} and {jarl}/{flir} make this easy and repeatable.

Now let's talk about something that is a bit more of a skill that can't be automated: how to write readable code.

We will work on one script and discuss it, and some principles on good coding.

Overview

Duration 90 minutes
Questions

- Why go back over code you have already written?
- What are the three passes of a review, and which of them can a machine do?
- What makes a variable name good enough?
- What happens when two packages have a function with the same name?
- How do I break a wall of code into readable pieces?
- Can I say what this code is for, and can I say it more simply?
- How do I review someone else's code without it going badly?

3.1. Clean as you go

There are two core ideas I'd like to get across this chapter:

> "Programs must be written for people to read, and only incidentally for machines to execute."
>
> – Harold Abelson and Gerald Jay Sussman, Structure and Interpretation of Computer Programs

Your code is only incidentally for computers - so we want it to be well written for us to understand, and protect our mental cycles!

The other idea is:

> You won't write your best code on the first pass.

Just like writing, I don't write my best code on the first pass. Your code gets better as you review it. The first time around you write code, is for getting it working, mostly. After that, you are improving readability, and expression.

To add another analogy into the mix: Professional kitchens clean as they go, because a tidy bench keeps your mind clear and because the alternative is chaos by service.

So a best practice to add to your learning is: when you finish a piece of code, go back over it briefly. That review is **three passes**, and we spend most of this chapter on them:

1. **The cheap pass.** Fix the style, then read it top to bottom and mark up anything that catches you. Neither move asks you to understand the code.
2. **The thinking pass.** Check the names, work out what each chunk is for, and ask whether there is a simpler way to say the same thing. No tool helps you here.
3. **The tidy-up pass.** Put like with like, throw out the code you commented out and never came back to, and run a linter over everything you moved.

The order matters, and the reason is in the middle one. The thinking pass moves code around, so the tidying and the linting have to come after it, not before.

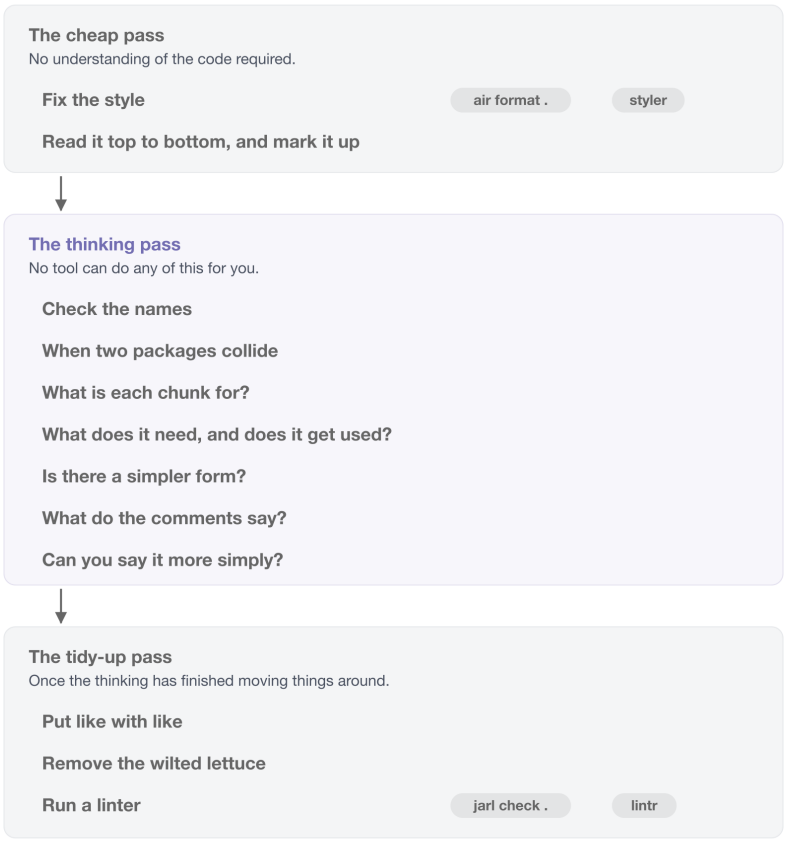


Figure 3.1.: Three passes. The middle one is the biggest because that is where the work is.

Now, you can't do a full review of your code, all the time.

But **doing some of it** will make your code better - and you'll get faster at it the more you practice.

Let's talk through these review practices by reviewing an example script.

3.2. The script

Imagine that someone hands you a script. It runs, it makes a plot at the end, and they would like another year of data added to it. Today, ideally.

Here's the script! Don't read it closely yet. Scroll through it. What do you notice?

```
library(readxl)
library(janitor)

# Read in the raw data
data <- read_excel(
  path = "data/Education and work, 2023, Datacube 2 (Table
   ↪  11).xlsx",
  sheet = "2014",
```

```
  skip = 4,
  n_max = 32,
  # use janitor::make_clean_names to turn `15-19 years` to
   ↪  "x15_19_years"
  .name_repair = make_clean_names
)

# sum(is.na(data))
# nrow(data)

# clean up the silly names from excel
names(data)
the_names <- names(data)
the_names[1] <- "state_territory"
the_names2 <- the_names

library(dplyr)
library(purrr)
# subset the data down to the number of educated people
 ↪  section
data
data_subset <- data %>%
  slice(4:11) %>%
  set_names(the_names2)


# sum(is.na(data_subset))
# nrow(data_subset)

data_subset

library(tidyr)
data_studying <- data_subset %>%
  pivot_longer(
    cols = -state_territory,
    names_to = "age_group",
    names_prefix = "x",
    names_pattern = "(.*)_years",
    values_to = "n_studying",
    values_transform = as.numeric
  ) %>%
  arrange(
    state_territory,
    age_group
  ) %>%
  # remove larger 15_24/64/74 age bands
  filter(
    age_group ≠ "15_24",
    age_group ≠ "15_64",
    age_group ≠ "15_74",
    age_group ≠ "18_24",
    age_group ≠ "25_64"
```

```
  )

# names(data_studying)

data_studying
# Population in age groups
data_population <- data %>%
  slice(24:31) %>%
  set_names(the_names2)

data_population <- data_population %>%
  pivot_longer(
    cols = -state_territory,
    names_to = "age_group",
    names_prefix = "x",
    names_pattern = "(.*)_years",
    values_to = "population",
    values_transform = as.numeric
  ) %>%
  arrange(
    state_territory,
    age_group
  ) %>%
  # remove larger 15_24/64/74 age bands
  filter(
    age_group ≠ "15_24",
    age_group ≠ "15_64",
    age_group ≠ "15_74",
    age_group ≠ "18_24",
    age_group ≠ "25_64"
  )

library(forcats)
# combine the data
# I guess it's only a study rather than the true numbers?
data_joined <- data_studying %>%
  left_join(data_population, by = c("state_territory",
  ↪  "age_group")) %>%
  mutate(
    age_group = as_factor(age_group),
    prop_studying = n_studying / population,
    year = 2014
  ) %>%
  relocate(
    year
  )

data_joined

library(ggplot2)
ggplot(data_joined, aes(x = prop_studying, y =
↪  state_territory)) +
```

```
  geom_col() +
  facet_wrap(~age_group)
```

This is a real script of mine, from a project called ozed that I used for an example project when teaching.

The goal of the analysis was to look at how many Australians are studying, by age group and by state, using data from the Australian Bureau of Statistics.

I want to be fair to it, because it is not **bad** code.

It runs. It's formatted. But I think we can do better.

Let's review it, using the three passes from above.

3.3. The cheap pass

Two moves, neither of which asks you to understand the code.

3.3.1. Fix the style

`air format .`, or `styler`. If you set up format-on-save back in Code style and readability 2, this has already happened and you can move on.

The other thing worth doing before you read a line: **run it from a clean session.** Restart R, run from the top. If it doesn't run, nothing else matters yet.

The linter comes at the *end*, in the tidy-up pass, and not here. That is deliberate - you are about to move code around, and there is no sense linting a layout you are about to change.

> Going deeper: if what you are reviewing is a package
>
> Everything above works on a folder of scripts. If you are reviewing an R package, there is a tool that checks the habits a package is supposed to have:
>
> ```
> goodpractice::gp()
> ```
>
> It runs `R CMD check`, a set of linters, and a pile of structural checks - whether there is a README, whether the DESCRIPTION is well formed, whether functions are too long or too complex, whether there are tests.
> It's tucked in here rather than named as a move of its own because most of what it checks only exists in a package. Pointed at an analysis project, the great majority of its checks have nothing to look at.
> Excellent tool, wrong shape for the code most of us review day to day. Worth knowing about for the day you write a package.

3.3.2. Read it top to bottom, and mark it up

Read the whole thing without fixing anything.

The urge to fix the first thing you see is strong. Resist!

On this pass you're only getting the shape of the thing, and collecting the places where the code caught you: where it didn't quite make sense, where something was unclear, where you had to read a line twice.

Mark them with a word you can search for. I use two: TODO and NOTE

```
# TODO: this filter drops 400 rows and I do not know why
# NOTE: same block as line 40, with different columns
```

`TODO` for something that needs doing, `NOTE` for something you want to remember but which might turn out to be fine.

This beats holding it in your head for two reasons.

You can find them all again with a search. Writing the note is cheaper than fixing, so you keep reading instead of disappearing down the first rabbit hole.

> What you are marking: code smells
>
> A **code smell** is a bit of code that is not wrong, but that makes you uneasy.
> The word is deliberate. It's the feeling of opening the fridge and knowing something in there has gone off before you have worked out what.
> Smells are useful because they're cheap. You don't have to prove anything to act on one - you notice that a chunk is doing three things, or that a block has been copied twice with one number changed, and you write a `NOTE`. That's all this step is.
> Refactoring, in the thinking pass, is what you do about them. Marking is not fixing.
> For a great introduction, watch Jenny Bryan's "Code smells and feels".

That is the cheap pass done. Everything from here works over the marks you just made.

3.4. The thinking pass

This is the pass that matters, and there is no tool for any of it.

Go back to the `TODO`s and `NOTE`s you left in the cheap pass. For each one, the question underneath everything else is: **can you say what the code is for, in a sentence?**

Here are two versions of the same calculation. Both give `0.27`, `0.33`, `0.81`.

3.5. Hard to say

```
d3 <- d2[d2$Month %in% c(5, 6, 7) & !is.na(d2$Ozone), ]
d3$f <- d3$Ozone > 30
res <- tapply(d3$f, d3$Month, mean)
```

3.6. Easy to say

```
prop_high_ozone <- airquality ▷
  filter(!is.na(Ozone), Month %in% 5:7) ▷
  group_by(Month) ▷
  summarise(prop_high = mean(Ozone > 30))
```

The second one has a sentence: *the proportion of days with high ozone, by month.*

For the first one I have to work out what `f` is, then what `tapply` is splitting by, then what `res` holds, and only then can I say the sentence. Same answer, and I had to run it in my head to get there.

Everything that follows is a way of moving code towards the second one. The rest of this pass is the questions I ask, in the order I ask them.

3.6.1. Check the names

Names go first, because they have the best return for the effort.

They sit in the thinking pass and not in the cheap one - renaming while you are still working out what the code does is how you end up with a confidently wrong name. Read the thing first, then name it.

> There are only two hard things in computer science: cache invalidation and naming things.
>
> – Phil Karlton[1]

It's comforting to know naming things is genuinely hard. I think we can sometimes let perfect become the enemy of good so when it comes to names - remember: **It is OK to have OK names**

> An OK name is better than a bad name An OK name is easier than a great name Use OK names

How do give things OK names?

> Say what it holds, in two or three words, joined by _. If a better name shows up later, rename it then.

[1]Phil Karlton was a principal developer at Xerox PARC, DEC, Silicon Graphics and Netscape. Martin Fowler has written up the provenance of the quote, which is more interesting than you'd expect. Nobody has ever found where Karlton first said it, and it seems to have travelled by word of mouth from about 1996.

If we can aim to make the names a little better - even just "OK", then that's good progress.

Here's a little demo of some names of four things:

3.7. data frame

```
# bad
d <- read_csv("data/penguins")
# OK
penguins <- read_csv("data/penguins")
# good
penguins_raw <- read_csv("data/penguins")
```

3.8. data filtered

```
# bad
df2 <- d |> filter(island == "Biscoe")
# OK
penguins_filtered |> filter(island == "Biscoe")
# good
penguins_biscoe |> filter(island == "Biscoe")
```

3.9. model

```
# bad
m <- lm(body_mass ~ species, data = df2)
# OK
model <- lm(body_mass ~ species, data = penguins_filtered)
# good
lm_mass_by_species <- lm(body_mass ~ species, data =
 ↪  penguins_biscoe)
```

3.10. plot

```
# bad
p1 <- ggplot(df2, aes(x = body_mass, colour = species)) +
 ↪  geom_boxplot()
# OK
mass_plot <- ggplot(penguins_filtered,
                    aes(x = body_mass, colour = species)) +
    geom_boxplot()
# good
```

```
boxplot_mass_species <- ggplot(penguins_biscoe,
                               aes(x = body_mass, colour =
                                ↪  species)) +
    geom_boxplot()
```

- The bad data propagates throughout the process - it becomes increasingly harder to know what you are dealing with when modelling and plotting
- `penguins_filtered` doesn't tell me *which* filter. I am OK with that because it tells me: "this is the penguin data, and it has been filtered".
- Notice the gap between bad and OK is much wider than the gap between OK and good.
- It's OK to aim for OK.

As you practice giving OK names to things, you'll get more practice at naming, and get better. But don't put too much into it.

Changing names

A name isn't a commitment! You can rename easily, and there's some good tricks.

I highly recommend doing

- "Rename in scope" (Cmd / Ctrl + Shift + Alt + M) with your cursor on the name.
  - Note that this is different to find + replace!

You can search for this in the command palette if you forget.

> **Your Turn: rename the script**
>
> The script is `10-hard-to-read/analysis.R` in the course repo, or copy it out of the listing above.
> Most of its names are already OK. `data_studying`, `data_population` and `data_joined` all tell you what they hold, and I would leave them alone. Three are worth five seconds each:
>
> - `data`
> - `data_subset`
> - `the_names2`
>
> 1. Rename those three. Aim for OK, not great.
> 2. Use **Rename in Scope** (`Cmd / Ctrl + Shift + Alt + M`) or `F2`, rather than find and replace. Find and replace on a name like `data` will reach into `data_studying` and `data_subset` too.
> 3. Read the script again afterwards. How does it feel?
> 4. `the_names2` is the interesting one. What *is* it, and why are there two?

What `the_names2` turned out to be

Here are those three lines on their own:

```
the_names <- names(data)
the_names[1] <- "state_territory"
the_names2 <- the_names
```

The third line copies the second variable into a third one and never touches it again. `the_names2` **is** `the_names`, and the only reason it exists is that somebody was not sure whether they would need the original.
The `2` is what hid it. A name ending in a number tells you there is another one somewhere, and nothing else, so you stop asking. Give it a real name, or delete the line.
While you are here: `data` is worth renaming for a second reason. There is already a `data()` function in `{utils}`, so naming your data frame `data` shadows it - which is the same hazard as When two packages collide, made by hand.

3.10.1. When two packages collide

So far this has been about names you chose. This one is about a name you didn't.

Look back at the script. It has six `library()` calls scattered through it, and further down it uses `filter()`.

Which `filter()`?

Do you know what this message here is telling us?

```
library(dplyr)
```

```
#>
#> Attaching package: 'dplyr'
#>
#> The following objects are masked from 'package:stats':
#>
#>     filter, lag
#>
#> The following objects are masked from 'package:base':
#>
#>     intersect, setdiff, setequal, union
```

R is telling you that there are now two functions called `filter()`, two called `lag()`, and that when you type one of those names it will hand you the `{dplyr}` one, because `{dplyr}` was attached most recently.

Which means the meaning of your code depends on the order of the `library()` calls at the top of your script. And that's a strange thing to be true.

Let's quickly show that as it hits you - does this look familiar?

`{MASS}` also has a `select()`:

```
library(dplyr)
```

```
Attaching package: 'dplyr'
```

```
The following objects are masked from 'package:stats':

    filter, lag
```

```
The following objects are masked from 'package:base':

    intersect, setdiff, setequal, union
```

```
library(MASS)
```

```
Attaching package: 'MASS'
```

```
The following object is masked from 'package:dplyr':

    select
```

```
starwars |> select(name, height)
```

```
Error in `select()`:
! unused arguments (name, height)
```

An error, straight away. Annoying. Familiar?

> **!** This is why library() order is not a style question
>
> Two scripts with the same `library()` but in a different order can give different answers. You also need to consider if you've got another `library()` call in the same session - a reason to make sure you have blank slate settings in Project organisation 1.

The `conflicted` pkg to the rescue!

The {conflicted} package is built for this!

Instead of silently picking the last package attached, R refuses to pick at all:

```
library(conflicted)
library(dplyr)
library(MASS)
starwars ▷ select(name)
```

```
Error:
! [conflicted] select found in 2 packages.
Either pick the one you want with `::`:
* MASS::select
* dplyr::select
Or declare a preference with `conflicts_prefer()`:
* `conflicts_prefer(MASS::select)`
* `conflicts_prefer(dplyr::select)`
```

An error, at the line that is ambiguous, naming both candidates and telling you how to fix it. I think that is about as good as an error message gets.

You then say which one you meant, once, at the top of the script:

```
library(conflicted)
library(dplyr)
library(MASS)

conflicts_prefer(dplyr::select)
```

```
[conflicted] Will prefer dplyr::select over any other package.
```

```
starwars ▷ select(name)
```

```
# A tibble: 87 x 1
   name
   <chr>
 1 Luke Skywalker
 2 C-3PO
 3 R2-D2
 4 Darth Vader
 5 Leia Organa
 6 Owen Lars
 7 Beru Whitesun Lars
 8 R5-D4
 9 Biggs Darklighter
10 Obi-Wan Kenobi
# i 77 more rows
```

This is a nice example of code that is documenting the reasoning.

You can also get {conflicted} to do a little reporting for you:

```
conflict_scout()
```

```
3 conflicts

* `filter()`: dplyr and stats

* `lag()`: dplyr and stats

* `select()`: dplyr
```

Going deeper: why not just write dplyr::select() everywhere?

You can, and some people do. It removes the ambiguity completely, and it is common practice in R package development.
I find in an analysis, it is a bit of noise. I have to look at this code a lot! A `{ggplot2}` plot block written that way is `ggplot2::` down the left hand side of every line, and the shape of the code disappears behind a prefix that is identical on every row.
`{conflicted}` gets you the same safety without the tax, because the machine does the checking instead of you.
So should you use `pkg::fun()`? Maybe - here's some thoughts on where it might be more worth it.

- A package you use once for one utility, and don't otherwise need attached.
- Deliberately disambiguating at a single call site.
- In R package development

Your Turn

1. Run `library(conflicted)` and then attach the packages your current project uses. What does `conflict_scout()` say?
2. Pick one conflict. Which function were you actually getting?
3. Add `library(conflicted)` as the first line of one of your scripts and run it. Did anything break? Anything that breaks was already ambiguous.

3.10.2. What is each chunk for?

The question to keep asking:

> How do I break this into pieces I can reason about one at a time?

Here is our script with every line coloured by the chunk it belongs to.

Figure 3.2.: Five ideas. The package colour turns up in five separate places, which is the whole argument for grouping like with like.

Five ideas in a hundred lines. And the orange band, the `library()` calls, turns up five separate times on the way down.

> Going deeper: why chunking works
>
> Here is a phone number:
>
> `1-8-0-0-1-3-1-0-8-6`
>
> Ten digits. Try to hold them in your head. Now try this:
>
> `1800 131 086`
>
> Same ten digits, and suddenly it's easy. Nothing about the information changed. Only the **chunking** did.
> This is not a party trick. Working memory holds roughly **seven, plus or minus two** chunks at once, from George Miller's 1956 paper "The magical number seven, plus or minus two". Memory isn't limited by the amount of information, but by the *number of chunks*.
> So make bigger chunks and you hold more. Thirty lines you can't hold. Five named ideas you can.

Back to our hundred lines, now with the names fixed.

Read it once, and every time the *subject* changes, put in a blank line and a comment saying what just finished.

Don't fix anything. You're drawing lines, not editing.

Here is what I get:

```
# 1. read the raw spreadsheet for one year
# 2. fix the column names Excel gave us
# 3. pull out the education rows, make them long, drop the
 ↪  wide age bands
# 4. pull out the population rows, make them long, drop the
 ↪  wide age bands
# 5. join them, and work out the proportion studying
# 6. plot it
```

Six ideas!

Run the code - look at the input, look at the output. Sometimes you can understand the inputs and outputs without understanding the nitty gritty.

Now look at what falls out, for free.

Chunks 3 and 4 are the same thing. Same `slice()`, same `pivot_longer()`, same `arrange()`, same five-line `filter()`. The only differences are which rows get sliced and what the value column is called. Forty lines doing one idea twice.

The change they asked for lives in chunk 1. Another year is another sheet in the same workbook, and nothing after that first chunk cares which year it is looking at. But the year is baked into `sheet = "2014"`, and `educated_2014`, and `population_2014`, and `year = 2014` - so adding one currently means copying eighty lines and editing five numbers inside them.

The six chunks help identify these factors.

> **Where this script goes next**
>
> Those six comments are a to-do list. Each line is a function waiting to be given a name, and the two identical steps are one function called twice.
> We pick the script up again in Writing functions 4, and take it the rest of the way in Designing functions A, until adding a year is one argument rather than eighty copied lines. The whole progression, one script per step, is in the ozed repo.

> **Your Turn: chunk something else**
>
> You have watched me do it. Now do it to something I have not touched. Use `07-needs-review/analysis.R` from the course repo, or better, one of your own from last year.
> Work it in order:
>
> 1. **The cheap pass.** `air format .` on it first, then read it top to bottom and mark it up with `TODO` and `NOTE`. Fix nothing.

2. **Check the names.** Rename three. Aim for OK.
3. **Find the chunks.** Where does one chunk end and the next begin? How many are there?

Then two questions the chunking will answer for you:

- Are any two of those chunks doing the same thing to different data?
- If someone asked you to add August, how many lines would you have to touch?

Those last two are where the work is, and you can answer both from your chunk comments alone.

3.10.3. What does it need, and does it get used?

Two questions for any chunk:

1. **what does this need in order to run**, and
2. **does it get used?**

```
penguins_raw <- read_csv("penguins.csv")                    ①
penguins <- clean_names(penguins_raw)

n_total <- nrow(penguins)                                   ②
n_missing <- sum(is.na(penguins$body_mass_g))
pct_missing <- n_missing / n_total

mass_summary <- penguins |>                                 ③
  group_by(species) |>
  summarise(mass_mean = mean(body_mass_g, na.rm = TRUE))

temp <- mass_summary |> arrange(desc(mass_mean))            ②

ggplot(mass_summary, aes(species, mass_mean)) +             ③
  geom_col()
```

① Used further down. Fine.
② **Never mentioned again.** Four objects, computed and abandoned.
③ Used. This is the actual work.

Four of the eight objects in that block are never used again.

Usually it means somebody explored, found their answer, and left the exploring in.

Sometimes it means a result was supposed to be used and quietly isn't.

A chunk with many inputs and many outputs is usually several jobs that happen to be sitting next to each other.

3.10.4. Is there a simpler form?

Sometimes code does something perfectly reasonable in a roundabout way, and R already has the direct version.

This can happen with statistics.

3.11. Mean

```
sum(x) / length(x)

mean(x)
```

3.12. Variance

```
sum((x - mean(x))^2) / (length(x) - 1)

var(x)
```

3.13. Standard deviation

```
sqrt(sum((x - mean(x))^2) / (length(x) - 1))

sd(x)
```

3.14. Counting rows

```
nrow(d[d$mass > 4000, ])

sum(d$mass > 4000)
```

3.15. Lengths of a list

```
sapply(my_list, function(i) length(i))

lengths(my_list)
```

Each pair gives the same answer. The first one says how, the second one says what. We want the reader to know thw what.

You can't spot these unless you happen to know the shortcut, which is a bit hard - it takes experience! So don't go hunting. Just notice when a line feels like more machinery than the idea deserves, and go and look.

3.15.1. What do the comments say?

Now read what the comments say, rather than reading past them.

The ones to delete are the ones that restate the code:

```
# read in the data
data <- read_excel(path)
```

The ones to keep are the ones that say *why*, which the code cannot:

```
# skip = 4 because the ABS puts four rows of notes above the
 ↪  header
data <- read_excel(path, skip = 4)
```

Our script has one of each. `# Read in the raw data` tells you nothing that `read_excel()` doesn't. `# use janitor::make_clean_names to turn 15-19 years to x15_19_years` is genuinely useful, and I would keep it.

There is a third kind, and it is the most useful one to notice. Some comments are not really comments - they are a name that has not been written yet. Our script has one at the top of the education block:

```
# subset the data down to the number of educated people
 ↪  section
```

That comment describes a step. Give the step a name and the comment has nothing left to say. The six chunk comments you just wrote are all of this kind, which is why they are a to-do list rather than documentation.

Worth reading on this: comment your code as little (and as well) as possible.

3.15.2. Can you say it more simply?

The last question is the useful one.

Can you re-express the intention of the code, and keep the output the same?

That's **refactoring**: changing code while keeping its behaviour.

It's like giving a car a service, or replacing parts that have worn. The car drives the same, it just does it more reliably.

This is where everything you marked gets cashed in. The copied block becomes a function. The four abandoned objects get deleted.

Then you run it again and check the output hasn't moved.

That check isn't optional. It's very easy to tidy something into a different answer.

> **Your Turn**
>
> 1. Run the cheap pass on `07-needs-review/analysis.R`. Actually run `air`, don't just read about it.
> 2. It still is not easy to follow. Mark three places where it caught you, with `TODO` or `NOTE`.
> 3. Pick one and re-express it, then check the output has not changed.

3.16. The tidy-up pass

The thinking is done, and it has left the code moved around. This pass puts the furniture back straight.

3.16.1. Put like with like

Once you can see the chunks, tidy them:

- All `library()` calls at the top.
- All function definitions together.
- If a variable is defined at line 10 and first used at line 150, move them closer.
- Chunks that do similar things should sit near each other.
- Add the commented headers, so the sections are visible without counting lines.
- If a section is getting long, should it be its own script?

None of this changes what the code does. All of it changes how much you've got to hold in your head.

Two questions worth asking here: **is the running order obvious** to someone who has never seen it, and **can you reason about each chunk on its own**, or do you need the whole thing in your head at once?

3.16.2. Remove the wilted lettuce

The commented-out code you might just come back to.[2]

Our script has four of them:

[2] Credit goes to Miles McBain for this name.

```
# sum(is.na(data))
# nrow(data)
# sum(is.na(data_subset))
# names(data_studying)
```

Those are someone checking their work as they went, which is a good thing to have done and not a thing to keep. They are noise now, and worse, a reader has to decide whether each one matters before ignoring it.

Delete them. If one really does matter, write a comment saying why it is there, rather than leaving a dead line lying around.

If you are worried about losing something: that is what version control is for, and it is much better at remembering than a commented-out line is.

3.16.3. Run a linter

`jarl check .`, or `lintr::lint_dir()`.

Last, not first. You have just moved code around, deleted lines and renamed things, so this is the sweep that catches what you disturbed on the way through.

> Going deeper: if what you are reviewing is a package
>
> Everything above works on a folder of scripts. If you are reviewing an R package, there is a tool that checks the habits a package is supposed to have:
>
> ```
> goodpractice::gp()
> ```
>
> It runs `R CMD check`, a set of linters, and a pile of structural checks - whether there is a README, whether the DESCRIPTION is well formed, whether functions are too long or too complex, whether there are tests.
> Most of what it checks only exists in a package. Pointed at an analysis project, the great majority of its checks have nothing to look at.
> Excellent tool, wrong shape for the code most of us review day to day. Worth knowing about for the day you write a package.

3.17. Treat your own code as someone else's

The person who wrote the code you are reviewing is not available for questions. It is you, six months ago. They had reasons for what they did, and they did not write them down.

That is the useful trick in all of this. Every question in the thinking pass gets easier if you read your own code the way you would read a colleague's: assume there was a reason, look for the intention before you judge the implementation, and be curious rather than embarrassed.

Reviewing code *for other people* is its own skill, with its own social rules, and it is what we spend the last session on in Putting it all together.

> **If you want a longer one**
>
> `08-curly-code/penguin-report.R` in the exercises repo is the same idea at greater length. It runs, it produces a report, and `lintr::lint()` finds 125 things in it across 13 rules before you get to anything a person has to see.
> Work the three passes on it end to end. The cheap pass takes about a second, and everything interesting happens after that.

> **Read more**
>
> Parts of this chapter are adapted from my blog post, How to get good with R.

4. Writing functions

> "You should consider writing a function whenever you've copied and pasted a block of code more than twice."
>
> – Hadley Wickham, Mine Çetinkaya-Rundel and Garrett Grolemund, R for Data Science

At some point you notice you have written code that looks something like the following:

```
library(dplyr)
adelie_mean <- penguins ▷
  filter(species == "Adelie") ▷
  pull(body_mass) ▷
  mean(na.rm = TRUE)
gentoo_mean <- penguins ▷
  filter(species == "Gentoo") ▷
  pull(body_mass) ▷
  mean(na.rm = TRUE)
chinstrap_mean <- penguins ▷
  filter(species == "Chinstrap") ▷
  pull(body_mass) ▷
  mean(na.rm = TRUE)
```

The issue with this is that you are repeating the same code three times, but changing only one part. In this case, only one part is changing:

```
filter(species == "Adelie")
filter(species == "Gentoo")
filter(species == "Chinstrap")
```

We can express this idea more clearly with a function, comparing the result above the function:

```
species_body_mass_mean <- function(species_name){
  penguins ▷
    filter(species == species_name) ▷
    pull(body_mass) ▷
    mean(na.rm = TRUE)
}
adelie_mean
```

```
[1] 3700.662
```

```
species_body_mass_mean("Adelie")
```

```
[1] 3700.662
```

```
gentoo_mean
```

```
[1] 5076.016
```

```
species_body_mass_mean("Gentoo")
```

```
[1] 5076.016
```

```
chinstrap_mean
```

```
[1] 3733.088
```

```
species_body_mass_mean("Chinstrap")
```

```
[1] 3733.088
```

In this chapter we discuss how to notice these moments of repition, and how to convert these into functions, so you can clearly express your ideas. We will also work through how to get inside a function when it misbehaves.

Overview

Duration 60 minutes
Questions

- What problem do functions actually solve?
- What are the parts of a function called, and why does that matter?
- What should I call a function, and what should I call its arguments?
- How do I look inside a function when it goes wrong?

4.1. The problem functions solve

I do not think I can overstate this: learning to write functions changed how I think about code, and how I think about solving problems. It does take time to develop, but the payoff is enormous.

A function isn't only a way to avoid typing something twice. It's a way to take complexity and express it in ways that you can reason with.

Once you can do that, you can hold a bigger problem in your head. It's like when you know a new word for a concept:

Benighted: Unexpectedly caught out in the dark

Scrumping: To steal fruit such as apples from trees

My journey into functions started in 2013. I was writing code to calculate a t-test from scratch. I had my reasons! At the top of my script was something like:

```
var_1 <- data$x1
var_2 <- data$x2
```

and then a pile of code that computed the result.

The whole script, if you want to see it

This is roughly what the code looked like. What I want you to notice is how much of it there is, and where the two lines I kept editing are.

```
var_1 <- data$x1                                                    ①
var_2 <- data$x2

n_1 <- length(var_1)
n_2 <- length(var_2)

mean_1 <- mean(var_1)
mean_2 <- mean(var_2)

var_within_1 <- sum((var_1 - mean_1)^2) / (n_1 - 1)
var_within_2 <- sum((var_2 - mean_2)^2) / (n_2 - 1)

pooled_var <- ((n_1 - 1) * var_within_1 + (n_2 - 1) *
 ↪  var_within_2) / (n_1 + n_2 - 2)
std_error <- sqrt(pooled_var * (1 / n_1 + 1 / n_2))

t_statistic <- (mean_1 - mean_2) / std_error
degrees_freedom <- n_1 + n_2 - 2
p_value <- 2 * pt(-abs(t_statistic), df =
 ↪  degrees_freedom)

t_statistic
p_value
```

① These two lines are the only part that ever changed.

Every time I wanted to look at a different pair of variables, I went back to the top, changed those two lines, and ran the whole script again.

Two lines out of twenty changed - and I re-ran all of them, by hand, every single time. Something felt off to me!

Figure 4.1.: Four runs of the same script. Two bars change colour.

Something about this felt wrong at the time. But I didn't have the words for it.

What I was reaching for sounded something like:

> I want to run this script, but get the top couple of lines to change depending on which variables I specify.

What I needed was a function, but I just didn't have the words for it.

I had been **taught** functions by then. And I had been **using** functions the whole time, things like `lm()`, `table()`, `subset()`, and `read.csv()`. I used them every day without ever thinking about where they came from, or that I could make my own.

But the examples I was taught with were "convert Celsius to Fahrenheit" and "identify odd numbers", and I couldn't see how those applied to anything I actually did.

They seemed like useful kitchen trinkets. An apple corer, a garlic press. Rather than a fundamental skill, like knowing how to use a knife.

Functions are the knife.

So rather than argue that, let me show you.

What does this code do?

We'll use `penguins`

```
head(penguins)
```

```
  species    island bill_len bill_dep flipper_len body_mass    sex year
1  Adelie Torgersen     39.1     18.7         181      3750   male 2007
2  Adelie Torgersen     39.5     17.4         186      3800 female 2007
3  Adelie Torgersen     40.3     18.0         195      3250 female 2007
4  Adelie Torgersen       NA       NA          NA        NA   <NA> 2007
5  Adelie Torgersen     36.7     19.3         193      3450 female 2007
6  Adelie Torgersen     39.3     20.6         190      3650   male 2007
```

```
bill_len <- penguins$bill_len
bill_dep <- penguins$bill_dep

bill_len_0 <- (bill_len - mean(bill_len, na.rm = TRUE)) /
 ↪  sd(bill_len, na.rm = TRUE)
bill_dep_0 <- (bill_dep - mean(bill_dep, na.rm = TRUE)) /
 ↪  sd(bill_len, na.rm = TRUE)
```

Before you read on, have a proper go at this one.

Your Turn

Here is the code again. Don't run it.

```
bill_len <- penguins$bill_len
bill_dep <- penguins$bill_dep

bill_len_0 <- (bill_len - mean(bill_len, na.rm = TRUE)) /
 ↪  sd(bill_len, na.rm = TRUE)
bill_dep_0 <- (bill_dep - mean(bill_dep, na.rm = TRUE)) /
 ↪  sd(bill_len, na.rm = TRUE)
```

1. In one sentence, what is this code doing?
2. There is a bug in it. Find it.
3. How long did that take you, and what made it hard?

So, what is it doing?

This is mean centering and scaling. It's a transformation that gives your data a mean of 0 and a standard deviation of 1, which some statistical models want.

But reading that code, it's hard to see that's what's happening. Nothing in it says "centre and scale". You have to reconstruct the intent from the arithmetic.

Worse, it invites errors, because you have to repeat `bill_len` or `bill_dep` three times per line.

And there **is** an error.

Look at the second line again. For `bill_dep_0`, I divide by the standard deviation of **`bill_len`**, not `bill_dep`.

That's a copy-paste bug, it's completely silent, and it produces numbers that look entirely plausible. If you found it, well done. If you didn't, that is why I would rather not rely on either of us spotting it.

A function fixes this by **abstracting away** the need to write the variable in each time. Break it into two steps:

```r
mean_center <- function(variable) {
  variable - mean(variable, na.rm = TRUE)
}

scale_sd <- function(variable) {
  variable / sd(variable, na.rm = TRUE)
}
```

Now we can write:

```r
bill_len ▷
  mean_center() ▷
  scale_sd()
```

Or write one function that does both steps:

```r
center_scale <- function(variable) {
  centered <- mean_center(variable)
  centered_scaled <- scale_sd(centered)
  centered_scaled
}
```

What is nice here is that each function **describes what it does** in its name. Reading `center_scale`, you can see it does a thing called `mean_center` first, then `scale_sd` next.

So compare the original:

```r
bill_len_0 <- (bill_len - mean(bill_len, na.rm = TRUE)) /
 ↪  sd(bill_len, na.rm = TRUE)
bill_dep_0 <- (bill_dep - mean(bill_dep, na.rm = TRUE)) /
 ↪  sd(bill_len, na.rm = TRUE)
```

with:

```r
bill_len_0 <- center_scale(bill_len)
bill_dep_0 <- center_scale(bill_dep)
```

Now you can't even write the bug, the function has removed that possibility, by clearly expressing the problem.

> **Someone has probably done this already**
>
> Base R has `scale()`. Its help page describes it as centring and scaling the columns of a numeric matrix, which is the two steps we just wrote, in one call.

```
head(center_scale(bill_len), 3)
#> [1] -0.8832047 -0.8099390 -0.6634077

head(scale(bill_len), 3)
#>            [,1]
#> [1,] -0.8832047
#> [2,] -0.8099390
#> [3,] -0.6634077
```

Same numbers. The only difference is that `scale()` hands back a matrix, because it is built to do every column of one at a time.
Which is a good argument for checking whether a solution exists before writing your own. You don't need to write everything from scratch.
At worst you find nobody has solved your problem, which is an opportunity. Usually you learn the words other people use to describe it, which makes your next search much better.

4.2. Anatomy of a function

Every function in R has the same few parts.

The reason to learn the names is practical. For example, knowing a term like a "default argument", means you can search for it, read the help page about it, and ask someone a question that gets you a useful answer. Without the words, you are stuck describing the shape of the problem and hoping.

We'll use two small functions here. The first counts how many values are missing:

```
n_missing <- function(x) {
  sum(is.na(x))
}

n_missing(airquality$Ozone)
```

```
[1] 37
```

The second uses that one to work out what **proportion** is missing:

```
prop_missing <- function(x, digits = 2) {
  prop <- n_missing(x) / length(x)
  round(prop, digits)
}

prop_missing(airquality$Ozone)
```

```
[1] 0.24
```

Notice `prop_missing()` calls `n_missing()`. You write a small function that does one thing, and then you use it inside the next one, instead of writing `sum(is.na(x))` again and hoping you type it the same way.

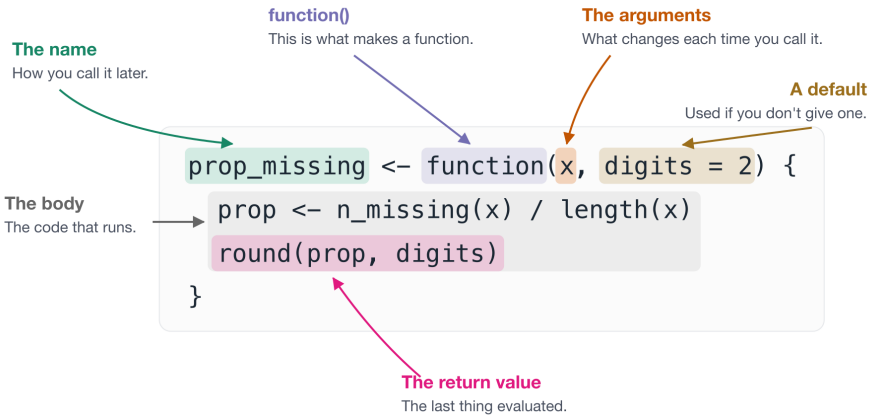


Figure 4.2.: The parts of a function.

Taking the parts one at a time:

- **The name.** `prop_missing`. This is how you call it later, so it is worth spending time to pick a good one. More on that just below.
- **`function()`.** This creates a function. `prop_missing` ← just gives it a name, the same way you would bind any other value.
- **The arguments.** `x` and `digits`. These are the what change each time you call the function. They are the answer to "what did I keep editing at the top of my script?"
- **A default.** `digits = 2`. If the caller doesn't supply `digits`, it is 2. Defaults let you make a function easy to call in the common case, without giving up control in the uncommon one.
- **The body.** Everything between the braces. This is the code that runs.
- **The return value.** R returns the **last thing evaluated**, so here that's the rounded proportion. You can write `return()` explicitly, and sometimes should for an early exit, but you don't have to.

4.2.1. Naming a function

We spent a whole chapter on naming variables, and functions get the same treatment. The difference is that a function **does** something, so the name should say what.

Watch this one improve:

```
myfun()                    # what?
temperature_conversion()   # better, but converting what to
 ↪  what?
celcius_to_fahrenheit()    # now I don't have to read the body
```

The pattern that gets you there is **`input_to_output()`**. Not every function fits it, but reaching for it is a fast way to find out whether you actually know what your function does. If you can't fill in both halves, that is worth knowing before you write the body.

The same care goes inside. Name the arguments and the intermediate values to match:

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

Read that and there is nothing left to work out. The argument says what goes in, the intermediate says what comes out, and the name says what happened in between.

4.2.2. Arguments, and where their values come from

The arguments are the part that trips people up, because `x` and `digits` do not have values when you write the definition. They are names waiting for something. The values arrive when you call the function, and where they arrive is decided by the order you write them in:

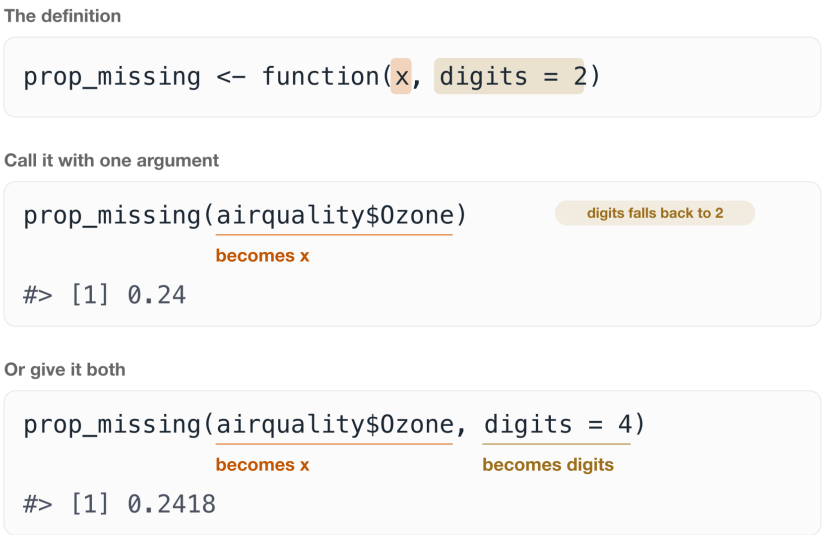


Figure 4.3.: Argument values arrive at the call, not at the definition.

The same function, two different answers, and nothing about the function changed. That is the whole job of an argument.

You can hand the values over in two ways. **By position**, in the order the definition lists them, or **by name**, using `name = value`.

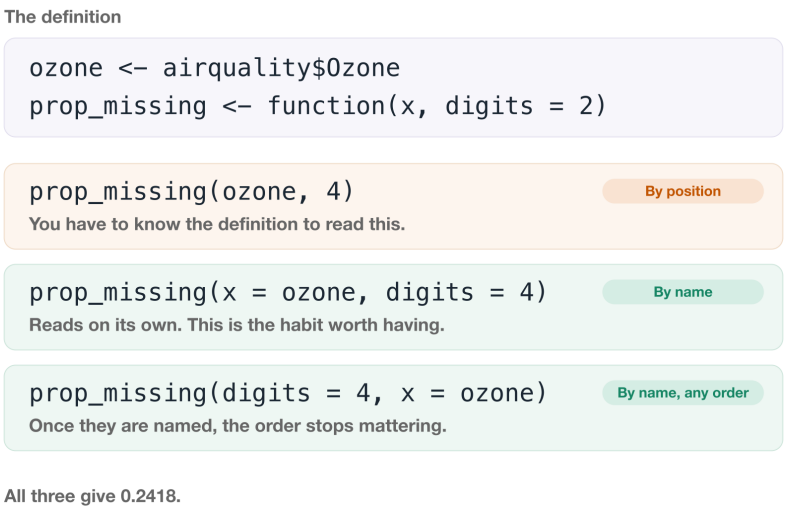


Figure 4.4.: Naming them costs a few keystrokes and buys you a line that reads on its own.

All three give the same answer. The difference is what you have to already know in order to read the line.

Name your arguments where you can. `prop_missing(ozone, 4)` requires you to remember what the second slot is for. `prop_missing(x = ozone, digits = 4)` tells you. The first argument is usually obvious enough to leave positional, and everything after it is worth naming.

> **Your Turn**
>
> Here is a function. Don't run it, and don't worry about what it computes.
>
> ```
> rate_per_1000 <- function(count, population, digits = 1)
>  ↪  {
>   rate <- (count / population) * 1000
>   round(rate, digits)
> }
> ```
>
> Name the parts:
>
> 1. What is the **name** of this function?
> 2. What are its **arguments**?
> 3. Which argument has a **default**, and what is it?
> 4. What is the **body**?
> 5. What does it **return**?
> 6. Now call it two ways: once letting `digits` take its default, and once setting it yourself.
>
> This one is really about the vocabulary. Once you can say "the third argument has a default", you can look it up and you can ask about it.

Two more parts, once the above is comfortable

These are worth knowing about eventually. Skip them for now if you are still getting the hang of the parts above.

Side effects. Anything a function does **other** than return a value:

- Printing to the console (`summary(lm_fit)`)
- Writing a file (e.g., a CSV, .rds, .png)
- Drawing a plot (e.g., `ggplot( ... )` or `plot( ... )`)
- Setting an option.

Side effects are not bad. Plenty of useful functions have them. But the issue is with reasoning - if a function returns something useful (a value) **and** quietly writes a file, it is harder to reason about than one that does a single one of those things.

Functions are values. An argument can be a function. Here is the situation where you would want that.

Say you want to know how many values are missing in each column of your data. With the function we already have, you could write:

```
n_missing(airquality$Ozone)
n_missing(airquality$Solar.R)
n_missing(airquality$Wind)
n_missing(airquality$Temp)
n_missing(airquality$Month)
n_missing(airquality$Day)
```

That is six lines to say one thing. You have to know every column name, you have to type each one correctly, and when the data gains a column next month you have to remember to come back and add a line.

It is the copy-paste problem from the start of this chapter, wearing a different hat.

What you want to say is "do this to every column". To say that, you need to hand one function to another function:

```
library(purrr)

map_int(airquality, n_missing)
```

```
  Ozone Solar.R    Wind    Temp   Month     Day
     37       7       0       0       0       0
```

Look closely at how `n_missing` is written there. **No brackets.** We are not calling it, we are handing it over, the same way we would hand over a number or a data frame. That is what "functions are values" means.

And you can accept a function in your own functions too:

```r
n_complete <- function(x) {
  sum(!is.na(x))
}

summarise_columns <- function(data, f) {
  map_dbl(data, f)
}

summarise_columns(airquality, n_complete)
```

```
  Ozone Solar.R    Wind    Temp   Month     Day
    116     146     153     153     153     153
```

A quick word on `map()`

`purrr` is the tidyverse package for applying a function to each element of something. The suffix tells you what comes back: `map()` gives a **list**, `map_int()` an **integer vector**, `map_dbl()` a **double vector**, and so on.

The typed ones error if the function cannot deliver that shape, which is the useful half. You say what you are expecting, and you find out straight away rather than three lines later.

Base R has its own versions, called `lapply()`, `vapply()` and `sapply()`.

Handing over a function you will only use once, without naming it, is worth knowing about too. That is in Anonymous functions B.

4.3. Other things worth turning into a function

Here are a few more ideas that you might turn into a function.

A calculation you keep redoing. Body mass index, a rate per thousand, a unit conversion.

```r
bmi <- function(weight_kg, height_m) {
  weight_kg / height_m^2
}
```

The function helps you know the units are right. The argument names do it for you.

```r
bmi(weight_kg = 70, height_m = 1.75)
```

```
[1] 22.85714
```

Unit conversions are the same idea:

`penguins` records body mass in grams, we can state `input_to_output()`:

```
grams_to_kg <- function(grams) {
  grams / 1000
}
library(dplyr)
penguins |>
  mutate(body_mass_kg = grams_to_kg(body_mass),
         .after = body_mass) |>
  head()
```

```
  species    island bill_len bill_dep flipper_len body_mass body_mass_kg    sex
1  Adelie Torgersen     39.1     18.7         181      3750         3.75   male
2  Adelie Torgersen     39.5     17.4         186      3800         3.80 female
3  Adelie Torgersen     40.3     18.0         195      3250         3.25 female
4  Adelie Torgersen       NA       NA          NA        NA           NA   <NA>
5  Adelie Torgersen     36.7     19.3         193      3450         3.45 female
6  Adelie Torgersen     39.3     20.6         190      3650         3.65   male
  year
1 2007
2 2007
3 2007
4 2007
5 2007
6 2007
```

- Also note that nice `.after` argument of `mutate()`

A check you want to run on every dataset that comes in. We built one of these a moment ago: `n_missing()`, in Anatomy of a function.

It's OK if the function is three lines long and doing something boring, or something repetitive. If they help express an idea more clearly, it is worth it.

> Small functions add up
>
> Here is one more of the same kind, to go with the two we already have:
>
> ```
> n_complete <- function(x) {
>   sum(!is.na(x))
> }
>
> n_complete(airquality$Ozone)
> ```
>
> ```
> [1] 116
> ```
>
> A couple of things to notice across these.

- `prop_missing()` **calls** `n_missing()` rather than writing `sum(is.na(x))` out a second time. Fix a bug in one, and you have fixed both.
- They are **named for the question they answer**, not for how they answer it. `prop_missing()`, not `round_na_ratio()`.

We come back to `prop_missing()` soon.

Your Turn

Look at your most recent analysis script.

1. Find a chunk of code you have written more than once, with something small changed each time
2. Write down, in one sentence, what that chunk is **for**.
3. That sentence is the name of the function - the idea. Don't write it yet, just note it down.

4.4. How to use a debugger

R is interactive. You run some code, you see the output immediately, and when something goes wrong you can poke around and look at it. R was designed as an interactive language - and this is distinct and different to scripted languages like C, C++, and Rust. The interactivity it special, and makes doing data analysis interactive. It is good.

Functions break that interactivity a little bit. A function is a box - you put code in, and return an output. By default you can't see inside a function while it runs.

For example, give `prop_missing()` an intermediate value on its way to the answer:

```
prop_missing <- function(x, digits = 2) {
  prop <- n_missing(x) / length(x)
  round(prop, digits)
}

prop_missing(airquality$Ozone)
```

```
[1] 0.24
```

We cannot look at `prop`:

```
prop
```

```
Error:
! object 'prop' not found
```

We know it existed! The function could not have given you an answer without it. But it lived inside the box, for a moment, and there is no way to reach it from out here.

We are used to interactively checking through code, but you can't print it, you can't `View()` it, you can't check its class.

To look inside, you need a special thing called a "debugger".

A confession

If you have ever copied a function into a new script, commented out function definition, then hard-coded the arguments at the top:

```
# prop_missing <- function(x, digits = 2) {
n_missing <- function(x) sum(is.na(x))

x <- airquality$Ozone
digits <- 2
prop <-  n_missing(x) / length(x)

round(prop, digits)
```

```
[1] 0.24
```

```
# }
# prop_missing(airquality$Ozone)
```

And run it line by line - this section is for you.
I say that as someone who did that for years.

4.4.1. A good case for a debugger

Small functions do not need one. `mean_center()` is two lines, and you can read it.

It gets harder the moment functions call other functions. Here is `prop_missing()` from earlier in the chapter:

```
n_missing <- function(x) {
  sum(is.na(x))
}

prop_missing <- function(x, digits = 2) {
  prop <- n_missing(x) / length(x)
  round(prop, digits)
}
```

When `prop_missing()` gives you a number you don't believe, the problem could be in either function, and `x` inside `n_missing()` is not an

object you can look at. It exists for a moment, in the middle of a call you did not make directly, and then it is gone.

4.4.2. Debugging with `browser()`

Let's give ourselves something to find. Here is `prop_missing()` again:

```
n_missing <- function(x) {
  sum(is.na(x))
}

prop_missing <- function(x, digits = 2) {
  prop <- n_missing(x) / length(x)
  round(prop, digits)
}

prop_missing(airquality$Ozone)
```

```
[1] 0.24
```

```
prop_missing(airquality)
```

```
[1] 7.33
```

It runs. Nothing errors, for `airquality$Ozone` it looks right, but then for `airquality` you get a proportion of 7.33. That's stange. And wrong.

Let's go and look.

`browser()` is like a stop sign that you put inside a function.

You write it on the line you want to stop at:

```
prop_missing <- function(x, digits = 2) {
  prop <- n_missing(x) / length(x)
  browser()
  round(prop, digits)
}
```

Now call the function the way you normally would:

```
prop_missing(airquality)
#> Called from: prop_missing(airquality)
#> Browse[1]>
```

That `Browse[1]>` is an R console, and it is standing **inside** the function. Every argument is now a real object you can look at, so look at the two numbers going into the division:

```
Browse[1]> length(x)
#> [1] 6

Browse[1]> n_missing(x)
#> [1] 44
```

There it is! `length(x)` gives us 6. Because length(data.frame()) returns the number of columns, not the number of elements.

We we divided 44 by 6 instead of the number of elements - rows x cols. A proportion has to divide by everything.

That didn't take too much work! Just a few questions.

The trick is a little bit more complex - and I think I'd like you to have a go at solving this in the next exercise.

A few commands are worth knowing at that prompt of `browser()`:

type this	and it does
`n`	run the next line
`c`	continue until the function ends
`Q`	quit, and go back to your console
`help`	print the list of these commands
anything else	evaluated as ordinary R, in the function's environment

That last row is the one that matters. Anything you would normally type at the console works here, which means you already know how to use a debugger. You just have to get inside one.

The catch with `browser()` is that it is committed code. It stops your script, and it will stop it for whoever runs it next, so you have to remember to take it out again.

> **i** We do this one live
>
> Reading about a debugger is close to useless. This is a demonstration in the session - I will put us inside a function, and we will poke at it together.
> If you are reading this on your own afterwards, run the Your Turn below rather than just reading it. The whole skill is in the doing.

> **💡** Without editing the function: debugonce(), debug() and undebug()
>
> Editing a function to add `browser()`, saving, sourcing, and then remembering to take it out again is a lot of steps. There is a way to do the same thing from the console.
> **`debugonce()`** says: next time this function runs, stop at the first line. Once only, so there is nothing to undo.

```
debugonce(prop_missing)

prop_missing(airquality)
#> debugging in: prop_missing(airquality)
#> Browse[1]>
```

Same prompt, same commands, no edit to the function and nothing left behind.
`debug()` and `undebug()` are the version that stays on. `debug()` marks the function, and it stays marked until you call `undebug()`.

```
debug(prop_missing)
undebug(prop_missing)
```

That is occasionally what you want, when you are stepping through the same function over and over. Mostly it is not. I've forgotten to call `undebug()` and felt a genuine madness descend as every subsequent call dropped me into the debugger.
So of the two, reach for `debugonce()`.

Your Turn

I want you to write n_missing(), n_complete(), and prop_complete(), so that they all work on a column, or a data.frame. Here's some starting code:

```
n_missing <- function(x) {
  sum(is.na(x))
}

n_complete <- function(x) {
  sum(!is.na(x))
}

prop_missing <- function(x, digits = 2) {
  prop <- n_missing(x) / length(x)
  round(prop, digits)
}
```

1. Start by using `browser()` inside prop_missing - play around inside it.
2. Now run `debugonce(prop_complete)`, then call it again. You will land inside the function.
3. Ask it questions - how can you get the number of elements? Are there existing functions inside of R you can use to get the total elements?
4. Type `Q` to quit, fix the function, and run it again. You should get `0.76`.
5. Do it once more, and this time step through with `n` rather than quitting.

Then do the same on a function of your own that you don't entirely trust.
The habit worth building: when a function surprises you, your first move is `debugonce()`, not copying the body into a new script.

Read more

Parts of this chapter are adapted from my blog post, How to get good with R.
The function design walkthrough, and the argument that DRY treats the symptom rather than the cause, come from my talk **Practical functions: practically magic** - slides and source.
Its take-home messages are worth repeating here. Good functions:

1. Manage **complexity**
2. Explain and express **ideas**
3. Can be **individually reasoned** with
4. Take **iteration**

4.5. Where this script goes next

We chunked a hundred line script back in Writing readable code, and left it with six comments in it, one per idea.

Once each of those ideas is a function with a name, the whole script is this:

```
source("packages.R")
lapply(list.files("./R", full.names = TRUE), source)

educated_population_2014 <- read_educated_population(year =
 ↪  "2014")

ggplot(educated_population_2014, aes(x = prop_studying, y =
 ↪  state_territory)) +
  geom_col() +
  facet_wrap(~age_group)
```

Ninety lines to six. The other eighty-four did not disappear. They moved into `R/`, one file per function, exactly the layout from Project organisation 1.

Getting there is a whole chapter of its own, and it is Designing functions A. That is where we take this script apart a chunk at a time, and where the two identical steps become one function called twice.

Summary

1. Functions are managing complexity so you can reason with it.

2. Name a function for what it does, not how it does it. `celcius_to_fahrenheit()`, not `myfun()`.
3. The parts that change between uses are the arguments. The parts that don't change are often defaults.
4. Small functions build on each other. `prop_missing()` calls `n_missing()` rather than repeating it.
5. When a function surprises you, look inside it with `browser()` or `debugonce()`.

If you remember nothing else from this chapter, remember this: **Functions are managing complexity so you can reason with it.**

5. Writing a reprex / getting unstuck

Everyone gets stuck. While avoiding being stuck in the first place is ideal - one of the best skills to learn is to get unstuck quickly, and to be able to ask for help in a way that actually gets you an answer.

This chapter is about the **reprex**, which is short for **repr**oducible **ex**ample. If you can demonstrate how you are stuck, you often solve the problem in the process of creating that demonstration. Then, if you can't, you have made it easier for someone else to see how your are stuck. That makes it easier for them to help you.

Overview

Duration 45 minutes
Questions

- What is a reprex, and why would I want one?
- Does something have to be broken before a reprex is useful?
- How do I cut a broken script down to something someone else can run, and why does that so often solve it?
- How do I find a bug that never errors?
- Which errors am I going to hit over and over, and what do they mean?

5.1. What is a reprex?

A reprex is code, plus the output that code produced, in a form somebody else can run.

Say we want to know why this is happening - how can this be NA?

```
mean(airquality$Ozone)
```

```
[1] NA
```

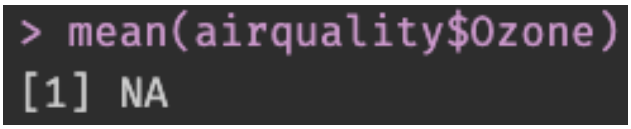
You can read what I ran, you can see what I got, and you can paste it into your own R session and get the same thing.

There are other ways to share this:

- You could send somebody just the code, `mean(airquality$Ozone)`

- But then they have to run it to see what you saw, and guess which packages you load.

You could send a screenshot of your console instead:



Now they can see the output, but they can't run it, and they can't copy a line out of it to try something.

You could send both, by hand, every time - the code, AND the screenshot And now it's on you to make sure the code and the output actually match.

It gets worse with a plot. If the interesting thing is a chart, the code on its own doesn't show it, and the picture on its own can't be ru

So what you want is both, together, and correct. It is fiddly to do by hand, and is why the `reprex()` package exists.

5.2. Using {reprex}

You can do all of that by hand. You make sure the environment is clean, the right packages are loaded, the code is formatted, and any images come out at a sensible size.

If your problem is tightly defined that might take a couple of minutes.

Other times it takes far longer. Say, when you've got a variable sitting in your environment that you've forgotten how you created, and didn't realise was load-bearing. *OH GOD HOW DID I EVEN DO THIS.* Ahem.

So, after some sensible swearing, you might ask:

> Surely there is an easy, reproducible way to make reproducible examples?

There is: the `{reprex}` package, by Jenny Bryan.

The workflow is four steps:

1. Copy your code to the clipboard.
2. Run `reprex()` and wait a moment for it to execute.
3. `{reprex}` shows you an HTML preview of the code *and its output.*
4. A markdown version is placed on your clipboard, ready to paste.

```
reprex::reprex()
```

By default the output is formatted for GitHub, which is also what you want for Slack, email, or pasting into a message to a colleague.

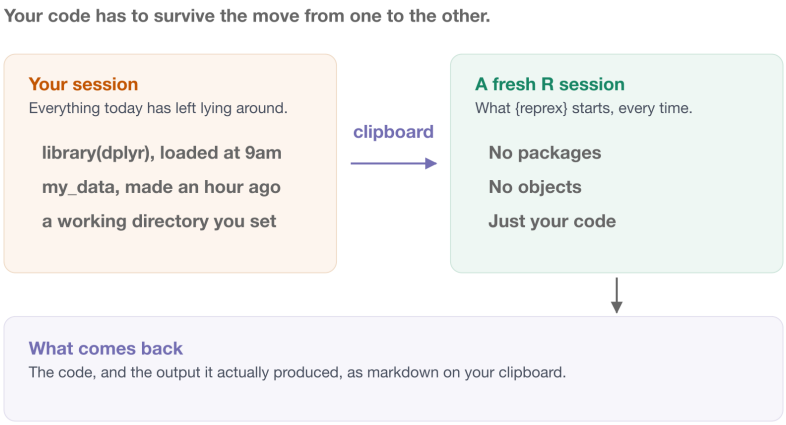


Figure 5.1.: None of the things holding your code up exist on the other side.

Why this actually works

`{reprex}` takes your copied code and runs `rmarkdown::render()` on it, **in a fresh R session**.
That's the whole trick. It means `{reprex}` will **fail hard and fail early** if your example isn't reproducible.
Try both of these. They work perfectly in your session, and neither survives a fresh one.
Forgot a `library()` call. Copy this and `reprex()` it:

```
penguins |> count(species)
#> Error in count(penguins, species) : could not find
 ↪  function "count"
```

Obvious once you see it. Not obvious at all when you have had `{dplyr}` loaded since nine o'clock this morning.
Relying on something you made an hour ago. Make an object, then copy only the line that uses it:

```
my_data <- penguins[penguins$species == "Gentoo", ]
```

```
library(dplyr)

my_data |> count(species)
#> Error: object 'my_data' not found
```

Your session knows what `my_data` is. Nobody else's does, and neither does the fresh one `{reprex}` starts.
So when `reprex()` gives you a working example, you *know* it works. Not because you were careful, but because it was run somewhere your local mess doesn't exist.

And it handles plots. If your example makes a chart, the chart comes

out the other side along with the code that made it, which is the bit that was hardest to do by hand.

Your Turn

Start as small as it goes.

1. Copy these two lines to your clipboard:

```
x <- c(1, 2, 3, NA)
mean(x)
```

2. Run `reprex::reprex()`.
3. Look at the preview. Then paste what landed on your clipboard into a fresh R script, and check it runs.

That's a reprex.
Now make it a keypress with a shortcut.
Copying to the clipboard and then typing `reprex::reprex()` is three separate actions.
So bind it to a key while you are here. There is a second function, `reprex::reprex_selection()`, which reprexes whatever is *selected* in your editor, with no clipboard step at all.
In RStudio. Tools → Modify Keyboard Shortcuts (or - Command Pallete, and type "Modify keyboard", and click "modiy keyboard shortcuts") then type `reprex` in the filter box. You get two: **Render reprex**, which opens a little window of options, and **Reprex selection**, which just goes. Click the shortcut column next to **Reprex selection**, press `Cmd / Ctrl + Shift + R`, and hit Apply.
That combination is already taken by Insert Section - RStudio noted this. Just overwrite it - or pick another combination for your reprexes.
In Positron. Open the command palette with `Cmd / Ctrl + Shift + P`, run **Open Keyboard Shortcuts (JSON)**, and add this:

```
{
  "key": "Cmd+Shift+R",
  "command": "workbench.action.executeCode.console",
  "when": "editorTextFocus",
  "args": {
    "langId": "r",
    "code": "reprex::reprex_selection()",
    "focus": true
  }
}
```

Use `"Ctrl+Shift+R"` on Windows or Linux. This one is lifted straight from Positron's own documentation, where reprex is the worked example. If you'd rather not bind anything, the palette

> also has **R: Run RStudio Addin**, which lists reprex alongside every other addin you have installed.
> **Then use it.** Select those same two lines in your editor and press your new shortcut. Same reprex, one keypress, and nothing to copy first.
> Everything else in this chapter is that, on harder code.

5.3. A reprex isn't only for when things break

Almost everything written about reprexes is about asking for help with a bug. That is how I first learned about them, and it left me with the idea that a reprex is a thing you make when something has gone wrong.

It isn't. A reprex is just a small, runnable piece of code with its output attached, and there are plenty of reasons to want one of those.

"Here's how I'd do it." Someone asks how to count the rows in each group. You could describe `count()`. It's faster to show them:

```
library(dplyr)

starwars ▷
  count(species, sort = TRUE) ▷
  head(3)
#> # A tibble: 3 × 2
#>   species     n
#>   <chr>   <int>
#> 1 Human      35
#> 2 Droid       6
#> 3 <NA>        4
```

"Is there a nicer way to write this?" Nothing is broken. You just suspect there is a neater version, and a reprex is how you ask without making somebody clone your project.

"Look at this, it's great." A function I have just found, a bit of syntax I did not know about. Code with its output is how you show somebody something works. Put it on a github gist.

Notes to yourself. I keep these on a github gist. Six months later, the code and its output together tell me what I worked out, and I don't have to run anything to remember.

The nice thing about starting here is that there is no pressure. Nothing is wrong, so there is nothing to diagnose. You are only practising the workflow.

> **Your Turn**
>
> 1. Find something in your own recent code that works, and that you would be happy to show somebody. Three or four lines.
> 2. `reprex()` it.
> 3. Send it to the person next to you, and have them paste it into their R session. Does it run?
>
> If it doesn't run on their machine, you have just learned something about your code that you did not know a minute ago.

5.4. Cutting a problem down

The harder skill, which you might have come across, is to cut down reprexes into the smallest unit.

When you hit an error you cannot solve by tinkering, it is worth spending some time building a small example of the code that breaks.

This takes practice. But the thing I really like is:

In the process of reducing the problem to its core, you often solve it.

It is sort of the same experience as describing a problem out loud to a colleague, and then by the time you've explained it, you manage to solve it. This is a whole concept called "rubber duck debugging" I have a rubby ducky on my desk for this purpose. You take your rubber ducky and explain the problem to then.

So this isn't only a way to ask for help. It's a debugging technique that happens to produce something shareable.

Let's tear one down. Say we are summarising the `diamonds` data:

```
library(tidyverse)
```

```
-- Attaching core tidyverse packages ----------------------- tidyverse 2.
v dplyr     1.2.1     v readr     2.2.0
v forcats   1.0.1     v stringr   1.6.0
v ggplot2   4.0.3     v tibble    3.3.1
v lubridate 1.9.5     v tidyr     1.3.2
v purrr     1.2.2
-- Conflicts ------------------------------------------ tidyverse_conflict
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
```

```
diamonds ▷
  mutate(price_per_carat = price / carat) ▷
  group_by(cut) ▷
  summarise(
    price_mean = mean(price_per_carat),
    price_sd = sd(price_per_carat),
    mean_color = mean(color)
  )
```

```
Warning: There were 5 warnings in `summarise()`.
The first warning was:
i In argument: `mean_color = mean(color)`.
i In group 1: `cut = Fair`.
Caused by warning in `mean.default()`:
! argument is not numeric or logical: returning NA
i Run `dplyr::last_dplyr_warnings()` to see the 4 remaining warnings.
```

```
# A tibble: 5 x 4
  cut       price_mean price_sd mean_color
  <ord>          <dbl>    <dbl>      <dbl>
1 Fair           3767.    1540.         NA
2 Good           3860.    1830.         NA
3 Very Good      4014.    2037.         NA
4 Premium        4223.    2035.         NA
5 Ideal          3920.    2043.         NA
```

The warning gives us a clue: it is something about `mean_color`. So try just that:

```
diamonds ▷
  mutate(mean_color = mean(color))
```

```
Warning: There was 1 warning in `mutate()`.
i In argument: `mean_color = mean(color)`.
Caused by warning in `mean.default()`:
! argument is not numeric or logical: returning NA
```

```
# A tibble: 53,940 x 11
   carat cut       color clarity depth table price     x     y     z mean_color
   <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>      <dbl>
 1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43         NA
 2  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31         NA
 3  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31         NA
 4  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63         NA
 5  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2.75         NA
 6  0.24 Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48         NA
 7  0.24 Very Good I     VVS1     62.3    57   336  3.95  3.98  2.47         NA
 8  0.26 Very Good H     SI1      61.9    55   337  4.07  4.11  2.53         NA
```

```
 9  0.22 Fair      E     VS2      65.1    61   337  3.87  3.78  2.49
10  0.23 Very Good H     VS1      59.4    61   338  4     4.05  2.39
# i 53,930 more rows
```

Same warning, and we've dropped `group_by()` and `summarise()` entirely. Strip it further:

```
mean(diamonds$color)
```

```
Warning in mean.default(diamonds$color): argument is not numeric or logica
returning NA
```

```
[1] NA
```

Still the same. We are now down to one line. So what is in `color`?

```
head(diamonds$color)
```

```
[1] E E E I J J
Levels: D < E < F < G < H < I < J
```

Ah. Does it make sense to take the mean of some letters? Of course not.

Notice what happened. We never needed to ask anyone.

Each step made the example smaller, and the smaller it got, the more obvious the problem became.

That's the technique.

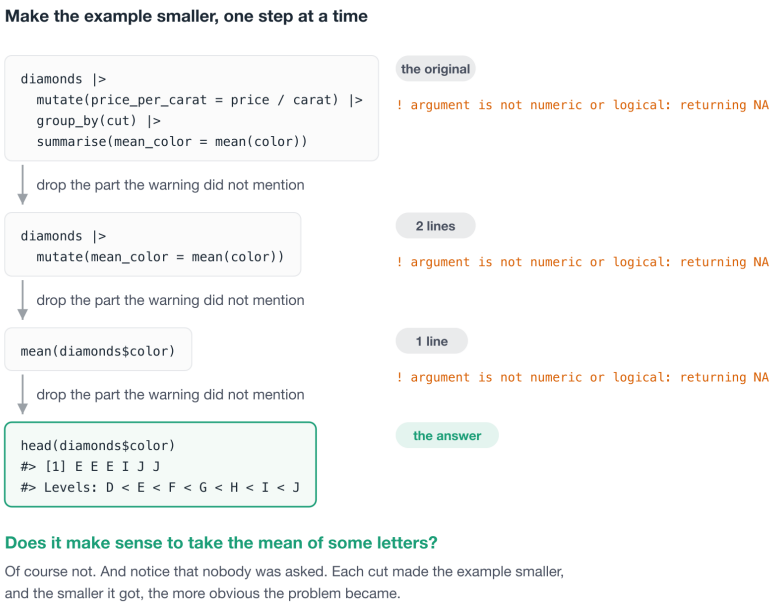


Figure 5.2.: Cutting the example down until the problem is obvious.

Your Turn

Here are two more that go wrong. Cut each one down the way we just did, one line at a time, until you can say what the problem is.
One

```
library(tidyverse)

starwars ▷
  filter(!is.na(species)) ▷
  group_by(species) ▷
  summarise(mean_hair = mean(hair_color))
```

Two

```
library(tidyverse)

msleep ▷
  group_by(vore) ▷
  summarise(mean_rem = mean(sleep_rem))
```

For each one:

1. What is the smallest piece of code that still shows the problem?
2. Look at the column. What is actually in it?
3. Say the problem in one sentence.

The second one is the more interesting of the two, and it is worth noticing why: R never complains at all.
Then, once you have them cut down:

4. `reprex()` your reduced version of each. Does the preview show the warning?
5. Now break one on purpose. Delete the `library(tidyverse)` line from your clipboard and `reprex()` it again. What happens?

Step 5 is worth doing. `{reprex}` runs your code in a fresh session, so it finds the thing you forgot before your colleague does.

5.5. A bug with no error

The `diamonds` example had a warning to follow. Most real bugs do not.

So let's do a harder one, on the script we have been working on all course.

5.5.1. The setup

At the end of Writing functions 4 we had `read_educated_population()`, which read a tidy spreadsheet.

Here is where that spreadsheet actually came from. Somebody sent over a folder of CSVs, one per year.

(This is run inside the ozed project)

```
library(tidyverse)
library(here)

education_2014 <-
 ↪  read_csv(here("data/raw/raw_education_2014.csv"))
education_2015 <-
 ↪  read_csv(here("data/raw/raw_education_2015.csv"))
education_2016 <-
 ↪  read_csv(here("data/raw/raw_education_2016.csv"))

education <- bind_rows(education_2014, education_2015,
 ↪  education_2016)
```

(Note, there are a couple of ways we can run this code in one line)

That runs without complaint, and gives you 216 rows across the three years.

Now the summary we came for. What proportion of people are studying, by age group?

```
education ▷
  group_by(age_group) ▷
  summarise(prop_mean = mean(prop_studying))
#> # A tibble: 27 x 2
#>    age_group prop_mean
#>    <chr>         <dbl>
#>  1 15---19       0.813
#>  2 15--19      -15.8
#>  3 15-19         0.811
#>  4 20---24     -19.5
#>  5 20--24      -10.6
#>  6 20-24         0.406
#>  7 25---29      -9.73
#>  8 25--29      -14.0
#>  9 25-29         0.212
#> 10 30---34     -14.0
#> # i 17 more rows
```

There is a lot wrong here - R said nothing at all.

There are **27 age groups**. There should be nine. And a *proportion* has come out as -15.8. No good.

5.5.2. Reducing it

Two things are wrong here, so find them one at a time, and start with the smaller of the two.

How many age groups are there really?

```
n_distinct(education$age_group)
#> [1] 27

sort(unique(education$age_group))
#>  [1] "15---19" "15--19"  "15-19"   "20---24" "20--24"
 ↪  "20-24"   "25---29"
#>  [8] "25--29"  "25-29"   "30---34" "30--34"  "30-34"
 ↪  "35---39" "35--39"
#> [15] "35-39"   "40---44" "40--44"  "40-44"   "45---49"
 ↪  "45--49"  "45-49"
#> [22] "50---54" "50--54"  "50-54"   "55---74" "55--74"
 ↪  "55-74"
```

There it is. `15-19` and `15--19` and `15---19` are the same age group typed three different ways, and `group_by()` has no way of knowing that. Nine real groups, spread across 27 spellings.

Now the other one. Why is a proportion negative?

```
summary(education$prop_studying)
#>      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
#> -99.00000   0.04563   0.09626  -9.43312   0.20003   0.90308
```

A minimum of -99, in a column that can only sit between 0 and 1.

`-99` is a missing value code. Somebody, at some point, decided that missing should be written as -99 rather than left blank. `read_csv()` has no way of knowing that, so it read it as the number minus ninety-nine.

5.5.3. The reprex

We are now down to two lines of it. Here is the whole thing, self-contained, with no data files and no packages beyond one:

```
library(dplyr)

age_group <- c("15-19", "15--19", "15---19")
prop_studying <- c(0.81, -99, 0.84)

tibble(age_group, prop_studying) ▷
  group_by(age_group) ▷
  summarise(prop_mean = mean(prop_studying))
#> # A tibble: 3 × 2
#>   age_group prop_mean
```

```
#>   <chr>          <dbl>
#> 1 15---19         0.84
#> 2 15--19        -99
#> 3 15-19           0.81
```

Three rows of made up data, and both bugs are visible at once: one age group has become three, and a proportion is -99.

Three files and 216 rows, down to four lines somebody can run on their own machine in a second. That is the reduction, done on something real.

5.5.4. Fixing it

Both fixes are one line, now that you can see what you are fixing.

The separators collapse with a regular expression that matches a *run* of anything that is not a digit:

```
education ▷
  mutate(age_group = str_replace_all(age_group, "[^0-9]+",
   ↪  "_"))
```

And the missing code becomes an actual missing value. `{naniar}` does this, and knowing what I know about that package's author, I would say it is a reasonable choice:

```
library(naniar)

education ▷
  replace_with_na(replace = list(prop_studying = -99))
```

Put the two together, and the summary finally says something sensible:

```
education ▷
  mutate(age_group = str_replace_all(age_group, "[^0-9]+",
   ↪  "_")) ▷
  replace_with_na(replace = list(prop_studying = -99)) ▷
  group_by(age_group) ▷
  summarise(prop_mean = mean(prop_studying, na.rm = TRUE))
#> # A tibble: 9 x 2
#>   age_group prop_mean
#>   <chr>          <dbl>
#> 1 15_19          0.810
#> 2 20_24          0.407
#> 3 25_29          0.198
#> 4 30_34          0.131
#> 5 35_39          0.113
```

```
#> 6 40_44         0.0898
#> 7 45_49         0.0647
#> 8 50_54         0.0530
#> 9 55_74         0.0189
```

Nine groups, and every proportion sitting between 0 and 1.

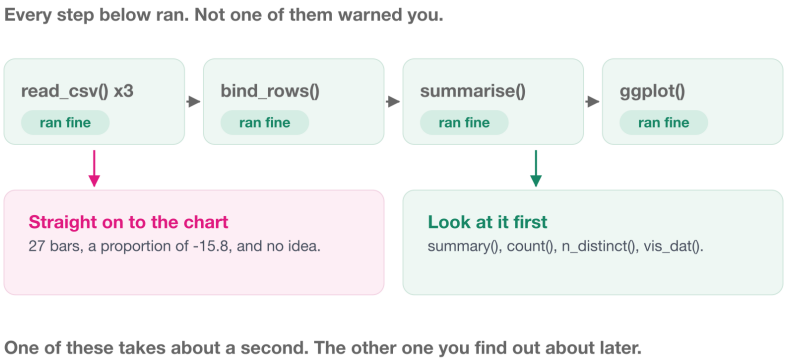


Figure 5.3.: Every box is green. That is what makes this one frightening.

! This is the failure mode to be afraid of

Nothing errored. Nothing warned. Every step ran.
If you had gone straight from reading the files to a plot, you would have got a chart with 27 bars on it and probably assumed the data was just messier than you remembered. If you had gone to a model, you would have got coefficients.
The reason this got caught is that somebody looked at the intermediate object before using it. I would take that habit over any tool in this course.
`summary()`, `count()`, `n_distinct()`, and `visdat::vis_dat()` all take about a second, and any one of them would have found this.

i Your Turn: the other three years

You now have a fix that works on 2014 to 2016. Here is the rest of the folder.
The files live in the ozed repo, under `data/raw/`. There are six years in there, not three: 2014 to 2019.

1. Read in all six years, and run your fix over the lot.
2. Run `summary()` and `n_distinct()` again. Is it clean?
3. It is not. Work out what changed, and when. Looking at one year at a time is much faster than staring at all 480 rows.
4. Rewrite the fix so it works on all six years.
5. Where would you put that fix? In the script, or inside the function that reads the file? What is the argument for each?

Question 3 is the real exercise here. Whoever sent these files did

not send you one problem.

The answer, once you have had a go

year	age separators	missing written as
2014 to 2016	`-`, `--`, `---`	`-99`
2017	`;`, `:`, `::`	`-1`
2018	`.`, `..`	`9999`
2019	`|`, `||`	`Inf`

The convention changed between years. A fix written by looking at 2014 will quietly not work on 2018, and nothing will tell you.

The separator fix survives it, because `[^0-9]+` never cared which character it was. The missing value fix does not, because `-99` is a specific number.

5.6. Some known error patterns

Before you build a reprex, it is worth checking whether you have hit one of the handful of errors that everyone hits.

R's error messages have a reputation for being unhelpful. I think that reputation is only half deserved. A lot of them are perfectly clear once you have seen them once, and the trouble is that nobody sits you down and shows you the once.

So here are the ones I see most, in roughly the order you will meet them.

5.6.1. `could not find function`

```
starwars |> select(name, height)
#> Error in select(starwars, name, height) : could not find
 ↪  function "select"
```

This is the most common error in R, and it almost always means the same thing:

You have not run `library()` yet.

The function exists. It is sitting on your computer right now. R just has not been told to go and look in that package.

```
library(dplyr)
starwars |> select(name, height)
```

It usually happens in one of three ways:

1. You restarted R, and forgot the `library()` calls are gone with the session.
2. You are running the script from the middle, and the `library()` calls are at the top.
3. You sent the code to someone else, and they don't have the same packages attached that you do.

That third one is why `{reprex}` runs your code in a fresh session. It catches this before your colleague does.

> **A habit that removes this one entirely**
>
> Restart R and run your script from the top, often. `Cmd / Ctrl + Shift + F10` restarts R in both RStudio and Positron.
> If it runs from a clean session, it will run on someone else's machine. If it only runs from *your* session, you have a dependency on something you cannot see, and you would much rather find that now than in a fortnight.

5.6.2. `there is no package called`

```
library(dplyr)
#> Error in library(dplyr) : there is no package called
 ↪  'dplyr'
```

Different error, and it is easy to confuse with the last one.

That one meant "installed, but not attached". This one means **not installed**.

```
install.packages("dplyr")
```

The way to tell them apart: `could not find function` complains about a *function*, and `there is no package called` complains about a *package*.

5.6.3. `object 'x' not found`

```
mean(penguin_data$body_mass_g)
#> Error: object 'penguin_data' not found
```

R does not know about a thing by that name. Almost always one of:

- A typo. `penguin_data` and `penguins_data` are different names.
- You have not run the line that creates it yet.
- You restarted R, and it was only ever in your environment, never in the script.

The last one is worth dwelling on, because it is the reason for the blank slate settings back in Project organisation 1. If your script only works because of an object you made three days ago and never wrote down, your script does not work.

5.6.4. `object of type 'closure' is not subsettable`

```
mean$x
#> Error in mean$x : object of type 'closure' is not
 ↪  subsettable
```

This one is famous for being baffling, to the point that Jenny Bryan named an entire keynote after it.

Translated: **a closure is a function, and you tried to pull a column out of one.**

Nearly always you have a name collision with a function that already exists. Someone writes `df ← data.frame( ... )`, then restarts R, then runs `df$x` before the line that makes `df`. There is a `df()` function in `{stats}`, so R finds *that*, and tells you you cannot subset it.

The fix is the fix for the error above: run the line that makes your object. The lesson is not to name your data after a function.

5.6.5. `$ operator is invalid for atomic vectors`

```
c(1, 2, 3)$a
#> Error in c(1, 2, 3)$a : $ operator is invalid for atomic
 ↪  vectors
```

You used `$` on something that has no columns. A plain vector has no names to reach for, so there is nothing for `$` to do.

Usually this means the thing you have is not the thing you thought you had. Check with `class()` before you check anything else.

5.6.6. `cannot open file ... : No such file or directory`

```
read.csv("data/penguins.csv")
#> Error in file(file, "rt") : cannot open file
 ↪  'data/penguins.csv': No such file or directory
#> In addition: Warning message: cannot open file
 ↪  'data/penguins.csv': No such file or directory
```

The file is not where you said it was, *relative to where R currently is*.

Two questions, in this order:

```
getwd()          # where does R think it is?
fs::dir_ls()     # what can it actually see from there?
```

Nine times in ten the answer is that your working directory is not the project root, which is the entire argument for projects and `here()` in Project organisation 1.

5.6.7. `unexpected symbol`, `unexpected ')'`, `unexpected end of input`

```
mean(c(1, 2)
#> Error in parse(text = ...) : <text>:2:0: unexpected end of
 ↪  input
#> 1: mean(c(1,2)
#>    ^
```

These are the ones where R could not even read your code, let alone run it. A bracket, a quote or a comma is missing.

The useful trick is that **the caret points at where R gave up, not at where you went wrong**. The mistake is usually a line or two *above* the mark.

Two things that make these mostly go away:

- Turn on rainbow parentheses, which we set up in Project organisation 1. A missing bracket becomes a colour you can see.
- Run `air format .`. A formatter cannot format code it cannot parse, so it will land you on the broken line straight away.

5.6.8. The one with no error at all

The worst pattern in this list produces no message.

Two attached packages both export a function called `lag()`, R hands you the one from whichever package you attached last, and you get a column of plausible looking numbers that are wrong.

That one gets a section of its own in When two packages collide, along with the one-line fix.

> **Your Turn**
>
> 1. Restart R, then run the second half of a script you are working on. Which of the errors above did you get first?
> 2. Reading the message alone, could you say which of the two "missing" errors it is: not installed, or not attached?

3. Fix it by adding to the top of the script rather than by typing into the console. What is the difference between those two fixes?

5.7. Practical tips on debugging

Your Turn

1. TODO: exercise for this section.

5.7.1. Keep solving vs cleaning up

Read more

Parts of this chapter are adapted from my blog posts How to get good with R and Magic reprex, the latter of which has gifs of the whole workflow in action.

6. Putting it all together

This is the part where you turn everything from this course loose on some real code.

We have some options! We can review:

1. Together, on some code I will share.
2. In pairs, on code you brought with you

Overview

Duration 60 minutes
Questions

- What do the three passes look like when somebody talks through them?
- Can I review code I have never seen before, in front of other people?
- How do I review someone else's code without it going badly?
- What do I do next, once the course is over?

6.1. The three passes

Everything in this session runs on the process built upon earlier in Writing readable code. Here is a summary:

1. The cheap pass:
    - `air format .` (or just Ctrl/Cmd + S to save!)
    - read it top to bottom marking `TODO` and `NOTE`
    - run it from a clean session
2. The thinking pass:
    - check the names
    - work out what each chunk is for,
    - ask what it needs
    - Ask whether there is a simpler way
3. The tidy up pass:
    - Put like with like
    - throw out the dead code
    - run the code linter, interpret and explore these

You might not finish - that is OK! Doing *some* of it is the whole idea.

6.2. Reviewing together

Open `06-bad-style/messy-analysis.R` from the course repository at https://github.com/njtierney/rbp-exercises/blob/main/06-bad-style/messy-analysis.R.

Somebody can share their screen, and walk us through it, out loud, while the rest of us watch.

This might feel like a lot! That's OK. Remember, nobody else read this file before either. And, often the most useful part of this review is hearing where someone slows down, backtracks, and changes their mind.

> **Your Turn: review it out loud**
>
> One person drives, sharing their screen. Everyone else watches, and can chip in.
> You should be able to answer this question:
>
> **What is this script for, in a sentence?**
>
> 1. **The cheap pass.**
>   - Run `air format .`.
>   - Then read it top to bottom, out loud, marking anything that makes you pause with `TODO` or `NOTE`.
>   - Fix nothing yet.
>   - Run it from a clean session.
> 2. **The thinking pass.** Now go back over the marks.
> 3. Read the names out - what do they tell you?
> 4. What is each chunk of code for?
> 5. Does everything it computes get used?
> 6. Is anything here being done twice?
> 7. **The tidy-up pass.**
>   - Where would you move things
>   - What would you delete?
>   - What does the linter tell you?

If you are reading this on your own rather than sitting in the session, do it anyway, and say it out loud to an empty room. The talking is what makes you notice you cannot explain something.

6.3. Round two: review each other's code

When you review someone else's code the questions are the same. What changes is that there is a person who can tell you things the code

cannot.

1. **Ask what they want first.**
    - *what kind of feedback are you looking for?* Someone who wants to know whether the statistics are right does not want thirty comments about variable names.
    - *what is this code meant to do?* Find the intention before you review the implementation, because half of what looks like a mistake turns out to be a decision you did not have the context for.
2. **Review the code, not the person.**
    - "This function is doing three things" is useful. "You always overcomplicate things" is not.
3. **Say what is good, specifically.**
    - This is not politeness. If you only ever name problems, people learn what to avoid and never what to aim for.
4. **Run it, don't just read it.**
    - Reading tells you whether code is comprehensible. Running it on something unfamiliar surfaces the assumptions the author did not know they were making.

This will be slow at first, and that is fine. Reading code you did not write is a skill, and it is slower than writing your own for quite a while. It gets faster. Nobody is quick at this on their first go.

> **Your Turn: swap code**
>
> Pair up, and use something of your own. A recent script, or something from other a year ago can be good, too.
>
> 1. Say what your code is meant to do, and what feedback you actually want.
> 2. Swap. Work the three passes on your partner's code.
> 3. Give three pieces of feedback. At least one should be something that is working well.
> 4. Swap back and talk about it.

> **When it breaks and you cannot see why**
>
> Cut it down. Remove everything not needed to make it fail, one piece at a time, until what's left is small enough to see through. That is a review technique, and it is also exactly how you make a **reprex**. We covered it in Writing a reprex / getting unstuck.

> **If we can see your code, we can trust it**
>
> Version control is not part of this course. It has its own.
> But the single biggest improvement to reviewability is putting

your code somewhere it can be seen, with a history. Review is much easier when the reviewer can see not just what the code is, but how it got that way.

If you want to see this done at full scale

Open software peer review is the most developed version of this practice, and it is all public. rOpenSci's reviewer guide sets out what a reviewer is asked to look at, and their review template is the checklist itself. Reviews happen in the open on GitHub and are signed, so you can read hundreds of real ones.
Two things there are worth stealing for a small analysis project. Reviews are explicitly **non-adversarial**, because the goal is better software rather than a verdict. And reviewers are asked about things a checklist cannot capture, such as whether the argument names read well and autocomplete sensibly.
That last one should sound familiar from the naming section.

6.4. Getting better, beyond the code

Everything in this course has been about the code itself. But a lot of getting good at R has nothing to do with typing R.

I've got a lot of thoughts on how to get better, and a lot of this is general advice. You don't have to follow all of these, but I think some of them will be useful. These are taken off of a blog post I wrote, "How to get good with R".

- **Learn how to type fast**. If you can type fast, you can operate closer to the speed of thought. See more on my blogpost, "Get Good with R: Typing Skills and Shortcuts", and the appendix, Working closer to the speed of thought.
- **Find a community of people who use R.** A local useR group, an online community, a Slack or Discord, a reading group at work. You will learn faster with people who are also learning.
- **Read other people's code.** Pick a package you use and read its source. You will find things you did not know were possible, and you will find things that look wrong to you - both are useful.
- **Practise reading the documentation.** Help files (`?function`) have a consistent structure, and once you can navigate one quickly you can navigate all of them. This is a skill, and it improves the most you use it. Read the examples.
- **Read books and learning material.** R for Data Science and Advanced R are both free online.
- **Offload ideas and tasks somewhere outside your head.** GitHub issues, a notebook, a task list - anything that means an

idea does not have to be carried around. We are good at having ideas, less good at holding on to them.

- **Ask yourself how relevant what you are doing right now is to the problem you set out to solve.** I get distracted often. Asking this question is how I redirect. It is remarkable how frequently the honest answer is "not very".
- **Write about your work publicly.** Knowing that someone might read it forced me to clean up my code and keep a tidy ship, and I have learnt a lot in the process of explaining things to others.
- **Have a strong desire to improve.** I cannot give you this one. But if you are reading this at the end of a course you chose to take, you probably already have it.

6.5. Not the end

There is more to all of this, and I do not have all the answers.

I am a decent R programmer. I am also still learning all the time, and I make mistakes a lot. I am not perfect, far from it. But I hope some of this helps you on your own journey of improving at R.

A. Designing functions

> "I want to get to a level where it works, and I can reason about it."
>
> – Joe Cheng

Chapter 4 was about what a function is and how to look inside one. This is the next question: given a piece of code, how do you decide what the function around it should be?

It is not part of the six hour course, and you do not need it to get value from everything else. It is here because it is the material I would teach next, and because the hundred line script we have been carrying since chapter 3 deserves to be finished.

Overview

Duration 50 minutes
Questions

- What complexity is this function managing?
- How do I turn a chunk of code into a function, one step at a time?
- What does it look like to take a whole script apart?
- When is a function worth writing, if the code was never repeated?

A.1. Good (simple) function design

Functions are **tools for managing complexity**. That is also called *abstraction*, or *abstracting away*.

So the question when writing one is always: **what complexity am I managing? What am I abstracting away?**

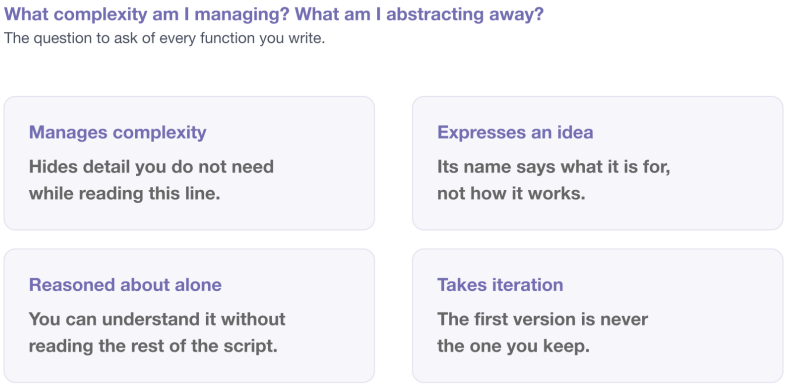


Figure A.1.: Four things that are true of a function worth keeping.

Here is a real piece of analysis code. It asks whether temperature differs on days when ozone is missing, compared with days when it is not:

```
is_different <- airquality$Temp
when_missing_index <- which(is.na(airquality$Ozone))
when_complete_index <- which(!is.na(airquality$Ozone))
is_different_miss <- is_different[when_missing_index]
is_different_complete <- is_different[when_complete_index]
result_ozone_temp <- t.test(is_different_miss,
 ↪  is_different_complete)
```

That works. But you can't use it on another pair of variables without editing it, and you can't tell at a glance what it's *for*.

Writing a function is writing. You don't get it right on the first pass, and you're not supposed to.

Here's the actual process.

Start with the bones. Make an empty function and paste the code into the body.

```
misstest <- function(Temp, Ozone) {
  # paste the text into the body
}
```

Ask what you are interested in. Which parts change? Here it is the variable we are testing, and the variable whose missingness we are splitting on.

Work out what to name things. Replace the hard-coded `airquality$Temp` with an argument. Comment out the old line rather than deleting it, so you can see what you are replacing:

```
misstest <- function(is_different, when_missing) {
  # is_different <- airquality$Temp
  when_missing_index <- which(is.na(when_missing))
  when_complete_index <- which(!is.na(when_missing))
```

```
  is_different_miss <- is_different[when_missing_index]
  is_different_complete <- is_different[when_complete_index]
  result_ozone_temp <- t.test(is_different_miss,
↪   is_different_complete)
}
```

Naming is tricky, and it's fine for it to take a few goes.

Return the last thing. The function currently assigns to `result_ozone_temp` and returns it invisibly. Put it on its own line at the end so the return is explicit to a reader:

```
  result_ozone_temp <- t.test(is_different_miss,
↪   is_different_complete)
  result_ozone_temp
}
```

Clean up the old lettuce. Remove the commented-out lines now that you no longer need them.

Name the function so it evokes the action. `misstest` was a placeholder. What does this actually do?

```
missingness_impact <- function(is_different, when_missing) {
  when_missing_index <- which(is.na(when_missing))
  when_complete_index <- which(!is.na(when_missing))
  is_different_miss <- is_different[when_missing_index]
  is_different_complete <- is_different[when_complete_index]
  t.test(is_different_miss, is_different_complete)
}
```

Then use it, and write the output to a variable:

```
temp_difference_ozone_missing <- missingness_impact(
  when_missing = airquality$Ozone,
  is_different = airquality$Temp
)
```

The process, in short:

1. **Copy** the text into the body.
2. **Identify** the complexity to manage.
3. **Abstract** that complexity.
4. **Iterate** - writing functions is iterative, just like regular writing.

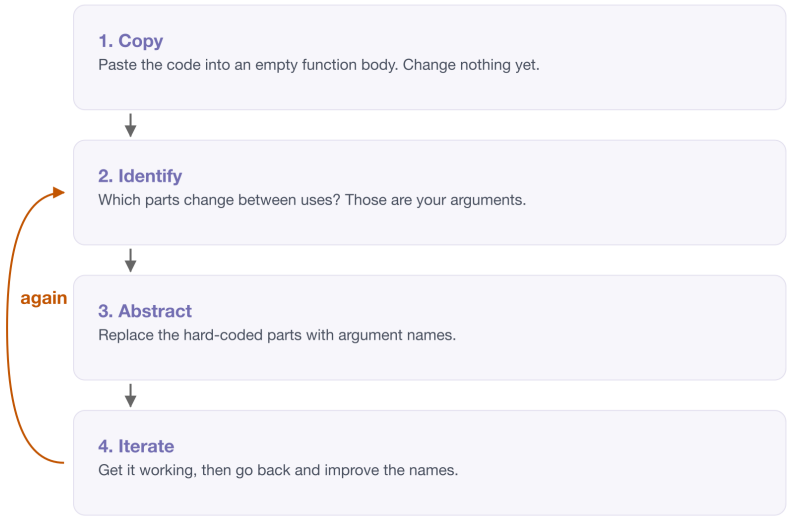


Figure A.2.: Steps 2 and 3 are the loop. You only copy the code in once.

DRY, and the better version of it

You will often be told **DRY**: *Don't Repeat Yourself*. If you copy and paste the same code three times, write a function.
That's true, but it treats the symptom. The real problem is **complex code**.
You only end up repeating yourself because you couldn't *express* the idea. So the cause is expression and reasoning, not repetition.
The version I prefer is **DRRY**: *Don't Re-Read Yourself*.[1]
The trigger is not "have I typed this three times". It is "have I had to read this block again to work out what it does, and what is different about it this time". That happens on the *first* copy, not the third, and it happens plenty of times when there is no copy at all.
A function isn't only needed when you repeat code. It's needed when you have an idea worth naming.

Your Turn

Earlier you wrote down a sentence describing a chunk of your own code. Now build the function around it, using the process above. If you don't have one to hand, use one of these. Both do the same thing more than once, with something small changed each time.
One

[1] Naming credit to Miles McBain.

```
adelie <- penguins[penguins$species == "Adelie", ]
adelie_mean <- mean(adelie$body_mass, na.rm = TRUE)

gentoo <- penguins[penguins$species == "Gentoo", ]
gentoo_mean <- mean(gentoo$body_mass, na.rm = TRUE)

chinstrap <- penguins[penguins$species == "Chinstrap", ]
chinstrap_mean <- mean(chinstrap$body_mass, na.rm = TRUE)
```

Two

```
ozone_high <- airquality$Ozone > 30
ozone_pct <- round(sum(ozone_high, na.rm = TRUE) /
 ↪  sum(!is.na(ozone_high)) * 100, 1)

temp_high <- airquality$Temp > 80
temp_pct <- round(sum(temp_high, na.rm = TRUE) /
 ↪  sum(!is.na(temp_high)) * 100, 1)
```

The second one has two things changing rather than one, so it needs two arguments. Work out which they are before you start typing.

1. **Copy.** Make an empty function, and paste the chunk into the body. Don't change anything yet.
2. **Identify.** Which parts of it change between uses? Those are your arguments.
3. **Abstract.** Replace the hard-coded parts with argument names. Comment the old lines out rather than deleting them, so you can see what you are replacing.
4. **Iterate.** Get it working first, then go back and improve the names. The first name you pick is rarely the one you keep.

Then call your function, and check you get the same answer you got before you started. That check matters. It is very easy to tidy something into a different result.
If you don't have a chunk of your own to hand, use `06-bad-style/messy-analysis.R` from the course repository. The loop at the bottom is doing the same thing to each species, one at a time.

A.2. Revisiting a script

Everything so far has been one function at a time. Let's do a whole script, because that is what you will actually be handed.

This is the hundred line script from What is each chunk for?. We left it with six comments in it:

```
# 1. read the raw spreadsheet for one year
# 2. fix the column names Excel gave us
# 3. pull out the education rows, make them long, drop the
 ↪  wide age bands
# 4. pull out the population rows, make them long, drop the
 ↪  wide age bands
# 5. join them, and work out the proportion studying
# 6. plot it
```

That list is a to-do list. Each line is a function waiting to be given a name, and we already know steps 3 and 4 are the same idea twice.

So let's work down it.

A.2.1. One chunk at a time

Step 1 was this:

```
raw_data <- read_excel(
  path = "data/Education and work, 2023, Datacube 2 (Table
   ↪  11).xlsx",
  sheet = "2014",
  skip = 4,
  n_max = 32,
  .name_repair = make_clean_names
)
```

Same process as before. Copy it into a function body, ask what changes between uses, and make that an argument.

The only thing that changes between uses is the year. It is a sheet in the same workbook.

```
read_raw_education_data <- function(path, year) {
  read_excel(
    path = path,
    sheet = year,
    skip = 4,
    n_max = 32,
    .name_repair = make_clean_names
  )
}
```

The year changes between calls, so it is an argument. The path changes too - not today, but the day the ABS publishes the 2024 datacube, or the day you point this at a different spreadsheet entirely. A function that reads a file should be told which file.

The script gets one line shorter, and one problem smaller:

```
raw_data_2014 <- read_raw_education_data(
  path = here("data/Education and work, 2023, Datacube 2
   ↪  (Table 11).xlsx"),
  year = "2014"
)
```

Notice what just happened to the request that started all this. "Can you add another year?" is now `read_raw_education_data(year = "2015")`. We are not finished, but the hard-coded `2014` has stopped being a fact buried in line 12 and started being an argument.

`here()` snuck in there too, from Project organisation 1. Building the path with `here()` means the call works the same whether R is sitting in the project root or somewhere else entirely.

A.2.2. The step that was written twice

Steps 3 and 4 were forty lines between them, doing one idea to two different sets of rows. Here they are side by side, with the pivot and filter trimmed out so the shape is visible:

```
# step 3: the education rows
data_subset <- data ▷
  slice(4:11) ▷
  set_names(the_names2)

data_studying <- data_subset ▷
  pivot_longer( ... ,  values_to = "n_studying") ▷
  arrange(state_territory, age_group) ▷
  filter( ... )

# step 4: the population rows
data_population <- data ▷
  slice(24:31) ▷
  set_names(the_names2)

data_population <- data_population ▷
  pivot_longer( ... , values_to = "population") ▷
  arrange(state_territory, age_group) ▷
  filter( ... )
```

Same `slice()`, same `set_names()`, same `pivot_longer()`, same `arrange()`, same five-line `filter()`. Two things differ: which rows get sliced, and what the value column is called.

Pull out the shared shape first:

```
extract_rows_set_names <- function(raw_data, row_numbers,
 ↪  names) {
  raw_data ▷
```

```
    slice(row_numbers) ▷
    set_names(names)
}
```

And now both callers are one line each:

```
educated_raw <- extract_rows_set_names(
  raw_data_2014,
  row_numbers = 4:11,
  names = new_names
)

population_raw <- extract_rows_set_names(
  raw_data_2014,
  row_numbers = 24:31,
  names = new_names
)
```

Those two lines are worth staring at, because they say something the original script could not. The education rows and the population rows are *the same kind of thing*, pulled from different parts of one spreadsheet. Forty lines of near-identical pipeline never said that. Two calls to one function say it immediately.

A.2.3. Grouping the small functions into bigger ones

Once each chunk has a name, the chunks themselves start to group:

```
prepare_education <- function(raw_data) {
  new_names <- extract_education_col_names(raw_data)

  educated_raw <- extract_rows_set_names(
    raw_data,
    row_numbers = 4:11,
    names = new_names
  )

  pivot_longer_educated(educated_raw)
}
```

Three small functions inside one slightly bigger one, and the bigger one reads like the comment it came from: get the names, pull out the rows, make it long.

Do the same for population and for the join, and the whole hundred lines collapse into one:

```
read_educated_population <- function(year = "2014") {
  raw_data <- read_raw_education_data(year = year)

  educated <- prepare_education(raw_data)

  population <- prepare_population(raw_data)

  combine_educated_population(
    education = educated,
    population = population,
    year = year
  )
}
```

Read that function. It is the six comments, in order, as code.

A.2.4. What the script looks like now

```
source("packages.R")
lapply(list.files("./R", full.names = TRUE), source)

educated_population_2014 <- read_educated_population(year =
 ↪  "2014")

ggplot(educated_population_2014, aes(x = prop_studying, y =
 ↪  state_territory)) +
  geom_col() +
  facet_wrap(~age_group)
```

Ninety lines to six. The other eighty-four did not disappear, they moved into `R/`, one file per function, exactly the layout from Project organisation 1.

And here is the bit I find genuinely satisfying. The original request was one more year of data. Now it is one more call:

```
educated_population_2015 <- read_educated_population(year =
 ↪  "2015")
```

In the hundred line version, that was eighty lines copied and pasted, with five numbers to edit inside them and no way to be sure you got them all.

> Extension: all ten years at once
>
> Not required, and we may not get to this in the session. But it is where the whole chapter has been heading.
> Once adding *one* year is one call, adding *ten* is `map()`, which we

met back in Anatomy of a function:

```
library(purrr)

study_years <- as.character(2014:2023)

educated_population <- study_years ▷
  map(\(year) read_educated_population(year = year)) ▷
  list_rbind()
```

Ten years, three lines. `map()` gives you a list with one data frame per year, and `list_rbind()` stacks them into one.
The thing to notice is that none of this was available before. You cannot `map()` over the original script, because there is no function to map. Every bit of work above is what bought you these three lines.
You may meet `map_dfr()` in older code, which did the mapping and the stacking in one call. purrr superseded it in version 1.0.0, in favour of the two steps above.

Read the whole thing

The full progression lives in the ozed repo, one script per step: `script_00.R` is the starting point, and `script_03.R` is the ten year version. Every intermediate step is there too, so you can see it happen a chunk at a time rather than as a before and after.
It is not finished, and I have left it that way on purpose. `pivot_longer_educated()` and `pivot_longer_population()` are still nearly the same function, differing in one argument. That is the next thing I would fix, and you can see exactly why by putting them side by side.

Your Turn

1. Take the script you chunked in the last chapter. Turn the *first* chunk into a function. Only the first. That is the `read_excel()` call at the top, the one commented `# 1. read the raw spreadsheet for one year` - the same chunk we turned into `read_raw_education_data()` above.
2. Run the script again. Same answer?
3. Now look at your list of chunks. Are any two of them the same idea on different data? That pair is the one worth doing next, and it is where most of the gain is.
4. Stop there. A script with two of its six chunks named is better than the one you started with, and you do not have to finish today.

Going faster with {fnmate}

The process above has one annoying step. You write the call you *wish* existed, then go and create the function somewhere else, with the right arguments, and switch back.
`{fnmate}` by Miles McBain removes that step. Write the call you want:

```
temp_difference <- missingness_impact(
  when_missing = airquality$Ozone,
  is_different = airquality$Temp
)
```

Put your cursor on it, trigger `{fnmate}`, and it generates the function definition - correctly named, with the right argument names - in your `R/` directory, ready for you to fill in the body.
This matters more than a keystroke saving. It makes the **outside-in** approach cheap. You imagine the interface you want first, then write the thing that satisfies it. Without tooling, outside-in is enough extra effort that most people default to inside-out.
Worth trying, with or without the package. Install it with `remotes::install_github("milesmcbain/fnmate")` - and if the install gives you trouble, skip it, because it is the habit that matters rather than the package.

1. Think of something you want to do but haven't written yet. Write **only the call**, with the argument names you wish existed.
2. Read what you wrote back. Could someone else tell what it does from the call alone? If not, rename things now, while it costs you nothing.
3. Now generate the function, with `{fnmate}` or by hand, and fill in the body.

Notice the order you just worked in. You designed the interface first and the implementation second, which is the opposite of how most of us write code.

B. Anonymous functions

> "Everything that exists is an object. Everything that happens is a function call."
>
> – John Chambers

Chapter 4 makes the point that functions are values: you can hand one to another function without calling it. This appendix is the next question, which comes up every time someone reads a bit of tidyverse code.

What is that `\(x)` doing there, why do older examples use `~ .x` instead, and where did any of it come from?

Functions without names

Duration 10 minutes
Questions

- When should a function have a name, and when should it not?
- What is `\(x)` short for?
- Why does older code use `~ .x`, and should I write it?

The examples here use `map_dbl()`, so:

```
library(purrr)
```

Everything so far has handed over a function that already had a name. Sometimes you want to hand over a function you will never need again, where naming it would just add clutter to the top of your script.
You can write the function right there, in the argument:

```
map_dbl(airquality, function(x) round(mean(x, na.rm =
↪  TRUE), 1))
```

```
  Ozone Solar.R    Wind    Temp   Month     Day
   42.1   185.9    10.0    77.9     7.0    15.8
```

That `function(x) round(mean(x, na.rm = TRUE), 1)` is an ordinary function. It simply never got a name, which is why it is called an **anonymous function**.
It does the same job as this:

```r
mean_rounded <- function(x) {
  round(mean(x, na.rm = TRUE), 1)
}

map_dbl(airquality, mean_rounded)
```

```
  Ozone Solar.R    Wind    Temp   Month     Day
   42.1   185.9    10.0    77.9     7.0    15.8
```

So which should you write? My rule is straightforward. If you would use it twice, or if a name would explain something, give it a name. Otherwise, leave it anonymous.
While you are learning, name them. A named function is something you can call on its own and poke at, which makes it far easier to debug.
The shorthand: `\(x)`
Since R 4.1 you can write `\(x)` instead of `function(x)`:

```r
map_dbl(airquality, \(x) round(mean(x, na.rm = TRUE), 1))
```

```
  Ozone Solar.R    Wind    Temp   Month     Day
   42.1   185.9    10.0    77.9     7.0    15.8
```

Identical to the version above, just shorter. `\(x) x + 1` and `function(x) x + 1` are the same thing.
You will see `\(x)` constantly in modern R code, so it is worth being able to read. When you are writing your own, use whichever you find clearer.

Some history: where the backslash comes from

`\(x)` is written with a backslash because it looks a little like λ, the Greek letter lambda.
That comes from lambda calculus, a way of describing computation worked out by Alonzo Church in the 1930s, where `λx.` means "a function of x".
Plenty of programming languages borrowed the idea and the word, which is why you will hear anonymous functions called "lambdas" in Python, JavaScript and elsewhere.

Some history: older purrr code and the `~ .x` shorthand

In older code, and in plenty of code still being written today, you will see this instead:

```r
map_dbl(airquality, ~ round(mean(.x, na.rm = TRUE), 1))
```

```
  Ozone Solar.R    Wind    Temp   Month     Day
   42.1   185.9    10.0    77.9     7.0    15.8
```

That `~` is a **formula**, and purrr treats it as a shorthand for a function. Inside it, `.x` stands for whatever is being passed in. So all three of these do the same thing:

```
map_dbl(airquality, ~ round(mean(.x, na.rm = TRUE), 1))
map_dbl(airquality, \(x) round(mean(x, na.rm = TRUE), 1))
map_dbl(airquality, function(x) round(mean(x, na.rm =
 ↪  TRUE), 1))
```

The formula version came first, because until R 4.1 there was no short way to write a function, and `function(x)` every time got tiring. Now that `\(x)` exists, purrr's own documentation recommends it, and it is what I would write in new code.
You still need to be able to **read** `~ .x`, because it is everywhere. Two more things you will meet alongside it. A bare `.` works in place of `.x`, and in two-argument functions like `map2()` the arguments are `.x` and `.y`:

```
map2_dbl(1:3, 4:6, ~ .x * .y)
```

```
[1]  4 10 18
```

C. Working closer to the speed of thought

Overview

Duration 15 minutes, then a 5-10 minutes a day for a couple of months
Questions

- Why would typing speed matter for writing code?
- Which keyboard shortcuts are actually worth learning?
- What is an RStudio addin, and which ones earn their keyboard shortcut?

C.1. Typing, and the speed of thought

Here's the jam: **if you can type faster, you operate closer to the speed of thought.** It's not really about working faster. It's about being able to clearly express what's in your head. And when you can do that, you put yourself in a good position to enter the flow state, being in the zone, completely absorbed, where things feel easy and make sense.

When you type quickly and accurately, you aren't making the small stumbling mistakes that pull you out of that state.

Keyboard shortcuts do the same job. More of what's in your head gets onto the screen without friction.

I don't think being a fast typist is **necessary** to be a better programmer. But tools that let you work with less resistance are good.

I once switched to a new keyboard and felt like I was walking around with ankle weights and a large unwieldy backpack. I simply couldn't get what was in my head onto the screen.

So I spent a couple of months working on typing every day. It made my typing faster and more accurate, which made my coding faster. I ended up ditching the keyboard and keeping the skill.

Two things worth knowing:

- **Accuracy first.** There's not much benefit in typing the wrong and unintelligible words very quickly. keybr is good for this, and I have heard good things about monkeytype.

- **You don't need a new keyboard.** I've spent a lot of time and money looking for a good one and I'm still not there. My favourite is still an old wireless Mac keyboard. Practising is a better use of your time than shopping. But if you want the keyboard, get the keyboard.

C.2. Shortcuts worth learning first

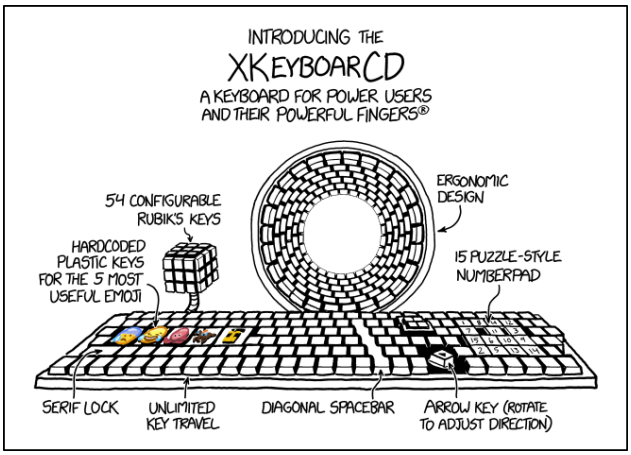


Figure C.1.: Sourced from xkcd 2150

You already use keyboard shortcuts every day without thinking about them. Saving with `Cmd / Ctrl + S`, or undoing with `Cmd / Ctrl + Z`.

So this is not a new skill. It's the same skill, pointed at the things you do most often in R.

These are the ones that come up repeatedly in this course:

- `Cmd / Ctrl + Enter` - run the current line
- `Cmd / Ctrl + Shift + Enter` - run the whole script or chunk
- `Cmd / Ctrl + Shift + F10` - restart R. We will use this a lot
- `Cmd / Ctrl + S` - save, and with air set up, reformat
- `Cmd / Ctrl + Shift + C` - comment or uncomment the current line
- `Cmd / Ctrl` + click on a function - jump to where it is defined
- `Ctrl + .` - go to a file or function by name
- `Alt / Option + Shift + J` - jump to a section of the current file
- `Cmd / Ctrl + Shift + F` - find in files, across a whole directory

> What about `Cmd / Ctrl + Shift + A`?
>
> That's RStudio's own "reformat section", and it's what people reached for before air existed.
> You can stop using it. Air does a better job, does it to the whole file, and does it on save without you asking.

More shortcuts, once those are automatic

Moving around

- `Alt + Left / Right` - move the cursor one word at a time. Add `Shift` to select
- `Cmd / Ctrl + Left / Right` - jump to the start or end of the line
- `Alt + Up / Down` - move the current line up or down
- `Cmd / Ctrl + Alt + Up / Down` - copy the current line up or down
- `Ctrl + Alt + Up / Down` - multiple cursors
- `Cmd / Ctrl + D` - delete the current line

Being able to move by word, or jump to the end of a line, matters more than it sounds. You stop leaning on the arrow key to get where you're going.

Elsewhere on your machine

- `Cmd / Ctrl + Z`, then `Cmd / Ctrl + Shift + Z` - undo, then undo the undo
- `Cmd / Ctrl + Tab` - change application
- `Cmd + Space` - Spotlight on macOS
- `Cmd / Ctrl + F` - search for a word. Works in nearly everything: PDFs, browsers, code
- `Cmd / Ctrl + T` / `+ W` / `+ Shift + T` - new browser tab, close tab, reopen the tab you just closed

Finding and setting your own

In RStudio, `Alt + Shift + K` brings up a shortcut menu (`Esc` to dismiss), and Tools → Keyboard Shortcuts Help lets you see and change all of them.

Some packages add their own commands, which you can then bind to a key. Those are addins, and they get a section of their own below.

Where to look things up

- The RStudio IDE cheat sheet has an index of the lot.
- Posit's guide to customising keyboard shortcuts covers rebinding them.

Your Turn

1. Bring up the shortcut reference with `Alt / Option + Shift + K`, and find the shortcut for inserting a pipe (`|>`).
2. Pick two shortcuts from the list above that you do not currently use, and write them on something you will see tomorrow.
3. Spend three minutes practising them on a script of your own.
4. Spend five minutes on keybr. Note your accuracy, not your

speed.

Ask me to demonstrate a few of my favourite combos: multiple cursors, moving and copying lines, and jumping to a function by name. They are much easier to understand when you watch them than when you read about them.

C.3. RStudio addins

There is a menu in RStudio that most people never open, sitting in the toolbar next to the code and knit buttons. It says **Addins**.

An addin is nothing exotic. It is an R function that a package has registered with RStudio, so that instead of typing the call, you pick it from a menu, or press a key.

The reason they are interesting is not the menu. It is that **anything in that menu can be bound to a keyboard shortcut**, which turns a two minute task into a keypress.

C.3.1. Binding one to a key

In RStudio: Tools → Modify Keyboard Shortcuts, then type "addin" into the filter box. You get a list of every addin you have installed. Click the shortcut column, press the keys you want, hit Apply.

That's the whole setup. It takes about twenty seconds, and I would do it for two addins rather than ten.

Positron asks a bit more of you, because there's no list to click. You write the binding yourself, in `keybindings.json`. There's a worked example of that, for reprex, in Writing a reprex / getting unstuck.

> **!** Addins in Positron
>
> Addins started life as an RStudio feature, and for a while that is all they were. Positron runs them now. Open the command palette and run **R: Run RStudio Addin**, and you get the same list, grouped by the package it came from.
> There is a catch. Positron doesn't implement every method in the RStudio API, so an addin that reaches for one it hasn't got will fall over. If you write your own, `rstudioapi::hasFun()` tells you whether a method is there before you call it.
> Either way, it's worth knowing the function underneath. `reprex::reprex()` works anywhere, and the addin is a convenience on top of it.

C.3.2. The ones I actually use

`{reprex}` - "Render reprex". Select some code, press the key, and a formatted reproducible example lands on your clipboard. Mine is bound to `Cmd / Ctrl + Shift + R`, and it is the addin I would give up last. See Writing a reprex / getting unstuck for what it does.

`{fnmate}` by Miles McBain - put your cursor on a call to a function that does not exist yet, press the key, and it writes the function definition for you, in the right file, with the arguments filled in.

This one changed how I write code. It means you can write the call you *wish* existed, then make it exist, rather than stopping to define things before you know what you want. It accelerated my functional programming considerably.

`{datapasta}`, also by Miles McBain - copy a block of cells from a spreadsheet, press the key, and it pastes into your script as a `tibble()` you can read.

I use this constantly for small lookup tables. It is the fastest way I know to get twenty rows out of somebody's email and into code where they can be version controlled.

`{shrtcts}` by Garrick Aden-Buie - for making your own. Write an R function, add a comment saying which key you want, and it becomes a shortcut. Good for the thing you do forty times a week that nobody has packaged.

C.3.3. A word of warning

Addins are easy to collect and hard to remember.

I have installed plenty that I used twice. The ones on this list stuck because they replace something I was doing by hand several times a day, and because I bound them to a key rather than leaving them in the menu.

If it lives in the menu, you will forget it exists. So bind it, or skip it.

> **Your Turn**
>
> 1. Open the Addins menu, or run **R: Run RStudio Addin** from the Positron command palette. What is already in there from packages you installed for other reasons?
> 2. Install `{fnmate}` or `{datapasta}`, and bind it to a key. Tools → Modify Keyboard Shortcuts in RStudio, or a line in `keybindings.json` in Positron.
> 3. Use it once, today, on real work. If you don't reach for it again this week, unbind it. That is a perfectly good outcome.

> **Read more**
>
> This appendix is adapted from my blog post Get good with R: typing skills and shortcuts, which has more detail, including a note on vim and emacs, and, importantly, how to get *out* of vim.

D. How ▷ differs from %>%

> "Ceci n'est pas un pipe."
>
> – René Magritte, quoted in the DESCRIPTION file of the {magrittr} package

This question comes up every time I teach anything with a pipe in it, and it's a fair question. R has two of them, they look like they do the same job, and nobody tells you which one to use.

This appendix is the answer I give. Short version: use ▷, and here is what you are giving up.

Overview

Duration 10 minutes
Questions

- Where did each pipe come from?
- What can `%>%` do that ▷ cannot?
- What is actually happening when R sees a pipe?
- Which one should I use, and does it matter?

D.1. The short answer

`x ▷ f()` is rewritten by R, before anything runs, into `f(x)`.

`x %>% f()` is a function call. `%>%` is a real operator that a package defines, and it works out what to do with your code while the code is running.

Almost every difference between them falls out of that one distinction.

Some history: where they both came from

`%>%` arrived first, in the `{magrittr}` package by Stefan Milton Bache, in 2014. The idea came from F#, which has had a ▷ operator for years, and there were a couple of earlier R attempts, including a `%.%` chain operator that did not stick.
It caught on enormously. The tidyverse re-exported it, so for most people `%>%` arrived without them ever installing `{magrittr}`, and for the best part of a decade "the pipe" meant `%>%`.
The name is a joke about René Magritte, which is why the DESCRIPTION file says "Ceci n'est pas un pipe."
The base R pipe, ▷, came much later, in R 4.1.0 in May 2021,

after a lot of discussion inside R Core. The _ placeholder followed in 4.2.0 in April 2022, and _$ extraction in 4.3.0 in April 2023.

D.2. What actually differs

D.2.1. ▷ needs the parentheses

```
c(1, 2, 3) %>% mean     # fine
c(1, 2, 3) ▷ mean       # error
#> Error in mean :
#>   The pipe operator requires a function call as RHS
```

%>% will accept a bare function name and call it for you. ▷ won't.

I've come around to preferring this. mean() with the brackets says "this is a function call", and now the pipe reads the same way as every other line of R you write.

D.2.2. The placeholder is _, not ., and it is fussier

Both pipes put the left hand side into the *first* argument by default. When you need it somewhere else, you use a placeholder.

```
mtcars %>% lm(mpg ~ cyl, data = .)
mtcars ▷ lm(mpg ~ cyl, data = _)
```

Three rules on _ that . does not have:

It must be a named argument.

```
x ▷ f(_, y)
#> Error: pipe placeholder can only be used as a named
 ↪  argument
```

It can only appear once.

With %>% you can write x %>% f(., .). With ▷ you can't.

It cannot be nested inside another call.

```
x ▷ f(g(_))
#> Error: invalid use of pipe placeholder
```

x %>% f(g(.)) is perfectly happy.

This is the one I miss most, and it's the difference people hit first.

Since R 4.3.0 there is one exception, which is extraction:

```
mtcars ▷ _$mpg ▷ head()
#> [1] 21.0 21.0 22.8 21.4 18.7 18.1
```

> ! Only if you can require R 4.3.0
>
> _$ needs R 4.3.0, released April 2023, and _ at all needs 4.2.0.
> If you're writing anything other people will run, that's a real constraint, not a detail.
> Check what you are willing to require before you use either.

D.2.3. When you need something _ cannot do

Write an anonymous function. This is the general escape hatch, and it covers every case above:

```
mtcars ▷ (\(d) f(g(d), d))()
```

That is not beautiful.

It is explicit, though, and it doesn't need you to remember any placeholder rules.

D.2.4. `%>%` needs to be attached, ▷ does not

▷ is part of the language. It works in a vanilla R session, in a package, or in a script someone sends you with no libraries at all.

`%>%` needs `{magrittr}`, or something that re-exports it like `{dplyr}`. Which means a script with `%>%` and no `library()` call is a script that doesn't run. See could not find function for how that turns up.

D.3. Under the hood

This is the bit I find genuinely delightful.

▷ is handled by the **parser**. By the time R runs anything, the pipe is gone:

```
quote(x ▷ f())
#> f(x)

quote(x ▷ f(y))
#> f(x, y)

quote(x ▷ f(y = _))
#> f(y = x)
```

Look at that output. There's no pipe in it. R read your code, rewrote it, and what runs is an ordinary nested call. The pipe is a way of *writing* code, not a thing that exists while it runs.

`%>%` is the other kind of thing entirely. It is a function, it is still there at run time, and it appears in your call stack:

```
f <- function(x) sys.calls()

g1 <- function() 1 ▷ f()
g1()
#> 1: g1()
#> 2: f(1)

g2 <- function() 1 %>% f()
g2()
#> 1: g2()
#> 2: 1 %>% f()
#> 3: f(.)
```

One extra frame. That matters when you're reading a traceback at 5pm, because every frame is one more thing between you and the line that actually broke.

To be fair to `{magrittr}`, this used to be much worse. Version 2.0 rewrote the internals and cut the stack down enormously. What you see above is the good version.

> Going deeper: the same trick shows up in `\()`
>
> `\(x) x + 1` is also a syntax transformation. R reads it and hands back an ordinary function:
>
> ```
> quote(\(x) x + 1)
> #> function(x) x + 1
> ```
>
> Same idea as the pipe. A shorter way to write something, which stops existing the moment R has read it.

D.4. Is one faster?

Yes, and it almost certainly doesn't matter.

Because ▷ disappears at parse time, it costs nothing at run time. `%>%` is a function call that does some work every time it runs. Measured on my machine:

```
x <- 42
bench::mark(
  "identity(x)"      = identity(x),
```

```
  "x |> identity()"  = x |> identity(),
  "x %>% identity()" = x %>% identity()
)
#> expression            min   median `itr/sec`
#> identity(x)         40.9ns    123ns  8277082.
#> x |> identity()       41ns     82ns  9333365.
#> x %>% identity()     738ns    861ns  1119400.
```

So `%>%` is roughly ten times slower.

Now be careful with that sentence, because it's exactly the kind of relative measurement that misleads people. Ten times slower is 800 nanoseconds. You'd need to run it a million times to lose a second.

If your pipeline is slow, the pipe is not why.

D.5. Which one do I use?

I use the base pipe.

There's nothing in `%>%` that I need for the way I write code, and `|>` is one less thing to attach, one less frame in a traceback, and part of the language rather than a package.

The honest caveat: if you already have a codebase full of `%>%` and it works, leave it.

Rewriting a thousand pipes to save 800 nanoseconds each is not an afternoon well spent. Use `|>` in the new code.

D.6. What about in a package?

I would use `|>`.

R 4.1.0 came out in May 2021, and 4.2.0, which brought `_`, in April 2022. My own view is that requiring an R released three or more years ago is plenty conservative.

If you want a stricter standpoint, the tidyverse team support the current version plus the previous four, which is a five year window. On that policy `|>` alone became safe in May 2026, and `_` becomes safe in April 2027.

Either way, write down which R version your package requires, in `DESCRIPTION`, so the decision is visible rather than implied.

> **Your Turn**
>
> 1. Take a `%>%` pipeline from your own code and rewrite it with `|>`. Which of the rules above did you hit?
> 2. Run `quote()` on one of your pipelines. Does the code it prints look like what you expected to be running?

3. Find a place where you used `.` more than once, or nested inside another call. What does the `\()` version look like, and do you prefer it?

Read more

This appendix is adapted from my draft blog post on the two pipes. The pieces I found most useful while writing it:

- The tidyverse post on base vs magrittr pipes, which is the definitive comparison.
- Albert Rapp's write-up.
- This Stack Overflow thread, which is remarkably comprehensive, and where I first found out about the experimental ⇒ operator. That one is not in the NEWS file and is very much experimental, so treat it as a curiosity rather than something to use.

E. What {} really does

> "To understand computations in R, two slogans are helpful: Everything that exists is an object. Everything that happens is a function call."
>
> – John Chambers, quoted in Advanced R

You've been typing curly braces since your first function, and there is a decent chance nobody ever told you what they are.

This appendix is short, and it will change how you read R.

Braces aren't punctuation. They're a function, and once you see that, several things that looked like arbitrary rules turn into one rule.

Overview

Duration 10 minutes
Questions

- What is { actually doing?
- Why does a function return its last line?
- When do I need braces, and when are they optional?
- What is `{ }`, and why do people write `{dplyr}`?

E.1. { is a function

Here is the thing that surprises people:

```
`{`
#> .Primitive("{")
```

{ is a function. You can call it like one:

```
`{`(10, 20, 30)
#> [1] 30
```

And that tells you what a braced block does. **It evaluates each expression in turn, and gives back the value of the last one.**

Which is exactly what this does:

```
x <- {
  10
  20
  30
}

x
#> [1] 30
```

The first two lines ran. Nothing kept their results, so they are gone. The last one is what came out.

That's the entire behaviour, and everything below is a consequence of it.

E.2. Why a function returns its last line

This is the one that pays off immediately.

```
mean_center <- function(variable) {
  variable - mean(variable, na.rm = TRUE)
}
```

There's no `return()` in that function, and it still returns something.

People often explain that as "R functions return their last line", as though it were a special rule about functions.

It isn't. The body of that function **is** a braced block, and a braced block gives you its last value. The function hands back whatever its body evaluated to, and its body is a `{`.

Which is also why this works, with no braces at all:

```
mean_center <- function(variable) variable - mean(variable,
 ↪  na.rm = TRUE)
```

The body is a single expression. There is nothing to group, so there is nothing for `{` to do.

> Going deeper: so when do I use braces in a function?
>
> The house rule for this course, and for my own code, is:
> **Use `\(x) ...` for a one line anonymous function. Use `function(x) { ... }` for everything else.**

```
# fine
purrr::map(my_list, \(x) x^2)

# braces, because there are two steps and a name
center_scale <- function(variable) {
  centered <- mean_center(variable)
  scale_sd(centered)
}
```

I put braces on any function I have named and saved, even a one-liner, because the day it grows a second line I don't want to be editing its shape as well as its content.

E.3. Where braces are load-bearing

`if`, `for` and `while` take **one** expression. Braces are how you give them more than one.

Without braces, only the next expression belongs to the `if`:

```
z <- 0

if (TRUE)
  z <- 1
  z <- 2

z
#> [1] 2
```

Read that carefully, because it does not do what it looks like it does. Only `z ← 1` is inside the `if`. The line `z ← 2` is an ordinary next line, and it runs no matter what the condition was.

This is a genuinely nasty bug, because the indentation says one thing and R does another.

```
if (TRUE) {
  z <- 1
  z <- 2
}
```

Now both lines are in the block, because the block is one expression.

> **!** Always brace your `if` and your `for`
>
> Even when the body is one line, even when it fits.
> The cost is two characters. The bug it prevents is invisible, survives review, and is indistinguishable from correct code at a glance.
> A formatter won't save you here either. `air format .` lays out

> what you wrote, and what you wrote was valid.

E.4. Braces do not make a new scope

This one trips up people arriving from other languages.

In C, or Java, or JavaScript with `let`, a braced block has its own scope, and a variable made inside it stays inside it.

In R it doesn't:

```
y <- 1

{
  y <- 99
}

y
#> [1] 99
```

The assignment reached straight out and changed `y`.

{ groups expressions. It does not build an environment. **Functions** build environments, which is why wrapping something in a function is how you keep its working variables to itself, and a bare block is not.

E.5. Braces in a pipe

With `%>%`, a braced block lets you use the placeholder somewhere awkward:

```
x %>% { f(.) + g(.) }
```

The base pipe does not support this. ▷ needs a function call on the right, and `{ ... }` is not one. The equivalent is an anonymous function:

```
x ▷ (\(d) f(d) + g(d))()
```

Which is more typing, and it is also clearer about what is happening: you are making a small function and calling it. See How ▷ differs from `%>%` D for the rest of that story.

E.6. The two other curly things

Both of these are different from everything above, and both get confused with it.

E.6.1. { } is not two blocks

```
count_by <- function(data, var) {
  data |> dplyr::count({{ var }})
}

count_by(palmerpenguins::penguins, species)
```

That is **embracing**, from `{rlang}`, and it is how you write a function that takes a bare column name the way `{dplyr}` verbs do. It says "take what the caller typed, and use that".

It is not a block inside a block. The doubled brace is deliberately chosen to be visually distinct, and it only means anything inside a tidy evaluation context.

If you write functions that wrap `dplyr` verbs, you need it. If you don't, you can leave it entirely alone.

E.6.2. {dplyr} is just how we write package names

Throughout this book you will see `{dplyr}`, `{here}`, `{air}`.

That is a convention from the R community, not R syntax. It is a way of saying "this is a package, not a function", which matters when a package and its main function share a name. Compare "use here" with "use `{here}`".

There is no code involved. You will see it in blog posts, on social media, and in this book.

Nothing breaks if you don't use it.

> **Your Turn**
>
> 1. Run `` `{`(1, 2, 3) ``. Which value comes back, and why?
> 2. Find an `if` or a `for` in your own code with no braces. Add them.
> 3. Take a function you have written that ends in `return(x)`. Remove the `return()` and check it still works. Do you prefer it?

Links

- https://r-best-practices.njtierney.com
- R: A Language for Data Analysis and Graphics
- Quarto: callout blocks
- https://github.com/njtierney/rbestpractices
- https://github.com/njtierney/rbp-exercises
- https://cran.r-project.org/

- RStudio
- Positron
- palmerpenguins
- R version 4.5.0 is out
- `basepenguins`
- Air
- Jarl
- Etienne Bacher
- “Workflow”
- RStudio, what and why
- “quarto for scientists”
- course exercises
- Posit’s RStudio guide to the Command Palette
- “Visual Studio Code - Open Source”
- Posit’s article on the topic
- “PNG”
- “Project-oriented workflow”
- “The R Proj File”
- {here}
- Common sense approaches to sharing tabular data alongside publication
- “Introduction to git and github”
- git
- The files chapter of the Tidyverse style guide
- “How to name files like a normie”
- “Good enough practices in scientific computing”
- Hadley Wickham
- Lionel Hutz
- The Simpsons
- Workflow: code style
- kerning
- style chapter of Advanced R, 1st edition
- Tidyverse style guide
- files chapter
- Code formatters
- `{styler}`
- Wikipedia page for lint
- `{lintr}`
- `{flir}`
- How to get good with R
- Structure and Interpretation of Computer Programs
- ozed
- Australian Bureau of Statistics
- Jenny Bryan’s “Code smells and feels”
- cache invalidation
- Phil Karlton
- Martin Fowler has written up the provenance of the quote
- the listing above
- When two packages collide
- the script
- {conflicted}

- “The magical number seven, plus or minus two”
- ozed repo
- comment your code as little (and as well) as possible
- Miles McBain
- Putting it all together
- How to get good with R
- R for Data Science
- `purrr`
- Anatomy of a function
- slides
- source
- Writing readable code
- `{reprex}`
- Positron’s own documentation
- “rubber duck debugging”
- `{naniar}`
- an entire keynote
- When two packages collide
- Magic reprex
- Writing readable code
- https://github.com/njtierney/rbp-exercises/blob/main/06-bad-style/messy-analysis.R
- Writing a reprex / getting unstuck
- rOpenSci’s reviewer guide
- review template
- “How to get good with R”
- “Get Good with R: Typing Skills and Shortcuts”
- Working closer to the speed of thought
- R for Data Science
- Advanced R
- What is each chunk for?
- `{fnmate}`
- lambda calculus
- Alonzo Church
- flow state
- keybr
- monkeytype
- xkcd 2150
- a section of their own below
- RStudio IDE cheat sheet
- customising keyboard shortcuts
- Miles McBain
- `{datapasta}`
- `{shrtcts}`
- Garrick Aden-Buie
- Get good with R: typing skills and shortcuts
- {magrittr}
- Stefan Milton Bache
- René Magritte
- could not find function
- The tidyverse post on base vs magrittr pipes

- Albert Rapp's write-up
- This Stack Overflow thread
- Advanced R
- `{rlang}`