OBESITY EXPERTISE

A first-principles curriculum in 196 books

BOOK 6 7th of 196 in the series

R, reproducibility engineering and data management · Rung 1

One dataset, one script, one command

KEEP THE RAW DATA UNTOUCHED AND LET CODE DO EVERY CHANGE

Dr. Harsh Maheshwari

MD Community Medicine Resident, AIIMS Raipur

drharshmaheshwari.com · version 1.0

OBESITY EXPERTISE · BOOK 6

R, reproducibility engineering and data management · Rung 1

One dataset, one script, one command

Dr. Harsh Maheshwari

MD Community Medicine Resident, AIIMS Raipur

contact@drharshmaheshwari.com · drharshmaheshwari.com · ORCID 0000-0001-8848-1918

SERIES	Obesity Expertise: A first-principles curriculum in 196 books
BOOK	Book 6, R, reproducibility engineering and data management · Rung 1 (7th of 196 in the series)
VERSION	1.0
LAST UPDATED	3 October 2026
AUTHOR	Dr. Harsh Maheshwari, MD Community Medicine Resident, AIIMS Raipur
CONTACT	contact@drharshmaheshwari.com
ORCID	https://orcid.org/0000-0001-8848-1918
NEWEST VERSION	drharshmaheshwari.com

© 2026 Harsh Maheshwari.

This book is shared free, for anyone to read, under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International licence (CC BY-NC-SA 4.0). You may copy, share and adapt it for non-commercial purposes, if you credit the author and share what you make under the same licence.

It quotes and adapts material from OpenStax textbooks (Rice University), which are licensed CC BY-NC-SA 4.0; that is why this book carries the same licence. It also quotes short passages from laws, reports and other works, each credited where it is used; those passages remain under their owners' terms. Fonts: Source Serif 4, Inter and JetBrains Mono, SIL Open Font License 1.1.

HOW THIS BOOK WAS MADE

Written for the author's own study, then shared. Drafted and checked with AI assistance (Claude, by Anthropic) under the author's direction. Every factual claim was checked against a source the checker had open, every quotation against its source text, and every worked answer recomputed. Mistakes can remain; the feedback page says how to report one.

NOT ADVICE

This book is for learning. It is not medical, legal or financial advice, and it does not replace a qualified professional who knows your situation.

About the author



Dr. Harsh Maheshwari

MD Community Medicine Resident, AIIMS Raipur

Harsh Maheshwari completed his MBBS at King George's Medical University (KGMU), Lucknow, and is a resident in MD Community Medicine at the All India Institute of Medical Sciences (AIIMS), Raipur. His MD thesis, on chrononutrition and obesity, was selected by the Indian Council of Medical Research (ICMR) for funding in 2026.

He learns best from first principles: rebuilding an idea from what came before it until the formula feels like the only reasonable answer. This series grew out of that habit.

His long-term aims are to reduce obesity in India and to help make India healthy.

Email	contact@drharshmaheshwari.com
Website	drharshmaheshwari.com
ORCID	0000-0001-8848-1918

Introduction

Why this book exists

Most analyses that cannot be repeated were not wrong in their statistics. They were done by hand: a cell edited in a spreadsheet, a number pasted from a screen, a step nobody wrote down. This book teaches the opposite habit from the first line. You learn R as a language with objects, functions and packages, the tidyverse verbs that import, clean, summarise and plot a data frame, and one rule that makes the rest worth doing: raw data is never edited and every change lives in code that another person can run.

How to read it

Read Book 0 first, or at least the sections this book names as it goes. Then read this one in order, with R open beside it and every example typed, not pasted. The book asks you to build one thing: a single dataset analysed from raw file to final figure in one script that another person can run with one command. Start the script in the first section and extend it as you read.

Every section has the same shape. **Definition** states the idea exactly. **In plain terms** says it again in everyday words. **Illustration** works through one real case, usually with a source you can open yourself. **Where this picture breaks** says where the illustration stops being true. **Must-know points** are what should change what you do. Then come **exercises** and, in most sections, a graded **practice set**.

Do the exercises before you look anything up. The worked answers are in the appendix at the back, under the same numbers. The references for each Part follow that Part. The glossary lists every term of art in the plain words it was first given. The last pages list all 196 books in the series and show where this one sits.

If you are new to this

Book 0, chiefly reading a table (F1), functions as input, rule and output (C4), average and spread (D5), graphs (C5, F2) and units (B1, B2, B6). The mathematics is Book 0's arithmetic only: percentages, a mean and a median, and BMI as weight divided by height squared; nothing beyond it is used. No programming, no knowledge of files and folders and no statistics software is assumed; you need a computer on which you can install R and RStudio.

On a first reading, read each section's definition, plain-terms explanation, illustration and must-know points, and do the exercises. Worked derivations can be skimmed and revisited once the idea feels familiar. Keep a calculator beside you: every worked answer can be followed with pen, paper and a calculator. Anything beyond the stated prerequisites is explained where it first appears, or marked as beyond this book. The **Symbols used in this book** page lists every sign and Greek letter with how to read it and where it is first used.

How to send feedback

If something is wrong, unclear, or could be better, please write to **contact@drharshmaheshwari.com**. Put the book number, the version and the section in the subject line (for example: *Book 6, v1.0, section C5*). Say what the text says, what you think it should say, and, if you can, the source.

Corrections are made in new versions. The version number and the date of the last update are on the copyright page of every book, and the newest version of every book is at **drharshmaheshwari.com**.

> **WHERE THIS BOOK SITS**
> Book 6 of the series' 196 books (Book 0 and 195 rung books). Before it: Book 5, Policy process, framing and advocacy craft · Rung 1. After it: Book 7, Question selection and research strategy · Rung 1. The full list is at the back.

Symbols used in this book

Every sign and Greek letter this book prints, how to read it aloud, what it means in this series, and the section where it first appears. Each is also explained where it is first used.

Symbol	Read as	Meaning	First in
SIGNS			
–	minus	Subtraction, or a negative number	1 · When data change silently: three documented failures
×	times	Multiplication	2 · Installing R and an editor; the console and the script
÷	divided by	Division	2 · Installing R and an editor; the console and the script
MARKS ON LETTERS			
x^2, 10^{-3}	superscript	Small raised characters are powers: $x^2 = x \times x$, and $10^{-3} = 0.001$	Before you start

Contents

Before you start 9

Rung 1 — Introductory 10

1 · When data change silently: three documented failures 10
2 · Installing R and an editor; the console and the script 16
3 · Objects and names 22
4 · Vectors, types and NA 28
5 · Functions and arguments 36
6 · Packages: where they come from and which version you have 43
7 · A project folder, and paths that work on someone else's computer 48
8 · Tidy data and the data frame 54
9 · Reading a raw file without changing it 62
10 · The data dictionary and the README 73
11 · Choosing rows and columns 80
12 · Making new columns 89
13 · Missing values: counting them and saying how many 100
14 · Categories as factors 109
15 · Dates 119
16 · Summaries by group 128
17 · Reshaping and joining 140
18 · Plotting with the grammar of graphics 152
19 · Checking the data in code 161
20 · Does shared code rerun? What studies found 172
21 · Top to bottom in a fresh session, with one command 177
22 · Worked journey: One dataset, end to end 185
References 198

Appendix · Worked answers 223

Glossary 313

The series: all 196 books 317

Before you start

This book uses the following Book 0 sections. Each is summarised where it is first needed; work through Book 0 itself if a summary is not enough.

- Book 0, A1 · Counting, place value, and what a calculator is actually doing
- Book 0, A3 · Decimals, rounding, and significant figures
- Book 0, A4 · Percentages, and the percentage of a percentage
- Book 0, A6 · Powers, roots and scientific notation
- Book 0, A8 · Orders of magnitude and the back-of-envelope sanity check
- Book 0, B1 · What a unit is, and the SI base and derived units
- Book 0, B2 · Converting units as multiplication by one
- Book 0, B6 · Body-size units: kg, m, cm and kg/m^2
- Book 0, C1 · Variables and algebra: what a letter stands for
- Book 0, C4 · Functions: input, rule, output
- Book 0, C5 · Graphs: axes, scale, slope and intercept
- Book 0, D3 · Conditional probability through the two-way table
- Book 0, D4 · Variation: what differs and by how much
- Book 0, D5 · Average and spread, and why the average is not the person
- Book 0, F1 · Reading a table
- Book 0, F2 · Reading a figure, and the ways a figure misleads
- Book 0, F3 · Checking a reference

Rung 1 — Introductory

1 · When data change silently: three documented failures

DEFINITION. A silent change is a change to data, or a choice about which data to use, that leaves no instruction a second person can read and run again, and that nothing on the screen announces.

Gene symbols turned into dates. Ziemann, Eren and El-Osta reported in 2016 that Microsoft Excel, used with its default settings, converted some gene symbols to dates: SEPT2 became "2-Sep" and MARCH1 became "1-Mar".

A spreadsheet formula and undeclared choices. One formula in Reinhart and Rogoff's working spreadsheet averaged rows 30 to 44 instead of rows 30 to 49, which left out five countries entirely.

Their papers' highest category was public debt above 90% of gross domestic product (GDP), a measure of the size of an economy.

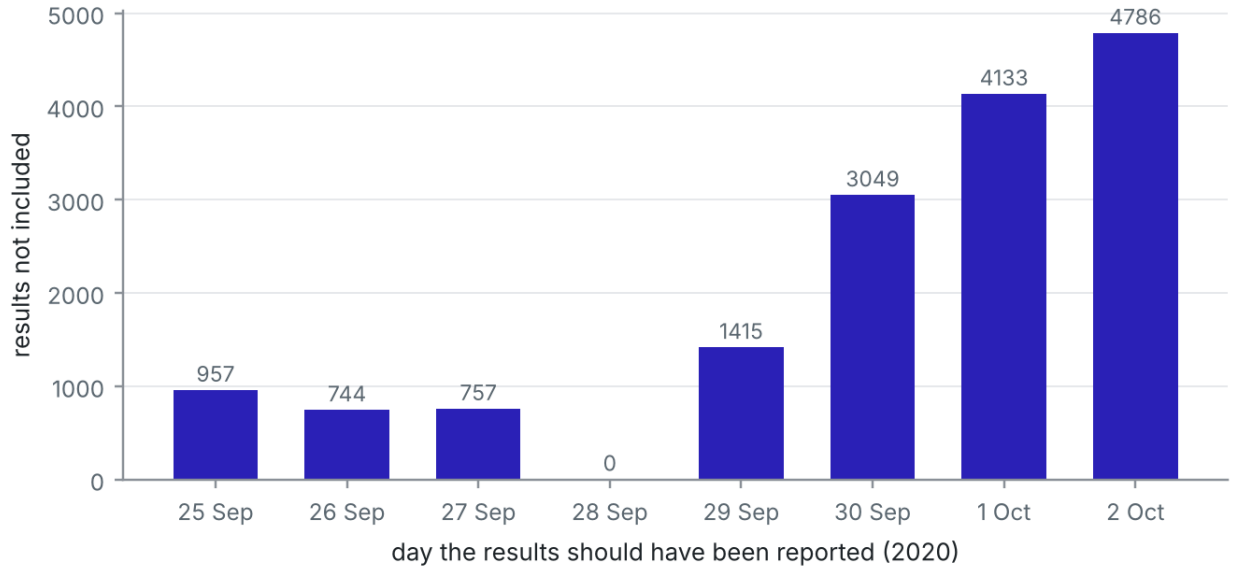
Rows lost in transfer. Public Health England (PHE) stated on 4 October 2020 that 15,841 positive test results for coronavirus disease 2019 (COVID-19), from 25 September to 2 October, had been left out of the reported daily figures. It gave the cause as files that "exceeded the maximum file size" of the process that loads them into central systems. It does not name the software.

The three cases differ in cause. What they share is that the change was silent: nothing announced it, and it came to light only later. [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38]

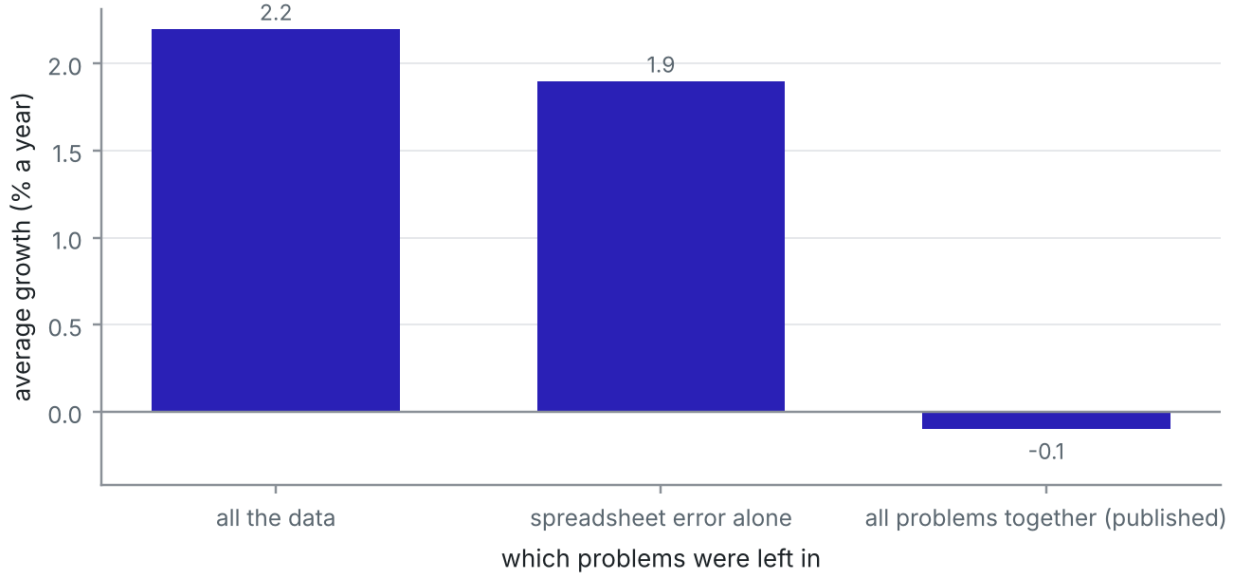
IN PLAIN TERMS. Here is the trap with data. A number can change, or a row can go missing, and nothing on the screen tells you. The table looks right, so you trust it.

- **A program changed values as the file was opened.** About one in five papers with Excel gene lists, in the 18 genomics journals the authors screened, carried such errors.
- **A formula covered the wrong rows, and choices went unsaid.** Other data had been left out on purpose, and the paper never said so.
- **A transfer lost rows.** Some files were too big for the system that loaded them.

So this book follows one rule. You never edit the raw data file. Every change you make to the data is a line of code, kept in a file, that anyone can read and run again. And you count your rows before and after each step. You see what a line of code and a comment look like in section 2.



Positive COVID-19 test results left out of England's daily figures, by the day each should have been reported, as Public Health England's statement of 4 October 2020 gives them: 957, 744, 757, 0, 1415, 3049, 4133 and 4786, a total of 15,841. The last three days hold 11,968 of them.



Average growth in the high-debt category, in % a year, three ways. With all the data it was 2.2%. The spreadsheet error alone moved it only to 1.9%, 0.3 of a percentage point. All the problems together, the exclusions, the weighting, the spreadsheet error and a copying slip, gave the published -0.1%.

Illustration 1. Start with the gene names, because the failure is easy to picture.

A gene symbol is a short code for one gene: SEPT2, MARCH1. Ziemann, Eren and El-Osta report that Excel, with its default settings, turns these into dates. SEPT2 becomes "2-Sep". MARCH1 becomes "1-Mar". Nobody typed a date. The program decided the text looked like one.

The authors then went looking for how often this reached published papers. They wrote programs to screen 35,175 supplementary Excel files from 18 journals, published from 2005 to 2015. Supplementary files are the extra files published alongside a paper. They found gene lists attached to 3597 papers. They confirmed errors in 704 of those papers. Those 704 papers held 987 files with errors.

Now check their headline figure yourself, as Book 0's section on percentages taught you. Ask first: per cent of what? Of the papers that had gene lists in Excel files. Not of all papers in those journals.

```
704 divided by 3597 = 0.1957…
0.1957 times 100 = 19.57
```

That rounds to 19.6%, the figure they state. It is about one paper in five.

Here is what made the error so hard to undo. A date carries no trace of the name it replaced. Of the 987 files with errors, 166 held nothing else, such as a gene's position, from which the original names could be worked out. Those names are lost from the file.

The world adjusted around the problem. In 2020 the committee that names human genes reported that it had renamed every symbol Excel turned into a date. SEPT1 is now SEPTIN1, and MARCH1 is now MARCHF1.

Figures quoted in this illustration are from [39].

WHERE THIS PICTURE BREAKS. The 19.6% describes papers with Excel gene lists in 18 journals from 2005 to 2015. It is not a rate for all papers, all journals or later years, and the authors could not screen supplementary files behind a paywall.

The finding is about Excel's behaviour in 2016, as the authors report it. Spreadsheet programs change between versions, and this course has not tested a current one. What carries over is the pattern: a program can change a value as it reads a file, and say nothing.

Illustration 2. Now a case where the error sat inside a formula, and the larger problems were choices nobody wrote down.

In 2010 Reinhart and Rogoff published figures on public debt and economic growth in 20 rich countries. Herndon, Ash and Pollin tried to rebuild them. From the data the authors had posted, they could not. The posted spreadsheets did not record which data series, years and methods the papers had used. Only when Reinhart and Rogoff sent their working spreadsheet could the replication begin.

Follow what the working spreadsheet held. Count the years in which a country carried public debt above 90% of its GDP, the highest category the papers used.

Step	Years counted
All the data	110
After the exclusions Reinhart and Rogoff chose	96
After those exclusions and the spreadsheet error	71

The spreadsheet error was one formula. It averaged rows 30 to 44 when the data ran to row 49. Five countries, Australia, Austria, Belgium, Canada and Denmark, fell out of every average. The authors note that the countries left out run in alphabetical order, a block of rows.

The exclusions were choices. Some early post-war years for Australia, Canada and New Zealand were left out, and the papers never said so.

The weighting was a third choice. Each country's average counted once, whether it rested on one year or nineteen.

Then read what each problem did to the answer. With all the data, average growth in the high-debt category was 2.2% a year. The published figure was −0.1%. Herndon and colleagues also put back each problem on its own. The spreadsheet error by itself moved the figure only from 2.2% to 1.9%. So did the exclusions alone, and so did the weighting alone. The three together, which the authors say interact strongly, brought it to 0.0%. A fourth slip, a figure copied wrongly, took it to the published −0.1%.

Reinhart and Rogoff acknowledged the spreadsheet errors. In their 2013 working paper, Herndon and colleagues drew a lesson for people who analyse economic data: the advantages of reproducible code over working spreadsheets.

Figures quoted in this illustration are from [40].

WHERE THIS PICTURE BREAKS. It is tempting to tell this as "a spreadsheet slip changed world policy". The replication does not say that. The slip alone moved the figure by 0.3 of a percentage point. The undeclared exclusions and the way countries were weighted mattered as much, and the authors stress that the problems work together.

Code would not have stopped anyone choosing to leave data out. What a script does is show the choice, as a line someone can read and question.

Illustration 3. The third case is a transfer that dropped rows, and the numbers let you check the statement against itself.

On 4 October 2020, PHE said that 15,841 positive COVID-19 test results had not reached the daily figures. The problem had been found overnight on Friday 2 October. The cause it gave: some files of results were bigger than the maximum file size the loading process could take. It put in a fix that splits large files.

The statement gives the missing results day by day. Each was recorded on one day and should have appeared in the next day's published figure.

Day it should have been reported	Results not included
25 Sep	957
26 Sep	744
27 Sep	757
28 Sep	0
29 Sep	1415
30 Sep	3049

Day it should have been reported	Results not included
1 Oct	4133
2 Oct	4786

Add the column. Then add the last three days alone.

```
957 + 744 + 757 + 0 + 1415 + 3049 + 4133 + 4786 = 15841
3049 + 4133 + 4786 = 11968
11968 divided by 15841 = 0.7555…
```

The total matches the 15,841 in the statement. The last three days hold 11,968 results, and 11,968 out of 15,841 is about 75.6%. The statement says "over 75% (11,968)". It checks.

Now look at the shape of the column. The missing results were fewer than a thousand a day at first, then rose over the last four days to 4786. A loss that grows while the daily table still looks complete is one nobody sees until they compare one count with another.

Here is what the statement does not say. It does not name the software, a file format or a row limit. You may read elsewhere that this was a spreadsheet problem. This statement does not say so, and neither does this book.

Figures quoted in this illustration are from [41].

WHERE THIS PICTURE BREAKS. PHE says the problem did not change anyone's test result: everyone who tested positive got their result as normal and was advised to self-isolate. What was delayed was the count in the daily figures. The outstanding cases, the statement says, were passed to contact tracing by 1am on 3 October.

The table gives the day each result should have been reported. The statement also describes the same results by the day they were recorded, one day earlier: 24 September to 1 October. Say which you mean when you quote the dates.

MUST KNOW POINTS FOR YOU.

- The error is to believe a table because it looks complete. Before you trust a table, compare its row count, or a total, with the count you expected.
- Never edit the raw data file, and never save over it. Make every change in code that reads the raw file and writes a new one.
- A choice about which data to leave out is part of the analysis. Write it as a line of code with a comment saying why.
- Do not repeat "a spreadsheet slip changed government policy" or "Excel lost the COVID cases". The replication found the slip alone moved the result from 2.2% to 1.9% a year. The PHE statement names no software. Quote what each source says, and no more.
- When you open someone else's data file, look for values that have changed type, such as text turned into dates.

- When a trainee says their numbers are right because they checked them on screen, ask how a second person would check them. If the answer is "by redoing it by hand", the work has no record.
- Writing every step in code does not make the steps right. What it changes is that the step is written down, so a reviewer can find the error and you can fix it once and rerun.

EXERCISE 1 (retrieval). From memory, name the three documented failures in this section. For each, say in one sentence what changed or went missing, and how it came to light.

EXERCISE 2 (critique). In a made-up department meeting, a senior colleague says: "Our thesis data are in one Excel file. Residents correct mistakes directly in the cells, so the file is always the cleanest version. That is good practice."

Using the three cases in this section, say what is lost when corrections are made directly in the file, and what you would propose instead.

EXERCISE 3 (interpretation). A classmate's draft introduction says: "Excel deleted 15,841 COVID-19 cases in England in 2020 because of its row limit." Check this sentence against what PHE's statement says, and rewrite it so that every part of it is supported.

EXERCISE 4 (teaching). A first-year resident has ten minutes and a whiteboard. They say: "My data are small, a few hundred rows. I can see every cell. Why should I bother with code?" Teach them, using one of the three cases.

2 · Installing R and an editor; the console and the script

DEFINITION. R is a program that reads instructions written in the R language and carries them out. R is installed first, then RStudio.

The console is the pane in which R shows a prompt, `>`, and waits for one instruction. If the instruction is incomplete, R shows `+` instead of `>` and waits for the rest.

A script is a plain-text file of R code, saved with a name ending in `.R`.

Everything after a `#` on a line is a comment. R ignores it; people read it.

An error message is R's statement that it could not carry out an instruction. It says what it could not do and, for a line it could not read, where on the line it stopped. [42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60]

IN PLAIN TERMS. RStudio is the window you work in. Install R first: download it from CRAN, the Comprehensive R Archive Network (cloud.r-project.org). Then download RStudio from Posit's website (posit.co/download/rstudio-desktop).

Open RStudio and you see several panes. The console is the one showing the prompt `>`.

The console is where R waits for you. Type `2 + 3` after it and press Enter. R prints `[1] 5`. The `[1]` only marks the first value on the line; ignore it for now. The answer is printed once and kept nowhere.

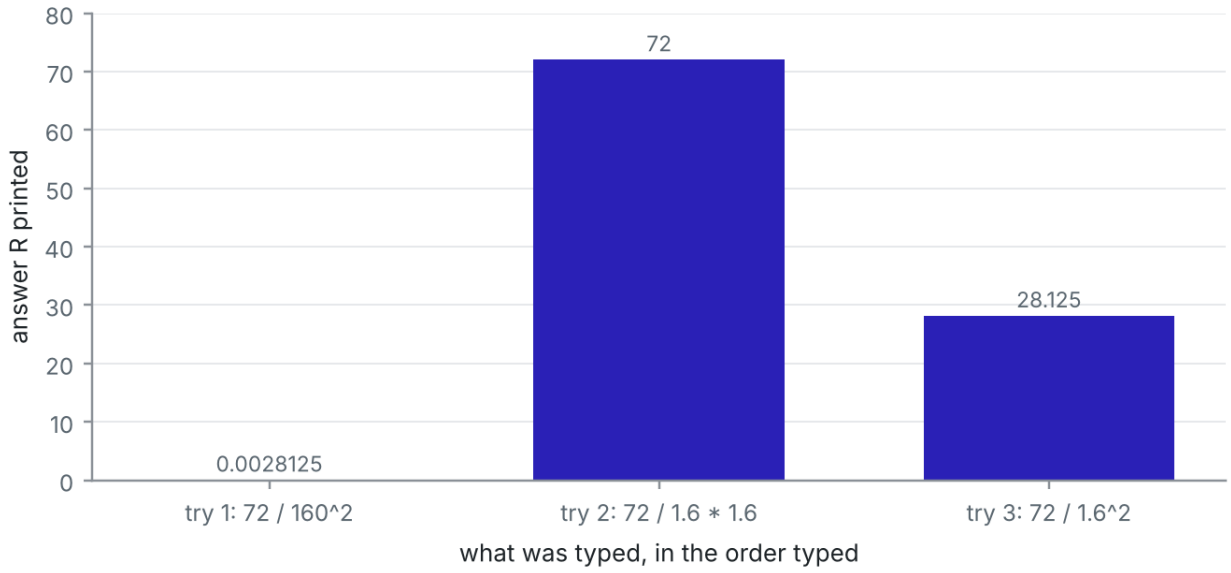
The script is a plain text file of instructions, one per line, saved with a name ending in `.R`. Open one from the File menu: New File, then R script. Put the cursor on a line and press Ctrl and Enter together. RStudio sends that line to the console, and the answer appears there. Select every line with the mouse and press Ctrl and Enter (Cmd and Enter on a Mac), and the whole file runs from the top.

The console is a place to try things. The script is what you keep.

Times is `*`. Divided by is `/`. The signs × and ÷ that you may have met at school do not work in R. Powers are `^`, so 1.6 squared is `1.6^2`. The order of working is the one you already know: brackets, powers, then times and divide, then plus and minus.

Use comments to say why a line is there.

When R cannot do what you asked, it prints an error message. Read it word by word.



The three answers R printed for the same made-up patient's body mass index (BMI). Height typed in centimetres, `72 / 160^2`, gives 0.0028125, a factor of 10,000 too small and too short to see as a bar. The power left out, `72 / 1.6 * 1.6`, gives back the weight, 72. Only `72 / 1.6^2` gives 28.125 kg/m^2, in the tens, as the rounded check expects.

Illustration 1. Start in the console, and make the mistakes on purpose. A made-up patient in a clinic register weighs 72 kg and is 160 cm tall. Book 0 gave you body mass index (BMI): weight in kilograms divided by height in metres, squared, in kg/m^2. Type the first thing that comes to mind.

```
72 / 160^2
```

```
[1] 0.0028125
```

R does exactly what you typed. Now do Book 0's order-of-magnitude check: round each input to one figure. 70 divided by 2 squared, which is 4, is about 20, so the BMI should be in the tens. This answer is thousands of times too small. The height went in as centimetres. 160 cm is 1.6 m, because a metre is 100 cm.

Try again, and this time leave the squaring out by accident:

```
72 / 1.6 * 1.6
```

```
[1] 72
```

The answer is the weight you started with. R worked left to right: 72 divided by 1.6, then that times 1.6. Multiplying by 1.6 twice was never what you meant. You meant 1.6 squared, and that needs a power or a bracket.

```
72 / 1.6^2
72/(1.6*1.6)
```

```
[1] 28.125
[1] 28.125
```

Both lines give 28.125, so the patient's BMI is about 28.1 kg/m^2. The power is worked out before the division, just as Book 0's order of operations says. The second line has no spaces, and R gives the same answer. This book puts spaces round the signs so a line is easy to read.

One more slip. You may have learnt the sign ÷ at school. Type it here:

```
72 ÷ 1.6^2
```

```
Error: <text>:1:4: unexpected input
1: 72 ÷
       ^
```

Read the message literally. "Unexpected input" means R met a character it cannot read. The ^ under the line points at the ÷ sign, which is not part of the R language. The `1:4` says where: line 1, fourth character. The fix is `/`.

Now scroll up through the console. Five tries sit there, three of them wrong, in the order you typed them. Nothing marks which ones were right. If you came back next week, you would have to work it out again.

WHERE THIS PICTURE BREAKS. BMI is used here only because Book 0 taught it and the numbers are easy to check. The patient is made up, and 28.1 kg/m^2 says nothing about anyone real.

The order-of-magnitude check caught the first slip because a factor of 10,000 is huge. It would not catch a weight typed as 27 instead of 72. Only checking the inputs catches that.

Illustration 2. Now put the right version where it will last. Open a new script from the File menu. Type these lines into it, and save it as `bmi_arithmetic.R`. A name with no spaces, all in small letters, and saying what the file does, is easy to find and easy to type later.

```
# BMI of a made-up clinic patient, from the register's units
# Weight is recorded in kg and height in cm.

# Height in metres: the register gives cm, and BMI needs m
160 / 100

# BMI in kg/m^2: weight divided by height in metres squared
72 / (160 / 100)^2
```

```
[1] 1.6
[1] 28.125
```

Select all the lines and press Ctrl and Enter. In the console each line appears above its answer; the book prints the answers only. The comments are skipped. The two calculations run, from the top, and their answers appear in the console.

Read the second calculation as R reads it. The bracket `(160 / 100)` is worked out first, giving 1.6. Then the power: 1.6 squared. Then 72 is divided by that.

Notice what the script holds that the console did not. It keeps only the right version. Its comments say why each line is there. The conversion from centimetres to metres is a line you can see, not a step you did in your head. Close RStudio, open the file tomorrow, and run it. You get 28.125 again.

This is the first file of the habit the book builds. Later sections add a folder, data files and a whole analysis. The habit does not change: what you want to keep goes in the script.

WHERE THIS PICTURE BREAKS. A script that runs is not a script that is right. If the register said 61 kg and you typed 72, the script would repeat the wrong answer every time, perfectly. Running from a file makes a mistake repeatable and visible. It does not remove it.

These two lines use typed-in numbers. Real work reads the numbers from a data file, and that comes later in this book.

MUST KNOW POINTS FOR YOU.

- The error is to think the console keeps your work. Put anything you want to keep into the script.
- Typing × or ÷ stops R with an error.
- `72 / 1.6 * 1.6` gives 72, not a BMI. R divides and multiplies from left to right. When you mean "divided by a product", put the product in brackets or use a power.
- Write a unit conversion as a line of code, with a comment, rather than doing it in your head and typing the result.
- When an answer comes out, check its size against what you expect before you use it. A BMI of 0.003 kg/m^2, or a BMI equal to the weight you typed, means the code is wrong, however cleanly it ran.
- When a trainee shows you a number from R, ask to see the script, not the console.

EXERCISE 1 (retrieval). Answer from memory, without R open. What does the console show when it is waiting for an instruction? What does it show when your line is unfinished? Where should a line you want to keep be written, and why?

EXERCISE 2 (build). Start the habit. In RStudio, open a new script. Save it with a name of your choosing that has no spaces and says what the file is for. In it, write lines that work out, for a made-up adult who weighs 58 kg and is 152 cm tall:

1. the height in metres;
2. the BMI in kg/m^2.

Give each line a comment saying why it is there. Run the whole script from the top and write down the two answers.

EXERCISE 3 (teaching). A first-year resident has ten minutes and a whiteboard. They have been doing their thesis calculations in the console "because it is faster". Teach them the difference between the console and a script, and why it matters for a thesis.

PRACTICE. Ten problems on this technique, easiest first. Work them on paper before turning to the answers, which are in the appendix at the back under this section's name.

1. Write one line of R that works out 45 plus 17 times 2. Before you run it, write down what you expect it to print.

2. Write one line of R that divides 150 by the sum of 20 and 30.

3. Predict what each line prints, then run them to check.

```
2^3 * 2
2^(3 * 2)
```

4. This book uses a file of 344 rows of penguin measurements from near Palmer Station, Antarctica. The first penguin in the file weighed 3750 g. Write one line of R that gives its weight in kilograms, with a comment saying what the line does.

5. A made-up adult in a clinic register weighs 81 kg and is 172 cm tall. Write a short script, with a comment on each line of code, that works out the height in metres and then the BMI in kg/m^2. Give the BMI to one decimal place in your answer.

6. The first three penguins in the Palmer file have flipper lengths of 181, 186 and 195 mm. Write a line of R that gives their average flipper length in centimetres, with a comment.

7. A made-up patient's weight was measured twice at one visit, 71.4 kg and 71.8 kg. A resident wants the average of the two and writes:

```
71.4 + 71.8 / 2
```

```
[1] 107.3
```

They record 107.3 kg in the register. Find what broke, and say why the wrong answer can look reasonable.

8. A resident saves this script and runs every line. R stops with an error. Find the cause.

```
# Height in metres
165 / 100
BMI of patient 2
68 / (165 / 100)^2
```

```
Error: <text>:3:5: unexpected symbol
2: 165 / 100
3: BMI of
       ^
```

9. A line from a made-up thesis results chapter reads: "The mean weight of the three participants in the pilot was 68.3 kg." The three weights in the pilot register are 64.2, 71.5 and 69.1 kg. Check the sentence in R. Then say what your check does not establish.

10. In a made-up lab meeting, someone says this: “The first three penguins in the Palmer file weighed 3750, 3800 and 3250 g. So the average penguin weighs 3.6 kg.” Turn the claim into R, run it, and say what the result does not establish.

3 · Objects and names

DEFINITION. An object is a value R holds in memory under a name. The assignment `name <- value` makes one: R works out the right-hand side completely, then stores the result under the name on the left. An assignment prints nothing. Typing the name alone prints the value.

Assigning to a name that already exists replaces its value.

Names are case sensitive: `bmi` and `BMI` are different names. A name may contain letters, digits, `.` and `_`, and no spaces. It must start with a letter, or with `.` not followed by a digit.

The objects that exist at a given moment form the workspace, which `ls()` lists and RStudio shows in its Environment pane.

R accepts `=` in most places; this book uses `<-`. [61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75]

IN PLAIN TERMS.

```
weight_kg <- 72
```

Read the arrow `<-` as "gets": weight_kg gets 72. R prints nothing. Type the name, and R prints the value back.

```
weight_kg
```

```
[1] 72
```

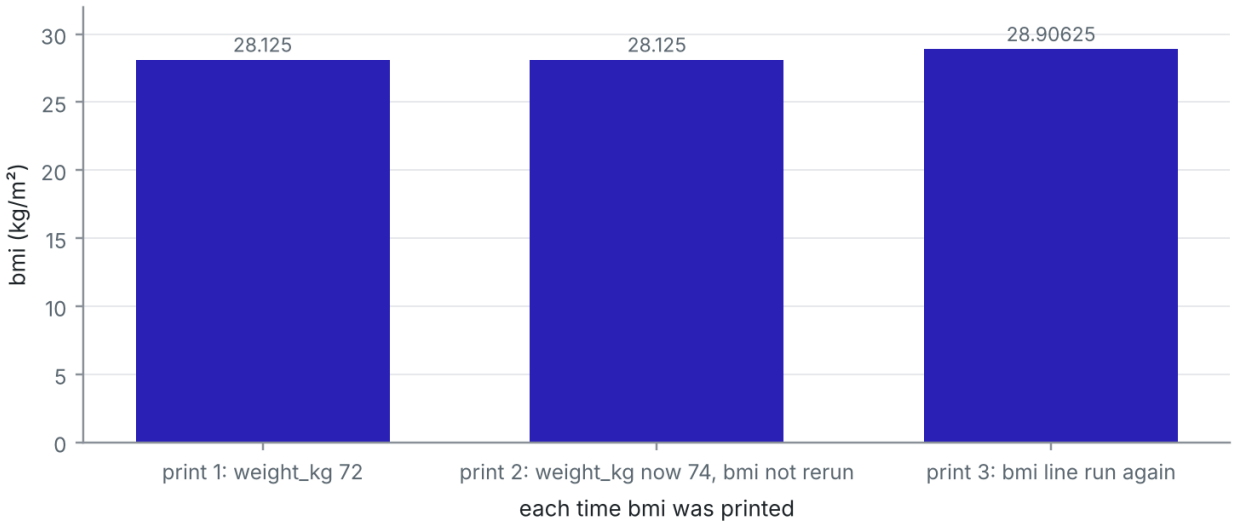
A name is a label stuck on a value.

- **Give the same name a new value, and the old one is gone.**
- **A name holds a value, not a sum.** It changes only when you run its line again.

The objects you have made so far sit in a list called the workspace. RStudio shows it in the Environment pane, at the upper right, and `ls()` prints it. That list is not your script. It holds every object made by whatever you ran today, in whatever order you ran it, including lines you typed only in the console.

Write names in small letters, with _ between words, and put the unit in the name: `height_cm`, `height_m`, `bmi`. The style guide this book follows, written for the tidyverse (a family of packages you meet in section 6), calls this snake case. It also asks for `<-`, not `=`, for assignment.

In Book 0 a variable was a letter standing for a number. In R, people also call a named object a variable. A third meaning, a column in a table of data, comes later in the book.



The value of bmi each time it was printed. First, with weight_kg 72, it is 28.125. After weight_kg is changed to 74, bmi still prints 28.125, because R stored the answer and not the recipe. Only when the line `bmi <- weight_kg / height_m^2` is run again does it become 28.90625, which is 74 divided by 1.6 squared.

Illustration 1. Start with a correction that does not arrive. Take the made-up patient from the last section, and give each number a name.

```
weight_kg <- 72
height_m <- 1.6
bmi <- weight_kg / height_m^2
bmi
```

```
[1] 28.125
```

Now the clinic tells you the weight was copied wrongly into the register. The form says 74 kg. You fix the weight, and print the BMI again.

```
weight_kg <- 74
bmi
```

```
[1] 28.125
```

The BMI has not moved. When R ran `bmi <- weight_kg / height_m^2`, it worked out the right-hand side, 72 divided by 1.6 squared, and stored the answer, 28.125. It did not store the recipe. Changing `weight_kg` afterwards changes `weight_kg` and nothing else.

Run the BMI line again, and the new weight is used.

```
bmi <- weight_kg / height_m^2
bmi
```

```
[1] 28.90625
```

Now look at what exists.

```
ls()
```

```
[1] "bmi"       "height_m"  "weight_kg"
```

Three objects, listed in alphabetical order. The list says nothing about the order the lines ran in, or about the weight that was 72 a minute ago. Suppose you had typed `weight_kg <- 74` in the console and never put it in the script. The workspace would still say 74, and you would believe it. Tomorrow, running the script from the top, you would get 28.125 again.

So the fix belongs in the script. Change the first line to 74, with a comment saying why. Then run every line from the top, so that each object is made from the ones above it.

WHERE THIS PICTURE BREAKS. Book 0's algebra letter is the wrong picture here. In Book 0's I = m divided by h^2, m stands for whatever number you substitute, and the formula holds for all of them at once. In R, `weight_kg` holds one value, the last one assigned, and `bmi` holds one number worked out once.

The patient and the copying error are made up. The correction itself is real practice: a correction is a line in the script, with a comment, never a value typed over in the console.

Illustration 2. Now break the naming rules on purpose, and read what R says each time.

A capital letter where there was none:

```
weight_kg <- 74
Weight_kg
```

```
Error: object 'Weight_kg' not found
```

R looked for an object called `Weight_kg` and found none. To R, the capital W makes it a different name.

A name that starts with a digit:

```
2nd_visit_kg <- 73
```

```
Error: <text>:1:2: unexpected symbol
1: 2nd_visit_kg
     ^
```

"Unexpected symbol" means R read the first part, `2`, as one thing, then met a second part it could not join to it.

Now name it so the rules and the reader are both satisfied:

```
weight_visit2_kg <- 73
weight_visit2_kg
```

```
[1] 73
```

Small letters, words joined by _, the visit and the unit in the name. Anyone reading `weight_visit2_kg` knows what it holds and in what unit, without a comment.

WHERE THIS PICTURE BREAKS. The rules R enforces and the style this book asks for are different things. R accepts `WeightVisit2` and `w2`, and both would run. The style guide's snake case is a choice the tidyverse made for consistency, and this book follows it so that every name reads the same way.

MUST KNOW POINTS FOR YOU.

- The error is to think a name holds a formula. After any correction, run the script from the top.
- The workspace is not your script. Trust a result only after it has come from running the script from the top.
- Give each thing its own name, such as `bmi_visit1` and `bmi_visit2`, rather than reusing `bmi`.
- "object 'x' not found" usually means a typing slip or a wrong capital letter, or a line that makes `x` was never run.
- A good name makes code readable, not correct. Names help a reviewer follow the steps; only checking the values tells you they are right.

EXERCISE 1 (retrieval). From memory: you run `bmi <- weight_kg / height_m^2`, then `weight_kg <- 74`. What does `bmi` hold now, and what must you do to bring it up to date? Then name two things an R name may not do.

EXERCISE 2 (design). A classmate's script reads:

```
W <- 66
H <- 158
x <- W / (H / 100)^2
x
```

```
[1] 26.43807
```

Rewrite it so that every name follows this book's style and says what it holds, with its unit. Add a comment for the step that would puzzle a reader. Check that your version gives the same answer.

EXERCISE 3 (teaching). A first-year resident has ten minutes and a whiteboard. They corrected a participant's height in the console and say their BMI results are now fixed. Teach them why they may not be.

PRACTICE. Ten problems on this technique, easiest first. Work them on paper before turning to the answers, which are in the appendix at the back under this section's name.

1. What does the last line print?

```
a <- 5
a <- a + 2
a
```

2. Which of these names will R accept for an object? Try each one with `<- 1` to check.

`bmi_2024`, `2024_bmi`, `bmi.2024`, `bmi 2024`, `.bmi`, `_bmi`

3. Predict what the last line prints, then run the lines to check.

```
x <- 10
y <- x * 2
x <- 3
y
```

4. The first penguin in the Palmer file weighed 3750 g. Store that value in an object whose name gives the unit. Then make a second object holding the weight in kilograms, and print it.

5. A made-up adult weighs 81 kg and is 172 cm tall. Write lines that store the weight and the height as they were recorded, make the height in metres, make the BMI, and print it. Use names that carry units.

6. The second and third penguins in the Palmer file weighed 3800 g and 3250 g. Store each weight in its own object, make an object holding the difference between them in kilograms, and print it.

7. A resident works out BMI for two made-up patients and reports patient A's value. Here is the script and what it printed.

```
bmi <- 72 / 1.6^2       # patient A
bmi <- 58 / 1.52^2      # patient B
bmi                     # patient A's BMI, for the report
```

```
[1] 25.10388
```

They write "Patient A: BMI 25.1 kg/m^2". Find what broke, and say why the wrong answer looks reasonable.

8. This script stops with an error. Find the cause.

```
weight_kg <- 60
height_m <- 1.62
weight_kg / Height_m^2
```

```
Error: object 'Height_m' not found
```

9. A line in a made-up thesis reads as follows. "The weight of participant 12 was corrected from 47 kg to 74 kg, a transposition error. The BMI was recalculated." The participant's height is 1.58 m. Write the lines that record this correction and give both BMIs. Then say what your lines do not establish.

10. A made-up thesis methods section says: "Height was recorded in centimetres and converted to metres before BMI was calculated." Participant 2's register entry reads: weight 64, height 1.63.

Write the lines the sentence describes for participant 2, with names that carry units. Run them. Then say what the methods sentence does not establish.

4 · Vectors, types and NA

DEFINITION. A vector is an ordered set of values that R holds as one object under one name. The function `c()` combines values into a vector, in the order given.

Every element of a vector has the same type. Numeric values are numbers (R keeps two kinds, integer and double, and treats both as numbers).

When values of different types are combined, R converts all of them to the highest type in the order logical, then numeric, then character. This conversion is called coercion. A single text value therefore turns a vector of numbers into a vector of text, and functions that compute on numbers, such as `mean()`, return no number for it. Logical values turned into numbers become 1 for `TRUE` and 0 for `FALSE`.

`NA`, for "not available", marks a value that is missing. Almost any arithmetic or comparison that involves `NA` returns `NA`, because the result cannot be known. The function `is.na()` returns `TRUE` for each element that is `NA`. The comparison `x == NA` returns `NA` for every element and finds nothing. [76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89]

IN PLAIN TERMS. In R, a set of values in a row, kept in order under one name, is a vector. You build one with `c()`, which stands for combine:

```
height_cm <- c(152, 160.5, 171)
height_cm
```

```
[1] 152.0 160.5 171.0
```

R prints every number in a vector with the same number of decimal places, so 152 shows as 152.0.

A vector holds one type of value. The three you need now are numbers (numeric), text (character) and `TRUE` or `FALSE` (logical).

```
sex <- c("female", "male", "female")
sex
```

```
[1] "female" "male"   "female"
```

Arithmetic works on the whole vector at once, element by element. Divide the heights by 100 and every height turns into metres:

```
height_cm / 100
```

```
[1] 1.520 1.605 1.710
```

R repeats a single number as often as the longer vector needs.

A comparison asks the same question of every element and answers `TRUE` or `FALSE` for each:

```
height_cm > 155
```

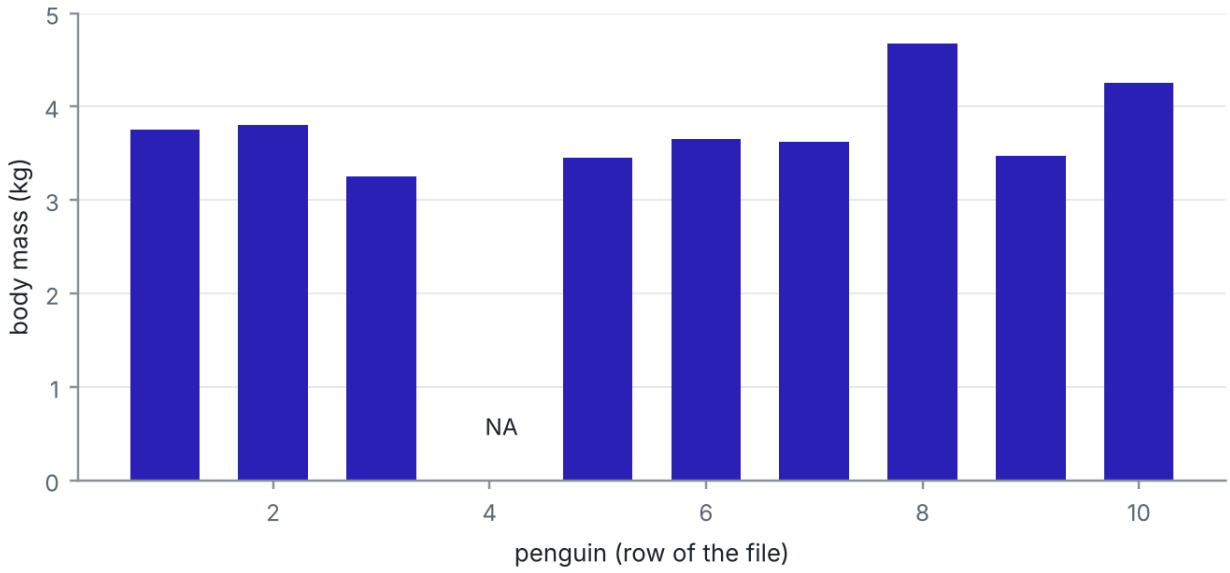
```
[1] FALSE  TRUE  TRUE
```

`>` means greater than. The others are `<`, `>=` (greater than or equal to), `<=`, `==` (equal to) and `!=` (not equal to).

Square brackets pick elements out. A number inside them is a position. A logical vector inside them keeps the elements where it says `TRUE`. `1:2` means positions 1 and 2:

```
height_cm[2]
height_cm[1:2]
height_cm[height_cm > 155]
```

```
[1] 160.5
[1] 152.0 160.5
[1] 160.5 171.0
```



The first ten body masses in the practice file, in kg, as R holds them after dividing by 1000. Penguin 4 has no bar: its value is NA, which is not zero and not small but unknown. Penguins 8 and 10 are the two above 4 kg.

Illustration 1. Start with a failure. A made-up clinic register lists four adults' weights. One entry was written as "71 kg", unit and all. You type the four entries as written:

```
weight_kg <- c(62.5, 70.1, "71 kg", 58)
weight_kg
```

```
[1] "62.5"  "70.1"  "71 kg" "58"
```

Look at the printout before anything else. Every value now sits in quotation marks, even the three that were plain numbers. One text value made R convert the whole vector to text. That

conversion is called coercion. R did it without a warning, because combining a number with text is allowed.

Ask R which type it holds:

```
typeof(weight_kg)
```

```
[1] "character"
```

Now ask for the average weight:

```
mean(weight_kg)
```

```
[1] NA
Warning message:
In mean.default(weight_kg) :
  argument is not numeric or logical: returning NA
```

You get `NA` and a warning, not a number. A warning means R carried on and gave a result you should distrust. `mean.default` is the version of `mean()` that handles plain numbers. Try arithmetic instead, such as converting to grams:

```
weight_kg * 1000
```

```
Error in weight_kg * 1000 : non-numeric argument to binary operator
```

That one stops with an error: R gave no result at all. Read it literally: “non-numeric argument” means one of the things being multiplied is not a number. It is the vector of text.

You can ask R to turn the text back into numbers with `as.numeric()`. Watch what it does with the entry it cannot read:

```
as.numeric(weight_kg)
```

```
[1] 62.5 70.1   NA 58.0
Warning message:
NAs introduced by coercion
```

Three numbers come back, and “71 kg” becomes `NA`, with a warning that says so. The weight of 71 kg is now lost from the vector. R did not guess what “71 kg” meant, and it should not. The person who knows it means 71 has to say so in code, where the decision can be seen:

```
weight_kg <- c(62.5, 70.1, 71, 58)
typeof(weight_kg)
mean(weight_kg)
```

```
[1] "double"
[1] 65.4
```

The four weights now average 65.4 kg.

Here the fix was easy, because you typed the values yourself. When the data arrive in a file, a column in which one cell says “71 kg” or “not weighed” arrives as text in exactly the same way. The section on reading a raw file shows how to catch that when you read it.

WHERE THIS PICTURE BREAKS. The register and its four weights are made up. Real registers carry many kinds of stray text, and each needs its own decision.

Typing a corrected value is right here only because nothing was read from a file. When the values come from a raw file, the correction belongs in a line of code that reads the file and changes the value, so the file stays as it was. Later sections build that habit.

Illustration 2. Now use real values. The book’s practice file, `penguins_raw.csv`, has 344 rows of penguin measurements. Its column “Body Mass (g)” gives each bird’s weight in grams. Here are its first ten values, typed in order. The fourth bird has no weight in the file, which R reads as `NA`:

```
body_mass_g <- c(3750, 3800, 3250, NA, 3450, 3650, 3625, 4675, 3475, 4250)
```

Reading the file itself comes in a later section. The same ten numbers come out.

Convert to kilograms by dividing by 1000. Each value is divided, and the missing one stays missing:

```
body_mass_kg <- body_mass_g / 1000
body_mass_kg
```

```
 [1] 3.750 3.800 3.250    NA 3.450 3.650 3.625 4.675 3.475 4.250
```

The fourth value is still `NA`, because R has no weight for that bird.

Ask which birds weigh more than 4000 g:

```
body_mass_g > 4000
```

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

Penguins 8 and 10 do. Penguin 4 gets `NA`, not `FALSE`. R does not know its weight, so it does not know the answer. An `NA` in the brackets gives an `NA` in the result:

```
body_mass_g[body_mass_g > 4000]
```

```
[1]   NA 4675 4250
```

Add the ten weights and the same thing happens to the total:

```
sum(body_mass_g)
```

```
[1] NA
```

One missing value made the whole sum unknown. That is the rule working as it should. A sum that quietly left penguin 4 out would look like the total of ten birds and be the total of nine.

Now the trap. To find the missing value, the natural thing to type is "equals NA":

```
body_mass_g == NA
```

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

Every answer is `NA`, so this finds nothing. Asking whether a value equals an unknown value can only give "unknown". The question R can answer is "is this value missing?", and `is.na()` asks it:

```
is.na(body_mass_g)
```

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

Only the fourth is `TRUE`. Put an exclamation mark, `!`, in front to turn each `TRUE` into `FALSE` and each `FALSE` into `TRUE`. Inside square brackets, that keeps the values that are not missing:

```
known_g <- body_mass_g[!is.na(body_mass_g)]
known_g
length(known_g)
```

```
[1] 3750 3800 3250 3450 3650 3625 4675 3475 4250
[1] 9
```

Nine weights are known. `length()` counts the elements of a vector.

Last, a warning about what you see. R prints a number rounded for display. It stores more digits than it prints. Body mass index (BMI) is weight in kg divided by the square of height in metres, in kg/m^2. Work out one, for a made-up adult of 70.1 kg and 1.68 m:

```
bmi <- 70.1 / 1.68^2
bmi
bmi * 1000
```

```
[1] 24.83702
[1] 24837.02
```

The first print shows 24.83702. Multiplying by 1000 shows digits that were there all along and were not printed. Your Book 0 rule still holds: the digits on the display are not the precision of the measurement. A weight to one decimal place does not give a BMI to five.

The stored value is not exact either. R keeps most decimal fractions as a close approximation in binary, so a test of exact equality between two decimals can fail:

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

```
[1] FALSE
```

The two sides differ by a tiny amount, far smaller than anything you measure. So never test decimals for exact equality. Round them to the precision you care about first, which the next section shows how to do.

Figures quoted in this illustration are from [90].

WHERE THIS PICTURE BREAKS. The ten weights are real values from the practice file, but they are the first ten rows only. Nothing here describes the other 334 rows, or penguins in general.

The BMI example uses one made-up adult. The digits it shows describe how R stores and prints a number, not how precise a weighing scale is.

MUST KNOW POINTS FOR YOU.

- The error is to believe a column that looks like numbers holds numbers. Before you compute anything, print the vector and look for quotation marks, or ask `typeof()`.
- `x == NA` never finds a missing value. Use `is.na(x)`, and refuse any count of missing values that was made with `==`.
- Do not make the `NA` go away by writing a 0 in its place. Find out how many values are missing first.
- "NA" typed inside quotation marks is the two letters N and A, not a missing value.
- Never test two decimals with `==`. Report a number to the precision of the measurement behind it, whatever R prints.
- `is.na()` finds only R's own `NA`. Suppose a raw file writes a missing weight as -99 or 999. To R, that is a weight of -99 or 999, and it goes into every mean until you tell R what the code means.
- When a trainee says R gave a wrong mean, ask them to print the vector and point to the quotation marks or the `NA`. The habit of looking is the thing worth teaching.

EXERCISE 1 (critique). A made-up thesis appendix shows this code and says "the mean haemoglobin of the five women could not be calculated because R gave an error".

```
hb_g_dl <- c(11.2, 12.5, "9.8*", 13.1, 10.4)
mean(hb_g_dl)
```

```
[1] NA
Warning message:
In mean.default(hb_g_dl) : argument is not numeric or logical: returning NA
```

Haemoglobin is measured in grams per decilitre (g/dL). Say what actually happened, what the asterisk probably was, and how the analysis should go on without editing anything by hand.

EXERCISE 2 (retrieval). Without looking back, name the three types of vector this section uses. Say which one wins when they are mixed in `c()`. Then give the line that finds the missing values in a vector `x`.

PRACTICE. Eleven problems on this technique, easiest first. Work them on paper before turning to the answers, which are in the appendix at the back under this section's name.

1. Make a vector called `height_cm` holding the heights 98, 102.5 and 110 cm. Then print the same heights in metres.

2. Before running them, say what type of vector each line makes and what it prints. Then run them and check.

```
c(1, TRUE, "no")
c(0, TRUE, FALSE)
```

3. Before running them, say what each line prints.

```
c(2, NA, 5) + 1
c(2, NA, 5) > 3
```

4. The column "Flipper Length (mm)" in `penguins_raw.csv` gives flipper lengths in millimetres. Its first ten values are 181, 186, 195, a missing value, 193, 190, 181, 195, 193 and 190. Make a vector of them with a name that carries the unit. Print them in centimetres, and say which penguins have a flipper longer than 190 mm.

5. The first ten body masses in `penguins_raw.csv` are 3750, 3800, 3250, a missing value, 3450, 3650, 3625, 4675, 3475 and 4250 g. Print the eighth penguin's body mass in kilograms. Then print the body masses of the first three penguins in grams.

6. Using the same ten body masses, make a vector of only the known body masses, in kilograms, and count how many there are.

7. A colleague wants the tallest child's height from a made-up school register. Their code and its output are below; `max()` gives the largest value. They report "the tallest child is 98 cm". Find what broke, and say why the answer looks reasonable.

```
height_cm <- c("98", "102", "110")
max(height_cm)
```

```
[1] "98"
```

8. A colleague checks a made-up list of six BMI values, in kg/m^2, for missing entries. Their code and output are below. They write "no BMI values are missing". Find what broke.

```
bmi <- c(22.4, 31.0, NA, 27.9, NA, 19.6)
bmi == NA
```

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

9. A colleague marks which of five made-up adults have a BMI of 25 kg/m^2 or more. One BMI was not recorded. Their code and output are below, and they report "four of the five adults have a BMI of 25 or more". Find what broke, and say why it looks reasonable.

```
bmi <- c(22.1, 27.4, "NA", 30.2, 25.8)
bmi >= 25
```

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

10. A made-up draft report says: “Body mass was recorded for all of the first ten rows of the practice file.” The first ten values of “Body Mass (g)” are as in problem 5. Check the sentence with code. Then say what your check does not tell you.

11. In a made-up meeting, someone says: “In the first ten rows of the file, two body masses are above 4000 g.” The first ten body masses are as in problem 5. Check the claim with code, and then say what the result does not establish.

5 · Functions and arguments

DEFINITION. A function call is a function's name followed by round brackets. The values written inside the brackets are its arguments, and the call returns one result, which R prints or which you assign to a name.

Each argument has a name. An argument may be given by name, as `digits = 1`, or by position, in the order the function lists its arguments.

Most arguments have a default value, which is used when the call does not give one. For `mean()`, `median()` and `sd()`, the argument `na.rm` says whether missing values (`NA`) are removed before the computation. Its default, `FALSE`, makes the result `NA` whenever a value is missing.

A help page, opened with `?name` or `help(name)`, describes one function in fixed sections: Description, Usage, Arguments, Value (what it returns) and Examples. [91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104]

IN PLAIN TERMS.

```
heights_cm <- c(152, 160.5, 171)
mean(heights_cm)
```

```
[1] 161.1667
```

Whatever goes inside the brackets is an argument. Many functions take more than one. `round()` takes the number to round, called `x`, and how many decimal places to keep, called `digits`:

```
round(161.1667, digits = 1)
```

```
[1] 161.2
```

Leave out an argument altogether and R uses its default. The default for `digits` is 0, so `round(161.1667)` gives a whole number.

One call can sit inside another, and R works from the inside out:

```
round(mean(heights_cm), digits = 1)
```

```
[1] 161.2
```

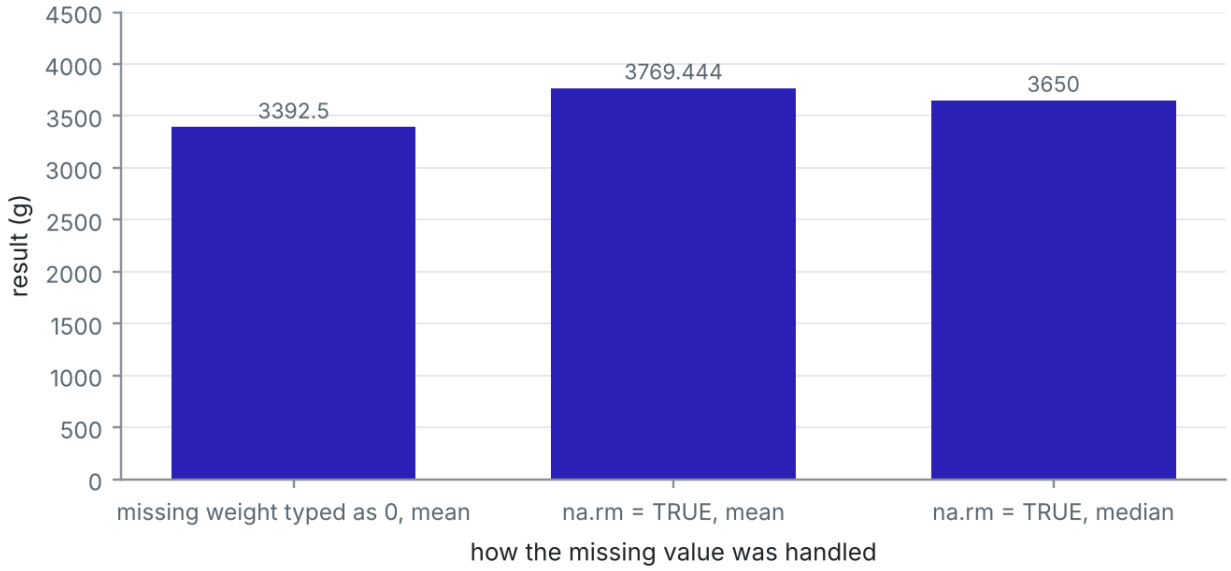
The pipe, written `|>`, puts the steps in the order they happen. Read it as "then". Type it as two characters, `|` then `>`. A line that ends in `|>` is unfinished, so R waits for the next line, as with the `+` prompt in section 2. The indent is only for reading:

```
heights_cm |>
  mean() |>
  round(digits = 1)
```

```
[1] 161.2
```

The pipe hands the thing on its left to the call on its right, as the first argument. So `round(digits = 1)` here means `round(the mean, digits = 1)`.

To find out what a function's arguments are called and what they do, open its help page. Type a question mark and the name: `?round`.



The same ten body masses, with penguin 4 missing, summarised three ways. Typing the missing weight as 0 gives a mean of 3392.5 g. Leaving it out with na.rm = TRUE gives a mean of 3769.444 g and a median of 3650 g, both from 9 values.

Illustration 1. Start where the last section stopped. Here again are the first ten body masses in the practice file, in grams, with the fourth missing. Ask for their mean:

```
body_mass_g <- c(3750, 3800, 3250, NA, 3450, 3650, 3625, 4675, 3475, 4250)
mean(body_mass_g)
```

```
[1] NA
```

`NA` again. Open the help page:

```
?mean
```

Read the Usage section first. It shows the call with every argument and its default: `mean(x, trim = 0, na.rm = FALSE, ...)`. Then read Arguments, which says what each one does. `na.rm` is "a logical evaluating to 'TRUE' or 'FALSE' indicating whether 'NA' values should be stripped before the computation proceeds". Its default is `FALSE`: do not remove them. That is why one missing value made the mean `NA`. You did not ask for anything to be removed.

Ask for it:

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

```
[1] 3769.444
```

Now look at what that number is. It is not the mean of ten values. Check it with the tools from the last section, adding the known values and dividing by how many there are:

```
known_g <- body_mass_g[!is.na(body_mass_g)]
sum(known_g)
length(known_g)
sum(known_g) / length(known_g)
```

```
[1] 33925
[1] 9
[1] 3769.444
```

The nine known masses add up to 33925 g, and 33925 divided by 9 is the same 3769.444 g. So `na.rm = TRUE` changed the count behind the mean from ten to nine. That is the honest answer, as long as you report the nine with it. A short way to count the values that are present is to add up a logical vector. In arithmetic R counts each `TRUE` as 1 and each `FALSE` as 0:

```
sum(!is.na(body_mass_g))
```

```
[1] 9
```

Here is the dishonest answer, which is easy to reach by accident. Suppose whoever typed the register wrote 0 where the weight was not taken:

```
body_mass_zero_g <- c(3750, 3800, 3250, 0, 3450, 3650, 3625, 4675, 3475, 4250)
mean(body_mass_zero_g)
```

```
[1] 3392.5
```

No warning, no `NA`, and a number that looks like a body mass. It is the 33925 g divided by 10, not by 9, as though one penguin weighed nothing. And `median()` has the same argument:

```
median(body_mass_g, na.rm = TRUE)
```

```
[1] 3650
```

how the missing value was handled	result (g)
missing weight typed as 0, mean	3392.5
na.rm = TRUE, mean	3769.444
na.rm = TRUE, median	3650

Last, put the steps together for a report in kilograms. Nested, it reads inside out. Piped, it reads in order: take the masses, then convert to kilograms, then take the mean of the known values, then round to two decimal places.

```
round(mean(body_mass_g / 1000, na.rm = TRUE), digits = 2)

(body_mass_g / 1000) |>
  mean(na.rm = TRUE) |>
  round(digits = 2)
```

```
[1] 3.77
[1] 3.77
```

Both give 3.77 kg. The sentence for the report is "mean body mass 3.77 kg (n = 9)".

WHERE THIS PICTURE BREAKS. The ten masses are real rows of the practice file. The register with a 0 typed in is made up, to show what a zero does to a mean.

`na.rm = TRUE` gives the mean of the values present. It is the mean of all ten only if the missing one would have sat at that same mean, and nothing here can tell you whether it would. Which values go missing, and why, is a question for a later book in this course.

Illustration 2. Error messages are about a call. Read them as sentences. Leave out an argument that has no default:

```
mean()
```

```
Error in mean.default() : argument "x" is missing, with no default
```

The message names the function (`mean.default` is the version of `mean()` that handles plain numbers) and says exactly what is wrong: the argument `x` is missing and has no default.

Misspell a value:

```
mean(body_mass_g, na.rm = true)
```

```
Error: object 'true' not found
```

R knows `TRUE` in capitals. To it, `true` is the name of an object, and no object with that name exists. Give a function the wrong kind of value:

```
round("3.77", digits = 1)
```

```
Error in round("3.77", digits = 1) :
  non-numeric argument to mathematical function
```

`round()` needs a number and was given text.

Errors are the friendly failures, because they stop you. The dangerous mistake is the call that runs. Suppose you want the mean of three body masses and forget `c()`:

```
mean(3750, 3800, 3250)
```

```
[1] 3750
```

No error and no warning, and 3750 g is a believable body mass. It is wrong. Without `c()`, the three numbers are three separate arguments, matched by position. `mean()` took 3750 as `x`, 3800 as its second argument, `trim`, and 3250 went into the `...` that collects extra arguments. The help page says `trim` is "the fraction (0 to 0.5) of observations to be trimmed from each end of 'x' before the mean is computed", and that values outside that range are taken as the nearest end. So `trim` became 0.5, and the "mean" of the single value 3750 is 3750. The call you meant:

```
mean(c(3750, 3800, 3250))
```

```
[1] 3600
```

The mean of the three is 3600 g.

WHERE THIS PICTURE BREAKS. Not every wrong call is silent and not every one errors; which happens depends on the function. The trim behaviour is particular to `mean()`. The general lesson is to put the values you mean to combine inside `c()` and to name every argument after the first.

MUST KNOW POINTS FOR YOU.

- The error is to read `mean(x, na.rm = TRUE)` as the mean of all your values. It is the mean of the values present. Report the count with it, "3.77 kg (n = 9)", every time.
- A missing value typed as 0 gives no `NA` and no warning, and pulls the mean down.
- Name every argument after the first. `round(x, digits = 1)` says what the 1 is. `mean(3750, 3800, 3250)` runs, gives 3750 and is wrong, because unnamed values were matched by position to arguments you did not intend.
- Before you use a function for the first time, open its help page and read Usage and Arguments.
- Read an error message as a sentence about one call: which function, which argument, what was wrong. Fix that argument and nothing else, then run the line again.
- No argument can tell you why a value is missing.
- If you find yourself copying the same block of calls three times with one thing changed, you need a function of your own. Writing one is taught in a later book in this course.

EXERCISE 1 (teaching). A first-year resident has ten minutes and R open. They have never read a help page. Teach them to find out, from `?round` alone, what `round(3.14159)` will give and why.

EXERCISE 2 (retrieval). Without looking back, say what `na.rm = TRUE` does in `mean(x, na.rm = TRUE)` and what its default is. Then say what you must report beside a mean computed this way.

PRACTICE. Eleven problems on this technique, easiest first. Work them on paper before turning to the answers, which are in the appendix at the back under this section's name.

1. A made-up adult's BMI is 24.83702 kg/m^2. Write one call that rounds it to one decimal place and one that rounds it to a whole number.

2. Find the mean of the four numbers 4, 8, a missing value and 6, with the default behaviour. Then find it using only the values that are present.

3. Rewrite this nested call with the pipe, one step per line, and check that it gives the same answer.

```
round(mean(c(61.2, 58.4, 70.9)), digits = 1)
```

4. The first ten flipper lengths in `penguins_raw.csv` are 181, 186, 195, a missing value, 193, 190, 181, 195, 193 and 190 mm. Give their mean and their median in centimetres, each to one decimal place, and say how many values each is based on.

5. Using only the help page of `sd()`, find how to make it leave out missing values. The first ten body masses in `penguins_raw.csv` are 3750, 3800, 3250, a missing value, 3450, 3650, 3625, 4675, 3475 and 4250 g. Give their standard deviation, rounded to the nearest gram.

6. Without using `mean()`, write code that gives the same number as `mean(body_mass_g, na.rm = TRUE)` for the ten body masses in problem 5. Then show that the two agree.

7. A colleague wants the mean of three body masses. Their code and output are below, and they report "the mean body mass was 3750 g". Find what broke, and say why the answer looks reasonable.

```
mean(3750, 3800, 3250)
```

```
[1] 3750
```

8. A colleague's script stops at this line. The error is below. Find what broke.

```
body_mass_g <- c(3750, 3800, 3250, NA, 3450)
mean(body_mass_g, na.rm = true)
```

```
Error: object 'true' not found
```

9. A colleague reports the mean of the first ten body masses (problem 5) in kilograms, to two decimal places, as 3.78 kg. Their code and output are below. Find what broke, and say why the answer looks reasonable.

```
body_mass_g <- c(3750, 3800, 3250, NA, 3450, 3650, 3625, 4675, 3475, 4250)
round(body_mass_g / 1000) |>
  mean(na.rm = TRUE) |>
  round(digits = 2)
```

```
[1] 3.78
```

10. A made-up draft report says: “The average body mass in the first ten rows was 3392.5 g.” The first ten body masses are as in problem 5. Work out with code how that number could have been produced. Say what the sentence gets wrong, and what a corrected sentence still does not establish.

11. A made-up thesis table reports “median body mass 3650 g (n = 10)” for the first ten rows of the practice file. The values are as in problem 5. Check both numbers with code, and say what the checked result does not establish.

6 · Packages: where they come from and which version you have

DEFINITION. A package is a collection of R functions, data and documentation that extends what base R, the set of standard packages installed with R itself, can do.

Installing a package, with `install.packages()`, downloads it from a repository such as CRAN (the Comprehensive R Archive Network) and stores it on the computer. It is done once per computer, and again only to change the version. Loading a package, with `library()`, makes its functions available in the current R session (R from the moment it starts until it is closed or restarted). `install.packages()` and `packageVersion()` need the name in quotation marks; `library()` also accepts it without.

Every package has a version number, read with `packageVersion()`. R itself has one, held in `R.version.string`. A package's behaviour, and the messages it prints, can change between versions. [105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126]

IN PLAIN TERMS. A package is a bundle of functions, with the help pages that explain them and sometimes some data. Most are kept on CRAN, a set of websites from which R downloads them.

Using a package takes two steps, and they happen at different times.

First, install it, once per computer. You type it in the console, not in your script:

```
install.packages("dplyr")
```

Then load it, in every session. It goes at the top of your script:

```
library(dplyr)
```

Some packages share a function name with another package. Loading dplyr, for example, hides another `filter()`, which belongs to stats, a package that comes with R. To be sure which one you get, write the package name and two colons in front: `dplyr::filter()` or `stats::filter()`.

The tidyverse is a family of packages: loading it loads nine, among them dplyr and readr.

Packages change. Each has a version number, and so does R. Check yours with:

```
R.version.string
packageVersion("dplyr")
```

```
[1] "R version 4.3.3 (2024-02-29)"
[1] '1.1.4'
```

Illustration 1. Start a fresh R session (in RStudio, Session, then Restart R) and try a function from dplyr before loading it. `n_distinct()` counts how many different values a vector holds. Here it is given the home districts of four made-up clinic patients:

```
n_distinct(c("Raipur", "Durg", "Raipur", "Bilaspur"))
```

```
Error in n_distinct(c("Raipur", "Durg", "Raipur", "Bilaspur")) :
  could not find function "n_distinct"
```

R looked through everything loaded and found no function by that name. The package is installed, but the session has not been told to use it. This is a common error in a script that worked yesterday: yesterday's session had loaded the package, and today's has not. Load it:

```
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
```

Read the message. "Masked" means hidden. dplyr has functions called `filter()` and `lag()`, and so does stats. It also has four functions whose names are already used in base. From now on in this session, `filter` means dplyr's `filter()`. Now the call works:

```
n_distinct(c("Raipur", "Durg", "Raipur", "Bilaspur"))
```

```
[1] 3
```

Three different districts. To see the order R searches in, ask for the search list:

```
search()
```

```
 [1] ".GlobalEnv"        "package:dplyr"     "package:stats"
 [4] "package:graphics"  "package:grDevices" "package:utils"
 [7] "package:datasets"  "package:methods"   "Autoloads"
[10] "package:base"
```

`.GlobalEnv` is the workspace, where the objects you make are kept, and R looks there first. dplyr now comes before stats, which is why its `filter()` wins.

The two-colon form needs no `library()` call at all. It names the package and the function together. Read the practice file and count its rows, using readr without loading it. Two new pieces appear here. `here::here()` builds the path to the file from the top of your project, which section 7 sets up; if you have none yet, run this after that section. `show_col_types = FALSE` turns off a report

that `read_csv()` prints, which the section on reading files explains. `nrow()` counts the rows of a table:

```
readr::read_csv(here::here("data-raw", "penguins_raw.csv"), show_col_types = FALSE) |>
  nrow()
```

```
[1] 344
```

The file has 344 rows. One row is one adult penguin's record on one sampling expedition, a nesting observation; the same bird can appear in more than one row. The two-colon form is useful when you need one function from a package once. When a script uses a package throughout, load it with `library()` at the top, so a reader sees at once what the script needs.

WHERE THIS PICTURE BREAKS. The district names are made up. The search list printed here is the one from a fresh session in this book's setup. Yours will list more if you load more packages, and RStudio may show extra entries.

Illustration 2. Here is the failure that versions cause. A script written two years ago uses a function that a newer version of a package has changed. The script stops with an error, or, worse, runs and prints something slightly different. Nothing in the script tells you which version it was written for.

So record the versions. Ask for the version of R and of each package your script loads:

```
R.version.string
packageVersion("dplyr")
packageVersion("readr")
```

```
[1] "R version 4.3.3 (2024-02-29)"
[1] ‘1.1.4’
[1] ‘2.1.5’
```

These are the versions behind every output in this book: R 4.3.3, dplyr 1.1.4 and readr 2.1.5. They are not the newest. When this book was written (checked 2 October 2026), R 4.6.1 (June 2026) and dplyr 1.2.0 (February 2026) had been released. The dplyr 1.2.0 changelog, the list of what changed in that version, adds new functions and, among other changes, now requires R 4.1.0 or later.

For you this means two things. First, if your output differs from the book's in a message or a small detail, compare your versions with the book's before you suspect your code. Second, your own script should say which versions it ran on. `packageVersion()` returns a version, not plain text, so R compares it with the text part by part, as numbers:

```
packageVersion("dplyr") >= "1.1.0"
```

```
[1] TRUE
```

Printing the versions at the end of every script, in one line, is the subject of a later section on running a script from top to bottom.

WHERE THIS PICTURE BREAKS. A version number tells you which release you have, not what changed between releases. For that you read the package's changelog. Even with the same versions, a different operating system can, rarely, give a different result.

MUST KNOW POINTS FOR YOU.

- The error is to put `install.packages()` in your script. Install once, in the console.
- "could not find function" means R has no function by that name loaded. Check the spelling, then check that its package is loaded in this session.
- Read the message that `library()` prints. After `library(dplyr)`, `filter()` means dplyr's.
- Record the version of R and of every package your analysis loads, in the script's output.
- When a trainee's output differs from the book's in a message or a detail, have them compare versions with `packageVersion()` before they rewrite their code.
- A version number names a release. It does not pin it: the next person to run `install.packages()` gets whatever version is current that day.

EXERCISE 1 (design). You are starting a script, `analysis.R`, that will read a raw file with readr and reshape it with dplyr. Write its first lines: the packages it needs and a record of the versions it ran on. Say why `install.packages()` does not appear in it.

EXERCISE 2 (retrieval). Without looking back, say how often you run `install.packages()` and how often you run `library()`, and where each one goes.

PRACTICE. Seven problems on this technique, easiest first. Work them on paper before turning to the answers, which are in the appendix at the back under this section's name.

1. Write the line you would type in the console to install the package readr. Then write the line that goes at the top of a script that uses it.

2. In a fresh session you load dplyr and then type `filter(...)`. Which package's `filter()` runs? Write the call you would use to run the other one, and the line that shows you what dplyr hides when it loads.

3. Write the lines that print the version of R you are running and the version of the package here.

4. Without loading any package, count the rows of `penguins_raw.csv` in your project's `data-raw` folder, and say what one row stands for. (The project is set up in section 7; if you have none yet, do this after it.)

5. A colleague's script begins as below and stops at its second line with the error shown. The script worked for them yesterday. Find what broke.

```
raw <- here::here("data-raw", "penguins_raw.csv")
penguins_raw <- read_csv(raw, show_col_types = FALSE)
```

```
Error in read_csv(raw, show_col_types = FALSE) :
  could not find function "read_csv"
```

6. A colleague wants to report the version of dplyr. Their line and its error are below. Find what broke, and say why the mistake is easy to make.

```
packageVersion(dplyr)
```

```
Error: object 'dplyr' not found
```

7. A made-up thesis methods section says: “All analyses were carried out in R.” Write the lines that would record what this sentence leaves out, for an analysis that uses readr and dplyr. Run them, and say what even your fuller record does not establish.

7 · A project folder, and paths that work on someone else's computer

DEFINITION. A path is the address of a file. An absolute path starts from the top of one computer's disk, such as `C:/Users/priya/Documents/thesis/data-raw/heights.csv`, so it names a place that exists on one computer only. A relative path starts from the working directory, the folder R treats as home. `data-raw/heights.csv` is one. [127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143]

IN PLAIN TERMS. Give every analysis its own folder. Inside it, make three folders:

Folder	What goes in it	Who writes to it
data-raw/	the data exactly as it arrived, and its documentation	you, by hand; never the code
code/	your scripts	you
output/	tables, figures and cleaned files the scripts make	only the scripts

Put a short plain-text file called `README.txt` at the top. It says what the project is, where the data came from, and how to run it again.

`C:/Users/priya/Documents/thesis/data-raw/heights.csv` is the full address on Priya's laptop. On your laptop there is no `C:/Users/priya` folder, so that address leads nowhere. `data-raw/heights.csv` says "from where I am now, go into data-raw".

So use `here()`, which finds the top of the project for you: `here("data-raw", "heights.csv")` gives the full address on whichever computer runs it.

Never write a full address from your own computer into a script. And never start a script with `setwd()`, the function that moves R's current home to a named folder.

Illustration 1. Start with the failure. A colleague, Priya, emails you the script for her thesis. You copy her `data-raw` file into your own project and run her first lines. (`read_csv()`, from the readr package, reads a CSV file (comma-separated values, a plain-text table) into R; the section on reading a raw file explains it.) Here they are, run on your computer:

```
library(readr)
penguins_raw <- read_csv("C:/Users/priya/Documents/thesis/data-raw/penguins_raw.csv")
```

```
Error: 'C:/Users/priya/Documents/thesis/data-raw/penguins_raw.csv' does not exist.
```

Read the error literally. R went looking for a folder called `C:/Users/priya` and found none. Nothing is wrong with her data or her code. The address is wrong, because it is the address of a place on her laptop. Every line after this one fails too, because `penguins_raw` was never made.

Now ask R where it is. The function `getwd()`, short for "get working directory", prints the folder R is treating as home:

```
getwd()
```

```
[1] "~/project"
```

On your computer this prints your own folder. Here it is the project folder (~ is short for your home folder). The file sits in `data-raw` inside it, so give the address from there:

```
penguins_raw <- read_csv("data-raw/penguins_raw.csv", show_col_types = FALSE)
nrow(penguins_raw)
```

```
[1] 344
```

The file opens and has 344 rows. The argument `show_col_types = FALSE` stops `read_csv()` printing a report about the columns; that report matters, but not for this section.

One more trap hides in addresses copied from Windows. The Windows file window writes paths with backslashes, `data-raw\penguins_raw.csv`. A backslash means something special to R, so a path pasted with backslashes usually stops with an error, and can quietly point somewhere else. Type forward slashes, which R reads on every system.

WHERE THIS PICTURE BREAKS. The relative path worked here because R's working directory was the top of the project. If R is started from another folder, `data-raw/penguins_raw.csv` fails in the same way the absolute path did. The next illustration shows that happening, and the fix.

The error message you see may be worded differently on a newer version of readr. What stays the same is the cause: R looked in a folder that does not exist on your computer.

Illustration 2. Now build the project and make the address safe. Load the here package. It prints the folder it has taken as the project root:

```
library(here)
```

```
here() starts at ~/project
```

It found the root by climbing upwards from the working directory until it met a marker file. In this book's project the marker is an empty file called `.here`. An RStudio project file works the same way; in RStudio, File > New Project makes one. Now build addresses from that root:

```
here()
here("data-raw", "penguins_raw.csv")
```

```
[1] "~/project"
[1] "~/project/data-raw/penguins_raw.csv"
```

Each argument is one folder or file name, given in order from the top. The function only builds the address. It does not check that the file exists, and it creates nothing.

Make the two missing folders from code, then list what the project holds:

```
dir.create(here("code"))
dir.create(here("output"))
list.files(here())
```

```
[1] "code"     "data-raw" "output"
```

Now the test that matters. Save a three-line script in `code/` that reads the file with a relative path:

```
library(readr)
penguins_raw <- read_csv("data-raw/penguins_raw.csv", show_col_types = FALSE)
nrow(penguins_raw)
```

Run it the way a colleague might, from inside the `code` folder. These are commands for the computer's terminal, the window where you type commands to the computer itself rather than to R. `cd code` moves into the folder, and `Rscript` runs an R script from start to end. A later section, on running a script top to bottom in a fresh session, explains `Rscript` fully. Here:

```
cd code
Rscript count_rows.R
```

```
Error: 'data-raw/penguins_raw.csv' does not exist in current working directory ('~/
project/code').
Execution halted
```

The relative path failed because R started in `code/`, and there is no `data-raw` folder inside `code/`. Change one line, so that the address is built by `here()`:

```
library(readr)
library(here)
penguins_raw <- read_csv(here("data-raw", "penguins_raw.csv"), show_col_types = FALSE)
nrow(penguins_raw)
```

Run it from the same place:

```
cd code
Rscript count_rows.R
```

```
here() starts at ~/project
[1] 344
```

R started in `code/` again. The here package climbed one folder up, found the marker, and built the full address from there. The script now finds its file from any folder inside the project, on any computer that has the project folder.

WHERE THIS PICTURE BREAKS. `here()` finds the root only if the marker is there. Copy the project without its `.here` file or RStudio project file and `here()` keeps climbing, so it can settle on a folder higher up that holds a marker. Keep the marker in the project folder, and copy the whole folder when you share it.

A safe address fixes one cause of failure, the location of files. A script can still fail on another computer because a package is missing or is a different version. Running the whole script in a

fresh session on a second computer is a separate check. Keeping every version of the project's files, with version control, is the next rung's subject.

MUST KNOW POINTS FOR YOU.

- An address that starts with `C:/Users/` or `/home/` names a folder on one computer. Before you send a script, search it for such addresses and replace each one with `here()`.
- Build every file address with `here()`, starting from the project root: `here("data-raw", "file.csv")` to read, `here("output", "table.csv")` to write.
- A script that starts with `setwd()` and a folder name works only on the computer it was written on.
- No code ever writes into `data-raw/`, and no one edits a raw file there. Put the correction in code, and write the corrected file to `output/`.
- Never put anything in `output/` that the code cannot make again.
- A trainee's script fails on your computer with "does not exist". Look at the address in the failing line before you look at the data, and check whether it names a folder on their computer.
- Name files with no spaces and no brackets. `Final data (2).xlsx` and `final data.xlsx` are a quarrel waiting to happen.
- A tidy folder and safe addresses fix only where files are. Do not accept "the paths are relative" as proof that someone else can rerun the work.

EXERCISE 1 (retrieval). Without looking back, write the line that reads a file called `heights.csv` from the project's `data-raw` folder so that it works on any computer. Say why a line starting with `setwd()` does not belong above it.

EXERCISE 2 (design). You are starting a thesis analysis of weight and height measured in a school health camp. The camp team will give you one CSV file, a plain-text table, exported from their data-entry form. Lay out the project folder. Give it a name, and list the folders and the files you expect in each. Say which folders your code may write to. Then write the one line that reads the camp file.

EXERCISE 3 (teaching). A first-year resident has ten minutes and a whiteboard. Her script ran on her laptop and stopped on yours with "does not exist". Teach her why, and what to change.

PRACTICE. Nine problems on this technique, easiest first. Work them on paper before turning to the answers, which are in the appendix at the back under this section's name.

1. Say whether each address is absolute or relative.

1. `C:/Users/ravi/Documents/thesis/data.csv`
2. `data-raw/camp_export.csv`
3. `/home/ravi/thesis/output/table1.csv`
4. `output/figures/bmi_histogram.png`
5. `D:/MD thesis/final.xlsx`

2. The project root is `~/project`. Predict what this line prints, then run it. Does running it create a folder called `figures`?

```
here::here("output", "figures", "bmi_histogram.png")
```

3. Read `penguins_raw.csv` from the project's `data-raw` folder, building the address with `here()`. Give its number of rows and columns with `dim()`.

4. Where does each of these files belong in the project folder: `data-raw/`, `code/`, `output/`, or the top?

1. `penguins_raw.csv`, exactly as downloaded
2. `analysis.R`
3. `species_counts.csv`, written by your script
4. `mass_histogram.png`, saved by your script
5. `README.txt`
6. a copy of `penguins_raw.csv` in which you corrected one value by hand

5. In the project, make the `output` folder from code, check that it now exists, and list the folders at the top of the project.

6. A colleague's script begins like this. Run it. Find the line that broke, and say why it looked reasonable to the person who wrote it.

```
library(readr)
setwd("C:/Users/priya/Documents/thesis")
penguins_raw <- read_csv("data-raw/penguins_raw.csv", show_col_types = FALSE)
```

```
Error in setwd("C:/Users/priya/Documents/thesis") :
  cannot change working directory
```

7. A resident wants a quick test run on fewer rows. Her script keeps the first 300 rows and saves them. She runs it twice. (`head(x, 300)` keeps the first 300 rows; `write_csv()` writes a table to a file.) Run the two runs below. Find the line that broke, and say why nothing looked wrong on the first run.

```
library(readr)
library(here)
# first run
penguins_raw <- read_csv(here("data-raw", "penguins_raw.csv"), show_col_types = FALSE)
nrow(penguins_raw)
penguins_test <- head(penguins_raw, 300)
write_csv(penguins_test, here("data-raw", "penguins_raw.csv"))
# second run, the same lines again
penguins_raw <- read_csv(here("data-raw", "penguins_raw.csv"), show_col_types = FALSE)
nrow(penguins_raw)
```

```
here() starts at ~/project
[1] 344
[1] 300
```

8. Your guide says at a meeting: “Send me the project folder. If it runs on your laptop, it will run on mine.”

Test the second half of that claim before you send anything. The project holds this script:

```
library(readr)
penguins_raw <- read_csv(here::here("data-raw", "penguins_raw.csv"), show_col_types =
FALSE)
nrow(penguins_raw)
```

Then say what your test does not establish.

9. A co-author writes: “Just email me the cleaned file. I don’t need your code. The cleaned file is the result.”

The project holds this script. It reads the raw penguin file and writes the rows that have a body mass recorded to `output/penguins_with_mass.csv`:

```
library(readr)
library(here)
penguins_raw <- read_csv(here("data-raw", "penguins_raw.csv"), show_col_types = FALSE)
# keep the rows whose body mass is not missing
with_mass <- penguins_raw[!is.na(penguins_raw$`Body Mass (g)`), ]
dir.create(here("output"), showWarnings = FALSE)
write_csv(with_mass, here("output", "penguins_with_mass.csv"))
```

Run these terminal commands. They run the script, print a checksum of the cleaned file, delete the file (`rm`), remake it and print its checksum again. `tools::md5sum()` computes an MD5 checksum, a 32-character code calculated from every byte of a file; section 9 teaches it.

```
Rscript code/clean.R
Rscript -e 'unname(tools::md5sum("output/penguins_with_mass.csv"))'
rm output/penguins_with_mass.csv
Rscript code/clean.R
Rscript -e 'unname(tools::md5sum("output/penguins_with_mass.csv"))'
```

```
here() starts at ~/project
[1] "2d547e4c975aa165159abbda25d7fc9e"
here() starts at ~/project
[1] "2d547e4c975aa165159abbda25d7fc9e"
```

What does the result tell the co-author, and what does it not establish?

8 · Tidy data and the data frame

DEFINITION. A data frame is R's table. Different columns may hold different types, but all the values within one column share one type.

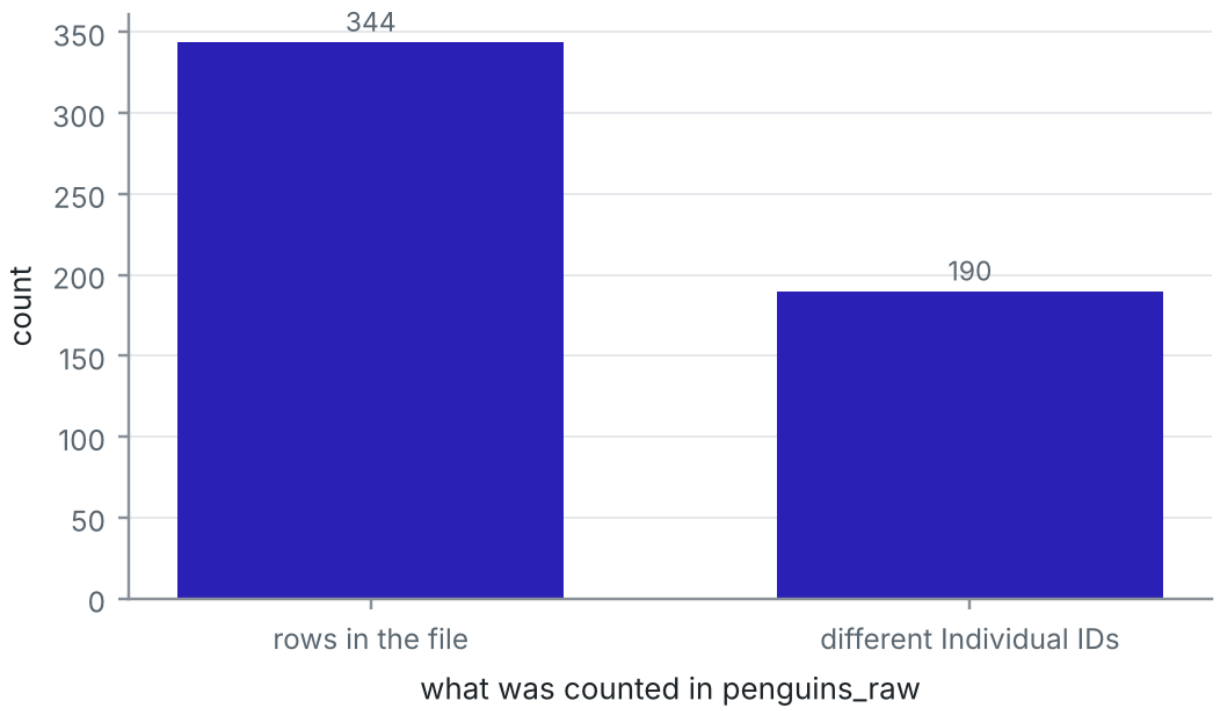
Data are tidy (Wickham 2014) when three rules hold. Each variable forms a column. Each observation forms a row. Each type of observational unit forms a table: facts about a child that do not change between visits go in one table, and measurements made at each visit go in another. [144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160]

IN PLAIN TERMS. Each column is a vector, a set of values of one type: all numbers, or all text. Every column has the same length. A tibble is the tidyverse's kind of data frame. It tells you its size and the type of each column when it prints.

A table is tidy when three things are true:

- **One column, one variable.**
- **One row, one observation.** A second visit is a second row, not a second column.
- **One table, one kind of unit.** Weight at each visit goes in a table with one row per visit.

Data typed into a spreadsheet often break the rules. Weights at three visits sit in three columns. Blood pressure is typed as "104/66" in one cell. Girls are marked by colouring their rows pink. Each stops the computer from using the data.



The penguin file has 344 rows but only 190 different values of Individual ID, so 154 rows repeat an ID that an earlier row already carries. One row is not one bird, and N1A1 alone names two birds of different species.

Illustration 1. Start with a register the way a clinic might keep it in a spreadsheet. The numbers are made up. Each child has a row, each visit's weight has a column, and blood pressure at the first visit is typed as one piece of text. Blood pressure is measured in millimetres of mercury, written mmHg.

Build it in R. The function `tibble()`, from the tibble package, makes a tibble from named vectors, one per column:

```
library(tibble)
register <- tibble(
  child_id = c("C01", "C02", "C03"),
  weight_v1_kg = c(21.4, 25.0, 19.8),
  weight_v2_kg = c(22.1, 25.6, 20.3),
  bp_v1 = c("104/66", "110/70", "98/62")
)
register
```

```
# A tibble: 3 × 4
  child_id weight_v1_kg weight_v2_kg bp_v1
  <chr>           <dbl>        <dbl> <chr>
1 C01              21.4         22.1 104/66
2 C02              25           25.6 110/70
3 C03              19.8         20.3 98/62
```

In printed output, # marks R's summary lines, not comments. Under each name the tibble prints the column's type: `<chr>` text, `<dbl>` numbers, `<date>` dates.

Now ask two simple questions of it. First, what is the average systolic pressure, the upper number?

```
mean(register$bp_v1)
```

```
[1] NA
Warning message:
In mean.default(register$bp_v1) :
  argument is not numeric or logical: returning NA
```

R cannot average “104/66”. It is text, because one cell holds two numbers and a slash. Second, what is the average weight over all visits? There is no single column of weights to average. You would have to name each visit’s column, and change the code when a third visit is added.

Rebuild it tidy. A weight is measured on one child at one visit, so one row is one visit, and the visit number becomes a column of its own:

```
visits <- tibble(
  child_id = c("C01", "C01", "C02", "C02", "C03", "C03"),
  visit = c(1, 2, 1, 2, 1, 2),
  weight_kg = c(21.4, 22.1, 25.0, 25.6, 19.8, 20.3)
)
visits
mean(visits$weight_kg)
```

```
# A tibble: 6 × 3
  child_id visit weight_kg
  <chr>    <dbl>     <dbl>
1 C01          1      21.4
2 C01          2      22.1
3 C02          1      25
4 C02          2      25.6
5 C03          1      19.8
6 C03          2      20.3
[1] 22.36667
```

The `$` sign picks one column out of a data frame by its name: `visits$weight_kg` is the weight column, a numeric vector. Its mean, worked by hand, is 134.2 divided by 6:

```
21.4 + 22.1 + 25.0 + 25.6 + 19.8 + 20.3 = 134.2
134.2 / 6 = 22.36667
```

The blood pressure becomes two columns, each holding one number:

```
bp <- tibble(
  child_id = c("C01", "C02", "C03"),
  systolic_mmhg = c(104, 110, 98),
  diastolic_mmhg = c(66, 70, 62)
)
mean(bp$systolic_mmhg)
```

```
[1] 104
```

One more rule is built into R itself. Every column must have the same length. Leave a value out and `tibble()` refuses:

```
tibble(child_id = c("C01", "C02", "C03"), weight_kg = c(21.4, 25.0))
```

```
Error in `tibble()`:
! Tibble columns must have compatible sizes.
• Size 3: Existing data.
• Size 2: Column `weight_kg`.
ℹ Only values of size one are recycled.
```

`!` gives the error, `•` the details, **ℹ** a hint. Three names and two weights cannot line up into rows, so R stops rather than guess which child has no weight. A missing weight is written `NA`, which keeps the place.

WHERE THIS PICTURE BREAKS. "Tidy" depends on the question. For a question about how weight changes between visit 1 and visit 2 in each child, a table with the two weights side by side is the handy shape, and Wickham's own paper finds it "surprisingly difficult" to define variables in general. The rules give a default for storing data, not a ban on other shapes made from it by code for one use.

The register's numbers are made up. The average of 22.36667 kg describes these six made-up visits and nothing else.

Illustration 2. Now the real file. Read the penguin data and look at it with `glimpse()`, from the tibble package, which prints each column on its own line with its type and its first few values. Lines too long for the screen end in …:

```
library(readr)
library(tibble)
library(here)
penguins_raw <- read_csv(here("data-raw", "penguins_raw.csv"), show_col_types = FALSE)
glimpse(penguins_raw)
```

```
here() starts at ~/project
Rows: 344
Columns: 17
$ studyName             <chr> "PAL0708", "PAL0708", "PAL0708", "PAL070…
$ `Sample Number`       <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 1…
$ Species               <chr> "Adelie Penguin (Pygoscelis adeliae)", "…
$ Region                <chr> "Anvers", "Anvers", "Anvers", "Anvers", …
$ Island                <chr> "Torgersen", "Torgersen", "Torgersen", "…
$ Stage                 <chr> "Adult, 1 Egg Stage", "Adult, 1 Egg Stag…
$ `Individual ID`       <chr> "N1A1", "N1A2", "N2A1", "N2A2", "N3A1", …
$ `Clutch Completion`   <chr> "Yes", "Yes", "Yes", "Yes", "Yes", "Yes"…
$ `Date Egg`            <date> 2007-11-11, 2007-11-11, 2007-11-16, 200…
$ `Culmen Length (mm)`  <dbl> 39.1, 39.5, 40.3, NA, 36.7, 39.3, 38.9, …
$ `Culmen Depth (mm)`   <dbl> 18.7, 17.4, 18.0, NA, 19.3, 20.6, 17.8, …
$ `Flipper Length (mm)` <dbl> 181, 186, 195, NA, 193, 190, 181, 195, 1…
$ `Body Mass (g)`       <dbl> 3750, 3800, 3250, NA, 3450, 3650, 3625, …
$ Sex                   <chr> "MALE", "FEMALE", "FEMALE", NA, "FEMALE"…
$ `Delta 15 N (o/oo)`   <dbl> NA, 8.94956, 8.36821, NA, 8.76651, 8.664…
$ `Delta 13 C (o/oo)`   <dbl> NA, -24.69454, -25.33302, NA, -25.32426,…
$ Comments              <chr> "Not enough blood for isotopes.", NA, NA…
```

`nrow()` and `ncol()` count rows and columns:

```
nrow(penguins_raw)
ncol(penguins_raw)
```

```
[1] 344
[1] 17
```

Several column names have spaces and brackets in them, such as `Body Mass (g)`. To use such a name in code, wrap it in backticks, the slanted quote marks on the key left of 1:

```
mean(penguins_raw$`Body Mass (g)`, na.rm = TRUE)
```

```
[1] 4201.754
```

The average body mass of the penguins with a mass recorded is 4201.754 grams. Now ask Book 0's first question. What does one row stand for? The help page of the palmerpenguins package (`?palmerpenguins::penguins_raw`, once the package is installed) says the file holds nesting observations of adult penguins, and calls `Individual ID` the unique ID for each individual. If one row were one bird, there would be as many IDs as rows. Count them (`unique()` keeps one copy of each value, and `length()` counts them):

```
length(unique(penguins_raw$`Individual ID`))
```

```
[1] 190
```

There are 190 different IDs in 344 rows. So some ID appears more than once. Take one and look at the species and island of every row that carries it:

```
penguins_raw$Species[penguins_raw$`Individual ID` == "N1A1"]
penguins_raw$Island[penguins_raw$`Individual ID` == "N1A1"]
```

```
[1] "Adelie Penguin (Pygoscelis adeliae)"
[2] "Gentoo penguin (Pygoscelis papua)"
[1] "Torgersen" "Biscoe"
```

The ID `N1A1` belongs to an Adélie penguin on Torgersen island and to a Gentoo penguin on Biscoe island. Those are two different birds. The ID alone does not pick out one penguin. One row is one nesting observation of one adult, and you need more than one column to say which bird it is.

Check the file against the rules, and against four pieces of Broman and Woo's advice: one thing per cell, one rectangle with one header row, no colour as data, and no calculations in the raw file. Each column holds one attribute. The units sit in the column names, `(mm)` and `(g)`, as Broman and Woo suggest, and not inside the cells. Free-text notes have their own column, `Comments`, rather than being typed into a measurement cell. By the rules, this file is close to tidy. What it does not do is tell you, from any one column, which rows are the same bird.

WHERE THIS PICTURE BREAKS. The average of 4201.754 grams is a fact about the rows in this file that have a body mass. It mixes three species and both sexes, so it describes no actual group of penguins.

The check above shows that one ID can name two birds. It does not show how many birds there are. That depends on which columns together pick out one bird, and the help page does not say.

MUST KNOW POINTS FOR YOU.

- The error is to read the number of rows as the number of people, or birds, or children. Ask first what one row stands for.
- When a register has one column per visit, the visit is hiding in the column names. Store it as a column called `visit` with one row per visit.
- One cell, one value. "104/66", "45 kg", "2.3 or 2.4" and "0 (below detection)" each stop a column from being numbers.
- Colour and highlighting are not data. Add a column, such as `recheck` with TRUE or FALSE, instead.
- Facts that do not change between visits, such as sex and date of birth, go in a table with one row per person.
- Before a data-entry team fills its first form, look at the sheet they designed. Ask for one header row, one rectangle, one value per cell and no merged cells.
- When a sentence says "variable", ask which of three things it means: Book 0's letter, an R object, or a column.
- Tidy data makes the next step easy. It does not make the values right.

EXERCISE 1 (retrieval). Without looking back, say what a data frame is made of, and what one row of the penguin file stands for.

EXERCISE 2 (critique). Here is a made-up data-entry sheet from a school health camp. It has one header row, with these column headings: `Name`, `Class/Section`, `Ht 1 (cm)`, `Ht 2 (cm)`, `Wt (kg)`,

`BMI`, `Remarks`. Rows for children referred to the clinic are highlighted red. Find every place where the sheet breaks the tidy rules or Broman and Woo's advice, and give the tidy layout.

PRACTICE. Nine problems on this technique, easiest first. Work them on paper before turning to the answers, which are in the appendix at the back under this section's name.

1. How many rows and columns does this made-up tibble have? Check with `nrow()` and `ncol()`.

```
tibble::tibble(x = c(1, 2, 3), y = c("a", "b", "c"))
```

2. Make a tibble of three made-up children with columns `child_id`, `height_cm` and `weight_kg`. Print it, then pick out the weight column with `$` and find its mean.

3. Read `penguins_raw.csv` from the project. Give its number of rows and columns, and the mean flipper length of the penguins with one recorded, with its unit.

4. Which islands does the penguin file cover? Use `unique()` on the right column.

5. In the penguin file, how many rows come from each of the three expeditions named in the column `studyName`? Use `table()`, which counts how often each value occurs. Then say what one row of the file stands for, and why it is not "one penguin".

6. A resident tries to find the average body mass. Run her code. Find what broke and say why the code looked right to her.

```
library(readr)
library(here)
penguins_raw <- read_csv(here("data-raw", "penguins_raw.csv"), show_col_types = FALSE)
```

```
here() starts at ~/project
```

```
mean(penguins_raw$Body Mass (g), na.rm = TRUE)
```

```
Error: <text>:1:24: unexpected symbol
1: mean(penguins_raw$Body Mass
                           ^
```

7. A camp register is entered with blood pressure as text. The data are made up. A resident runs this code to find the average blood pressure and writes "BP data not available" in her report. Run it. Find what broke, and say why her conclusion looks reasonable.

```
camp <- tibble::tibble(
  child_id = c("K1", "K2", "K3", "K4"),
  bp = c("102/64", "108/70", "96/60", "110/72")
)
mean(camp$bp)
```

```
[1] NA
Warning message:
In mean.default(camp$bp) : argument is not numeric or logical: returning NA
```

8. A made-up thesis draft says: “Weight was recorded at three visits for each of the three children (Table 2).” Table 2 has columns `child`, `visit_1`, `visit_2` and `visit_3`. The weights are, in kilograms: child P1 18.2, 18.6, 19.1; P2 20.4, 20.9, 21.0; P3 17.5, 17.8, 18.4. Build the tidy table, check its number of rows against the sentence, and find the mean weight. Then say what your table does not establish.

9. In a meeting someone says: “The penguin file has data on 344 penguins.” Decide what to compute to test the claim, compute it, and say what your answer does not establish.

9 · Reading a raw file without changing it

DEFINITION. A CSV file, short for comma-separated values, is plain text. Nothing in the file says what type each column is.

`read_csv()`, from the readr package, reads a CSV file. It guesses each column's type from a sample of its values and prints the guess, the column specification. Values listed in its `na` argument become `NA`, R's code for a missing value; by default the list is the empty string (`""`, an empty cell) and the text `NA`. The `col_types` argument states the types you expect instead of letting readr guess. A value that does not fit its stated type becomes `NA`, readr warns, and `problems()` lists each such value with its row and column.

Other formats need other functions. `read_excel()`, from readxl, reads Excel `.xls` and `.xlsx` files, one sheet (one tab of a workbook) at a time, optionally from a stated cell range such as `A5:F15`. SPSS and SAS are two statistics programs. From haven, `read_sav()` reads SPSS `.sav` files and `read_xpt()` reads SAS transport `.xpt` files; both keep each column's label, the description stored with it in the file, as a label attribute (extra information stored with a column). [161, 162, 163, 138, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177]

IN PLAIN TERMS. A raw file is the data as it first reached you.

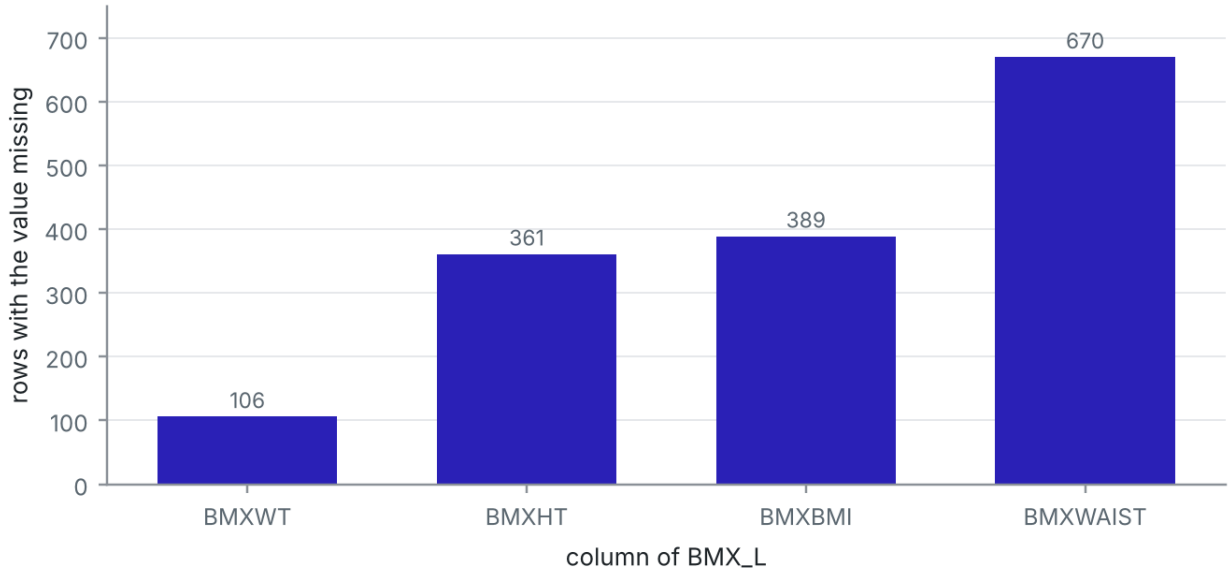
The commonest raw file is a CSV. The first line holds the column names. Nothing in it says what type each column is.

So `read_csv()` has to guess. Then it prints what it decided. Read that report every time. A column you expected to be numbers that comes out as text means some cell in it is not a number.

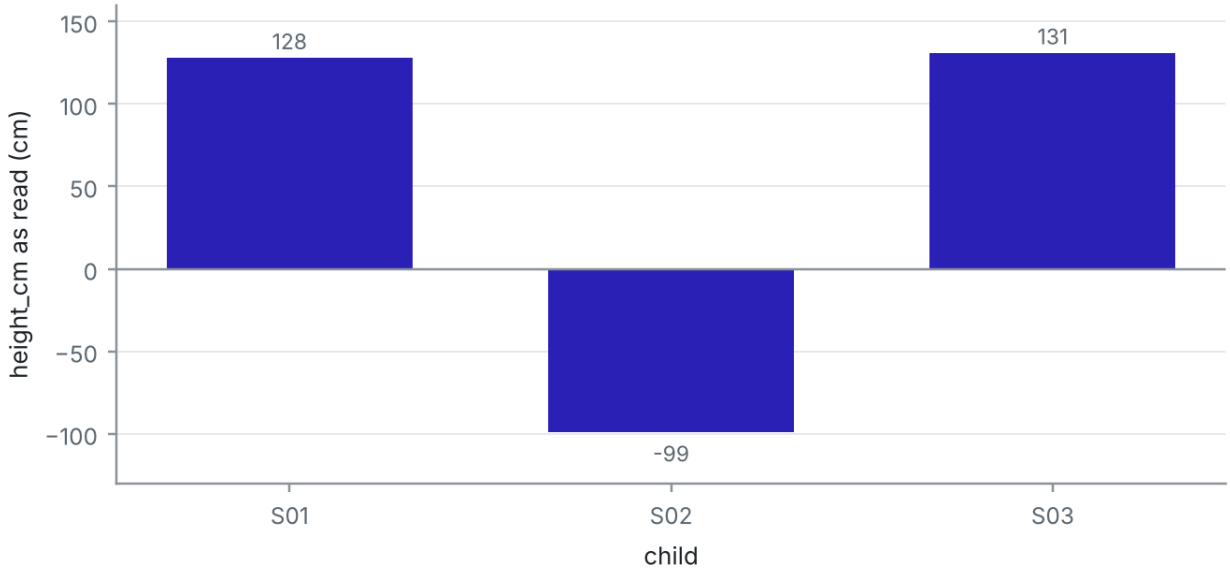
Two habits keep the guess honest:

- **Name the missing-value codes.** If the file uses -99, or "NR", for "not recorded", say so in `na = c("", "NA", "-99", "NR")`. Otherwise -99 is read as a real number.
- **Say what you expect.** `col_types` tells `read_csv()` which type each column should have.

Reading never changes the file.



Missing values in four body-measure columns of the NHANES 2021–2023 body measures file, as read_xpt() reads it: weight 106, height 361, BMI 389, waist 670 rows. BMI is missing in exactly the 389 rows that lack a weight or a height. These are counts of rows in the file, not survey estimates.



Heights in the made-up camp file as read_csv() reads them when it is not told the code: child S02's -99 becomes a height of -99 cm, the bar below zero, and min() returns -99. Declaring na = c("","NA", "-99") makes it NA, and the shortest measured child is 128 cm.

Illustration 1. Start with a failure. A school camp team sends you a file. Where a child could not be measured, they typed -99. The values below are made up. Here is the file's text, held in R as one piece of text that runs over several lines; `writeLines()` prints it as a file would hold it:

```
library(readr)
camp_text <- "child_id,height_cm,weight_kg
S01,128,26.0
S02,-99,24.5
S03,131,-99"
writeLines(camp_text)
```

```
child_id,height_cm,weight_kg
S01,128,26.0
S02,-99,24.5
S03,131,-99
```

That is the whole of a CSV file: a header line, then one line per child. Read it with `read_csv()`. Wrapping the text in `I()` tells `read_csv()` that this is the data itself, not the name of a file:

```
camp <- read_csv(I(camp_text))
camp
```

```
Rows: 3 Columns: 3
── Column specification ────────────────────────────────────────────────────────
Delimiter: ","
chr (1): child_id
dbl (2): height_cm, weight_kg

ℹ Use `spec()` to retrieve the full column specification for this data.
ℹ Specify the column types or set `show_col_types = FALSE` to quiet this message.
# A tibble: 3 × 3
  child_id height_cm weight_kg
  <chr>        <dbl>     <dbl>
1 S01            128      26
2 S02            -99      24.5
3 S03            131     -99
```

The report says `height_cm` and `weight_kg` were read as `dbl`, numbers. That is true, and it is the trap. -99 is a number, so readr had no reason to object. Ask for the shortest child:

```
min(camp$height_cm)
```

```
[1] -99
```

A height of -99 cm. Every summary of this column is now wrong, and nothing warned you. Tell `read_csv()` the code. The vector given to `na` replaces the default, so keep the empty string and `"NA"` in it as well:

```
camp <- read_csv(I(camp_text), na = c("", "NA", "-99"), show_col_types = FALSE)
camp
min(camp$height_cm, na.rm = TRUE)
```

```
# A tibble: 3 × 3
  child_id height_cm weight_kg
  <chr>        <dbl>     <dbl>
1 S01            128      26
2 S02             NA      24.5
3 S03            131      NA
[1] 128
```

The -99s are `NA` now, and the shortest measured child is 128 cm.

Now a second file from the same camp, where one entry says “not measured” in words:

```
camp_text_2 <- "child_id,height_cm,weight_kg
S01,128,26.0
S02,not measured,24.5
S03,131,27.5"
camp_2 <- read_csv(I(camp_text_2))
```

```
Rows: 3 Columns: 3
── Column specification ────────────────────────────────────────────────────────
Delimiter: ","
chr (2): child_id, height_cm
dbl (1): weight_kg

ℹ Use `spec()` to retrieve the full column specification for this data.
ℹ Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

The report says `height_cm` is `chr`, text. One word in one cell turned the whole column into text, and `mean()` of it would fail. Say what you expect instead, with `col_types`. The function `cols()` takes one entry per column; `col_character()` means text and `col_double()` means numbers. A code such as 0042 read as a number becomes 42, so read codes as text:

```
camp_2 <- read_csv(
  I(camp_text_2),
  col_types = cols(
    child_id = col_character(),
    height_cm = col_double(),
    weight_kg = col_double()
  )
)
camp_2
```

```
Warning message:
One or more parsing issues, call `problems()` on your data frame for
details, e.g.:
  dat <- vroom(...)
  problems(dat)
# A tibble: 3 × 3
  child_id height_cm weight_kg
  <chr>        <dbl>     <dbl>
1 S01            128      26
2 S02             NA      24.5
3 S03            131      27.5
```

The `vroom(...)` line is readr's generic example; read it as your own data frame's name. Now readr warns, keeps `height_cm` as numbers, and puts `NA` where the words were. Ask which values did not fit. The last column of `problems()` names a temporary file that changes from run to run, so keep the first four, named inside square brackets (`<int>` marks whole numbers):

```
problems(camp_2)[c("row", "col", "expected", "actual")]
```

```
# A tibble: 1 × 4
    row   col expected actual
  <int> <int> <chr>    <chr>
1     3     2 a double not measured
```

Row 3 of the file, counting the header line as row 1, column 2: readr expected a number and found "not measured". You now know the value, and the raw file still says exactly what the camp team typed.

WHERE THIS PICTURE BREAKS. The camp files are made up and three rows long. In a real file of thousands of rows, a code such as -99 does not stand out by eye, which is the reason to declare it when you read.

Declaring `na` works only for codes you know about. A file may use 999 for one column and 0 for another, and 0 can be a real value. The codes come from the file's own documentation, not from guessing; where there is none, ask the person who made the file.

Illustration 2. Now the real file. Read the penguin data and let `read_csv()` print its report:

```
library(readr)
library(here)
penguins_raw <- read_csv(here("data-raw", "penguins_raw.csv"))
```

```
here() starts at ~/project
Rows: 344 Columns: 17
── Column specification ────────────────────────────────────────────────────────
Delimiter: ","
chr  (9): studyName, Species, Region, Island, Stage, Individual ID, ...
dbl  (7): Sample Number, Culmen Length (mm), Culmen Depth (mm), Flip...
date (1): Date Egg

ℹ Use `spec()` to retrieve the full column specification for this data.
ℹ Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Read the report line by line. `Rows: 344 Columns: 17` gives the size. `Delimiter: ","` says the values were separated by commas. Then the guesses: nine columns as `chr`, text; seven as `dbl`, numbers; one as `date`. The list after each type is cut short to fit the line. `Date Egg` became a date because its values are written year-month-day, such as 2007-11-11, which readr recognises.

Check each guess against what you know. Body mass should be a number, and it is in the `dbl` line. To make that expectation part of the script, name the column in `col_types`. Columns you do not name are still guessed, unless `.default =` inside `cols()` sets the type of every column you do not name:

```
penguins_raw <- read_csv(
  here("data-raw", "penguins_raw.csv"),
  col_types = cols(`Body Mass (g)` = col_double())
)
```

It reads quietly. When `col_types` is given, readr stops printing the report. If a later copy of this file ever arrives with "3750 g" typed in that column, this line warns, and the `problems()` list names the row.

Last, prove that reading did not touch the file. `tools::md5sum()` computes an MD5 checksum, a 32-character code calculated from every byte (the units a file is stored in) of a file. Change one character in the file and the code changes. Take it, read the file again, and take it again (`unname()` drops the file name that `md5sum()` attaches):

```
before <- tools::md5sum(here("data-raw", "penguins_raw.csv"))
penguins_raw <- read_csv(here("data-raw", "penguins_raw.csv"), show_col_types = FALSE)
after <- tools::md5sum(here("data-raw", "penguins_raw.csv"))
unname(before)
unname(before == after)
```

```
[1] "049da101568e078f9845c8b366481810"
[1] TRUE
```

Same code before and after. Write the checksum into the project's README the day a raw file arrives. Then anyone can check, later, that the file the analysis reads is the file that arrived.

WHERE THIS PICTURE BREAKS. The checksum shows that the bytes of the file are unchanged since you computed it. It says nothing about whether the data were right when they arrived.

The type report comes from a sample of rows. In a large file, a stray word far down a column may be missed by the guess. Stating `col_types` for the columns that matter closes that gap.

Illustration 3. Not every raw file is a CSV. Here are the three other kinds this book's reader meets most.

Excel. The readxl package comes with a small example workbook that is laid out the way people lay out sheets for printing, with notes above and below the table. `readxl_example()` gives the address of that file. Read it without instructions and count the rows:

```
library(readxl)
nrow(read_excel(readxl_example("deaths.xlsx")))
```

```
New names:
• `` -> `...2`
• `` -> `...3`
• `` -> `...4`
• `` -> `...5`
• `` -> `...6`
[1] 18
```

readxl gave the blank headers made-up names such as `...2`. The notes became rows: 18 rows, the real header row among them, and every column text. Give the sheet and the cells that hold the table:

```
read_excel(readxl_example("deaths.xlsx"), sheet = "other", range = "A5:F15")
```

```
# A tibble: 10 × 6
   Name           Profession   Age `Has kids` `Date of birth`
   <chr>          <chr>      <dbl> <lgl>      <dttm>
 1 Vera Rubin     scientist     88 TRUE       1928-07-23 00:00:00
 2 Mohamed Ali    athlete       74 TRUE       1942-01-17 00:00:00
 3 Morley Safer   journalist    84 TRUE       1931-11-08 00:00:00
 4 Fidel Castro   politician    90 TRUE       1926-08-13 00:00:00
 5 Antonin Scalia lawyer        79 TRUE       1936-03-11 00:00:00
 6 Jo Cox         politician    41 TRUE       1974-06-22 00:00:00
 7 Janet Reno     lawyer        78 FALSE      1938-07-21 00:00:00
 8 Gwen Ifill     journalist    61 FALSE      1955-09-29 00:00:00
 9 John Glenn     astronaut     95 TRUE       1921-07-28 00:00:00
10 Pat Summit     coach         64 TRUE       1952-06-14 00:00:00
# ℹ 1 more variable: `Date of death` <dttm>
```

Now there is one header row and ten rows of data. Ages are numbers, `<lgl>` marks TRUE or FALSE, and the two date columns are `<dttm>`, date-times, printed with a time of midnight that readxl attaches. If you want to look at an Excel file before reading it, open a copy and leave the original alone.

SPSS. The haven package comes with an SPSS copy of a classic dataset of iris flowers. `system.file()` finds a file installed with a package:

```
library(haven)
iris_spss <- read_sav(system.file("examples", "iris.sav", package = "haven"))
head(iris_spss, 3)
```

```
# A tibble: 3 × 5
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
         <dbl>       <dbl>        <dbl>       <dbl> <dbl+lbl>
1          5.1         3.5          1.4         0.2 1 [setosa]
2          4.9         3            1.4         0.2 1 [setosa]
3          4.7         3.2          1.3         0.2 1 [setosa]
```

Look at `Species`. Its type is `dbl+lbl`, a number with a label: SPSS stored the number 1 and the label `setosa`, and haven keeps both. Missing-value codes declared inside an SPSS file become `NA` when it is read, unless you ask otherwise.

SAS transport. NHANES, the National Health and Nutrition Examination Survey of the United States, publishes its files as `.xpt`. This book's project holds the body measures file of the 2021–2023 cycle. Both files are on the NHANES data pages of the US Centers for Disease Control and Prevention; download `BMX_L.xpt` and `DEMO_L.xpt` into `data-raw/`. Read the body measures file with `read_xpt()`, and its weight column's label with `attr()`, which reads an attribute:

```
bmx <- read_xpt(here("data-raw", "BMX_L.xpt"))
dim(bmx)
attr(bmx$BMXWT, "label")
```

```
[1] 8860   22
[1] "Weight (kg)"
```

The file has 8860 rows, one per person examined, and 22 columns. The column names are codes, and the label carries their meaning: `BMXWT` is weight in kg. The label came with the file; nobody typed it. Look at the labels of four columns, and count the missing values in each. `bmx[c(...)]` keeps the named columns. `sapply()` asks each column in turn for its label, and `colSums()` adds up the missing values column by column:

```
body <- bmx[c("BMXWT", "BMXHT", "BMXBMI", "BMXWAIST")]
sapply(body, attr, "label")
colSums(is.na(body))
```

```
                    BMXWT                      BMXHT
             "Weight (kg)"      "Standing Height (cm)"
                   BMXBMI                    BMXWAIST
"Body Mass Index (kg/m**2)"  "Waist Circumference (cm)"
   BMXWT    BMXHT   BMXBMI BMXWAIST
     106      361      389      670
```

BMI is missing more often than weight or height. The documentation says BMI was calculated from weight and height, so it is missing whenever either one is. Check that, with `|` meaning "or":

```
sum(is.na(bmx$BMXWT) | is.na(bmx$BMXHT))
```

```
[1] 389
```

Exactly 389 rows lack a weight or a height, the same as the rows that lack a BMI. The four counts also match the code tables printed in the survey's own codebook (its published description of every column, with a count for each value). These are counts of rows in the file, unweighted. They are not survey estimates (figures for the whole US population): the survey's documentation says its sample weights should be used to analyse these data, and no count or average in this book uses them. Sample weights are columns of the demographics file, nothing to do with body weight; another book of this series teaches them.

Figures quoted in this illustration are from [178].

WHERE THIS PICTURE BREAKS. The Excel and SPSS files are examples that come inside the readxl and haven packages. Their contents are not health data; they are there to show the reading.

The NHANES counts describe the people examined in one American survey cycle, at all ages, infants included. They say nothing about how many people in India, or in the United States, lack a measurement.

MUST KNOW POINTS FOR YOU.

- The error is to think that a file which reads without an error has been read correctly. -99 read as a height raises no warning. Read the column report, and declare every missing-value code from the file's documentation in `na` before you compute anything.
- Giving `na` replaces the default list. Write `na = c("", "NA", "-99")`, not `na = "-99"`; the second form stops reading empty cells and the text "NA" as missing.

- A number column that the report calls `chr` holds at least one value that is not a number. Never open the raw file and retype the cell.
- Record the MD5 checksum of every raw file in the README on the day it arrives.
- Open an Excel file you were sent only as a copy. Read the original into R with `read_excel()`, giving the sheet and range.
- When a department hands over an SPSS file, read it with `read_sav()` rather than asking for an Excel export.
- Any average you compute from an NHANES file without its sample weights (columns of the demographics file, nothing to do with body weight) describes the people in the file only.
- Reading carefully catches codes and types. It cannot catch a wrong value that is a valid number: a height of 182 cm typed for 128 cm reads perfectly.

EXERCISE 1 (retrieval). Without looking back, write the argument that makes `read_csv()` treat -99 as missing without losing its default missing values. Then name the function that lists the values that did not fit the types given in `col_types`.

EXERCISE 2 (design). A district team sends you `screening_2026.csv`, exported from their form. Its documentation says four things:

- height and weight are in cm and kg;
- 999 means “not measured”;
- “NR” means “not recorded”;
- `village_code` is a code with leading zeros, such as 0042.

Write the reading line, with every argument you need, and say what each argument protects you from.

EXERCISE 3 (interpretation). You read a made-up clinic file and `read_csv()` prints this report:

`Rows: 1250 Columns: 6`, then `chr (3): patient_id, sex, weight_kg` and `dbl (3): age_years, height_cm, visit_no`.

What does the report tell you, what does it not tell you, and what do you do next?

PRACTICE. Nine problems on this technique, easiest first. Work them on paper before turning to the answers, which are in the appendix at the back under this section’s name.

1. Read this made-up CSV text with `read_csv()` and print the result. Wrap the text in `I()`.

```
"district,children,with_obesity
Raipur,410,23
Durg,385,19"
```

2. Read this made-up text with `col_types` given as a short string, one letter per column: `c` for text and `d` for numbers. Print the result.

```
"ward,beds,occupied
A,30,27
B,24,24"
```

3. Read the Excel example workbook that comes with readxl, `readxl_example("deaths.xlsx")`, taking only the cells `A5:C8` of the sheet called `arts`. How many rows and columns do you get?

4. Read the penguin file with its column report showing. How many columns were read as text, as numbers and as dates? Which numeric columns would you expect, from their names?

5. Read the NHANES demographics file, `DEMO_L.xpt`, from the project's `data-raw` folder. Give its number of rows and columns, and the label of the column `RIDAGEYR`.

6. A resident wants all the measurements read as numbers. She writes this, gets a warning, and notes "child IDs missing in the file". Run it. Find what broke and why her note looks reasonable. The data are made up.

```
library(readr)
camp <- read_csv(I("child_id,height_cm,weight_kg
S01,128,26.0
S02,133,24.5
S03,131,27.5"), col_types = cols(.default = col_double()))
camp
```

```
Warning message:
One or more parsing issues, call `problems()` on your data frame for
details, e.g.:
  dat <- vroom(...)
  problems(dat)
# A tibble: 3 × 3
  child_id height_cm weight_kg
     <dbl>     <dbl>     <dbl>
1       NA       128      26
2       NA       133      24.5
3       NA       131      27.5
```

7. The penguin file has no -99 codes, but a resident adds `na = "-99"` to be safe. Then she counts the rows with no sex recorded and finds none. Run her code. Find what broke, and why the answer looked believable.

```
library(readr)
library(here)
penguins_raw <- read_csv(here("data-raw", "penguins_raw.csv"), na = "-99")
sum(is.na(penguins_raw$Sex))
```

```
here() starts at ~/project
Rows: 344 Columns: 17
── Column specification ────────────────────────────────────────────────────────
Delimiter: ","
chr  (15): studyName, Species, Region, Island, Stage, Individual ID,...
dbl   (1): Sample Number
date  (1): Date Egg

ℹ Use `spec()` to retrieve the full column specification for this data.
ℹ Specify the column types or set `show_col_types = FALSE` to quiet this message.
[1] 0
```

8. A slide at a journal club says: “NHANES 2021–2023 measured 8860 people; their mean weight was 70.5 kg.” Check the two numbers against the body measures file, and say what the slide’s sentence does not establish.

9. A colleague says: “I only opened the CSV in Excel to look at it and saved it. Nothing changed; it is still the original.” The project’s README recorded the file’s MD5 checksum on the day it arrived: `049da101568e078f9845c8b366481810`. Decide what to compute, and compute it for the project’s copy. Then show what a single hand edit does to the checksum, using a copy in `output/`. Say what the check does not establish.

10 · The data dictionary and the README

DEFINITION. A data dictionary is a separate file, laid out as a table, that explains every variable in one data file. It has one row per variable.

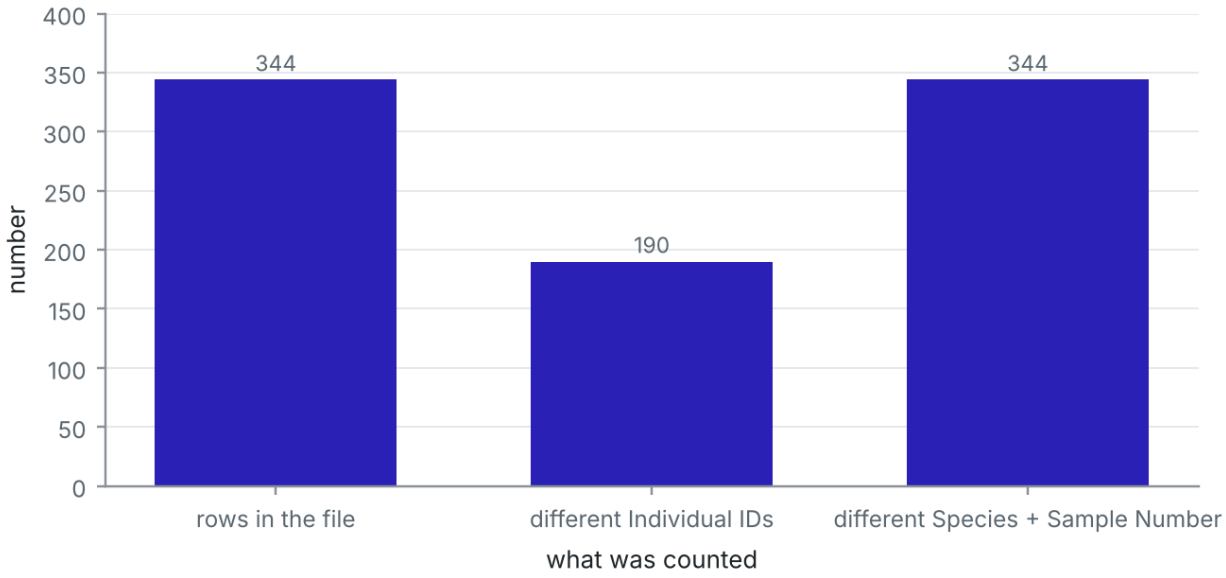
A README is a short plain-text file at the top of the project folder. It gives the one command that reruns the analysis, and the folder to run it from. [179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203]

IN PLAIN TERMS. A data dictionary is itself a small table, one row per column of the data. For each column it says:

- the exact name, letter for letter, as it is in the file;
- what the column means, in plain words;
- its unit;
- its type: number, text or date;
- the values it may take, or the range you expect;
- how a missing value is written;
- where you got each of these facts.

A README is the note on the front of the project folder. It is a plain-text file that anyone can open. Someone who has never met you should be able to read it and rerun your analysis. That someone is often you, a year later.

A project put online for others has two more files beside the README. A LICENSE says what others may do with it. A CITATION says how to cite it.



Different values in the penguin file against its 344 rows. Individual ID, which the help page calls unique, has only 190 different values, so it does not pick out one row each. Species with Sample Number has 344 different pairs, one for every row, and that is what the dictionary records as the identifier.

Illustration 1. Write a dictionary for the penguin file, and watch the documentation and the file disagree.

The documentation is the help page of the palmerpenguins package. It describes "a tibble with 344 rows and 17 variables", one paragraph each. Copy its names into the first column of your dictionary: studyName, Sample Number, Species, and on to Culmen Length, Culmen Depth, Flipper Length, Body Mass, Sex, Delta 15 N, Delta 13 C, Comments.

Now check them against the file. Read it, then ask `spec()` for the column specification that `read_csv()` used. It lists every column's name exactly as the file has it, with the type it was read as.

```
library(readr)
library(dplyr, warn.conflicts = FALSE)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)

spec(penguins_raw)

nrow(penguins_raw)
```

```
cols(
  studyName = col_character(),
  `Sample Number` = col_double(),
  Species = col_character(),
  Region = col_character(),
  Island = col_character(),
  Stage = col_character(),
  `Individual ID` = col_character(),
  `Clutch Completion` = col_character(),
  `Date Egg` = col_date(format = ""),
  `Culmen Length (mm)` = col_double(),
  `Culmen Depth (mm)` = col_double(),
  `Flipper Length (mm)` = col_double(),
  `Body Mass (g)` = col_double(),
  Sex = col_character(),
  `Delta 15 N (o/oo)` = col_double(),
  `Delta 13 C (o/oo)` = col_double(),
  Comments = col_character()
)
[1] 344
```

`warn.conflicts = FALSE` turns off the masking message that section 6 explained. In the output, `col_date(format = "")` means the date was read in readr's default form, year-month-day.

17 columns and 344 rows, as the help page said. But read the names. The file says `Culmen Length (mm)`, `Flipper Length (mm)`, `Body Mass (g)` and `Delta 15 N (o/oo)`, with a unit in brackets after the name. Six of the names you copied are not the names in the file. Any line of code written from your dictionary would fail on them. The dictionary's first column must hold the name exactly as the file has it.

The units are a second difference. The help page gives millimetres and grams in words. It gives no unit for the two isotope columns; the file's own header writes "o/oo" after them. So the unit column of your dictionary says "o/oo, as written in the column name; the help page gives no unit". Neither source says what "o/oo" stands for, so the cell copies it and does not translate it. That cell records exactly what you know and where it came from, and no more.

The third difference is in the identifiers. The help page calls `Individual ID` "the unique ID for each individual in dataset". Count how many different values the column holds:

```
n_distinct(penguins_raw$`Individual ID`)
```

```
[1] 190
```

190 different IDs in 344 rows. So an ID can appear on more than one row. The help page does not say why, and this book does not guess. What the dictionary records is what you found. Then find something that does pick out one row each. Species with sample number does:

```
n_distinct(penguins_raw$Species, penguins_raw$`Sample Number`)
```

```
[1] 344
```

344 combinations for 344 rows: one each.

Last, the allowed values and the missing code. The help page says only that Sex is "a character string". `count()` lists each value with the number of rows that hold it. Inside dplyr functions you name a column without `$`; the function looks it up in the data frame it was given.

```
penguins_raw |>
  count(Sex)
```

```
# A tibble: 3 × 2
  Sex        n
  <chr>  <int>
1 FEMALE   165
2 MALE     168
3 <NA>      11
```

`<int>` marks whole numbers (integers), and `<NA>` is how a tibble prints a missing text value. Two values, FEMALE and MALE, in capitals, and 11 rows missing. To see how the file writes a missing value, look at one line of it as plain text. `sed -n 5p`, a terminal command, prints a file's fifth line, which is its fourth row after the header. `cut -c 64-124` keeps characters 64 to 124 of it, from the Stage column to the measurements:

```
sed -n 5p data-raw/penguins_raw.csv | cut -c 64-124
```

```
"Adult, 1 Egg Stage",N2A2,Yes,2007-11-16,NA,NA,NA,NA,NA,NA
```

The fourth row, N2A2, has `NA` where its measurements should be. So the file writes a missing value as `NA`, which `read_csv()` reads as missing without being told. The quotes round `"Adult, 1 Egg Stage"` mark it as one value, so its comma does not start a new column.

Here is the dictionary, for the columns this book uses most, printed in two halves to fit the page; in the file it is one table. Each entry says where it came from.

Variable, as in the file	Meaning	Unit	Type
Species	penguin species	none	text
Island	island where the sample was taken	none	text
Individual ID	identifier of the penguin	none	text
Sample Number	running number of the sample	none	number
Culmen Length (mm)	length of the ridge along the top of the bill	mm	number
Body Mass (g)	body mass	g	number
Sex	sex of the penguin	none	text
Delta 15 N (o/oo)	ratio of the nitrogen isotopes 15N to 14N	o/oo, from the column name	number

Variable	Allowed values	Source
Species	three spellings, listed by count(Species)	help page; values from the file
Island	Biscoe, Dream, Torgersen	help page; values from the file
Individual ID	not one per row: 190 values in 344 rows	help page says unique; the file disagrees

Sample Number	with Species, one per row	help page; checked in the file
Culmen Length (mm)	expected range not stated	help page; name from the file
Body Mass (g)	expected range not stated	help page; name from the file
Sex	FEMALE, MALE	help page; values from the file
Delta 15 N (o/oo)	expected range not stated	help page; unit from the file

The full dictionary has 17 rows, one for each column, and one more fact for the whole file: missing values are written `NA`. Save it as a CSV file, `penguins_raw_dictionary.csv`, beside the raw file in `data-raw/`. Wilson and colleagues keep raw data and the facts about it (its metadata) in the same folder. You type the dictionary once, by hand, before any cleaning. No code ever writes to that folder.

Notice the cells that say "expected range not stated". Broman and Woo suggest giving the expected minimum and maximum of each variable. The help page gives none. Do not fill the cell with the smallest and largest values in this file: those describe the file, not what is possible, so they could not catch an error. Leave it saying what you know, until a source or a decision fills it.

WHERE THIS PICTURE BREAKS. The penguin file came with documentation written by the people who made it. Many files do not: a register handed over from a clinic may come with nothing. Then the dictionary is written from asking the people who collected it, and its source column says so, name and date.

A dictionary says what the values should be. It does not check them. Turning "allowed values" and "expected range" into lines of code that stop the script comes later in this book.

Illustration 2. Now the README. Picture the project folder handed to a colleague who has never seen it. They can see `data-raw/`, `code/` and `output/`. What can they not tell from the folder alone? Where the raw file came from, and whether they may use it. Which file to run, and from where. Which output is the final table. Whom to ask.

The README answers those, in that order. Type it into a plain-text file called `README.txt` at the top of the folder:

> *Penguin body mass by species*
>
> *An analysis of adult penguin body mass by species and sex, written while learning R.*
>
> *Raw data:* `data-raw/penguins_raw.csv`*, copied unchanged from the palmerpenguins R package, version 0.1.1, licence CC0. The data were collected by Palmer Station Antarctica LTER (Long Term Ecological Research) and K. Gorman, and first published by Gorman, Williams and Fraser in PLoS ONE in 2014. MD5 checksum on arrival: 049da101568e078f9845c8b366481810.*
>
> *Data dictionary:* `data-raw/penguins_raw_dictionary.csv`.
>
> *To rerun: open a terminal in this folder and type* `Rscript code/analysis.R`*. The script reads the raw file, writes everything it makes into* `output/`*, and changes nothing in* `data-raw/`.
>
> *Outputs: a table of body mass by species and a figure of it, both in* `output/`.
>
> *Written with R 4.3.3. Contact: your name and email address.*

The checksum is the one the section on reading a raw file taught you to take on the day the file arrives. A terminal is explained in section 7; how to open one depends on your computer and is not covered in this book.

CC0 is the licence the package's own files state. Wilson and colleagues describe it as the "No Rights Reserved" licence.

Read it back as the colleague. Each question they had is answered in a line. The line "To rerun" is the one that matters most. If you cannot write it yet, because the analysis needs a click somewhere or a step done by hand, the analysis cannot be rerun, and the README has told you so before anyone else finds out.

Notice what the README does not do. It does not describe each column; the dictionary does that. It does not explain each step of the analysis; the comments in the script do that. It points to both.

WHERE THIS PICTURE BREAKS. The README's "To rerun" line is a claim. It is true only if someone has run that command, in a fresh session, and got the outputs it promises. Writing the line does not make it true; running it from a clean start, later in this book, is how you find out.

The outputs line here is general, because the script is not written yet. A finished README names each output file.

MUST KNOW POINTS FOR YOU.

- The error is to think the documentation describes the file. Check every entry of your dictionary against the file in code: the names, the values, and whether an ID is really one per row.
- Write the variable name in the dictionary exactly as the file has it, letter for letter, with its spaces and brackets.
- Write the dictionary before you clean. A dictionary written after cleaning describes your cleaned data and hides what the raw file held.

- Record how missing values are written in the raw file, for each column if they differ: blank, `NA`, -99, "not recorded".
- Never fill an "expected range" cell with the smallest and largest values in your own file. Write "not stated" in it, and where you looked, until a source or a protocol gives the range.
- Write the README's rerun line as the exact command.
- When a trainee hands you a thesis dataset, ask for its dictionary before you look at a single number. If there is none, ask them to write one row: one variable, its unit, its missing code, and where they got each.
- A dictionary and a README describe the data and how to rerun the analysis. They do not make either one right.

EXERCISE 1 (design). A made-up thesis dataset, `school_heights.csv`, has these columns: `child_id`, `school`, `sex` (coded 1 and 2), `dob`, `visit_date`, `height_cm` (to 0.1 cm), `weight_kg` (to 0.1 kg). The data-entry operator typed -99 wherever a measurement was not taken. The study protocol says `sex` is 1 for boys and 2 for girls, and gives no expected ranges. Write the data dictionary.

EXERCISE 2 (critique). A colleague's project folder has this README, in full: "Analysis code for my thesis. Run the scripts in order. Data from the department." Name what is missing, and say what each gap would cost the next person.

EXERCISE 3 (retrieval). Without looking back, list what a data dictionary records for each variable, and what a README says.

EXERCISE 4 (teaching). A health secretary asks, on one page: "Why should a district survey's data come with a data dictionary? We have the data."

11 · Choosing rows and columns

DEFINITION. Four functions of the dplyr package choose parts of a data frame. These functions are called verbs, because each does one thing to a data frame. Inside them you name columns without `$`. None of them changes the data frame it was given. To keep a result, assign it to a name.

`select()` keeps the columns you name, in the order you name them, and drops the rest. Every row is kept.

`rename()` changes the names of columns you name, with `new_name = old_name`, and keeps every column, in its place.

`filter()` keeps the rows for which a condition is `TRUE`. A condition is built from comparisons (`==` equal to, `!=` not equal to, `<` less than, `>` greater than, `<=` at most, `>=` at least) joined by `&` (and), `|` (or) and `!` (not). R works out each comparison before it joins them with `&` or `|`, so each side of `|` must be a whole comparison. Several conditions separated by commas must all be `TRUE`. A row whose condition is `NA` is dropped. So a test for a missing value is written `is.na(x)`, never `x == NA`.

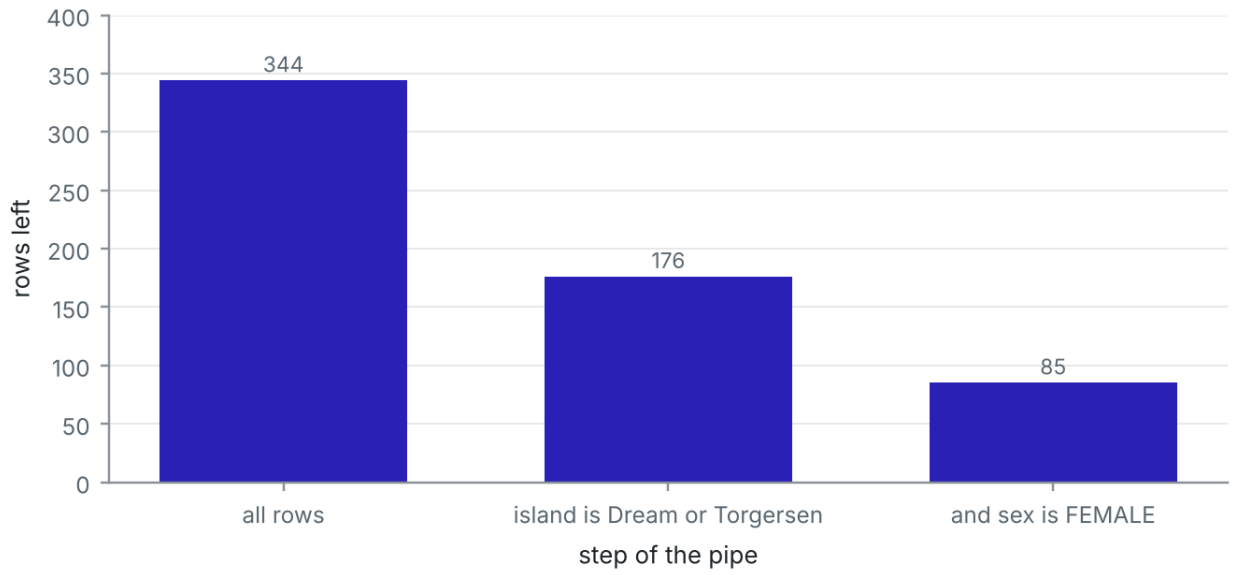
`arrange()` sorts the rows by the values of the columns named, smallest first. `desc()` sorts largest first. Missing values go to the end either way.

The pipe passes the data frame on its left to the function on its right, as that function's first argument: `x |> f(y)` is `f(x, y)`. [204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 199, 220, 221, 200]

IN PLAIN TERMS. `filter()` keeps rows. You write a test, and a row stays only if the test says `TRUE` for it.

Here is the trap that catches everyone once. A test on a missing value gives neither `TRUE` nor `FALSE`. It gives `NA`, "don't know". `filter()` drops every row where the test says "don't know". To find the missing ones, ask `is.na()`, which answers `TRUE` or `FALSE`.

Read each `|>` as "then": take the penguins, then keep these rows, then keep these columns, then sort.



Rows left after each filter() step in the search for the heaviest females on Dream and Torgersen: 344 rows in the file, 176 on Dream or Torgersen, and 85 of those with sex recorded as FEMALE. Each condition can only take rows away.

Illustration 1. Start with a question that fails. Which rows have no body mass recorded?

Load the two packages and read the raw file, as the section on reading a raw file showed. `show_col_types = FALSE` turns off the column report that `read_csv()` prints, because you have already read it once.

```
library(readr)
library(dplyr, warn.conflicts = FALSE)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)
```

`warn.conflicts = FALSE` turns off the masking message that section 6 explained. Every block in this section loads dplyr that way.

The column is called `Body Mass (g)`, with spaces and brackets. A name like that breaks R's rules for names, so you wrap it in backticks, as the section on tidy data showed. Now write the question the way it sounds in English: keep the rows where body mass equals missing.

```
penguins_raw |>
  filter(`Body Mass (g)` == NA)
```

```
# A tibble: 0 × 17
# ℹ 17 variables: studyName <chr>, Sample Number <dbl>, Species <chr>,
#   Region <chr>, Island <chr>, Stage <chr>, Individual ID <chr>,
#   Clutch Completion <chr>, Date Egg <date>, Culmen Length (mm) <dbl>,
#   Culmen Depth (mm) <dbl>, Flipper Length (mm) <dbl>,
#   Body Mass (g) <dbl>, Sex <chr>, Delta 15 N (o/oo) <dbl>,
#   Delta 13 C (o/oo) <dbl>, Comments <chr>
```

R says: a tibble with 0 rows. No error, no warning. It would be easy to write down "no row is missing its body mass".

That is wrong, and here is why. `==` compares each value with `NA`. A comparison with an unknown value is itself unknown, so the answer on every row is `NA`. `filter()` keeps a row only when the answer is `TRUE`, and drops it when the answer is `NA`. So it drops all 344.

Ask the question with `is.na()` instead. It answers `TRUE` for a missing value and `FALSE` for anything else. Keep four columns, so the answer fits on the screen.

```
penguins_raw |>
  filter(is.na(`Body Mass (g)`)) |>
  select(Species, `Individual ID`, `Body Mass (g)`, Comments)
```

```
# A tibble: 2 × 4
  Species                      `Individual ID` `Body Mass (g)` Comments
  <chr>                        <chr>                     <dbl> <chr>
1 Adelie Penguin (Pygoscelis a… N2A2                         NA Adult n…
2 Gentoo penguin (Pygoscelis p… N38A2                        NA Adult n…
```

Two rows, N2A2 and N38A2. Their `Comments` column starts "Adult n", cut short to fit the screen; in the file it reads "Adult not sampled". The empty answer above was not a finding about the penguins. It was a property of `==`.

WHERE THIS PICTURE BREAKS. `is.na()` finds what R reads as missing. A missing value written in the file as -99, or as "not recorded", is not `NA` unless you told `read_csv()` about it when you read the file. For such a column `is.na()` finds nothing, for a different reason, and the section on missing values deals with it.

Newer dplyr (1.2.0) adds `filter_out()`, which names the rows to drop; its changelog says it is "particularly useful when missing values are involved". This book's outputs come from dplyr 1.1.4.

Illustration 2. Now answer a real question in steps: among the female penguins on Dream and Torgersen islands, which were the heaviest?

First keep the columns you need, and give them names you can type. Inside `select()`, `new_name = old_name` renames as it keeps. `starts_with("Culmen")` keeps both culmen columns (the culmen is the ridge along the top of the bill) without typing either name.

```
penguins_small <- penguins_raw |>
  select(
    species = Species,
    island = Island,
    sex = Sex,
    body_mass_g = `Body Mass (g)`,
    starts_with("Culmen")
  )

penguins_small
```

```
# A tibble: 344 × 6
   species                 island sex   body_mass_g `Culmen Length (mm)`
   <chr>                   <chr>  <chr>       <dbl>                <dbl>
 1 Adelie Penguin (Pygosc… Torge… MALE         3750                 39.1
 2 Adelie Penguin (Pygosc… Torge… FEMA…        3800                 39.5
 3 Adelie Penguin (Pygosc… Torge… FEMA…        3250                 40.3
 4 Adelie Penguin (Pygosc… Torge… <NA>           NA                 NA
 5 Adelie Penguin (Pygosc… Torge… FEMA…        3450                 36.7
 6 Adelie Penguin (Pygosc… Torge… MALE         3650                 39.3
 7 Adelie Penguin (Pygosc… Torge… FEMA…        3625                 38.9
 8 Adelie Penguin (Pygosc… Torge… MALE         4675                 39.2
 9 Adelie Penguin (Pygosc… Torge… <NA>         3475                 34.1
10 Adelie Penguin (Pygosc… Torge… <NA>         4250                 42
# ℹ 334 more rows
# ℹ 1 more variable: `Culmen Depth (mm)` <dbl>
```

Still 344 rows: `select()` never drops a row. Six columns. The two culmen columns keep their old names, because `starts_with()` chose them and nothing renamed them. Long text is cut short to fit the screen: `FEMA…` is "FEMALE". A tibble prints a missing text value as `<NA>`; it is the same `NA`.

`rename()` does the renaming without dropping anything. Check that it keeps all 17 columns:

```
penguins_raw |>
  rename(body_mass_g = `Body Mass (g)`) |>
  ncol()
```

```
[1] 17
```

Now the rows. Keep the two islands with `%in%`, which asks "is this value one of these?". Then count the rows with `nrow()`.

```
penguins_small |>
  filter(island %in% c("Dream", "Torgersen")) |>
  nrow()
```

```
[1] 176
```

Add the second condition after a comma. Both must be `TRUE` for a row to stay. The file writes sex in capitals, so the test is `sex == "FEMALE"`, not `"female"`.

```
penguins_small |>
  filter(island %in% c("Dream", "Torgersen"), sex == "FEMALE") |>
  nrow()
```

```
[1] 85
```

Then sort the heaviest first, and look at the top five: `print(n = 5)` shows only the first 5 rows.

```
penguins_small |>
  filter(island %in% c("Dream", "Torgersen"), sex == "FEMALE") |>
  arrange(desc(body_mass_g)) |>
  print(n = 5)
```

```
# A tibble: 85 × 6
  species                 island sex   body_mass_g `Culmen Length (mm)`
  <chr>                   <chr>  <chr>       <dbl>                <dbl>
1 Chinstrap penguin (Pygo… Dream  FEMA…        4150                 46
2 Chinstrap penguin (Pygo… Dream  FEMA…        3950                 45.2
3 Chinstrap penguin (Pygo… Dream  FEMA…        3900                 47.5
4 Chinstrap penguin (Pygo… Dream  FEMA…        3850                 47.6
5 Adelie Penguin (Pygosce… Torge… FEMA…        3800                 39.5
# ℹ 80 more rows
# ℹ 1 more variable: `Culmen Depth (mm)` <dbl>
```

The heaviest of those 85 rows has a body mass of 4150 g, a Chinstrap penguin on Dream.

Read the pipe aloud, with "then" for each `|>`: take the small table, then keep the rows on Dream or Torgersen where sex is FEMALE, then sort by body mass, largest first.

Here is what each step left.

Step	Rows left
all rows	344
island is Dream or Torgersen	176
and sex is FEMALE	85

Last, check that none of this changed the raw data.

```
nrow(penguins_raw)
```

```
[1] 344
```

Still 344. Each verb handed back a new data frame. Only `penguins_small` was kept, because only it was given a name.

WHERE THIS PICTURE BREAKS. The 85 is a count of rows in this file with "FEMALE" in the sex column. Eleven rows have no sex recorded, and some of them may be female. So 85 is the number of rows with sex recorded as FEMALE on those two islands, not a count of female birds.

The count holds for this file only. It says nothing about how many penguins live on either island.

MUST KNOW POINTS FOR YOU.

- The error is to write `filter(x == NA)` to find missing values. Write `filter(is.na(x))`.
- When a filter returns 0 rows, check the test before you report the zero: print the values you tested against.
- `filter()` drops a row whose test is `NA`. Whenever you report a count after a filter, say whether missing values were dropped by it.
- Never type a filter into the raw file by deleting rows. Write the `filter()` line in the script.
- A verb that is not assigned changes nothing.

- Rename awkward column names once, near the top of the script, into snake case (lowercase words joined by _).
- `filter()` keeps the rows that pass the test you wrote. It cannot tell you whether that test matches your question.

EXERCISE 1 (teaching). A first-year resident has ten minutes and a whiteboard. Their code `filter(hb == NA)` on an anaemia survey file returned no rows, and they have written "no missing haemoglobin values" in their results. Explain what happened and how to check.

EXERCISE 2 (retrieval). Without looking back, take each of `select()`, `rename()`, `filter()` and `arrange()` in turn. Say what it does to the rows and to the columns. Then say what it does to the data frame you gave it.

EXERCISE 3 (design). Write the first block of your analysis script after the line that reads the penguin file. It should keep the species, island, sex, flipper length and body mass columns, and give each a snake-case name that carries its unit. Show that the result has 344 rows and 5 columns.

PRACTICE. Eleven problems on this technique, easiest first. Work them on paper before turning to the answers, which are in the appendix at the back under this section's name.

1. A made-up clinic register has three rows. dplyr makes `tibble()` available too.

```
library(dplyr, warn.conflicts = FALSE)

clinic <- tibble(
  id = c(101, 102, 103),
  sex = c("F", "M", "F"),
  weight_kg = c(61.2, 78.5, NA)
)
```

Write one line for each. (a) Keep the rows for women. (b) Keep only the `id` and `weight_kg` columns, in that order. (c) Sort it heaviest first; before you run it, say where the row with no weight will go.

2. Read the penguin file.

```
library(readr)
library(dplyr, warn.conflicts = FALSE)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)
```

How many rows are Chinstrap penguins? The species is written in the file exactly as "Chinstrap penguin (Pygoscelis antarctica)".

3. Read the penguin file.

```
library(readr)
library(dplyr, warn.conflicts = FALSE)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)
```

How many rows from Biscoe island have a flipper length of 210 mm or more? Give the number with its unit of count and its condition.

4. Read the penguin file.

```
library(readr)
library(dplyr, warn.conflicts = FALSE)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)
```

The two isotope columns are `Delta 15 N (o/oo)` and `Delta 13 C (o/oo)`. How many rows are missing both? How many are missing at least one?

5. A colleague wants the number of rows for Adelie penguins and writes:

```
library(readr)
library(dplyr, warn.conflicts = FALSE)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)

penguins_raw |>
  filter(Species == "Adelie penguin (Pygoscelis adeliae)") |>
  nrow()
```

```
[1] 0
```

They report “there are no Adelie penguins in the file”. Find what broke.

6. A colleague wants the rows with no sex recorded and writes:

```
library(readr)
library(dplyr, warn.conflicts = FALSE)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)

penguins_raw |>
  filter(Sex == NA) |>
  nrow()
```

```
[1] 0
```

They write in their notes: “sex was recorded for every penguin”. Find what broke, and give the right count.

7. A colleague wants the male and female penguins on Dream island and writes:

```
library(readr)
library(dplyr, warn.conflicts = FALSE)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)
```

```
penguins_raw |>
  filter(Island == "Dream", Sex == "MALE" | "FEMALE")
```

```
Error in `filter()`:
ℹ In argument: `Sex == "MALE" | "FEMALE"`.
Caused by error in `Sex == "MALE" | "FEMALE"`:
! operations are possible only for numeric, logical or complex types
```

Read the error. Say what broke and fix it.

8. A colleague wants the mean body mass without the two missing values and writes:

```
library(readr)
library(dplyr, warn.conflicts = FALSE)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)

penguins_raw |>
  filter(!is.na(`Body Mass (g)`))

mean(penguins_raw$`Body Mass (g)`)
```

```
# A tibble: 342 × 17
   studyName `Sample Number` Species Region Island Stage `Individual ID`
   <chr>               <dbl> <chr>   <chr>  <chr>  <chr> <chr>
 1 PAL0708                 1 Adelie… Anvers Torge… Adul… N1A1
 2 PAL0708                 2 Adelie… Anvers Torge… Adul… N1A2
 3 PAL0708                 3 Adelie… Anvers Torge… Adul… N2A1
 4 PAL0708                 5 Adelie… Anvers Torge… Adul… N3A1
 5 PAL0708                 6 Adelie… Anvers Torge… Adul… N3A2
 6 PAL0708                 7 Adelie… Anvers Torge… Adul… N4A1
 7 PAL0708                 8 Adelie… Anvers Torge… Adul… N4A2
 8 PAL0708                 9 Adelie… Anvers Torge… Adul… N5A1
 9 PAL0708                10 Adelie… Anvers Torge… Adul… N5A2
10 PAL0708                11 Adelie… Anvers Torge… Adul… N6A1
# ℹ 332 more rows
# ℹ 10 more variables: `Clutch Completion` <chr>, `Date Egg` <date>,
#   `Culmen Length (mm)` <dbl>, `Culmen Depth (mm)` <dbl>,
#   `Flipper Length (mm)` <dbl>, `Body Mass (g)` <dbl>, Sex <chr>,
#   `Delta 15 N (o/oo)` <dbl>, `Delta 13 C (o/oo)` <dbl>,
#   Comments <chr>
[1] NA
```

The `filter()` printed 342 rows, so they expected a number. They got `NA`. Find what broke.

9. A colleague wants the heaviest penguin in the file and writes:

```
library(readr)
library(dplyr, warn.conflicts = FALSE)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)

penguins_raw |>
  arrange(`Body Mass (g)`) |>
  select(Species, `Body Mass (g)`) |>
  print(n = 1)
```

```
# A tibble: 344 × 2
  Species                                   `Body Mass (g)`
  <chr>                                              <dbl>
1 Chinstrap penguin (Pygoscelis antarctica)           2700
# ℹ 343 more rows
```

They report “the heaviest penguin was a Chinstrap at 2700 g”. Find what broke.

10. A line in a made-up field report says: “More than half of the Adelie penguins on Torgersen were female.” Check it against the file, and say what your answer does not establish.

11. In a made-up lab meeting, someone says: “The Gentoo penguins over 6 kg were all males.” Turn this into code, run it, and say what the result does not establish.

12 · Making new columns

DEFINITION. `mutate()`, from the dplyr package, adds columns to a data frame. A new column goes on the right. If its name is the name of an existing column, it replaces that column.

`if_else(condition, true, false)` gives one value where the condition is `TRUE` and another where it is `FALSE`. Where the condition is `NA`, the result is `NA`, unless a value is given for `missing`.

`case_when()` takes a list of cases, each written `condition ~ value`. It checks them in order, and each row gets the value of the first case whose condition is `TRUE`. A row that matches no case gets `.default` if one is given, written `.default = "label"` after the cases, and `NA` otherwise.

`round(x, digits)` rounds to that many decimal places. A 5 at the cut is rounded to the even digit, so `round(2.5)` is 2.

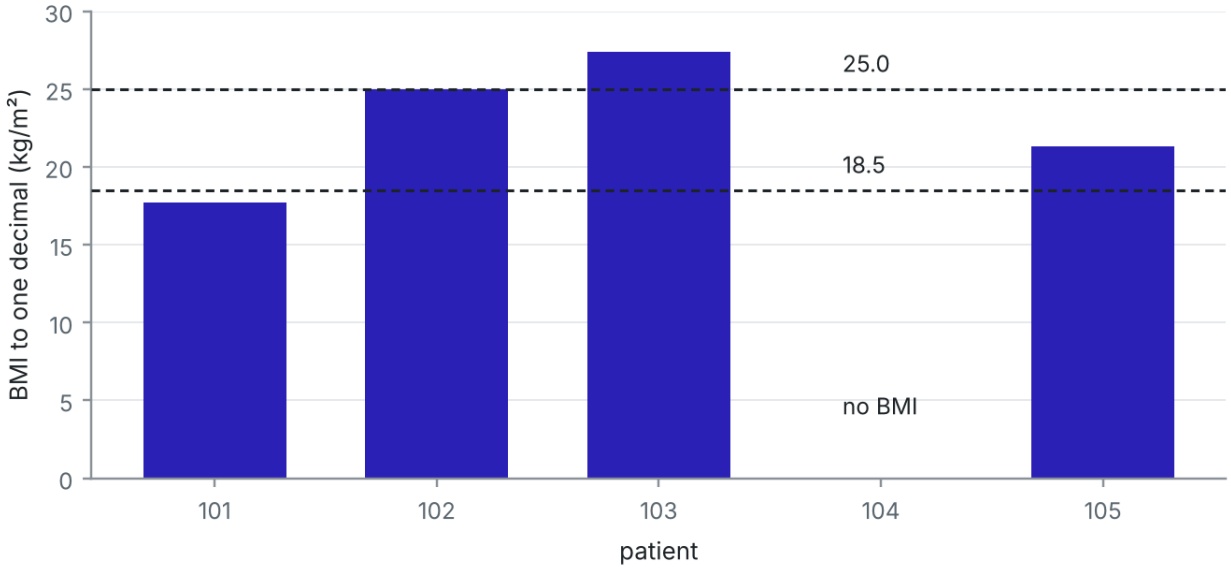
`str_trim()` removes spaces from the start and end of a piece of text. `str_to_lower()` turns text into lower case. Both come from the stringr package. [222, 223, 224, 225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 200]

IN PLAIN TERMS. `mutate()` adds a column. You give the new column a name and a rule, and R applies the rule to every row. The number of rows does not change.

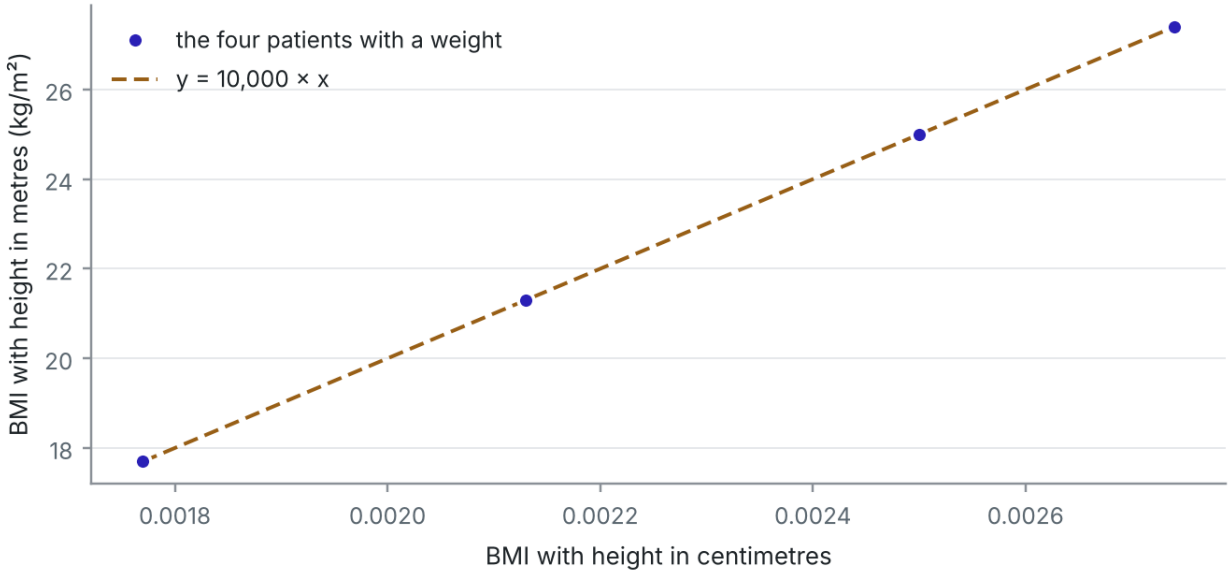
`if_else()` sorts rows into two groups by one test. `case_when()` sorts them into several groups. A row with a missing value passes no test and gets `NA`, which is what you want. Be careful with `.default`, the label for "everything else": a missing value lands there too.

`round()` is for showing a number, not for storing it. Keep full precision in the column you compute with, and round only what you print.

Text columns get cleaned the same way. `str_trim()` takes spaces off the ends. `str_to_lower()` turns capitals into small letters. Together they turn "Male", "male" and "MALE" into "male".



The four made-up patients with a BMI, at one decimal place, against the two NFHS-5 cut-points: 18.5 and 25.0 kg/m^2. Patient 102 sits exactly on 25.0, which the survey's rule puts in the upper group. Patient 104 has no weight, so no BMI.



Each patient's BMI worked out with height in centimetres (across) against the same BMI with height in metres (up). Every point lies on the line y = 10,000 × x: a height 100 times larger, squared, is 10,000 times larger, so every BMI from centimetres is 10,000 times too small.

Illustration 1. Start with BMI, on a made-up clinic register of five adults. Heights are in centimetres, as a clinic writes them; weights are in kilograms. Patient 104 has no weight recorded. dplyr makes `tibble()` available too, and `warn.conflicts = FALSE` turns off the masking message that section 6 explained.

```
library(dplyr, warn.conflicts = FALSE)

clinic <- tibble(
  id = c(101, 102, 103, 104, 105),
  height_cm = c(152, 160, 171, 158, 165),
  weight_kg = c(41, 64, 80.2, NA, 58)
)
```

Write Book 0's rule straight in, with the height as it is:

```
clinic |>
  mutate(bmi = weight_kg / height_cm^2)
```

```
# A tibble: 5 × 4
     id height_cm weight_kg      bmi
  <dbl>     <dbl>     <dbl>    <dbl>
1   101       152      41    0.00177
2   102       160      64    0.0025
3   103       171      80.2  0.00274
4   104       158      NA    NA
5   105       165      58    0.00213
```

The code runs and every BMI is about 0.002. Nobody has a BMI of 0.002. The rule wants height in metres, and a height in centimetres is 100 times larger. Squared, it is 10,000 times larger, so every BMI came out 10,000 times too small. The order-of-magnitude check from Book 0 catches it in one glance.

Convert first, in its own column, then use it. Within one `mutate()`, a column made on one line can be used on the next.

```
clinic <- clinic |>
  mutate(
    height_m = height_cm / 100,
    bmi = weight_kg / height_m^2
  )

clinic
```

```
# A tibble: 5 × 5
     id height_cm weight_kg height_m   bmi
  <dbl>     <dbl>     <dbl>    <dbl> <dbl>
1   101       152      41       1.52  17.7
2   102       160      64       1.6   25
3   103       171      80.2     1.71  27.4
4   104       158      NA       1.58  NA
5   105       165      58       1.65  21.3
```

Now the numbers are BMIs, in kg/m^2. Patient 104 has `NA`: a missing weight gives a missing BMI, which is honest. The height in centimetres is still there beside the height in metres. Nothing was overwritten, so anyone reading the script can check the conversion row by row.

WHERE THIS PICTURE BREAKS. The five patients are made up, and their BMIs describe nobody. The code is the point, not the numbers.

The conversion is right only if every height in the column is in centimetres. A register where some heights were written in metres would give a mix of sensible and absurd BMIs, and no line of code here would notice. Checking a range in code, so the script stops on a height of 1.6 cm, comes later in this book.

Illustration 2. Now put the patients into groups. The National Family Health Survey, NFHS-5 (2019–21), uses two BMI cut-points in its India fact sheet. A BMI below 18.5 kg/m^2 is its "below normal" group. A BMI of 25.0 kg/m^2 or more is its "overweight or obese" group. Notice the boundary rule: 25.0 itself goes in the upper group, and 18.5 itself does not go in the lower one.

Write the three groups with `case_when()`, labelled by their ranges. The cases are checked in order, so the second case only sees rows that failed the first.

```
clinic <- clinic |>
  mutate(
    bmi_group = case_when(
      bmi < 18.5 ~ "below 18.5",
      bmi < 25 ~ "18.5 to below 25",
      bmi >= 25 ~ "25 or more"
    )
  )

clinic |>
  select(id, bmi, bmi_group)
```

```
# A tibble: 5 × 3
     id   bmi bmi_group
  <dbl> <dbl> <chr>
1   101  17.7 below 18.5
2   102  25   18.5 to below 25
3   103  27.4 25 or more
4   104  NA   <NA>
5   105  21.3 18.5 to below 25
```

Look at patient 102. The BMI prints as 25, and the group says "18.5 to below 25". By the survey's rule, 25.0 belongs in "25 or more". Something is wrong.

Ask R whether that BMI equals 25, and then print it with more digits.

```
clinic$bmi[2] == 25

print(clinic$bmi[2], digits = 17)
```

```
[1] FALSE
[1] 24.999999999999996
```

In `print()`, `digits` counts all the digits shown, 17 here, not decimal places as in `round()`.

The stored number is a hair below 25. On paper, 64 divided by 1.6^2 is exactly 25. But a computer stores numbers in binary, and many decimals, 1.6 among them, have no exact binary form, in the same way that one third has no exact decimal form. The tiny error survives the arithmetic,

and `bmi < 25` is `TRUE`. The help page for `round()` warns of this: rounding applies to the number as stored, not as printed.

Here is the trap. Printed digits are not stored digits, and a cut-point is exactly where a hair matters. The fix is a rule you choose and write down. The survey states its cut-points to one decimal place, so round the BMI to one decimal in a separate column, and group on that column. `round(bmi, 1)` replaces the hair-below number with the number R stores for `25`, so `bmi_1dp < 25` is `FALSE`.

```
clinic <- clinic |>
  mutate(
    bmi_1dp = round(bmi, 1),
    bmi_group = case_when(
      bmi_1dp < 18.5 ~ "below 18.5",
      bmi_1dp < 25 ~ "18.5 to below 25",
      bmi_1dp >= 25 ~ "25 or more"
    )
  )

clinic |>
  select(id, bmi_1dp, bmi_group)
```

```
# A tibble: 5 × 3
     id bmi_1dp bmi_group
  <dbl>   <dbl> <chr>
1   101    17.7 below 18.5
2   102    25   25 or more
3   103    27.4 25 or more
4   104    NA   <NA>
5   105    21.3 18.5 to below 25
```

Patient 102 is now in “25 or more”. Patient 104, with no BMI, has `NA` as a group: every test was `NA`, no case matched, and `case_when()` gives `NA` when no case matches and no `.default` is set. The full-precision `bmi` column is still there for any calculation.

Reusing the name `bmi_group` replaced the first, wrong grouping. That is acceptable here, because `bmi_group` is a column your own script made two steps earlier, not a column from the file. Never reuse the name of a column that came from the raw file.

The four patients with a BMI, as grouped:

Patient	BMI to one decimal (kg/m^2)	Group
101	17.7	below 18.5
102	25.0	25 or more
103	27.4	25 or more
105	21.3	18.5 to below 25

WHERE THIS PICTURE BREAKS. Rounding to one decimal before grouping is a rule chosen here because the cut-points are written to one decimal. It is not taken from the fact sheet, which does not say how it rounds. A protocol has to state its own rule, and the code has to follow it.

The labels name ranges, not conditions. Which range means what for a person's health is a decision made by a body, set out in the books on measurement and on obesity in adults, not in this one. The survey's cut-points apply to the adults it covers.

Illustration 3. The same moves work on the penguin file. Its body mass is in grams; suppose your report wants kilograms. Its sex column is in capitals. Its species names are long. Fix all three in one `mutate()`, each into a new column.

```
library(readr)
library(dplyr, warn.conflicts = FALSE)
library(stringr)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)

penguins <- penguins_raw |>
  mutate(
    body_mass_kg = `Body Mass (g)` / 1000,
    sex = str_to_lower(Sex),
    species = case_when(
      Species == "Adelie Penguin (Pygoscelis adeliae)" ~ "Adelie",
      Species == "Chinstrap penguin (Pygoscelis antarctica)" ~ "Chinstrap",
      Species == "Gentoo penguin (Pygoscelis papua)" ~ "Gentoo"
    )
  )

penguins |>
  select(Species, species, Sex, sex, `Body Mass (g)`, body_mass_kg)
```

```
# A tibble: 344 × 6
   Species              species Sex   sex    `Body Mass (g)` body_mass_kg
   <chr>                <chr>   <chr> <chr>            <dbl>        <dbl>
 1 Adelie Penguin (Pyg… Adelie  MALE  male              3750         3.75
 2 Adelie Penguin (Pyg… Adelie  FEMA… fema…             3800         3.8
 3 Adelie Penguin (Pyg… Adelie  FEMA… fema…             3250         3.25
 4 Adelie Penguin (Pyg… Adelie  <NA>  <NA>                NA        NA
 5 Adelie Penguin (Pyg… Adelie  FEMA… fema…             3450         3.45
 6 Adelie Penguin (Pyg… Adelie  MALE  male              3650         3.65
 7 Adelie Penguin (Pyg… Adelie  FEMA… fema…             3625         3.62
 8 Adelie Penguin (Pyg… Adelie  MALE  male              4675         4.68
 9 Adelie Penguin (Pyg… Adelie  <NA>  <NA>              3475         3.48
10 Adelie Penguin (Pyg… Adelie  <NA>  <NA>              4250         4.25
# ℹ 334 more rows
```

Each new column sits beside the column it came from. `sex` and `Sex` are two different columns, because R tells capitals from small letters.

A `case_when()` with no `.default` gives `NA` to any row it did not match. So check that every row matched. Count the new column with `count()`, which lists each value with the number of rows that hold it (the section on categories teaches it):

```
penguins |>
  count(species)
```

```
# A tibble: 3 × 2
  species       n
  <chr>     <int>
1 Adelie      152
2 Chinstrap    68
3 Gentoo      124
```

152, 68 and 124, and no `NA` row. Every species name in the file was caught by one of the three cases. If a fourth spelling had crept in, it would show up here as `NA`, and you would go and look.

Last, the column names. `rename_with()` applies a function to every name. Give it `str_to_lower`. Written without brackets, a function is handed over to be used on each name, not called.

```
penguins_raw |>
  rename_with(str_to_lower) |>
  names()
```

```
 [1] "studyname"           "sample number"       "species"
 [4] "region"              "island"              "stage"
 [7] "individual id"       "clutch completion"   "date egg"
[10] "culmen length (mm)"  "culmen depth (mm)"   "flipper length (mm)"
[13] "body mass (g)"       "sex"                 "delta 15 n (o/oo)"
[16] "delta 13 c (o/oo)"   "comments"
```

The names are now in lower case, but `body mass (g)` still has spaces and brackets, so it still needs backticks. Making a name safe in one step needs text patterns that this book does not teach. For a file with 17 columns, name the ones you use with `rename()` or inside `select()`, as the last section did.

WHERE THIS PICTURE BREAKS. Dividing by 1000 is exact only because grams and kilograms differ by a fixed factor. A column that mixed units, some rows in grams and some in kilograms, would need a test on each row before any conversion, and dividing every row by 1000 would hide the problem.

The three species labels work because the file holds exactly three spellings. The count proves that for this file, and only for this file.

MUST KNOW POINTS FOR YOU.

- Convert units in code, in a new column, before using them.
- A BMI computed from height in centimetres is 10,000 times too small.
- The error is to think a value that prints as 25 is 25. At a cut-point, decide a rounding rule, write it in a comment, and group on the rounded column.
- The order of the cases in `case_when()` is part of the rule. With `bmi >= 18.5` first and `bmi >= 25` second, nobody is ever put in "25 or more", and no error says so.
- Leave `.default` out unless you mean it. With `.default` set, a missing value is given that label as if it had been measured.

- Give a new column a new name.
- Round for the table you print, not for the column you compute with.
- A new column is only as right as the rule you wrote.

EXERCISE 1 (critique). A colleague's thesis dataset has a weight column in kilograms. They opened the spreadsheet, typed a formula into a new column to compute BMI, filled it down, and saved the file. Then they read it into R. Say what is wrong with this, and what they should do instead.

EXERCISE 2 (retrieval). Without looking back, name four ways a new column made with `mutate()` and `case_when()` can be wrong without any error message.

EXERCISE 3 (calculation). Add to your penguin script a column of flipper length in centimetres. Add another that labels each penguin's body mass as "4.5 kg or more" or "under 4.5 kg". The cut-point is chosen for practice; it is not a biological threshold. Count how many rows get each label, including any missing.

PRACTICE. Thirteen problems on this technique, easiest first. Work them on paper before turning to the answers, which are in the appendix at the back under this section's name.

1. A made-up table of three heights:

```
library(dplyr, warn.conflicts = FALSE)

heights <- tibble(height_cm = c(150, 162.5, 178))
```

Add a column `height_m` with the heights in metres.

2. A made-up table of three weights:

```
library(dplyr, warn.conflicts = FALSE)

weights <- tibble(weight_kg = c(72, 55, NA))
```

Add a column that says "60 or more" or "under 60". Before you run it, say what the third row will get.

3. Before you run it, say what this prints.

```
round(c(0.5, 1.5, 2.5))
```

4. Read the penguin file.

```
library(readr)
library(dplyr, warn.conflicts = FALSE)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)
```

Add a column of flipper length in centimetres. What is the first penguin's flipper length in centimetres?

5. Read the penguin file.

```
library(readr)
library(dplyr, warn.conflicts = FALSE)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)
```

Work out body mass in kilograms. Then put each penguin in one of three groups. The groups are "under 3.5 kg", "3.5 to under 4.5 kg" and "4.5 kg or more", cut-points chosen for practice. Count the rows in each group.

6. A made-up register typed sex by hand:

```
library(dplyr, warn.conflicts = FALSE)
library(stringr)

register <- tibble(sex = c("Male", " female", "MALE ", "Female", "male"))
```

How many different spellings are there now? Clean the column into a new one, and count the values of the new one.

7. A colleague computes BMI on a made-up register:

```
library(dplyr, warn.conflicts = FALSE)

register <- tibble(
  height_cm = c(158, 172, 149),
  weight_kg = c(62, 85, 47)
)

register |>
  mutate(bmi = weight_kg / height_cm^2) |>
  mutate(below_18_5 = bmi < 18.5)
```

```
# A tibble: 3 × 4
  height_cm weight_kg     bmi below_18_5
      <dbl>     <dbl>   <dbl> <lgl>
1       158        62 0.00248 TRUE
2       172        85 0.00287 TRUE
3       149        47 0.00212 TRUE
```

They report "all three patients have a BMI below 18.5 kg/m^2". Find what broke.

8. A colleague groups BMI on a made-up register:

```
library(dplyr, warn.conflicts = FALSE)

register <- tibble(bmi = c(17.2, 23.9, 26.4, 31.0))

register |>
  mutate(
    bmi_group = case_when(
      bmi < 18.5 ~ "below 18.5",
      bmi >= 18.5 ~ "18.5 to below 25",
      bmi >= 25 ~ "25 or more"
    )
  )
```

```
# A tibble: 4 × 2
    bmi bmi_group
  <dbl> <chr>
1  17.2 below 18.5
2  23.9 18.5 to below 25
3  26.4 18.5 to below 25
4  31   18.5 to below 25
```

They report "nobody in the register has a BMI of 25 or more". Find what broke.

9. A colleague groups BMI on a made-up register, and wants every row to have a group:

```
library(dplyr, warn.conflicts = FALSE)

register <- tibble(bmi = c(17.2, 23.9, NA, 26.4))

register |>
  mutate(
    bmi_group = case_when(
      bmi < 18.5 ~ "below 18.5",
      bmi < 25 ~ "18.5 to below 25",
      .default = "25 or more"
    )
  ) |>
  count(bmi_group)
```

```
# A tibble: 3 × 2
  bmi_group            n
  <chr>            <int>
1 18.5 to below 25     1
2 25 or more           2
3 below 18.5           1
```

They report "2 of the 4 patients have a BMI of 25 or more". Find what broke.

10. A colleague converts body mass to kilograms and reports the mean:

```
library(readr)
library(dplyr, warn.conflicts = FALSE)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)

penguins <- penguins_raw |>
  mutate(`Body Mass (g)` = `Body Mass (g)` / 1000)

mean(penguins$`Body Mass (g)`, na.rm = TRUE)
```

```
[1] 4.201754
```

Their table says "mean body mass 4.2 g". Find what broke.

11. A colleague writes:

```
library(dplyr, warn.conflicts = FALSE)

register <- tibble(
  height_cm = c(158, 172, 149),
  weight_kg = c(62, 85, 47)
)
```

```
register |>
  mutate(bmi = weight_kg / height_m^2) |>
  mutate(height_m = height_cm / 100)
```

```
Error in `mutate()`:
ℹ In argument: `bmi = weight_kg/height_m^2`.
Caused by error:
! object 'height_m' not found
```

Read the error. Say what broke and fix it.

12. A made-up clinic audit says: "2 of our 5 patients had a BMI of 25.0 kg/m^2 or more." Here is the register.

```
library(dplyr, warn.conflicts = FALSE)

audit <- tibble(
  id = c(1, 2, 3, 4, 5),
  height_cm = c(160, 155, 170, 165, 150),
  weight_kg = c(64, 52, 75, 62, 54)
)
```

Check the claim, and say what your answer does not establish.

13. A made-up draft thesis says: "Mean body mass of the Gentoo penguins was 5.08 kg." The penguin file records body mass in grams. Turn the sentence into code, run it, and say what the result does not establish.

13 · Missing values: counting them and saying how many

DEFINITION. In R, a missing value is the constant `NA`, "not available". R turns a code into `NA` only when it is told to, at reading or in a later line of code.

`is.na(x)` returns `TRUE` for each element of `x` that is `NA`. In arithmetic, `TRUE` counts as 1 and `FALSE` as 0, so `sum(is.na(x))` is the number of missing values in `x`.

`mean(x, na.rm = TRUE)` and similar summaries leave out the missing values. The number of values behind the result is then `sum(!is.na(x))`, not `length(x)`.

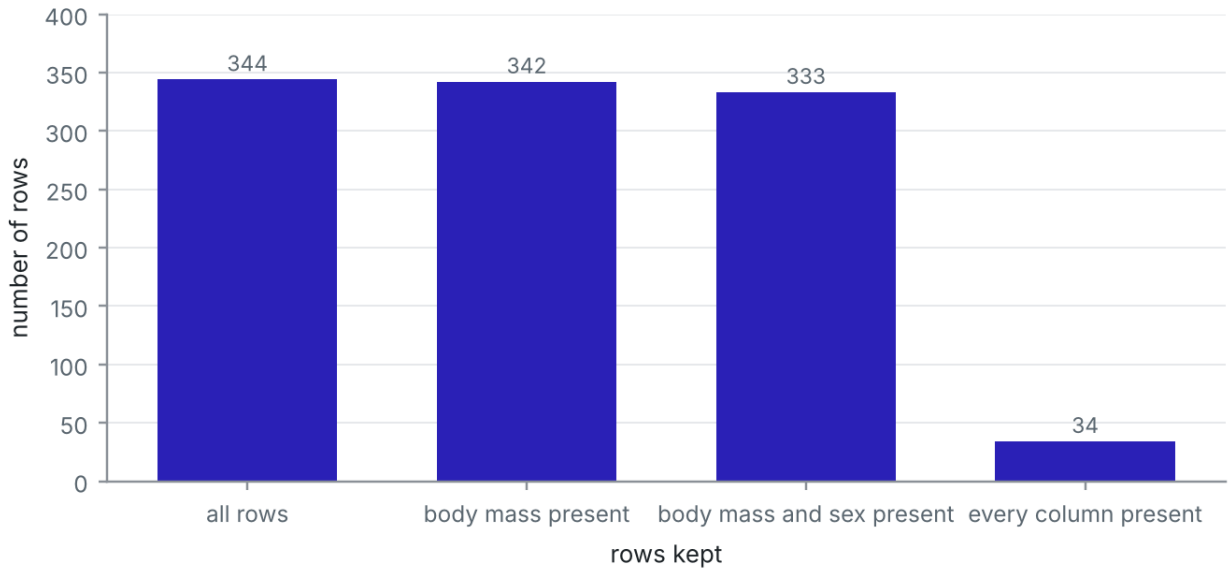
`drop_na()`, from the tidyr package, keeps only the rows with no missing value in the columns it is given, or in every column when it is given none. Analysing only such rows is called complete-case analysis. [237, 238, 239, 240, 241, 242, 243, 244, 245, 246, 247]

IN PLAIN TERMS. A missing value reaches R in one of two ways. Either it is already `NA`, or it arrives disguised as a real value: a weight of 999, a height of -99, the words "not recorded". R cannot tell a disguised code from a measurement.

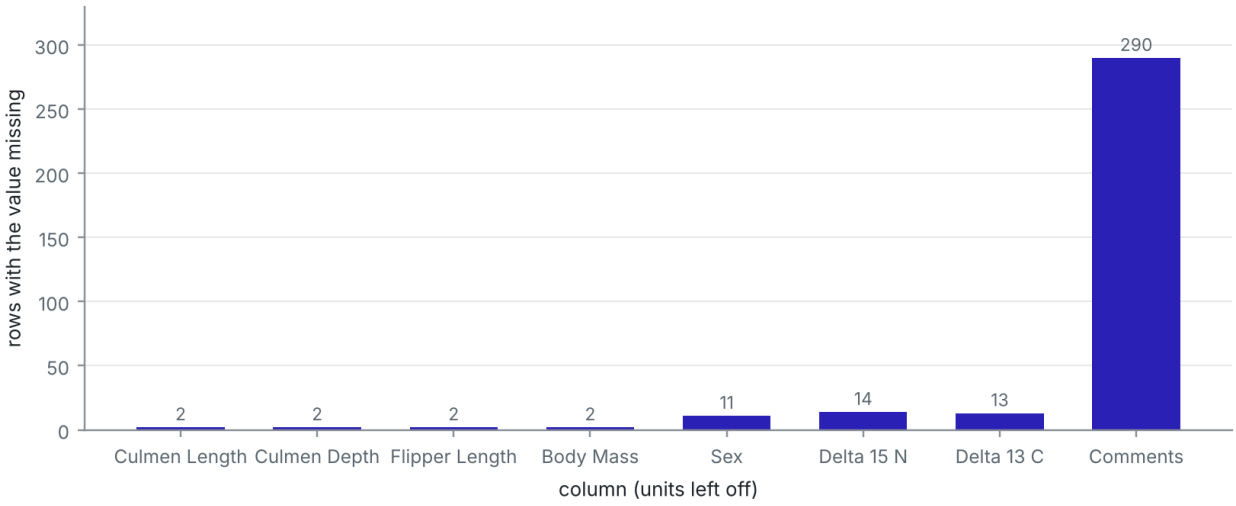
Read the data dictionary, then look at the largest and smallest values of each column. Turn every code into `NA` in your script, never in the file.

Then count. `sum(is.na(x))` counts the missing values in one column. `colSums(is.na(data))` counts them in every column at once.

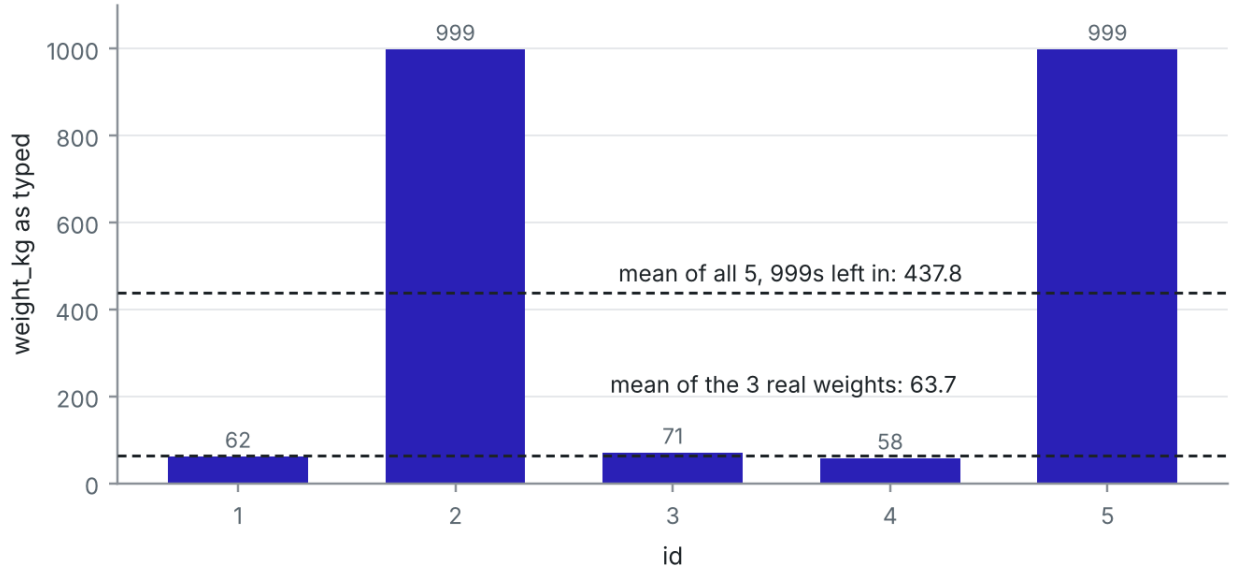
Dropping rows with `drop_na()` is a choice, not a clean-up. It changes which rows your answer is about.



Rows of the penguin file kept by four choices: all 344 rows; 342 with a body mass; 333 with a body mass and a recorded sex; 34 with no gap in any column. The last drop is driven by the Comments column, which is empty in 290 rows.



Missing values in each column of the penguin file that has any, as colSums(is.na()) counts them. Comments, a free note, is empty in 290 rows; every measurement column misses 14 or fewer. This is why drop_na() with no columns named keeps only 34 rows.



The made-up clinic register's weights as typed, with 999 where the scale was not available. Left in, the two 999s pull the mean of all 5 values to 437.8 kg; turned into NA, the mean of the 3 real weights is 63.7 kg.

Illustration 1. Start with the step that looks tidiest and does the most damage. Read the penguin file and keep only the rows with no missing value, using `drop_na()` with no columns named.

```
library(readr)
library(tidyr)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)

complete <- drop_na(penguins_raw)

nrow(penguins_raw)
nrow(complete)
```

```
[1] 344
[1] 34
```

Of 344 rows, 34 survive. Nine rows in ten are gone. Any summary of `complete` describes 34 rows, and nothing in its output says so.

Count the missing values in each column to see why. `is.na()` on the whole data frame gives a grid of `TRUE` and `FALSE`. `colSums()` adds up each column of the grid, counting each `TRUE` as 1.

```
colSums(is.na(penguins_raw))
```

```
          studyName       Sample Number             Species
                  0                   0                   0
             Region              Island               Stage
                  0                   0                   0
      Individual ID   Clutch Completion           Date Egg
                  0                   0                   0
 Culmen Length (mm)   Culmen Depth (mm) Flipper Length (mm)
                  2                   2                   2
      Body Mass (g)                 Sex   Delta 15 N (o/oo)
                  2                  11                  14
  Delta 13 C (o/oo)            Comments
                 13                 290
```

The `Comments` column is empty in 290 rows. It holds a note only when the field team had something to add, so a gap there is not a missing measurement at all. Yet `drop_na()` with no columns named treats it as one, and every row without a comment is thrown out.

Name the columns your question needs. For a question about body mass, that is one column:

```
nrow(drop_na(penguins_raw, `Body Mass (g)`))
```

```
[1] 342
```

342 rows: only the 2 with no body mass are left out. For a question about body mass and sex, name both:

```
nrow(drop_na(penguins_raw, `Body Mass (g)`, Sex))
```

```
[1] 333
```

333 rows. Each column you add can take more rows away.

WHERE THIS PICTURE BREAKS. The counts are of rows in this one file, which records nesting observations, not a survey of all penguins. They say how many rows each choice keeps, not whether the rows kept are like the rows left out.

A gap in `Comments` is harmless here because the column is a free note. In another file an empty cell can be a real missing measurement. The data dictionary says which.

Illustration 2. Now the disguised code. Here is a made-up register of five adults at a clinic, already read into R. Whoever entered it typed 999 when the scale was not available.

```
library(dplyr)

register <- tibble(
  id = c(1, 2, 3, 4, 5),
  weight_kg = c(62, 999, 71, 58, 999)
)

mean(register$weight_kg)
```

```
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

[1] 437.8
```

A mean weight of 437.8 kg. Nothing warned you, because 999 is a perfectly good number. Book 0's order-of-magnitude check catches it in one glance: no clinic's mean adult weight is in the hundreds.

The best place to fix this is the reading step: in section 9 you declared codes with `na =` in `read_csv()`. When the data are already in R, perhaps from a file type that has no such argument, fix them in code. Make a new column with `if_else()`, keeping the old one beside it:

```
register <- register |>
  mutate(weight_kg_clean = if_else(weight_kg == 999, NA, weight_kg))

register
```

```
# A tibble: 5 × 3
     id weight_kg weight_kg_clean
  <dbl>     <dbl>           <dbl>
1     1        62              62
2     2       999              NA
3     3        71              71
4     4        58              58
5     5       999              NA
```

Now count and summarise. `summarise()` collapses a data frame into one row of summaries; a later section uses it for groups. Here it gives three numbers together: the rows, the missing values, and the mean of the rest. `n()` counts the rows.

```
register |>
  summarise(
    rows = n(),
    missing = sum(is.na(weight_kg_clean)),
    mean_weight_kg = mean(weight_kg_clean, na.rm = TRUE)
  )
```

```
# A tibble: 1 × 3
   rows missing mean_weight_kg
  <int>   <int>          <dbl>
1     5       2           63.7
```

The tibble prints the mean as 63.7 kg, and it is the mean of 3 weights, not 5. Write it that way: "mean weight 63.7 kg (n = 3; 2 of 5 not weighed)". Book 0's mean is a sum divided by a count, and the count is part of the answer.

WHERE THIS PICTURE BREAKS. The five adults and their weights are made up. The numbers show the arithmetic, not anything about any clinic.

Replacing 999 is right only if 999 is never a real value in that column. For weight in kilograms it cannot be. For a column such as a cholesterol reading, 999 could be real, which is why the code list comes from the data dictionary, not from guessing.

Illustration 3. Last, watch the count change the answer. Take the mean body mass twice: once from every row with a body mass, and once from only those whose sex is also recorded, as a complete-case analysis on those two columns would.

```
library(readr)
library(dplyr)
library(tidyr)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)

penguins_raw |>
  summarise(
    n = sum(!is.na(`Body Mass (g)`)),
    mean_mass_g = mean(`Body Mass (g)`, na.rm = TRUE)
  )

penguins_raw |>
  drop_na(`Body Mass (g)`, Sex) |>
  summarise(
    n = n(),
    mean_mass_g = mean(`Body Mass (g)`)
  )
```

```
# A tibble: 1 × 2
      n mean_mass_g
  <int>       <dbl>
1   342       4202.
# A tibble: 1 × 2
      n mean_mass_g
  <int>       <dbl>
1   333       4207.
```

Two means, 4202 g from 342 rows and 4207 g from 333. The dot after each number means the tibble rounded it to fit the column. The difference is small here. It is still a difference, made by a choice about rows, not by anything measured. The 9 rows left out the second time have a body mass and no recorded sex, and nobody can tell from this file whether they were heavier or lighter than the rest.

That is why the n goes beside every number. A reader who sees “4207 g (n = 333)” next to “344 rows in the file” can ask where the other 11 went. A reader who sees “4207 g” cannot.

WHERE THIS PICTURE BREAKS. Here the two means are close. That does not mean dropping rows is safe in general: in another file the rows with gaps may be very different from the rest, and the means far apart. Whether that happens depends on why the values are missing, which is the subject of a later book on missing data.

MUST KNOW POINTS FOR YOU.

- A code such as 999 or -99 is an ordinary number to R until you say otherwise.
- Count missing values with `sum(is.na(x))`, or `colSums(is.na(data))` for every column.
- Beside every mean, median or percentage, write the n it was computed from. With `na.rm = TRUE` that n is `sum(!is.na(x))`, not the number of rows.
- `drop_na()` with no columns named drops a row for a gap in any column, including a free-text note. Always name the columns, and print `nrow()` before and after.

- Turn codes into `NA` in a new column made by code, never by editing the raw file.
- Say in the methods which columns you required, and how many rows that left.
- Counting missing values tells you how many are gone, never why. A count cannot show whether the people with gaps differ from the rest.

EXERCISE 1 (calculation). In your penguin script, count the missing values of flipper length. Then give the mean flipper length in millimetres, with the number of rows it was computed from, written the way a table would show it.

EXERCISE 2 (critique). A made-up thesis methods section says: "Participants with missing data were excluded. Mean BMI was 26.1 kg/m^2." The dataset had 412 participants and 28 columns. Say what is missing from these two sentences, and rewrite them.

EXERCISE 3 (retrieval). Without looking back, write the line that counts the missing values in one column `x`. Then write the line that counts them in every column of a data frame `d`. Last, say what `drop_na()` does when no column is named.

PRACTICE. Eleven problems on this technique, easiest first. Work them on paper before turning to the answers, which are in the appendix at the back under this section's name.

1. How many missing values does this made-up vector hold? Write the line that counts them, and say what it prints before you run it.

```
x <- c(3, NA, 7, NA, NA, 2)
```

2. Before you run them, say what each line prints, and how many values the second mean is computed from.

```
weights <- c(54, 61, NA, 70)
mean(weights)
mean(weights, na.rm = TRUE)
```

3. A made-up three-row table:

```
library(tidyr)

d <- tibble::tibble(
  height_cm = c(150, NA, 162),
  weight_kg = c(48, 55, NA)
)
```

How many rows does `drop_na(d)` keep? How many does `drop_na(d, height_cm)` keep? Say both before you run them.

4. Read the penguin file.

```
library(readr)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)
```

How many rows have no recorded sex? What percentage of the rows is that, to one decimal place?

5. Read the penguin file.

```
library(readr)
library(dplyr, warn.conflicts = FALSE)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)
```

In one `summarise()`, give the number of rows, the number of missing values of `Delta 15 N (o/oo)`, and the mean of the values present. Write the result as a table would show it. The data's help page describes Delta 15 N as a measure of the ratio of two forms of nitrogen, taken from a blood sample. Its unit, o/oo, is in the column name.

6. Read the penguin file.

```
library(readr)
library(tidyr)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)
```

A question needs culmen length, culmen depth and both blood measurements, `Delta 15 N (o/oo)` and `Delta 13 C (o/oo)`. How many rows have all four? How many does that leave out?

7. A colleague's made-up register used -9 for "not weighed". They report: "Mean weight was 40 kg, after removing missing values."

```
weights <- c(58, -9, 66, 71, -9, 63)
mean(weights[!is.na(weights)])
```

```
[1] 40
```

Run it. Find what broke, and why the report looks reasonable.

8. A made-up camp file has a height column where the field worker wrote "not recorded" for one child. A resident reads it and asks for the mean height:

```
library(readr)

camp <- read_csv(I("id,height_cm
1,131
2,not recorded
3,128"), show_col_types = FALSE)

mean(camp$height_cm)
```

```
[1] NA
Warning message:
In mean.default(camp$height_cm) :
  argument is not numeric or logical: returning NA
```

They report: “height could not be summarised; the data are missing”. Find what broke.

9. A colleague wants the mean body mass of the penguins whose sex is known. They write:

```
library(readr)
library(tidyr)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)

known <- drop_na(penguins_raw)

nrow(known)
mean(known$`Body Mass (g)`)
```

```
[1] 34
[1] 3877.206
```

Their table says “mean body mass of penguins of known sex, 3877.206 g”. Find what broke.

10. A made-up draft report says: “Mean body mass was 4202 g in the 344 penguins studied.” Check the sentence against the penguin file, rewrite it, and say what your rewritten sentence does not establish.

11. In a made-up department meeting, a resident presenting the penguin analysis says: “We did a complete-case analysis, so missing data were not a problem.” The analysis needs body mass, flipper length and sex. Find out what the complete-case analysis did, and say what the resident’s sentence claims that the data cannot show.

14 · Categories as factors

DEFINITION. A factor stores each value together with its levels: the list of allowed categories, in a set order.

`factor(x, levels = ...)` makes a factor with the levels given. A value of `x` that is not among the levels becomes `NA`, without an error. With no levels given, they are taken from the data in alphabetical order.

Three functions come from the forcats package. The first, `fct_recode()`, renames levels, written `new = "old"`, and leaves the others as they are. The second, `fct_relevel()`, moves the named levels to the front, written `fct_relevel(x, "level")`. The third, `fct_infreq()`, orders the levels by how many rows have each one, largest first. Each also takes a text column, and makes it a factor first, with its levels in alphabetical order.

`count()`, from dplyr, gives the number of rows for each value of one column, or for each combination of values of two or more columns. For a factor, `.drop = FALSE` also lists the combinations that never occur, with a count of 0. [248, 249, 250, 251, 252, 253, 198, 254, 255]

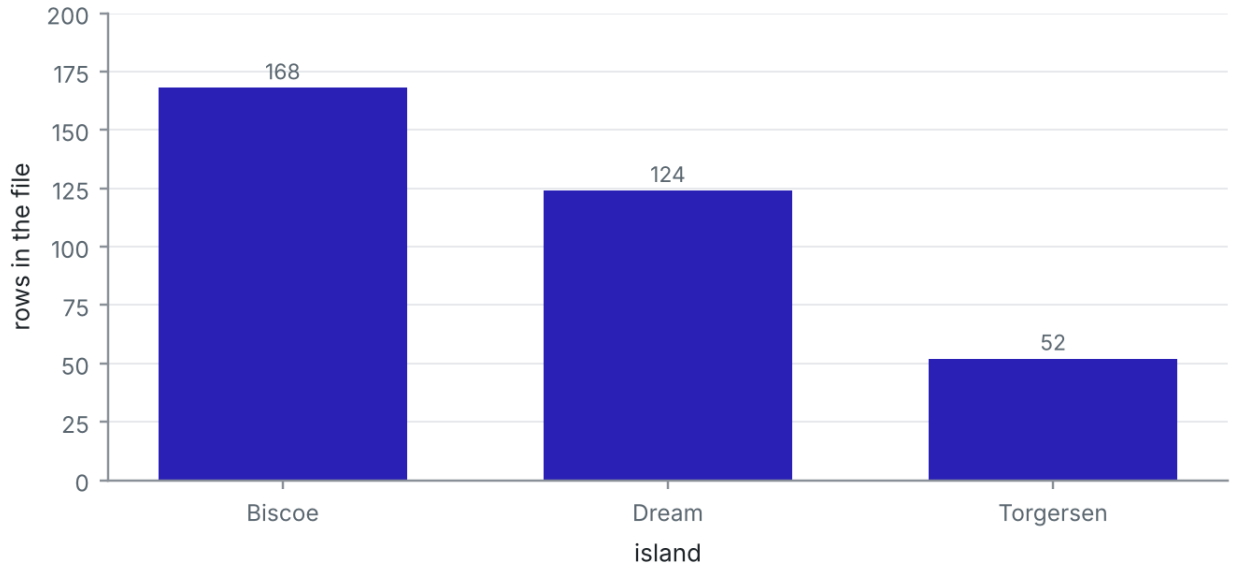
IN PLAIN TERMS. "Male", "male" with a space at the end, and "M" are three different values to R.

`count()` shows you every value a column actually holds, and how many rows hold it. A misspelling shows up as an extra row.

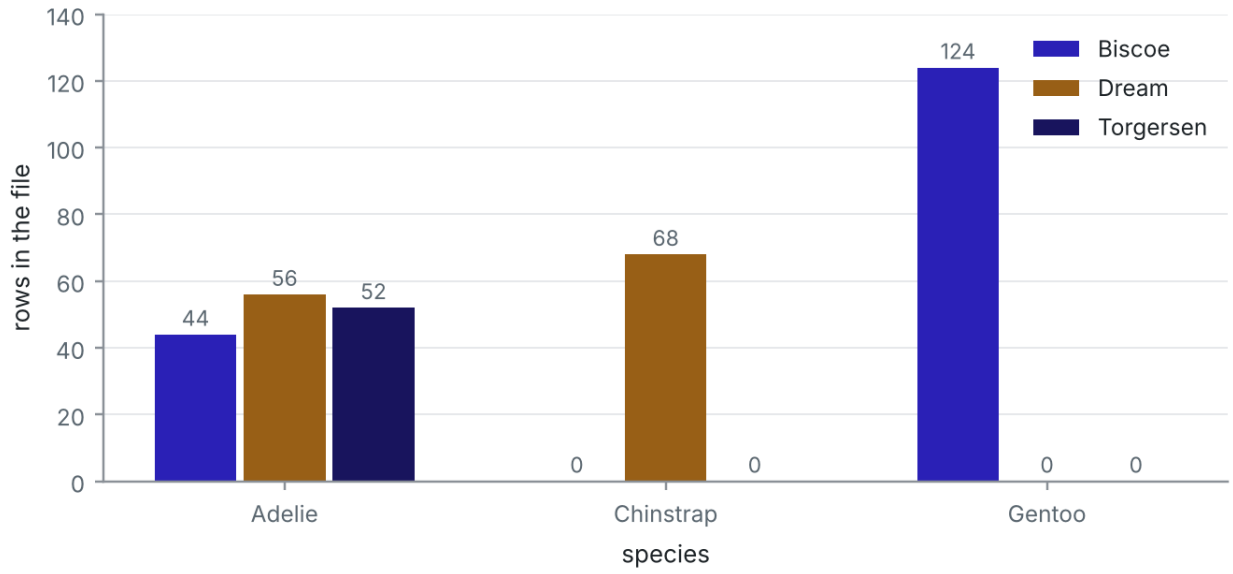
Here is the trap. When you make a factor and a value is not on your list, R turns it into `NA` and says nothing.

Codes need the same care. A file may store sex as 1 and 2. Never guess the codes from the data.

Two counts at once, `count(species, island)`, is Book 0's two-way table, written as one row per pair.



Rows of the penguin file on each island, in the order fct_infreq() gives: Biscoe 168, Dream 124, Torgersen 52. The order follows the counts; a fixed order from fct_relevel() would not.



Rows of the penguin file for each species on each island, as count(species, island, .drop = FALSE) gives them. The zeros are pairs that never occur in this file: Chinstrap rows come only from Dream, Gentoo rows only from Biscoe, and only Adelie rows come from all three islands.

Illustration 1. Start with a column typed by hand. Here is a made-up register of six patients, with sex as the staff wrote it.

```
library(dplyr)
library(stringr)
library(forcats)

register <- tibble(
  id = c(1, 2, 3, 4, 5, 6),
  sex = c("Male", "male ", "Female", "F", "female", "MALE")
)
```

```
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
```

You want two categories, female and male. Make a factor with those two levels, and count it:

```
register |>
  mutate(sex_f = factor(sex, levels = c("female", "male"))) |>
  count(sex_f)
```

```
# A tibble: 2 × 2
  sex_f      n
  <fct>  <int>
1 female     1
2 <NA>       5
```

One female, no male, and five missing. Not one value was missing in the register. `factor()` turned every value that did not match a level exactly into `NA`, and it gave no warning. A report built on this would say five patients' sex was "not recorded".

Go back a step and count the text as it is:

```
register |>
  count(sex)
```

```
# A tibble: 6 × 2
  sex          n
  <chr>    <int>
1 "F"          1
2 "Female"     1
3 "MALE"       1
4 "Male"       1
5 "female"     1
6 "male "      1
```

Six rows, six spellings, each held once. This is the count to run first, on every category column.

Now clean it in steps, each into a new column. Trim the spaces and lower the case, as the section on making new columns did. Then rename the one code that is left, “f”, with `fct_recode()`: the new name on the left, the old one on the right.

```
register <- register |>
  mutate(
    sex_clean = str_to_lower(str_trim(sex)),
    sex_f = fct_recode(sex_clean, female = "f"),
    sex_f = factor(sex_f, levels = c("female", "male"))
  )

register |>
  count(sex_f)
```

```
# A tibble: 2 × 2
  sex_f      n
  <fct>  <int>
1 female     3
2 male       3
```

Three female and three male, and no `NA`. The last line sets the order of the levels, so female comes first in every count and table, whatever order the data arrive in.

WHERE THIS PICTURE BREAKS. The six patients are made up. The spellings are the kind that hand entry produces; Broman and Woo’s advice on data entry warns against exactly this mix of “male” and ” male “.

Lowering the case and trimming spaces catches only these slips. “Mle” or “fem” would survive both steps, and the final `factor()` would turn them into `NA`. The count after the factor is what catches them, so never skip it.

Illustration 2. Now codes. Many forms store a category as a number. Here is a made-up thesis dataset of five students, where sex is stored as 1 or 2. The data dictionary for this made-up study says 1 means female and 2 means male.

```
library(dplyr)
library(forcats)

students <- tibble(
  id = c(101, 102, 103, 104, 105),
  sex_code = c(1, 2, 2, 1, 2)
)

mean(students$sex_code)
```

```
[1] 1.6
```

R gives a mean sex of 1.6. The arithmetic is right and the number means nothing: 1 and 2 are labels, not amounts. A numeric code invites exactly this kind of summary.

Turn the code into a factor with the dictionary’s labels. `fct_recode()` works on text, so turn the numbers into text first with `as.character()`. Keep the code column beside the new one.

```
students <- students |>
  mutate(
    sex = fct_recode(as.character(sex_code), female = "1", male = "2")
  )

students

students |>
  count(sex)
```

```
# A tibble: 5 × 3
     id sex_code sex
  <dbl>    <dbl> <fct>
1   101        1 female
2   102        2 male
3   103        2 male
4   104        1 female
5   105        2 male
# A tibble: 2 × 2
  sex        n
  <fct>  <int>
1 female     2
2 male       3
```

Two female, three male. The column type now reads `<fct>`, a factor. Every label came from the dictionary, written once in the script, where anyone can check it against the dictionary.

WHERE THIS PICTURE BREAKS. The codes and their meanings are made up for this example. Another study may use 1 for male, or 0 and 1, or M and F. The labels in your script must come from your own data dictionary, never from this page or from habit.

Illustration 3. Last, the penguin file. It has three category columns you will use often: species, island and sex. Count the islands first.

```
library(readr)
library(dplyr)
library(forcats)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)

penguins_raw |>
  count(Island)
```

```
# A tibble: 3 × 2
  Island        n
  <chr>     <int>
1 Biscoe      168
2 Dream       124
3 Torgersen    52
```

The column is text, `<chr>`, so `count()` lists the islands alphabetically. A factor is something you choose to make.

Make the factors. Shorten the species names with `fct_recode()`, and order the islands by how many rows each has with `fct_infreq()`.

```
penguins <- penguins_raw |>
  mutate(
    species = fct_recode(
      Species,
      Adelie = "Adelie Penguin (Pygoscelis adeliae)",
      Chinstrap = "Chinstrap penguin (Pygoscelis antarctica)",
      Gentoo = "Gentoo penguin (Pygoscelis papua)"
    ),
    island = fct_infreq(Island)
  )

levels(penguins$island)

penguins |>
  count(island)
```

```
[1] "Biscoe"    "Dream"     "Torgersen"
# A tibble: 3 × 2
  island        n
  <fct>     <int>
1 Biscoe      168
2 Dream       124
3 Torgersen    52
```

`levels()` lists a factor's levels in their order. Biscoe comes first now, because it has the most rows. The counts are the same; only the order changed.

Island	Rows
Biscoe	168
Dream	124
Torgersen	52

Now two columns at once. `count(species, island)` gives one row for each pair that occurs. With `.drop = FALSE`, it also gives the pairs that never occur, with 0:

```
penguins |>
  count(species, island, .drop = FALSE)
```

```
# A tibble: 9 × 3
  species   island        n
  <fct>     <fct>     <int>
1 Adelie    Biscoe       44
2 Adelie    Dream        56
3 Adelie    Torgersen    52
4 Chinstrap Biscoe        0
5 Chinstrap Dream        68
6 Chinstrap Torgersen     0
7 Gentoo    Biscoe      124
8 Gentoo    Dream         0
9 Gentoo    Torgersen     0
```

Nine rows: three species times three islands. Laid out as Book 0's two-way table, with species down the side and islands across the top, the same counts read:

Species	Biscoe	Dream	Torgersen
Adelie	44	56	52
Chinstrap	0	68	0
Gentoo	124	0	0

The zeros carry information. In this file, Chinstrap penguins were recorded only on Dream, and Gentoo only on Biscoe. Without `.drop = FALSE`, those zeros would not appear at all, and you might not notice that they exist.

WHERE THIS PICTURE BREAKS. The counts are rows of the file, which are nesting observations on the islands sampled. A zero says no such row was recorded. It does not say that no Chinstrap penguin lives on Biscoe.

`fct_infreq()` orders by the counts in this file. Another season's file could order the islands differently, which matters if a report compares tables across files. A fixed order from `fct_relevel()` or from the levels you list yourself does not move.

MUST KNOW POINTS FOR YOU.

- Run `count()` on every category column before you use it.
- `factor()` does not warn when a value matches no level. Count the factor after making it, and treat any new `NA` as a spelling you missed.
- Refuse a summary that averages a code.
- If you misspell an old level in `fct_recode()`, R warns "Unknown levels" and leaves that level unchanged.
- Set the order of the levels on purpose, with the levels you list or with `fct_relevel()`.
- A table that leaves out its zeros hides that a combination never occurred.
- A clean count shows how many rows hold each category. It cannot show that the category was recorded correctly in the first place.

EXERCISE 1 (calculation). In your penguin script, make `sex` a factor with the levels "female" and "male", in that order, from the file's `Sex` column. Count it, and say how many rows have no recorded sex.

EXERCISE 2 (retrieval). Without looking back, say what `factor()` does with a value that is not among the levels you gave it. Then name the line you run to catch that.

EXERCISE 3 (teaching). A first-year resident has ten minutes and a whiteboard. Their thesis data has a column `tobacco` coded 0, 1 and 2, and they have reported "mean tobacco use 0.8". Explain what is wrong, and show them how to fix it.

PRACTICE. Ten problems on this technique, easiest first. Work them on paper before turning to the answers, which are in the appendix at the back under this section's name.

1. Before you run it, say in what order the levels come.

```
factor(c("Dream", "Biscoe", "Dream", "Torgersen"))
```

2. Before you run it, say what the levels of the result are, in order.

```
library(forcats)

fct_infreq(c("rice", "roti", "rice", "millet", "rice", "roti"))
```

3. A made-up column codes the meal eaten as 1 (breakfast), 2 (lunch) or 3 (dinner). Turn it into a factor with those labels.

```
meal_code <- c(2, 3, 3, 1, 2)
```

4. Read the penguin file.

```
library(readr)
library(dplyr, warn.conflicts = FALSE)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)
```

Count the rows for each value of `Clutch Completion`. What does a "No" mean, according to the data's help page?

5. Read the penguin file.

```
library(readr)
library(dplyr, warn.conflicts = FALSE)
library(forcats)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)
```

Make a factor of island in which Torgersen comes first and the other islands keep their alphabetical order. Count it.

6. A colleague makes a factor from a made-up register and reports: "2 of the 6 patients were female."

```
library(dplyr, warn.conflicts = FALSE)

register <- tibble(sex = c("female", "Female", "male", "female", "Female ", "male"))

register |>
  mutate(sex = factor(sex, levels = c("female", "male"))) |>
  count(sex)
```

```
# A tibble: 3 × 2
  sex        n
  <fct>  <int>
1 female     2
2 male       2
3 <NA>       2
```

Find what broke, and why the report looks reasonable.

7. A colleague shortens the species names in the penguin file:

```
library(readr)
library(dplyr, warn.conflicts = FALSE)
library(forcats)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)

penguins_raw |>
  mutate(
    species = fct_recode(
      Species,
      Adelie = "Adelie Penguin (Pygoscelis adeliae)",
      Chinstrap = "Chinstrap Penguin (Pygoscelis antarctica)",
      Gentoo = "Gentoo penguin (Pygoscelis papua)"
    )
  ) |>
  count(species)
```

```
# A tibble: 3 × 2
  species                                     n
  <fct>                                   <int>
1 Adelie                                    152
2 Chinstrap penguin (Pygoscelis antarctica)    68
3 Gentoo                                    124
Warning message:
There was 1 warning in `mutate()`.
ℹ In argument: `species = fct_recode(...)`.
Caused by warning:
! Unknown levels in `f`: Chinstrap Penguin (Pygoscelis antarctica)
```

Their table has three rows, labelled “Adelie”, “Chinstrap penguin (Pygoscelis antarctica)” and “Gentoo”. They decide the file’s Chinstrap name is “too long to fix” and leave it. Find what broke.

8. A made-up study’s data dictionary says sex is coded 1 = male, 2 = female. A resident writes:

```
library(dplyr, warn.conflicts = FALSE)
library(forcats)

study <- tibble(sex_code = c(2, 2, 1, 2, 1, 2, 2))

study |>
  mutate(sex = fct_recode(as.character(sex_code), female = "1", male = "2")) |>
  count(sex)
```

```
# A tibble: 2 × 2
  sex        n
  <fct>  <int>
1 female     2
2 male       5
```

Their results say “5 of the 7 participants were male”. Find what broke.

9. A made-up draft report says: “Of the 333 penguins with sex recorded, 168 were male.” Check the sentence against the penguin file and say what it does not establish.

10. In a made-up meeting, someone says: “The penguin data show that Gentoo penguins live only on Biscoe.” Turn the claim into a count, run it, and say what the result does and does not establish.

15 · Dates

DEFINITION. In R, a date is an object of class `Date`; `class(x)` names the kind of object `x` is. It is stored as a number of days since 1 January 1970 and printed in the ISO 8601 form, year-month-day, such as 2021-04-03.

Text becomes a date by parsing: reading it according to the order of its parts. The lubridate functions are named for that order. `ymd()` reads year, month, day; `dmy()` reads day, month, year; `mdy()` reads month, day, year. Each accepts any separator. Text that does not fit the order becomes `NA`, with a warning: how many failed, or that all of them did. `read_csv()` recognises dates already written as year-month-day and leaves other forms as text.

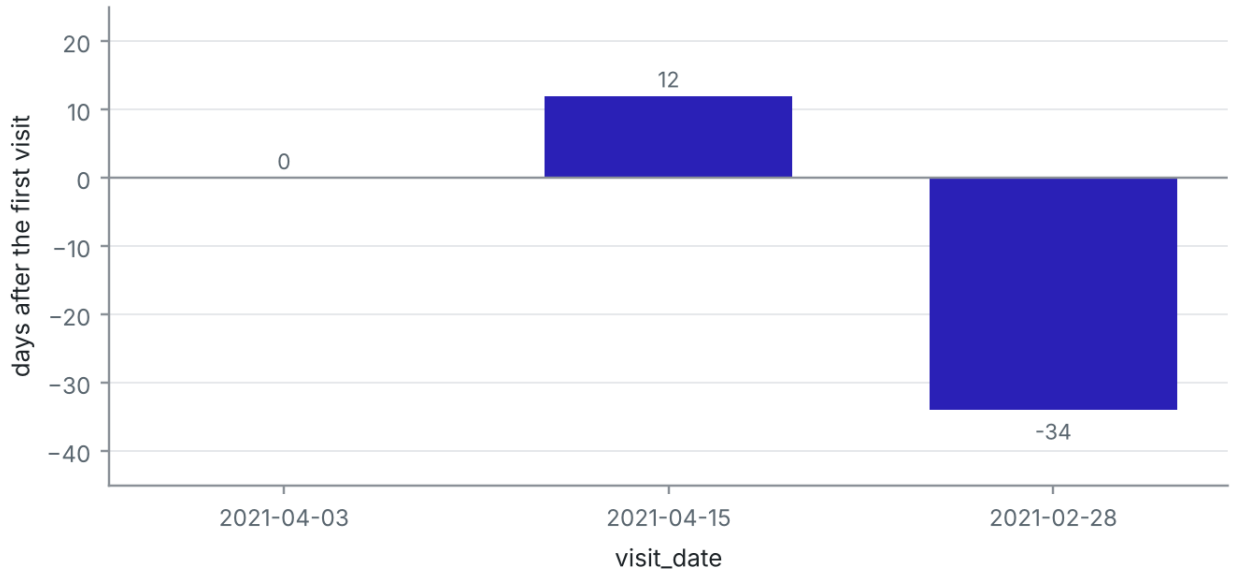
Subtracting one date from another gives a time difference in days. Adding a number to a date gives the date that many days later. [256, 257, 258, 259, 260, 261, 262, 263, 264, 265, 266, 6, 267, 268]

IN PLAIN TERMS. To R, a date is a number of days, counted from 1 January 1970, that prints as year-month-day. Because it is a number underneath, you can subtract one date from another and get days.

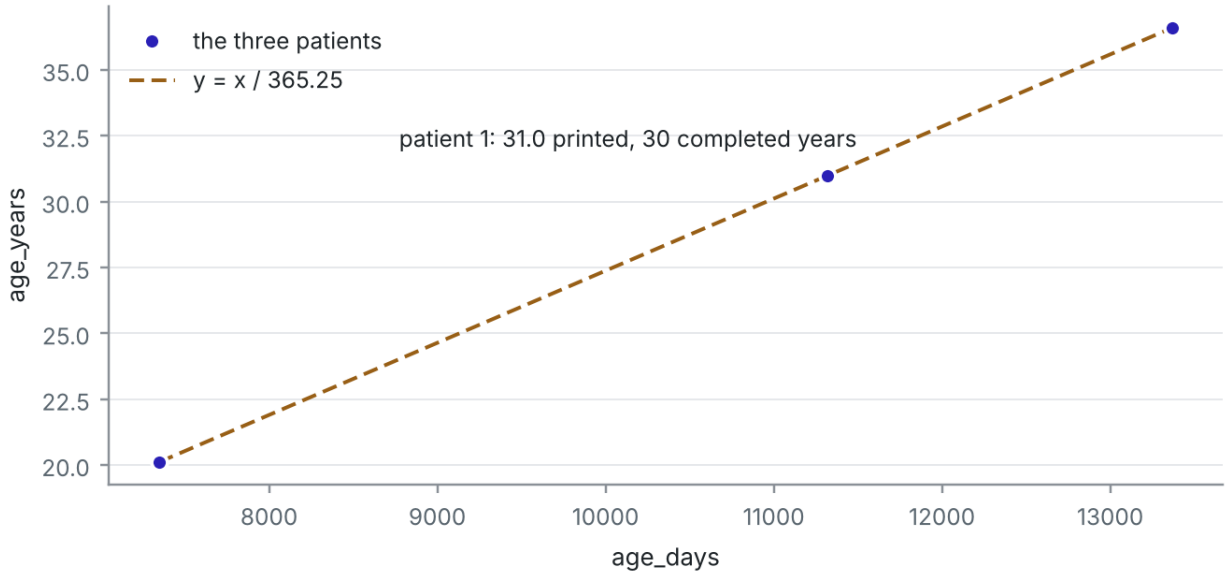
A date usually arrives as text, such as “03/04/2021”. Is that 3 April or 4 March? The text cannot tell you.

Here is the trap. If you pick the wrong order, some dates come out wrong and still look like dates. Others fail and become `NA`.

Broman and Woo recommend the ISO 8601 form, year-month-day: 2021-04-03.



Days from the first camp visit to each visit, from subtracting one Date from another: 0 for the first visit itself, 12 for the visit of 2021-04-15, and -34 for the visit of 2021-02-28, which came before the first.



Age at the visit in days against age in years for the three made-up patients. Every point lies on y = x / 365.25. Patient 1's 31.0 is printed rounded; the stored value is just under 31, so floor() gives 30 completed years.

Illustration 1. Start with a made-up camp register whose visit dates were typed with the day first. Here is its text, read with `I()` to mark it as data rather than a file name, as an earlier section did.

```
library(readr)
library(dplyr, warn.conflicts = FALSE)
library(lubridate)

camp <- read_csv(I("id,visit
1,03/04/2021
2,15/04/2021
3,28/02/2021"), show_col_types = FALSE)

camp
```

```
Attaching package: 'lubridate'

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

    date, intersect, setdiff, union

# A tibble: 3 × 2
     id visit
  <dbl> <chr>
1     1 03/04/2021
2     2 15/04/2021
3     3 28/02/2021
```

The `visit` column is `<chr>`, text. These dates are not year-month-day, so `read_csv()` left them as they were. Loading lubridate printed a masking message, like dplyr's in an earlier section.

Try the wrong order first. Year first:

```
ymd(camp$visit)
```

```
[1] NA NA NA
Warning message:
All formats failed to parse. No formats found.
```

Every date is `NA`, and the warning says no format fitted. That failure is loud, which is good. Now month first:

```
mdy(camp$visit)
```

```
[1] "2021-03-04" NA           NA
Warning message:
 2 failed to parse.
```

This one is worse. Two dates failed: there is no 15th or 28th month. But the first, 03/04/2021, became 4 March 2021, which looks like a perfectly good date. If every date in the column had a day of 12 or less, nothing would fail and every date would be wrong.

Day first is the order the register used:

```
camp <- camp |>
  mutate(visit_date = dmy(visit))

camp
```

```
# A tibble: 3 × 3
     id visit      visit_date
  <dbl> <chr>      <date>
1     1 03/04/2021 2021-04-03
2     2 15/04/2021 2021-04-15
3     3 28/02/2021 2021-02-28
```

All three are dates now, printed year-month-day. The new column is `<date>`, and the text column is still beside it.

A date is a number underneath, so you can do arithmetic with it. How many days after the first visit was each visit?

```
camp$visit_date - camp$visit_date[1]

as.numeric(ymd("2021-04-03"))
```

```
Time differences in days
[1]   0  12 -34
[1] 18720
```

Subtraction gives a time difference in days: 0, 12 and -34. The third visit, in February, came 34 days before the first, so its difference is negative. The last line shows the number behind a date: 3 April 2021 is 18720 days after 1 January 1970. Dates compare like numbers too: `visit_date >= ymd("2021-04-01")` is `TRUE` for the two April visits.

WHERE THIS PICTURE BREAKS. The register is made up. A column that mixes orders, some day first and some month first, cannot be fixed by any single function. The 03/04 dates in it are lost unless another record says which was meant.

One date with a day above 12 settles the order only if the whole column was written one way. Check the data dictionary or the form the data came from.

Illustration 2. Now a date that arrives as a number. A colleague exports a spreadsheet, and the visit date column comes out as 41822. Data Carpentry's spreadsheet lesson says Excel "counts the days from a default of December 31, 1899, and thus stores July 2, 2014 as the serial number 41822". Take the first half of that sentence and add the days:

```
library(lubridate)

ymd("1899-12-31") + 41822
```

```
[1] "2014-07-03"
```

3 July 2014. The lesson's own example says 2 July. Its two statements, taken literally, are a day apart. You do not need to know why to protect yourself. Find a start that gives the date you know:

```
ymd("1899-12-30") + 41822
```

```
[1] "2014-07-02"
```

2 July 2014, which matches. So the rule for this kind of file is: start from 30 December 1899. And the habit, which matters more than the rule: when you convert a spreadsheet's numbers to dates, check one date whose true value you know.

The lesson warns of a second system too. Excel on a Mac has used a 1904 starting date, so the same number can mean a date about four years away. Broman and Woo say the same: Excel stores dates "internally as a number, with different conventions on Windows and Macs".

Spreadsheets also make dates out of things that are not dates. Ziemann and colleagues found the gene symbol SEPT2 turned into "2006/09/02" in one supplementary file, a full date with a year the spreadsheet supplied. They found gene name errors in 19.6% of the papers they screened whose Excel files held gene lists, as the first section of this book showed. R does not do this. Read the same text with `read_csv()`:

```
library(readr)

genes <- read_csv(I("gene
SEPT2
MARCH1"), show_col_types = FALSE)

genes
```

```
# A tibble: 2 × 1
  gene
  <chr>
1 SEPT2
2 MARCH1
```

Still text. R parses a date only when you ask it to, with a function whose name says the order.

Figures quoted in this illustration are from [269, 39].

WHERE THIS PICTURE BREAKS. "Start from 30 December 1899" reproduces the lesson's one example, 41822 as 2 July 2014. This book has not checked it for dates before March 1900 or for files made on a Mac, and the lesson warns that both differ. Check against a known date in your own file.

That R leaves text alone does not undo a spreadsheet's change. A gene list saved after the spreadsheet turned SEPT2 into a date holds the date, and R reads the date.

Illustration 3. Last, the calculation every clinic register needs: age at a visit. Here is a made-up register of three people, with the date of birth and the visit date both written day first.

```
library(dplyr)
library(lubridate)

patients <- tibble(
  id = c(1, 2, 3),
  dob = dmy(c("14/08/1990", "02/01/1985", "30/06/2001")),
  visit = dmy(c("10/08/2021", "10/08/2021", "10/08/2021"))
)

patients <- patients |>
  mutate(
    age_days = as.numeric(visit - dob),
    age_years = age_days / 365.25,
    age_completed = floor(age_years)
  )

patients
```

```
# A tibble: 3 × 6
     id dob        visit      age_days age_years age_completed
  <dbl> <date>     <date>        <dbl>     <dbl>         <dbl>
1     1 1990-08-14 2021-08-10    11319      31.0            30
2     2 1985-01-02 2021-08-10    13369      36.6            36
3     3 2001-06-30 2021-08-10     7346      20.1            20
```

Read it one column at a time. `visit - dob` is the time difference in days, and `as.numeric()` keeps the number. Dividing by 365.25, the average length of a year counting leap years, gives age in years with a fraction. `floor()` rounds down to the whole number below, which is age in completed years, the way a register writes it.

Patient 1 is 30, not 31. The visit was on 10 August and the birthday is on 14 August, so the 31st birthday had not come. The tibble prints `age_years` as 31.0 because it rounds to fit the column;

the stored value is a little under 31, and `floor()` sees the stored value. Subtracting the years, 2021 minus 1990, would have said 31.

The same arithmetic works on the penguin file, whose `Date Egg` column `read_csv()` read as dates by itself:

```
library(readr)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)

min(penguins_raw$`Date Egg`)
max(penguins_raw$`Date Egg`)
max(penguins_raw$`Date Egg`) - min(penguins_raw$`Date Egg`)
```

```
[1] "2007-11-09"
[1] "2009-12-01"
Time difference of 753 days
```

`Date Egg`, says the data's help page, is the date the study nest was observed with one egg. The first is 9 November 2007 and the last 1 December 2009, 753 days later.

WHERE THIS PICTURE BREAKS. Dividing by 365.25 is a rule of thumb. On the day before or after a birthday it can put a person in the wrong whole year. For an exact age in completed years, lubridate has functions that count calendar years, which this book does not cover. Where a cut-point at an exact age matters, check the people near it by hand.

The three patients are made up. The penguin dates describe when study nests were observed, not anything about penguins in general.

MUST KNOW POINTS FOR YOU.

- Read day-first text with `mdy()`, and every date whose day is 12 or less comes out as a real, wrong date. Only the others fail.
- Before parsing a date column, find out the order of its parts from the data dictionary or the form.
- Read every "failed to parse" warning. It says how many dates became `NA`, or that all of them did.
- When you design a form or a register, write dates as year-month-day, ISO 8601, as Broman and Woo recommend.
- A spreadsheet date may reach you as a day count such as 41822. The starting date differs between systems, and a one-day or four-year slip looks like a real date.
- Compute age from the date of birth and the visit date, never by subtracting years.
- If a spreadsheet already turned a gene name or a date into something else, no function in R can tell you what was there before.

EXERCISE 1 (calculation). In your penguin script, check that `Date Egg` came in as a date, and count its missing values.

EXERCISE 2 (retrieval). Without looking back, say which lubridate function reads “28/02/2021”, and what `mdy()` would give for “03/04/2021”.

EXERCISE 3 (design). You are designing the data-entry sheet for a made-up school survey that records each child’s date of birth and the date they were measured. Write the two lines of the data dictionary for these columns, and say what the analysis script will do with them.

PRACTICE. Eleven problems on this technique, easiest first. Work them on paper before turning to the answers, which are in the appendix at the back under this section’s name.

1. Turn the text “2023-11-14” into a date. Say what class the result has.

2. Before you run them, say what each line gives.

```
library(lubridate, warn.conflicts = FALSE)

dmy("05/06/2022")
mdy("05/06/2022")
```

3. Before you run it, say how many days this gives, and why.

```
library(lubridate, warn.conflicts = FALSE)

ymd("2024-03-01") - ymd("2024-02-01")
```

4. Read the penguin file.

```
library(readr)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)
```

How many days separate the first nest observed in the file and the first nest of the second season, which starts on 2008-11-02?

5. Read the penguin file.

```
library(readr)
library(dplyr, warn.conflicts = FALSE)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)
```

How many rows have a `Date Egg` in November 2008?

6. A made-up register records two children measured at a school camp. Work out each child’s age at measurement in completed years.

```
library(dplyr, warn.conflicts = FALSE)
library(lubridate, warn.conflicts = FALSE)

children <- tibble(
  id = c("C1", "C2"),
  dob = dmy(c("21/06/2014", "05/12/2013")),
  measured = dmy(c("20/06/2024", "20/06/2024"))
)
```

7. A colleague reads a made-up register whose dates were typed day first, and reports: "Visit dates were missing for all three patients."

```
library(lubridate, warn.conflicts = FALSE)

visits <- c("12/01/2022", "19/01/2022", "26/01/2022")
ymd(visits)
```

```
[1] NA NA NA
Warning message:
All formats failed to parse. No formats found.
```

Find what broke.

8. A colleague reads a made-up register whose dates were typed day first. They report: "1 of the 3 screening dates was missing; the first patient was screened on 3 February 2023."

```
library(lubridate, warn.conflicts = FALSE)

screened <- c("02/03/2023", "11/03/2023", "17/03/2023")
mdy(screened)
```

```
[1] "2023-02-03" "2023-11-03" NA
Warning message:
 1 failed to parse.
```

Find what broke.

9. A made-up spreadsheet export gives two visit dates as day counts, 45292 and 45306. The colleague who exported it remembers that the first visit was on 1 January 2024. A resident converts them:

```
library(lubridate, warn.conflicts = FALSE)

ymd("1899-12-31") + c(45292, 45306)
```

```
[1] "2024-01-02" "2024-01-16"
```

They report: "The first visit was on 2 January 2024." Find what broke.

10. A made-up clinic note says: "The median time from screening to the first clinic visit was 14 days." Here are the dates for the five patients it describes, typed day first.

```
library(dplyr, warn.conflicts = FALSE)
library(lubridate, warn.conflicts = FALSE)

clinic <- tibble(
  id = c(1, 2, 3, 4, 5),
  screened = dmy(c("02/05/2024", "02/05/2024", "09/05/2024", "09/05/2024", "16/05/2024")),
  visited = dmy(c("13/05/2024", "20/05/2024", "23/05/2024", "06/06/2024", "30/05/2024"))
)
```

Check the claim, and say what your answer does not establish.

11. A made-up draft report on the penguin file says: “All penguins in the 2008–09 season laid their first egg in November 2008.” The `studyName` column names the season; PAL0809 is 2008–09. Check the claim in code, and say what the file can and cannot support.

16 · Summaries by group

DEFINITION. `summarise()`, from the dplyr package, makes a new data frame. It has one row for each group, or a single row for the whole data frame when there are no groups.

The function `n()` gives the number of rows in the group, whatever they hold. The function `sd()` gives the standard deviation, dividing by one less than the count. A quantile is the value below which a given share of the data lies. The function `quantile()` prints the 0%, 25%, 50%, 75% and 100% ones; 0% and 100% are the smallest and largest values. `IQR()` gives the interquartile range. Both use the seventh of nine rules for placing a quantile between two values, unless told otherwise.

`group_by()` marks a data frame as grouped by one or more columns, so that the next verbs work group by group. `ungroup()` removes the marking. The `.by` argument of `summarise()` groups for that one step only, and leaves the result ungrouped.

`count(a)` gives the number of rows for each value of `a`. `write_csv(data, path)`, from the readr package, writes a data frame to a CSV file. [270, 247, 271, 272, 273, 274, 275, 276, 277, 278, 279, 198, 280, 281, 282, 283]

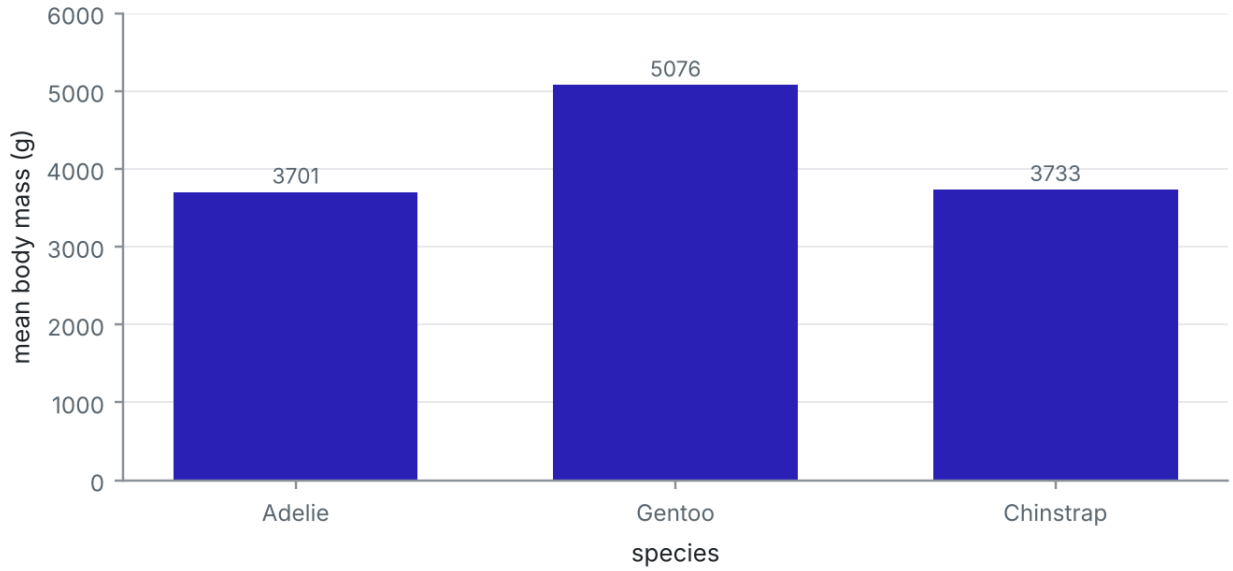
IN PLAIN TERMS. The shape of the answer tells you what you asked for. Three species give three rows. If you expected three rows and got eight, you grouped by something else as well.

Put the count beside every summary.

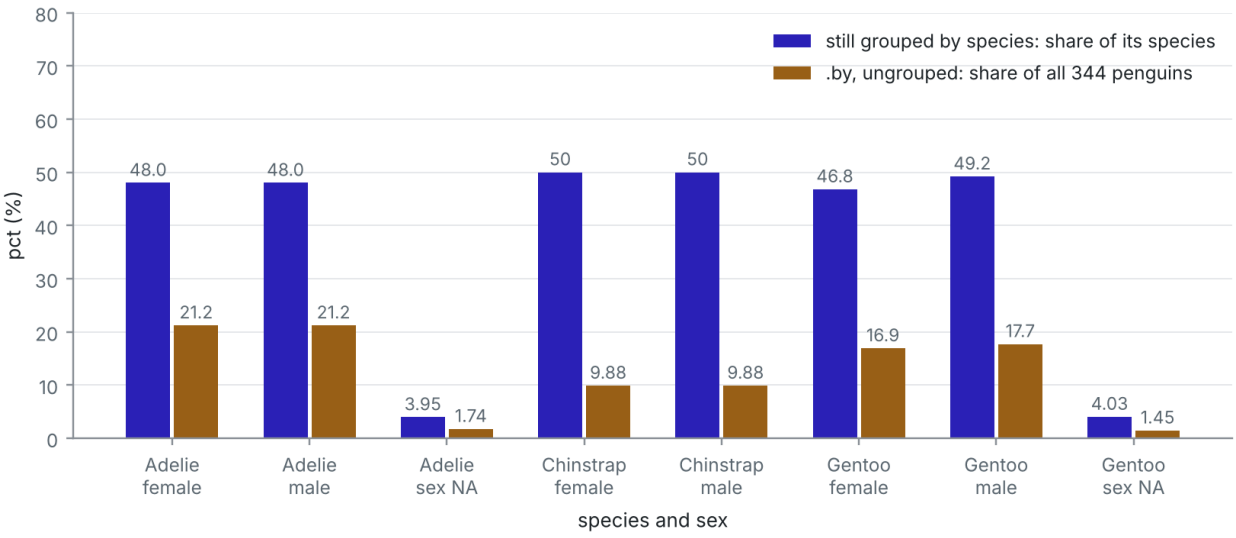
Quartiles have more than one recipe. None is wrong. Say which you used.

Never copy a number off the screen into a report. If the data change, rerun the script and the file changes with them.

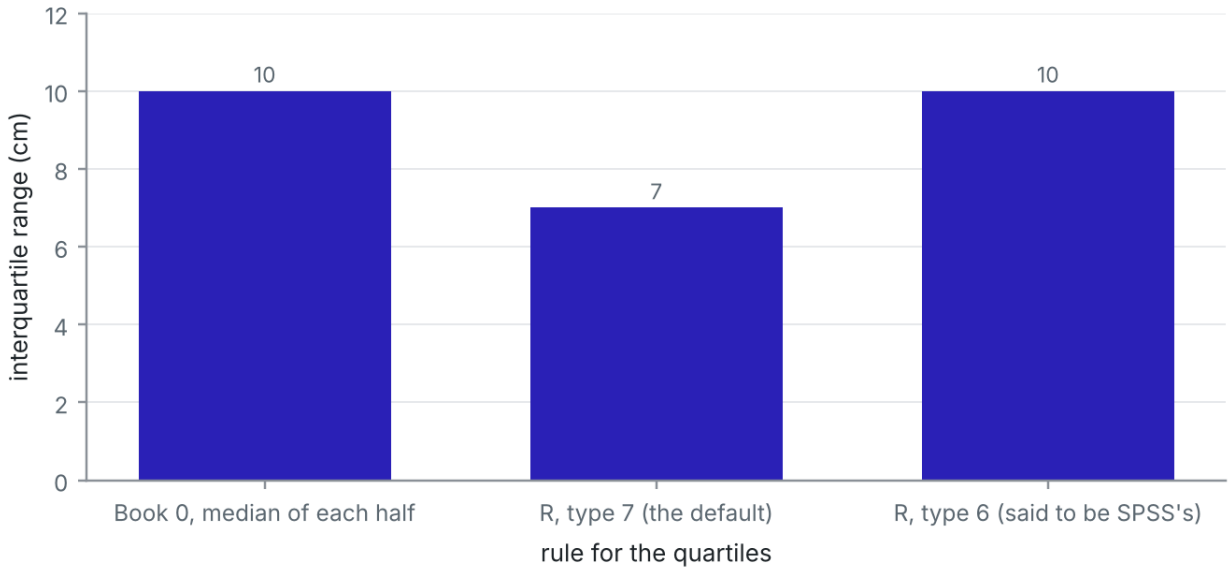
A summary describes the rows in your file. It is not, by itself, a figure for anyone outside it.



Mean body mass of the penguins in the file, by species, from the summary by species: Adelie 3701 g from 151 body masses, Gentoo 5076 g from 123, Chinstrap 3733 g from 68. The bars start at zero, so their lengths compare the means. The means describe the rows of this file.



The same eight counts of penguins by species and sex, turned into percentages two ways. Still grouped by species, sum(n) adds up within each species, so 73 Adelie females print as 48.0%, a share of the 152 Adelie penguins. Grouped with .by for one step only, sum(n) adds up all 344 rows, and the same 73 are 21.2%. The code ran both times; only the second answers “what share of all the penguins?”.



The interquartile range of the nine children's heights, 121 to 141 cm, by three rules. Book 0's median of each half gives 136 - 126 = 10 cm; R's default, type 7, gives 134 - 127 = 7 cm; type 6 gives 10 cm again. The data are the same in every bar: only the rule for placing a quartile between two values differs, so a methods section names the software and the rule.

Illustration 1. Start with the penguin file, and give the species short names as the section on making new columns did. Keep a copy of body mass under a name without spaces.

```
library(readr)
library(dplyr, warn.conflicts = FALSE)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)

penguins <- penguins_raw |>
  mutate(
    species = case_when(
      Species == "Adelie Penguin (Pygoscelis adeliae)" ~ "Adelie",
      Species == "Chinstrap penguin (Pygoscelis antarctica)" ~ "Chinstrap",
      Species == "Gentoo penguin (Pygoscelis papua)" ~ "Gentoo"
    ),
    body_mass_g = `Body Mass (g)`
  )
```

Ask for the mean body mass of all the penguins:

```
penguins |>
  summarise(mean_g = mean(body_mass_g))
```

```
# A tibble: 1 × 1
  mean_g
   <dbl>
1     NA
```

One row, as promised, and the answer is `NA`. Two penguins have no body mass, and one missing value makes the whole mean missing. That is R refusing to guess. Tell `mean()` to leave the missing values out, and ask for the count beside it:

```
penguins |>
  summarise(
    n_rows = n(),
    mean_g = mean(body_mass_g, na.rm = TRUE)
  )
```

```
# A tibble: 1 × 2
  n_rows mean_g
   <int>  <dbl>
1    344  4202.
```

Read the two numbers together. `n_rows` is 344, every row in the file. The mean was worked out from 342 of them, because `na.rm = TRUE` dropped the two without a body mass. The count beside the mean is wrong by two, and nothing on the screen says so. Count the values the mean used, not the rows:

```
mass_by_species <- penguins |>
  summarise(
    n_rows = n(),
    n_mass = sum(!is.na(body_mass_g)),
    mean_g = mean(body_mass_g, na.rm = TRUE),
    sd_g = sd(body_mass_g, na.rm = TRUE),
    median_g = median(body_mass_g, na.rm = TRUE),
    iqr_g = IQR(body_mass_g, na.rm = TRUE),
    .by = species
  )

mass_by_species
```

```
# A tibble: 3 × 7
  species   n_rows n_mass mean_g  sd_g median_g iqr_g
  <chr>      <int>  <int>  <dbl> <dbl>    <dbl> <dbl>
1 Adelie       152    151  3701.  459.     3700  650
2 Gentoo       124    123  5076.  504.     5000  800
3 Chinstrap     68     68  3733.  384.     3700  462.
```

`.by = species` asked for one row per species, and there are three rows. Each row has its own two counts. Adelie has 152 rows and 151 body masses; Gentoo has 124 rows and 123; Chinstrap has 68 of each. Those are the two missing values, one in each of two species. The tibble prints the means rounded to fit the screen.

Now put the table where a report can use it, without a hand touching a number. In your own project the `output/` folder already exists; the first line makes it if it does not, and does nothing if it does.

```
dir.create(here::here("output"), showWarnings = FALSE)

write_csv(
  mass_by_species,
  here::here("output", "mass_by_species.csv")
)

readLines(here::here("output", "mass_by_species.csv"))
```

```
[1] "species,n_rows,n_mass,mean_g,sd_g,median_g,iqr_g"
[2] "Adelie,152,151,3700.662251655629,458.5661259101348,3700,650"
[3] "Gentoo,124,123,5076.016260162602,504.11623665709163,5000,800"
[4] "Chinstrap,68,68,3733.0882352941176,384.3350813871914,3700,462.5"
```

`readLines()` prints a text file's lines as they are, to show what was written. The file holds the full numbers, not the rounded ones on the screen. A report that quotes the Adelie mean takes it from this file. If a corrected body mass arrives next month, you rerun the script, the file changes, and the report's number can be checked against it.

WHERE THIS PICTURE BREAKS. The table describes the 344 rows of this file: penguins measured on three islands in three seasons. It is not a figure for all penguins of these species, and the file's documentation makes no such claim.

Leaving out missing values is a choice. Here two body masses out of 344 are missing, and the choice barely matters. When a quarter of the values are missing, the mean of the rest may describe a different group of penguins, or patients, from the one you meant. The section on missing values says how to count and report them.

Illustration 2. Here is a trap that catches people moving between Book 0 and R. Take nine made-up heights of children, in centimetres, already in order: 121, 125, 127, 128, 132, 133, 134, 138, 141.

Work the quartiles by Book 0's method first. There are nine values, an odd count, so leave the middle one, 132, out of both halves. The lower half is 121, 125, 127, 128. Its median, the first quartile, is halfway between 125 and 127: 126. The upper half is 133, 134, 138, 141. Its median, the third quartile, is halfway between 134 and 138: 136. The interquartile range is 136 minus 126, which is 10.

```
first quartile = (125 + 127) divided by 2 = 126
third quartile = (134 + 138) divided by 2 = 136
interquartile range = 136 - 126 = 10
```

Now ask R:

```
heights_cm <- c(121, 125, 127, 128, 132, 133, 134, 138, 141)

quantile(heights_cm)
IQR(heights_cm)
```

```
  0%  25%  50%  75% 100%
 121  127  132  134  141
[1] 7
```

R says the first quartile is 127, the third is 134, and the interquartile range is 7. Your hand calculation said 126, 136 and 10. The median, 132, agrees.

Neither of you made a mistake. There is more than one rule for where a quartile sits when it falls between two values. The help page for `quantile()` lists nine, and R uses type 7 unless told otherwise. The same page says SPSS uses type 6. Ask for type 6:

```
quantile(heights_cm, type = 6)
```

```
  0%  25%  50%  75% 100%
 121  126  132  136  141
```

Type 6 gives 126 and 136, the same as Book 0's method on these nine values. That agreement is a property of this set of numbers. It is not a rule that the two methods always agree.

Method	First quartile (cm)	Third quartile (cm)	Interquartile range (cm)
Book 0, median of each half	126	136	10
R, type 7 (the default)	127	134	7
R, type 6 (said to be SPSS's)	126	136	10

So when a colleague's SPSS table and your R output disagree on a quartile, check the rule before you check the data. And in your own methods section, name the software and the rule.

WHERE THIS PICTURE BREAKS. The nine heights are made up; they show the arithmetic and describe no children.

With hundreds of values the rules land very close together, because neighbouring values in a big sorted list are close. The disagreement is largest with few values, which is exactly when a thesis table is most likely to be checked by hand.

Illustration 3. Now group by two columns at once, species and sex, and work out each group's share of all the penguins. Continue in the same session.

```
penguins |>
  group_by(species, Sex) |>
  summarise(n = n()) |>
  mutate(pct = 100 * n / sum(n))
```

```
`summarise()` has grouped output by 'species'. You can override using
the `.groups` argument.
# A tibble: 8 × 4
# Groups:   species [3]
  species   Sex        n   pct
  <chr>     <chr>  <int> <dbl>
1 Adelie    FEMALE    73 48.0
2 Adelie    MALE      73 48.0
3 Adelie    <NA>       6  3.95
4 Chinstrap FEMALE    34 50
5 Chinstrap MALE      34 50
6 Gentoo    FEMALE    58 46.8
7 Gentoo    MALE      61 49.2
8 Gentoo    <NA>       5  4.03
```

Eight rows, not six. Two species have penguins whose sex was not recorded, and `NA` became a group of its own: 6 Adelie and 5 Gentoo. A summary never hides a missing category. It prints it, and you decide what to say about it.

Read the message too. `summarise()` removed only the last grouping column, `Sex`, so the result is still grouped by species. The line `# Groups: species [3]` says so.

The first row says 48.0% for Adelie females. But 73 is not 48% of 344. Because the result was still grouped by species, `sum(n)` added up within each species: 73 divided by 152, not by 344. The code ran, gave tidy numbers, and answered a different question.

Group for one step only with `.by`, and the result comes back ungrouped:

```
penguins |>
  summarise(n = n(), .by = c(species, Sex)) |>
  mutate(pct = 100 * n / sum(n))
```

```
# A tibble: 8 × 4
  species   Sex        n   pct
  <chr>     <chr>  <int> <dbl>
1 Adelie    MALE      73 21.2
2 Adelie    FEMALE    73 21.2
3 Adelie    <NA>       6  1.74
4 Gentoo    FEMALE    58 16.9
5 Gentoo    MALE      61 17.7
6 Gentoo    <NA>       5  1.45
7 Chinstrap FEMALE    34  9.88
8 Chinstrap MALE      34  9.88
```

Now 73 Adelie females are 21.2% of all 344. The rows come in the order each group first appears in the file, whereas `group_by()` sorted them. If you do use `group_by()`, end the pipe with `ungroup()` before anything else touches the result.

For counts alone there is a shorter way, met in the section on categories. `count(species, Sex)` gives the same eight rows and counts, and an ungrouped result.

WHERE THIS PICTURE BREAKS. The percentages describe the rows of this file. Whether 21.2% of all penguins on these islands are female Adelies is a question about how the birds were chosen for study, which no summary can answer.

Grouping and summarising is also what a database query does; the next rung runs the same summaries on a whole survey in a database.

The leftover grouping trap needs two grouping columns. With one column, `summarise()` removes it and the result comes back ungrouped, so the trap waits until the day you add a second.

MUST KNOW POINTS FOR YOU.

- Print the count beside every summary, and print the right count. `n()` counts rows. With `na.rm = TRUE`, a mean uses only the rows that have a value, so count those with `sum(!is.na(x))`.
- The error is to think `na.rm = TRUE` fixes missing values. It only leaves them out, without a word.
- Check the shape of the result before reading a number.
- After `group_by()` on two columns, `summarise()` leaves the result grouped by the first. A percentage worked out next is a percentage within each group, not of the total.
- When two tables disagree on a quartile, ask which rule each used before you suspect the data.
- A number copied off the screen into a thesis cannot be traced back.
- When a trainee shows you a table of means, ask first for the count in each row and for how many values were missing. Then ask for the median beside the mean, as Book 0 taught.
- A summary describes the rows in your file. It is not an estimate for any population unless the rows were chosen to stand for one. For a survey, it also needs the survey's sample weights, which a later book teaches.

EXERCISE 1 (calculation). Add to your penguin script a summary of flipper length by island. Give, for each island, the number of rows, the number of flipper lengths used, the mean and the median in millimetres. Write the table to `output/flipper_by_island.csv`, and say which island's mean rests on the fewest measurements.

EXERCISE 2 (critique). A colleague's thesis table gives "Mean haemoglobin 11.2 g/dL (n = 312)" for the women in her study. She ran `mean(hb, na.rm = TRUE)` in the console and read the answer

off the screen. She typed it into the Word table, with `nrow()` of her data frame as the n. Say what can be wrong with that number and its n, and how her script should produce them instead.

EXERCISE 3 (teaching). A first-year resident has ten minutes and a whiteboard. She has just made a table of mean BMI by district from a screening register. She does not see why you want a count in every row. Teach her.

EXERCISE 4 (retrieval). Without looking back, name three ways a table of summaries by group can be wrong, or misleading, without any error message.

PRACTICE. Twelve problems on this technique, easiest first. Work them on paper before turning to the answers, which are in the appendix at the back under this section's name.

1. A made-up table:

```
library(dplyr, warn.conflicts = FALSE)

scores <- tibble(score = c(4, 9, 5))
```

Write one line that gives the number of rows and the mean score, in one row.

2. A made-up table of two groups:

```
library(dplyr, warn.conflicts = FALSE)

visits <- tibble(
  clinic = c("A", "B", "A", "B", "A"),
  patients = c(12, 7, 9, 11, 6)
)
```

Give the total number of patients for each clinic. Before you run it, say how many rows the answer will have.

3. Use Book 0's method, the median of each half, on the four numbers 2, 4, 6 and 8. Work out the first quartile, the third quartile and the interquartile range. Then run `quantile()` and `IQR()` on the same four numbers in R. Say why the answers differ.

4. Read the penguin file.

```
library(readr)
library(dplyr, warn.conflicts = FALSE)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)
```

For each island, give the number of rows and the number of different species found there. (`n_distinct(x)` counts the different values in `x`.)

5. Read the penguin file.

```
library(readr)
library(dplyr, warn.conflicts = FALSE)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)
```

Give the median body mass in grams for each value of `Sex`, with the number of rows and the number of body masses used. Say what the third row is, and why it is there.

6. Read the penguin file.

```
library(readr)
library(dplyr, warn.conflicts = FALSE)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)
```

Make a table of culmen length by species: the number of values used, the mean and the standard deviation. The culmen is the ridge of the bill, measured in millimetres. Write it to `output/culmen_by_species.csv`. Then show that the file exists and how many lines it has.

7. A colleague runs this and writes in her notes: “Adelie and Gentoo body masses were not recorded; only Chinstrap has data.”

```
library(readr)
library(dplyr, warn.conflicts = FALSE)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)

penguins_raw |>
  summarise(mean_g = mean(`Body Mass (g)`), .by = Species)
```

```
# A tibble: 3 × 2
  Species                                   mean_g
  <chr>                                      <dbl>
1 Adelie Penguin (Pygoscelis adeliae)          NA
2 Gentoo penguin (Pygoscelis papua)            NA
3 Chinstrap penguin (Pygoscelis antarctica)  3733.
```

Find what broke.

8. A colleague’s table of Gentoo body mass reads “n = 124, mean 5076 g”. Here is his code.

```
library(readr)
library(dplyr, warn.conflicts = FALSE)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)

penguins_raw |>
  filter(Species == "Gentoo penguin (Pygoscelis papua)") |>
  summarise(
    n = n(),
    mean_g = mean(`Body Mass (g)`, na.rm = TRUE)
  )
```

```
# A tibble: 1 × 2
      n mean_g
  <int>  <dbl>
1   124  5076.
```

The mean is right. Find what is wrong with the table.

9. A colleague wants each species-and-sex group's share of all the penguins, and reports "Gentoo males are 49.2% of the penguins in the file."

```
library(readr)
library(dplyr, warn.conflicts = FALSE)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)

penguins_raw |>
  group_by(Species, Sex) |>
  summarise(n = n()) |>
  mutate(pct = 100 * n / sum(n)) |>
  filter(Sex == "MALE")
```

```
`summarise()` has grouped output by 'Species'. You can override using
the `.groups` argument.
# A tibble: 3 × 4
# Groups:   Species [3]
  Species                                   Sex       n   pct
  <chr>                                     <chr> <int> <dbl>
1 Adelie Penguin (Pygoscelis adeliae)       MALE     73  48.0
2 Chinstrap penguin (Pygoscelis antarctica) MALE     34  50
3 Gentoo penguin (Pygoscelis papua)         MALE     61  49.2
```

Find what broke.

10. A colleague runs this to get mean body mass by short species name:

```
library(readr)
library(dplyr, warn.conflicts = FALSE)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)
```

```
penguins_raw |>
  summarise(
    mean_g = mean(`Body Mass (g)`, na.rm = TRUE),
    .by = species
  ) |>
  mutate(
    species = case_when(
      species == "Adelie Penguin (Pygoscelis adeliae)" ~ "Adelie",
      species == "Chinstrap penguin (Pygoscelis antarctica)" ~ "Chinstrap",
      species == "Gentoo penguin (Pygoscelis papua)" ~ "Gentoo"
    )
  )
```

```
Error in `summarise()`:
! Can't subset columns that don't exist.
✖ Column `species` doesn't exist.
```

Read the error. Say what broke and fix it.

11. A made-up draft thesis says: “Gentoo penguins (n = 124) were heavier than Adelie penguins (n = 152), with mean body masses of 5076 g and 3701 g.” Check every number in the sentence against the penguin file. Then say what the result does not establish.

12. In a made-up lab meeting, someone says: “Male penguins in this dataset average about 4.5 kg.” Turn the sentence into code, run it, and say what the result does not establish.

17 · Reshaping and joining

DEFINITION. `pivot_longer()`, from the tidyr package, makes a data frame longer: it turns a set of columns into two, one holding the old column names and one holding their values, and repeats the other columns. It increases the rows and decreases the columns. `pivot_wider()` does the reverse.

A key is a column, or set of columns, whose values pick out the row they belong to. A join combines two data frames, `x` and `y`, by matching rows whose keys are equal. `left_join(x, y)`, from the dplyr package, keeps every row of `x` and adds the columns of `y`. A row of `x` with no match in `y` gets `NA` in the new columns. A row of `x` that matches several rows of `y` appears once for each match. `anti_join(x, y)` returns the rows of `x` that have no match in `y`.

`by` names the key. [284, 285, 286, 287, 288, 289, 290, 291, 292, 293, 294, 295, 296, 297]

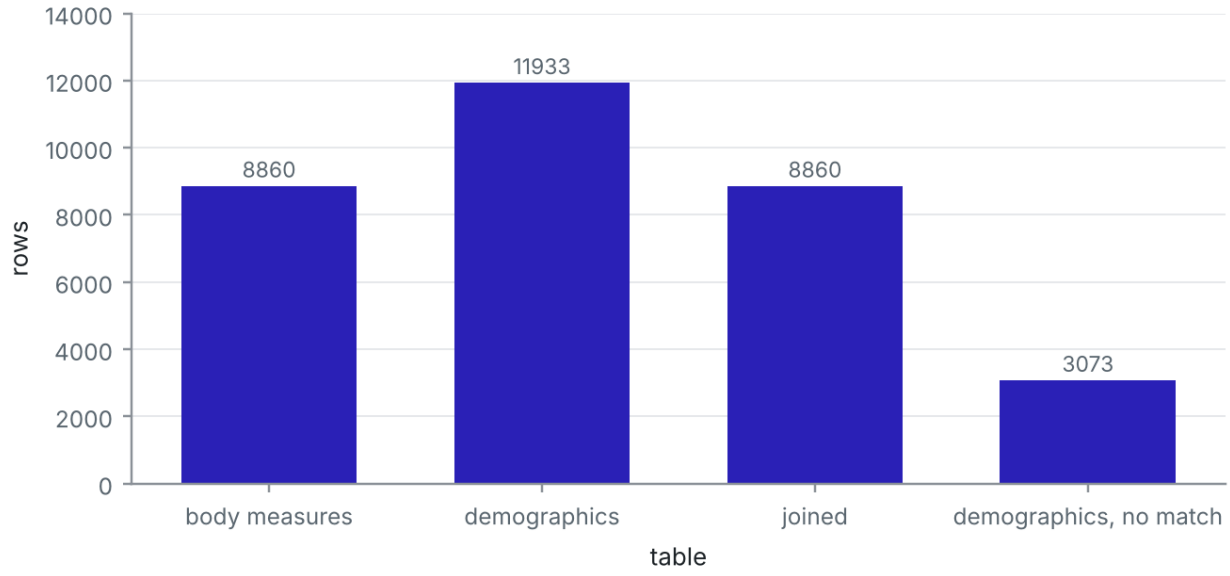
IN PLAIN TERMS. **The wide table.** A clinic writes a patient's weight at three visits in three columns: visit 1, visit 2, visit 3. But "visit" is a variable, and here its values sit in the column names. In the column names, `summarise(.by = visit)` cannot reach them.

`pivot_longer()` turns the three columns into two: one saying which visit, one saying the weight.

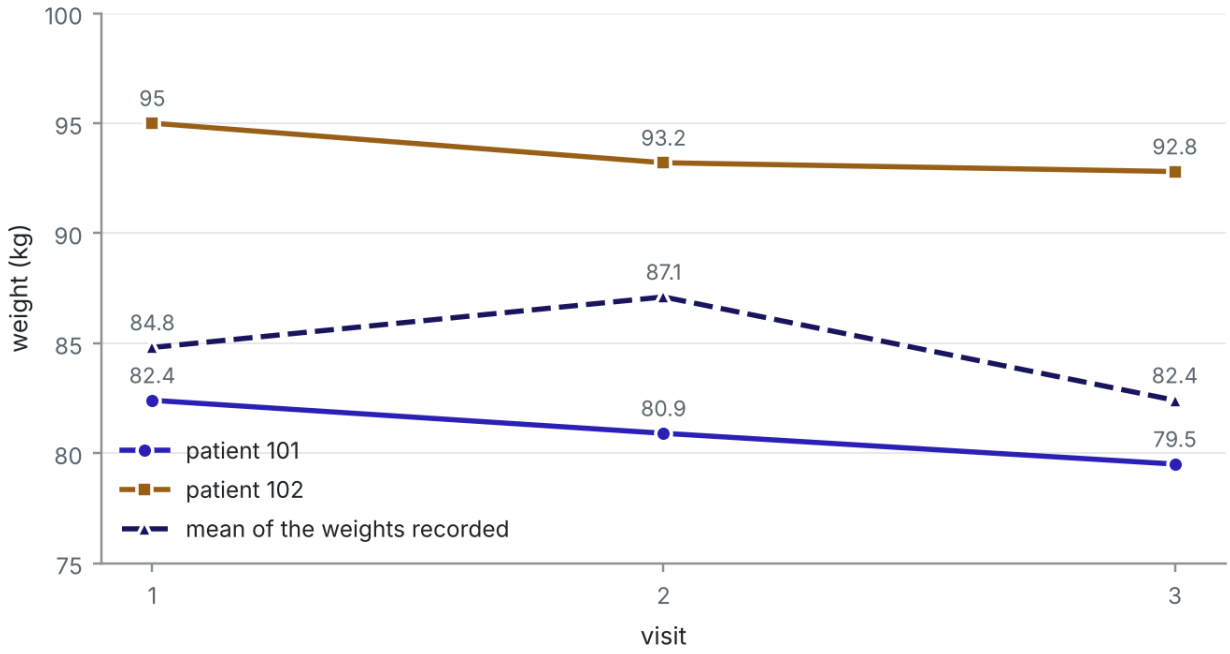
The two files. A survey sends the demographics in one file and the measurements in another, with a participant number in both.

Here is the trap with joins. Suppose a key appears twice in the second file, by a slip in data entry. Then the matching row of the first file appears twice in the result. No error, no warning.

So count the rows before and after every join. Find keys that appear more than once. Find rows that did not match with `anti_join()`.



Rows in the two NHANES 2021–2023 files and in their join on SEQN. The body measures file has 8860 rows and the demographics file 11933. The left join of body measures to demographics keeps 8860, and the 3073 demographic rows with no body measures are participants who were interviewed but not examined. These are counts of rows, not survey estimates.



Weights in the made-up register at visits 1, 2 and 3, after pivot_longer() made the visit a column. Patients 101 and 102 lose weight at every visit, yet the mean of the weights recorded rises from 84.8 kg at visit 1 to 87.1 kg at visit 2, then falls to 82.4 kg. Visit 2 has 2 weights, not 3: patient 103, the lightest (77.1 kg at visit 1, 75 kg at visit 3), was not weighed there and has no line here, so the visit 2 mean is a mean of different people.

Illustration 1. Start with a made-up weight register in the wide shape a clinic would write by hand. Each patient's weight in kilograms at three visits sits in three columns. Patient 103 missed visit 2.

```
library(dplyr, warn.conflicts = FALSE)
library(tidyr)

weights_wide <- tibble(
  id = c(101, 102, 103),
  weight_v1 = c(82.4, 95.0, 77.1),
  weight_v2 = c(80.9, 93.2, NA),
  weight_v3 = c(79.5, 92.8, 75.0)
)
```

Try to get the mean weight at each visit. The visit number is not in any column. It is part of three column names, so there is nothing to group by. You could write three `mean()` calls, one per column, and a fourth when a visit is added. Instead, make the visit a column:

```
weights_long <- weights_wide |>
  pivot_longer(
    cols = starts_with("weight_v"),
    names_to = "visit",
    names_prefix = "weight_v",
    values_to = "weight_kg"
  )

weights_long
```

```
# A tibble: 9 × 3
     id visit weight_kg
  <dbl> <chr>     <dbl>
1   101 1          82.4
2   101 2          80.9
3   101 3          79.5
4   102 1          95
5   102 2          93.2
6   102 3          92.8
7   103 1          77.1
8   103 2          NA
9   103 3          75
```

Read the arguments in order. `cols` names the three columns to fold. `names_to` names the new column that receives their names. `names_prefix` takes “weight_v” off the front, so the visit column holds 1, 2 and 3. `values_to` names the column that receives the weights.

Three patients times three visits is nine rows. Each row is one patient at one visit, which is what one observation is here. Patient 103’s missing visit 2 is a row with `NA`, not a gap. Now the visit is a column, and the last section’s tools work:

```
weights_long |>
  summarise(
    n_weights = sum(!is.na(weight_kg)),
    mean_kg = mean(weight_kg, na.rm = TRUE),
    .by = visit
  )
```

```
# A tibble: 3 × 3
  visit n_weights mean_kg
  <chr>     <int>   <dbl>
1 1             3    84.8
2 2             2    87.1
3 3             3    82.4
```

Look at the means: 84.8, 87.1, 82.4. The mean went up at visit 2, though every patient weighed there lost weight. The count explains it. Visit 2 has 2 weights, not 3: the lightest patient was missing, so the visit 2 mean is a mean of two different people. A mean over visits compares visits only when the same people are in every visit. The count beside the mean is how you see that.

For the printed table, turn it back:

```
weights_long |>
  pivot_wider(
    names_from = visit,
    values_from = weight_kg,
    names_prefix = "weight_v"
  )
```

```
# A tibble: 3 × 4
     id weight_v1 weight_v2 weight_v3
  <dbl>     <dbl>     <dbl>     <dbl>
1   101      82.4      80.9      79.5
2   102      95        93.2      92.8
3   103      77.1      NA        75
```

The same three rows and four columns you started with. `names_from` names the column whose values become the new column names, and `values_from` the column that fills them. The remaining columns, here `id`, pick out the rows. `values_fill = 0` puts 0 where a combination has no value; use it only for counts. Long for working, wide for reading.

WHERE THIS PICTURE BREAKS. The three patients are made up. Their weights show the reshaping and say nothing about any treatment or any person.

`pivot_longer()` is right only when the folded columns hold the same variable in the same unit. Folding a weight column and a height column into one value column would make a column with two units in it. The tools would compute a mean of it without complaint.

Illustration 2. Now two files. A made-up clinic keeps demographics in one table and BMI in another, with the patient number `id` in both. Patient 102 was entered twice in the demographics table, by a slip at the registration desk. Patient 106 has a BMI and no demographic record.

```
demographics <- tibble(
  id = c(101, 102, 102, 103, 104, 105),
  sex = c("F", "M", "M", "F", "M", "F"),
  age = c(34, 51, 51, 45, 29, 60)
)

bmi <- tibble(
  id = c(101, 102, 103, 105, 106),
  bmi = c(27.1, 31.4, 24.9, 29.8, 22.3)
)

nrow(bmi)

bmi_demo <- bmi |>
  left_join(demographics, by = "id")

nrow(bmi_demo)
bmi_demo
```

```
[1] 5
[1] 6
# A tibble: 6 × 4
     id   bmi sex     age
  <dbl> <dbl> <chr> <dbl>
1   101  27.1 F        34
2   102  31.4 M        51
3   102  31.4 M        51
4   103  24.9 F        45
5   105  29.8 F        60
6   106  22.3 <NA>     NA
```

Five rows went in and six came out. Patient 102 is there twice, because the key 102 matched two rows of `demographics`. R printed no warning. Every later count and mean would include patient 102 twice, and a summary of 6 BMIs would be reported for 5 patients.

Patient 106 is there once, with `NA` for sex and age. `left_join()` kept the row, because it keeps every row of the first table, and had nothing to fill in.

Three lines find both problems. Count each key, and keep any that appear more than once. Then ask which BMI rows have no partner:

```
demographics |>
  count(id) |>
  filter(n > 1)

bmi |>
  anti_join(demographics, by = "id")
```

```
# A tibble: 1 × 2
     id     n
  <dbl> <int>
1   102     2
# A tibble: 1 × 2
     id   bmi
  <dbl> <dbl>
1   106  22.3
```

Key 102 appears twice. Patient 106 has no demographic record. Neither is fixed by editing the raw file. Patient 106's missing record is a question for the clinic. The duplicate is removed by a line of code, with a comment saying why; a later section shows how to make the script stop on it.

Last, make the join refuse to add rows. The `relationship` argument states how many matches each row is expected to have, and the join stops with an error if the data break it. Each BMI row should match at most one row of demographics, which `relationship = "many-to-one"` says:

```
bmi |>
  left_join(demographics, by = "id", relationship = "many-to-one")
```

```
Error in `left_join()`:
! Each row in `x` must match at most 1 row in `y`.
i Row 2 of `x` matches multiple rows in `y`.
```

Now the script stops at the join and names the row that broke the rule, instead of carrying a wrong count into every table after it.

WHERE THIS PICTURE BREAKS. The register is made up, and its five patients describe nobody.

A duplicated key is not always an error. A table of visits has one row per visit, so a patient with three visits has the same id three times, and joining demographics to it should repeat each patient's age three times. The question is never "are there repeats?" but "should there be?", and the answer comes from what one row stands for.

Illustration 3. Now the real case. NHANES, the American survey whose body measures file you read in the section on raw files, sends its demographics in a second file. Both files carry `SEQN`, the participant's sequence number. Read both, keep a few columns of each, and count rows before and after the join.

```
bmx <- haven::read_xpt(here::here("data-raw", "BMX_L.xpt"))
demo <- haven::read_xpt(here::here("data-raw", "DEMO_L.xpt"))

nrow(bmx)
nrow(demo)

body <- bmx |>
  select(SEQN, BMXWT, BMXHT, BMXBMI) |>
  left_join(
    demo |> select(SEQN, RIDSTATR, RIAGENDR, RIDAGEYR),
    by = "SEQN",
    relationship = "one-to-one"
  )

nrow(body)
```

```
[1] 8860
[1] 11933
[1] 8860
```

The body measures file has 8860 rows and the demographics file 11933. The join kept 8860, the same as went in. `relationship = "one-to-one"` expects each key at most once on each side; it did not stop, so no `SEQN` appears twice in either file.

The demographics file has 3073 more rows than the body measures file. Who are they? Ask which demographic rows found no body measures, and look at their `RIDSTATR`. The codebook says that column records whether a participant was interviewed and examined, or only interviewed.

```
bmx |>
  anti_join(demo, by = "SEQN") |>
  nrow()

demo |>
  anti_join(bmx, by = "SEQN") |>
  count(RIDSTATR)
```

```
[1] 0
# A tibble: 1 × 2
  RIDSTATR     n
     <dbl> <int>
1        1  3073
```

Every body-measures row found its demographics: 0 failed to match. The 3073 demographic rows with no body measures all have `RIDSTATR` 1. In the codebook's words, they were only interviewed, and never examined, so there was nothing to measure. The two counts agree:

```
11933 - 8860 = 3073
```

Table	Rows
body measures	8860
demographics	11933

Table	Rows
joined	8860
demographics, no match	3073

Now the joined table can answer a question neither file could alone. How many adults aged 20 or over have a body measures record, and what is their BMI, by the codebook's `RIAGENDR`, where 1 is male and 2 is female?

```
body |>
  filter(RIDAGEYR >= 20) |>
  summarise(
    n_rows = n(),
    n_bmi = sum(!is.na(BMXBMI)),
    mean_bmi = mean(BMXBMI, na.rm = TRUE),
    median_bmi = median(BMXBMI, na.rm = TRUE),
    .by = RIAGENDR
  )
```

```
# A tibble: 2 × 5
  RIAGENDR n_rows n_bmi mean_bmi median_bmi
     <dbl>  <int> <int>    <dbl>      <dbl>
1        1   2720  2680     29.3       28.1
2        2   3344  3290     30.3       28.7
```

2720 men and 3344 women, 6064 adults in all, of whom 2680 and 3290 have a BMI. Their mean BMI is 29.3 and 30.3 kg/m^2, and the medians are 28.1 and 28.7 kg/m^2.

These are the mean and median of the rows in the file, unweighted. The survey's documentation says its sample weights should be used to analyse these data. Without them, these numbers describe the people examined, and estimate nothing about adults in the United States. The book on survey data teaches the weights.

WHERE THIS PICTURE BREAKS. A join on a key is also the core of a database query; the next rung writes the same joins in SQL (Structured Query Language).

The join worked cleanly because NHANES built both files around one key and published it. A thesis dataset with a register typed in one place and laboratory results in another rarely has a key that clean: names spelled two ways, IDs with and without a leading zero. The checks are the same; expect them to find more.

The adult summary includes everyone aged 20 or over who was examined. It does not exclude anyone for whom BMI means something different, such as women who were pregnant at the examination. A real analysis would decide that, in code, before summarising, as the last section of this book does.

MUST KNOW POINTS FOR YOU.

- Count the rows before and after every join. A `left_join()` that returns more rows than went in has found a key that appears more than once in the second table. It says nothing about it.

- The error is to think a join only adds columns. It also adds rows when a key repeats, and every count and mean after it counts those rows again.
- Before a join, check the key in each table: count each value with `count()` and keep those with `n > 1`.
- Always write `by`. Left out, the join matches on every column the two tables share by name. A column called `year` or `date` may mean different things in each. It will then be matched too. dplyr names the columns it used in a message that is easy to miss.
- When a wide table has values of a variable in its column names, make it long with `pivot_longer()` before summarising.
- A mean at each visit compares visits only if the same people are in every visit. Print the count at each visit.
- When a trainee's merged thesis file "has a few more rows than it should", do not let them delete rows by hand.
- The join checks can tell you that keys match and how often. They cannot tell you that two rows with the same key are really the same person.

EXERCISE 1 (calculation). Add to a script the join of the NHANES body measures file to the demographics file on `SEQN`. Print five counts: the rows of each file, the rows of the result, the body-measures rows with no demographics, and the joined rows with an age of 20 or over. State what each count tells you.

EXERCISE 2 (critique). A colleague merged her thesis questionnaire file with the laboratory results in a spreadsheet. She sorted both by participant ID, pasted the laboratory columns beside the questionnaire columns, and saved the result as a new file. Say what can go wrong, and how to do it in code.

EXERCISE 3 (retrieval). Without looking back, say what each of these returns: `pivot_longer()`, `pivot_wider()`, `left_join(x, y)` and `anti_join(x, y)`.

PRACTICE. Eleven problems on this technique, easiest first. Work them on paper before turning to the answers, which are in the appendix at the back under this section's name.

1. A made-up table:

```
library(tidyr)
library(dplyr, warn.conflicts = FALSE)

screen <- tibble(child = "A", height_2024 = 121, height_2025 = 127)
```

Make it long, with a column `year` and a column `height_cm`. How many rows will it have?

2. A made-up long table:

```
library(tidyr)
library(dplyr, warn.conflicts = FALSE)

long <- tibble(
  district = c("Durg", "Durg", "Korba", "Korba"),
  round = c("r1", "r2", "r1", "r2"),
  screened = c(410, 455, 380, 362)
)
```

Make it wide, with one row per district and one column per round.

3. Two made-up tables:

```
library(dplyr, warn.conflicts = FALSE)

x <- tibble(id = c(1, 2, 3), weight_kg = c(60, 72, 55))
y <- tibble(id = c(2, 3, 3, 4), ward = c("A", "B", "C", "D"))
```

Before you run `left_join(x, y, by = "id")`, say how many rows the result will have and which `ward` each row gets. Then run it.

4. Read the two NHANES files.

```
library(dplyr, warn.conflicts = FALSE)

bmx <- haven::read_xpt(here::here("data-raw", "BMX_L.xpt"))
demo <- haven::read_xpt(here::here("data-raw", "DEMO_L.xpt"))
```

Check that `SEQN` identifies one row in each file: count how many values of `SEQN` appear more than once in each.

5. Read the penguin file.

```
library(readr)
library(dplyr, warn.conflicts = FALSE)
library(tidyr)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)
```

Make a table with one row per species and one column per island, holding the number of rows of each species on each island. Put 0 where a species was not found on an island, and say why that is the right value here.

6. Read the two NHANES files.

```
library(dplyr, warn.conflicts = FALSE)

bmx <- haven::read_xpt(here::here("data-raw", "BMX_L.xpt"))
demo <- haven::read_xpt(here::here("data-raw", "DEMO_L.xpt"))
```

Join the age column `RIDAGEYR` to the body measures file. Then count how many examined participants are children under 2, and how many of those have a standing height (`BMXHT`). Say why the second count is what it is. The BMX codebook lists the ages at which each body measure is taken.

7. A colleague wants one tidy column of penguin measurements. He writes this, and reports "the mean measurement of Gentoo penguins is 1780".

```
library(readr)
library(dplyr, warn.conflicts = FALSE)
library(tidyr)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)

penguins_raw |>
  pivot_longer(
    cols = c(`Culmen Length (mm)`, `Flipper Length (mm)`, `Body Mass (g)`),
    names_to = "measure",
    values_to = "value"
  ) |>
  summarise(mean_value = mean(value, na.rm = TRUE), .by = Species)
```

```
# A tibble: 3 × 2
  Species                                   mean_value
  <chr>                                          <dbl>
1 Adelie Penguin (Pygoscelis adeliae)            1310.
2 Gentoo penguin (Pygoscelis papua)              1780.
3 Chinstrap penguin (Pygoscelis antarctica)      1326.
```

Find what broke.

8. A made-up clinic keeps two tables. In `visits`, `year` is the year of the visit. In `register`, `year` is the year of birth. A colleague joins them and reports "no patient's sex is recorded".

```
library(dplyr, warn.conflicts = FALSE)

visits <- tibble(id = c(1, 2, 3), year = c(2025, 2025, 2026), bmi = c(26.2, 31.0, 23.4))
register <- tibble(id = c(1, 2, 3), year = c(1980, 1972, 1991), sex = c("F", "M", "F"))

visits |>
  left_join(register)
```

```
Joining with `by = join_by(id, year)`
# A tibble: 3 × 4
     id  year   bmi sex
  <dbl> <dbl> <dbl> <chr>
1     1  2025  26.2 <NA>
2     2  2025  31   <NA>
3     3  2026  23.4 <NA>
```

Find what broke.

9. A colleague joins a made-up laboratory table to a register. The laboratory file was read with every column as text.

```
library(dplyr, warn.conflicts = FALSE)

register <- tibble(id = c(101, 102, 103), sex = c("F", "M", "F"))
lab <- tibble(id = c("101", "102", "103"), hb = c("12.1", "14.3", "11.8"))
```

```
register |>
  left_join(lab, by = "id")
```

```
Error in `left_join()`:
! Can't join `x$id` with `y$id` due to incompatible types.
ℹ `x$id` is a <double>.
ℹ `y$id` is a <character>.
```

Read the error. Say what broke and fix it.

10. A made-up methods section says: “We merged the NHANES 2021–2023 body measures and demographic files on the sequence number; 11,933 participants had body measures.” Check the claim, and say what the merge does not establish.

11. In a made-up journal club, someone says: “In NHANES, women have a higher BMI than men.” Turn the sentence into code on the joined files, for adults aged 20 or over, run it, and say what the result does not establish.

18 · Plotting with the grammar of graphics

DEFINITION. `ggplot()`, from the ggplot2 package, starts a plot from a data frame and a set of aesthetic mappings. An aesthetic mapping, made with `aes()`, says which column of the data controls which visual property of the marks: position along x, position along y, colour, and others. A geom is the kind of mark that represents each observation or group, and each `geom_` function adds one layer of marks. Layers are added to the plot with `+`.

The function `geom_histogram()` divides the x axis into bins and draws a bar whose height is the count of observations in each bin. Unless told otherwise it uses 30 bins. A value on the edge between two bins is then counted in the lower bin. The arguments `binwidth`, `boundary` and `closed` set the width of the bins, where an edge falls, and which edge a bin includes.

The function `geom_boxplot()` draws the median and a box from the first to the third quartile. Its whiskers reach the furthest values within 1.5 times the interquartile range beyond each end of the box (below the first quartile, above the third). Values beyond the whiskers are drawn one by one.

A visual property written inside `aes()` is mapped: it varies with a column. Written outside `aes()`, as an argument of the geom, it is set: one fixed value for every mark. The function `facet_wrap()` draws one panel for each value of a column. The function `ggsave()` writes a plot to a file, at a size you give. [298, 299, 300, 301, 302, 303, 304, 305, 306, 307, 308, 309, 310, 311, 312, 313, 314, 315, 316, 317, 318]

IN PLAIN TERMS. A plot in ggplot2 is built from three parts.

The data: a data frame, one row per observation, as every section since tidy data has used.

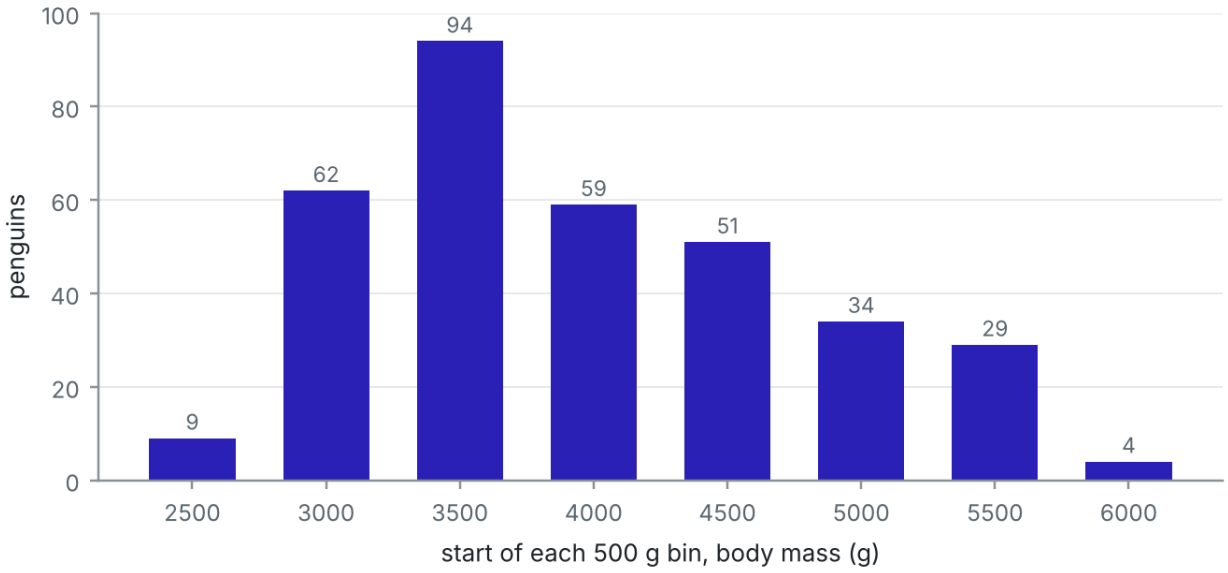
The mappings: which column goes where. You write these inside `aes()`, short for aesthetics, the visual properties of a mark.

The geom: what kind of mark. Points for a scatter plot, bars for a histogram or a count, boxes for a boxplot.

You join the parts with `+`.

Two traps come up early. Inside `aes()` means “let this vary with a column”. Outside it means “make every mark this”. Write `colour = "blue"` inside and ggplot2 treats “blue” as data. And a histogram’s bars depend on bins you did not choose unless you choose them: set their width and where they start.

Save it to `output/` with `ggsave()`, from the script, at a size you write down. Never export it with the mouse.



The counts behind the histogram of penguin body mass with its bins chosen in code: 500 g wide from 2500 g, each including its left edge. The tallest bin, 3500 to under 4000 g, holds 94 of the 342 penguins with a body mass. The same counts are what the code writes to output/ beside the figure.

Illustration 1. Start with a histogram of penguin body mass. Load the packages, read the file, and make the short columns used in the summaries section.

```
library(readr)
library(dplyr, warn.conflicts = FALSE)
library(ggplot2)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)

penguins <- penguins_raw |>
  mutate(
    species = case_when(
      Species == "Adelie Penguin (Pygoscelis adeliae)" ~ "Adelie",
      Species == "Chinstrap penguin (Pygoscelis antarctica)" ~ "Chinstrap",
      Species == "Gentoo penguin (Pygoscelis papua)" ~ "Gentoo"
    ),
    body_mass_g = `Body Mass (g)`,
    flipper_mm = `Flipper Length (mm)`
  )

range(penguins$body_mass_g, na.rm = TRUE)
```

```
[1] 2700 6300
```

Now the plot: the data, one mapping, one geom.

```
ggplot(penguins, aes(x = body_mass_g)) +
  geom_histogram()
```

```
`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
Warning message:
Removed 2 rows containing non-finite values (`stat_bin()`).
```

A histogram appears, and R prints two lines. Read both. The first says ggplot2 used 30 bins because you did not choose any. Thirty is not a property of penguins. It is a default, and the help page calls it “not a good default”. The second says 2 rows were removed: the two penguins with no body mass. The figure shows 342 penguins, and nothing on it says so.

Choose the bins. Body masses run from 2700 g to 6300 g, so bins 500 g wide, starting at 2500 g, cover them in eight bars. Say also which edge a bin includes. By default a penguin of exactly 3000 g goes in the bin “2500 to 3000”. `closed = "left"` makes each bin run from its left edge up to just under its right edge, so the same penguin counts in “3000 to under 3500”; the last bin also keeps its right edge.

```
ggplot(penguins, aes(x = body_mass_g)) +
  geom_histogram(binwidth = 500, boundary = 2500, closed = "left")
```

```
Warning message:
Removed 2 rows containing non-finite values (`stat_bin()`).
```

The warning about the 2 removed rows stays, and it should. Now count the same bins yourself, so the numbers behind the bars exist somewhere other than in a picture. Dividing by 500, rounding down with `floor()` and multiplying by 500 gives each penguin’s bin start. `count()` can make the column it counts; the name goes on the left of `=`.

```
mass_bins <- penguins |>
  count(bin_start_g = floor(body_mass_g / 500) * 500)

mass_bins

dir.create(here::here("output"), showWarnings = FALSE)
write_csv(mass_bins, here::here("output", "mass_histogram_counts.csv"))
```

```
# A tibble: 9 × 2
  bin_start_g     n
        <dbl> <int>
1        2500     9
2        3000    62
3        3500    94
4        4000    59
5        4500    51
6        5000    34
7        5500    29
8        6000     4
9          NA     2
```

The eight counts add up to 342, and the `NA` row holds the other 2. These are the heights of the eight bars. Rule 7 of Sandve and colleagues’ ten rules for reproducible research asks for exactly this: keep the counts behind a histogram, not only the picture.

Choosing the bins in code matters for one more reason. The outputs in this book come from ggplot2 3.4.4. ggplot2 4.0.0 changed how it places bins by default, and its own notes say this "may affect plots that use default binning". A histogram left on defaults can change shape when a colleague runs your script on a newer version. One with its width and boundary written down cannot, unless that version has a bug: the notes for 4.0.1 list a fixed bug in which `boundary` was ignored.

WHERE THIS PICTURE BREAKS. The bars describe the 342 penguins with a body mass in this file. They are not the shape of body mass in any wider population of penguins.

Another bin width tells a different story. Bins of 100 g show more bumps, and some of them are chance; bins of 1000 g hide real ones. There is no correct width. Try two or three, pick one that shows what the data show, and write it in the code.

Illustration 2. Now two variables, and the commonest early mistake. Continue in the same session. You want every point blue, and you write the colour inside `aes()`:

```
ggplot(penguins, aes(x = flipper_mm, y = body_mass_g, colour = "blue")) +
  geom_point()
```

```
Warning message:
Removed 2 rows containing missing values (`geom_point()`).
```

The book prints no picture here, but on your screen the points are not blue. They are ggplot2's first default colour, a pinkish red, and a legend appears titled "colour" with one entry: "blue". Inside `aes()`, "blue" was treated as data, a column whose every value is the word "blue". ggplot2 gave that one value a colour of its own choosing and explained it in a legend.

Mapping and setting are different acts. Map a colour to a column when you want it to vary with the data. Set it outside `aes()`, in the geom, when you want one colour for every point:

```
ggplot(penguins, aes(x = flipper_mm, y = body_mass_g, colour = species)) +
  geom_point()

ggplot(penguins, aes(x = flipper_mm, y = body_mass_g)) +
  geom_point(colour = "blue")
```

```
Warning message:
Removed 2 rows containing missing values (`geom_point()`).
Warning message:
Removed 2 rows containing missing values (`geom_point()`).
```

The first gives each species its own colour and a legend that means something. The second makes every point blue and needs no legend.

Colour is one way to show a third variable. Facets are another: one small panel per species, on the same axes, so the panels can be compared by eye. Finish the figure as Book 0 asked: axis titles that name the quantity and its unit, and a source line. Then save it from the script, at a size written in the code, into `output/`.

```
mass_flipper <- ggplot(penguins, aes(x = flipper_mm, y = body_mass_g)) +
  geom_point() +
  facet_wrap(~species) +
  labs(
    x = "Flipper length (mm)",
    y = "Body mass (g)",
    caption = "Source: palmerpenguins raw data, as read from data-raw/"
  )

mass_flipper

dir.create(here::here("output"), showWarnings = FALSE)
ggsave(
  here::here("output", "mass_flipper.png"),
  plot = mass_flipper,
  width = 18,
  height = 10,
  units = "cm"
)

file.exists(here::here("output", "mass_flipper.png"))
```

```
Warning message:
Removed 2 rows containing missing values (`geom_point()`).
Warning message:
Removed 2 rows containing missing values (`geom_point()`).
[1] TRUE
```

Three panels, one per species, each on the same scales. `~species` reads "by species". The figure is in `output/`, made by the same line every time the script runs. The `dir.create()` line matters for a reader on a newer version: from ggplot2 3.5.0, `ggsave()` makes a missing folder only when its new `create.dir` argument tells it to.

WHERE THIS PICTURE BREAKS. A scatter plot shows that heavier penguins in this file tend to have longer flippers. It does not show that one causes the other, and it says nothing about penguins outside the file.

Facets compare panels well only when they share axes, which `facet_wrap()` does unless told otherwise. Free axes, one scale per panel, make every panel look equally spread out, and the eye then compares shapes that are drawn on different rulers.

Illustration 3. Last, the two plots most used in a thesis to compare groups: the boxplot and the bar chart. Continue in the same session.

```
ggplot(penguins, aes(x = species, y = body_mass_g)) +
  geom_boxplot() +
  labs(x = "Species", y = "Body mass (g)")
```

```
Warning message:
Removed 2 rows containing non-finite values (`stat_boxplot()`).
```

Each box runs from the first quartile to the third, with a line at the median. Which quartiles? The ones `quantile()` gives, R's type 7 from the section on summaries; ggplot2's notes for the newer 4.0.3 name 7 as the default. Check them:

```
penguins |>
  summarise(
    q1_g = quantile(body_mass_g, probs = 0.25, na.rm = TRUE),
    median_g = median(body_mass_g, na.rm = TRUE),
    q3_g = quantile(body_mass_g, probs = 0.75, na.rm = TRUE),
    .by = species
  )
```

```
# A tibble: 3 × 4
  species    q1_g median_g  q3_g
  <chr>     <dbl>    <dbl> <dbl>
1 Adelie     3350      3700  4000
2 Gentoo     4700      5000  5500
3 Chinstrap 3488.      3700  3950
```

The Adelie box runs from 3350 g to 4000 g with its median at 3700 g. The Gentoo box runs from 4700 g to 5500 g. A boxplot's side axis need not start at zero. Every box, whisker and point carries its value by position, read off the printed scale, as Book 0 said of a line chart.

A bar chart is different. A bar's value is its length, so its value axis must start at zero. `geom_bar()` counts the rows in each category, as `count()` does:

```
ggplot(penguins, aes(x = species)) +
  geom_bar() +
  labs(x = "Species", y = "Rows in the file")
```

On your screen the bars stand at 152, 68 and 124. A bar chart of means, with the axis cut to start at 3000 g, would make the Gentoo bar look about three times the Adelie bar when its mean is about 1.4 times as large. Keep bar charts for counts and amounts that start at zero, and show distributions with boxes or histograms.

WHERE THIS PICTURE BREAKS. A box hides how many values it summarises: a box drawn from 5 values looks as solid as one drawn from 150. Print the n per group beside the figure, or in its caption.

The 1.4 is rounded: 5076 divided by 3701 is about 1.37. It compares two means in this file and says nothing about how different a typical Gentoo and Adelie are.

MUST KNOW POINTS FOR YOU.

- Write a colour or size inside `aes()` only when it should vary with a column. Inside, a fixed value such as "blue" is treated as data.
- Read every message and warning ggplot2 prints. "Removed 2 rows" means your figure shows fewer observations than your data frame holds.
- The error is to think a histogram shows the shape of the data. It shows the data cut into bins someone chose.
- A bar chart's value axis starts at zero, because a bar's value is its length. A boxplot or a scatter plot need not, because position carries the value.
- Every axis title names the quantity and its unit, and every figure carries a source line.

- Save every figure with `ggsave()` from the script, with its width, height and units written in.
- When a trainee's thesis figure looks wrong, ask for the code before the picture.
- The grammar draws exactly what the code says. It cannot guarantee that the data, the bins or the rows it silently drops are right.

EXERCISE 1 (calculation). Add to your penguin script a histogram of flipper length. Use bins 10 mm wide starting at 170 mm, each including its left edge. Give it an axis title with its unit, and a source line. Save it to `output/flipper_histogram.png` at 15 cm by 10 cm. Write the counts per bin to `output/flipper_histogram_counts.csv`. Say how many penguins the figure shows.

EXERCISE 2 (critique). A colleague makes the figures for her thesis by running code in the console. Then she clicks Export in RStudio's plot pane and chooses "Save as image". She drags the size until it looks right. After a data correction, she has to remake eleven figures. Say what is wrong with this way of working and what she should do.

EXERCISE 3 (teaching). A journalist has three minutes. She has a bar chart from a press release. One district's bar for "adults screened" looks five times another's, and the axis starts at 9,000. Explain what she should ask for, without jargon.

EXERCISE 4 (retrieval). Without looking back, name the three parts of a ggplot2 plot, and two choices that change a histogram's bars without changing the data.

PRACTICE. Twelve problems on this technique, easiest first. Work them on paper before turning to the answers, which are in the appendix at the back under this section's name.

1. In this line, name the data, each aesthetic mapping, and the geom.

```
ggplot(clinic, aes(x = age_years, y = bmi)) + geom_point()
```

2. In each line, say whether the colour is mapped or set, and what the points will look like.

```
ggplot(clinic, aes(x = age_years, y = bmi, colour = sex)) + geom_point()
ggplot(clinic, aes(x = age_years, y = bmi)) + geom_point(colour = "darkgreen")
```

3. Six made-up values: 1.2, 1.8, 2.1, 2.9, 3.0 and 3.4. A histogram uses `binwidth = 1` and `boundary = 1`, with ggplot2's default for which edge a bin includes. How many values go in the bins from 1 to 2, 2 to 3 and 3 to 4? Then answer again with `closed = "left"`.

4. Read the penguin file.

```
library(readr)
library(dplyr, warn.conflicts = FALSE)
library(ggplot2)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)
```

Draw a boxplot of body mass by island, with axis titles. Then give the median and the quartiles the boxes are drawn from, with the number of body masses behind each box.

5. Read the penguin file.

```
library(readr)
library(ggplot2)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)
```

Draw a scatter plot of culmen depth against culmen length, one panel per species. The culmen is the ridge of the bill. Give axis titles that carry units, and a source line. Save it to `output/culmen.png` at 18 cm by 8 cm, and show that the file exists.

6. Read the penguin file.

```
library(readr)
library(dplyr, warn.conflicts = FALSE)
library(ggplot2)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)
```

Draw a bar chart of the number of rows for each value of `Sex`, with axis titles. Give the heights of the bars, and say what the third bar is.

7. A colleague wants all points dark red, and writes:

```
ggplot(penguins_raw, aes(x = `Flipper Length (mm)`, y = `Body Mass (g)`, colour =
"darkred")) +
  geom_point()
```

The points come out pinkish red, and a legend says “darkred”. She decides her version of R “cannot show dark red”. Find what broke.

8. You and a colleague run the same script on the same penguin file. Your histogram of body mass, made with `geom_histogram()` and no other arguments, has a different shape from hers. You have ggplot2 3.4.4; she has 4.0.3. She concludes one of the two data files must be corrupt. Say what is more likely, and how to make the two figures agree.

9. A colleague used to the pipe writes:

```
library(readr)
library(ggplot2)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)
```

```
ggplot(penguins_raw, aes(x = `Flipper Length (mm)`, y = `Body Mass (g)`)) |>
  geom_point()
```

```
Error in `geom_point()`:
! `mapping` must be created by `aes()`
ℹ Did you use `%>%` or `|>` instead of `+`?
```

Read the error. Say what broke and fix it.

10. A made-up thesis figure caption says: “Figure 3. Body mass of all 344 penguins, by species.” The code was:

```
library(readr)
library(ggplot2)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)

ggplot(penguins_raw, aes(x = Species, y = `Body Mass (g)`)) +
  geom_boxplot()
```

```
Warning message:
Removed 2 rows containing non-finite values (`stat_boxplot()`).
```

Find what is wrong with the caption, using only what R printed.

11. A made-up draft says: “As Figure 1 shows, most penguins weigh between 3.5 and 4 kg.” Draw the histogram that would back the claim, and count the penguins in that range. Say whether the claim holds, and what the figure does not establish.

12. A made-up district officer says: “Give me one figure comparing BMI in men and women from that American survey, for my slides.” Make it from the joined NHANES files for adults aged 20 or over, save it to `output/`, and say what the figure does not establish.

19 · Checking the data in code

DEFINITION. A check, in this book, is a line of code that states something you believe about the data and stops the script with an error when it is false.

`stopifnot()`, from base R, takes one or more expressions. If any of them is not `TRUE` in every element, it stops with an error naming the first expression that failed, and the expressions after it are not evaluated. A missing value counts as not `TRUE`, so an expression that gives `NA` fails. When an expression is given a name, the name is printed in place of the expression. Without a name it prints the expression, as in `all(x > 0) is not TRUE`. When every expression is `TRUE`, `stopifnot()` prints nothing.

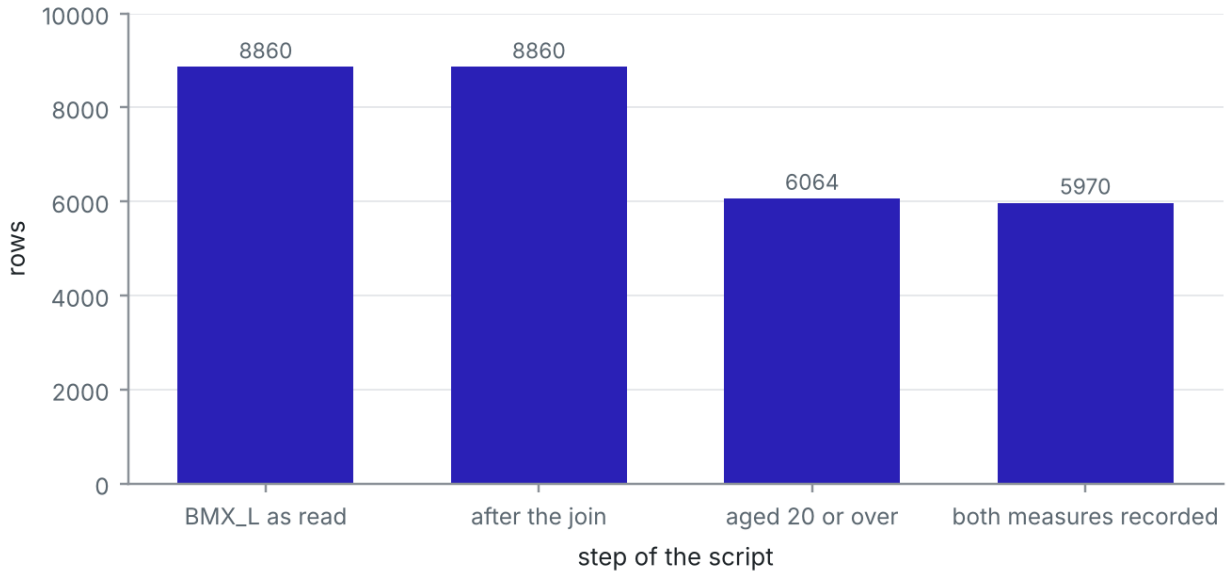
Three checks recur. A range check tests that every recorded value of a column lies between a lower and an upper limit written in the code. The limits come from a source or a stated decision, never from the smallest and largest values in the same data. A key is the column, or set of columns, meant to pick out one row each. A key check tests that the key has as many different values as the data have rows. `n_distinct()`, from dplyr, counts the different values or combinations. A row-count check stores the number of rows before a step that could change it, such as a join, and compares it with the number after. [319, 320, 321, 322, 323, 197, 324, 325, 326, 327, 328, 329, 330]

IN PLAIN TERMS. `stopifnot()` is the tool. Give each test a name that says the belief in words, and the error prints that name.

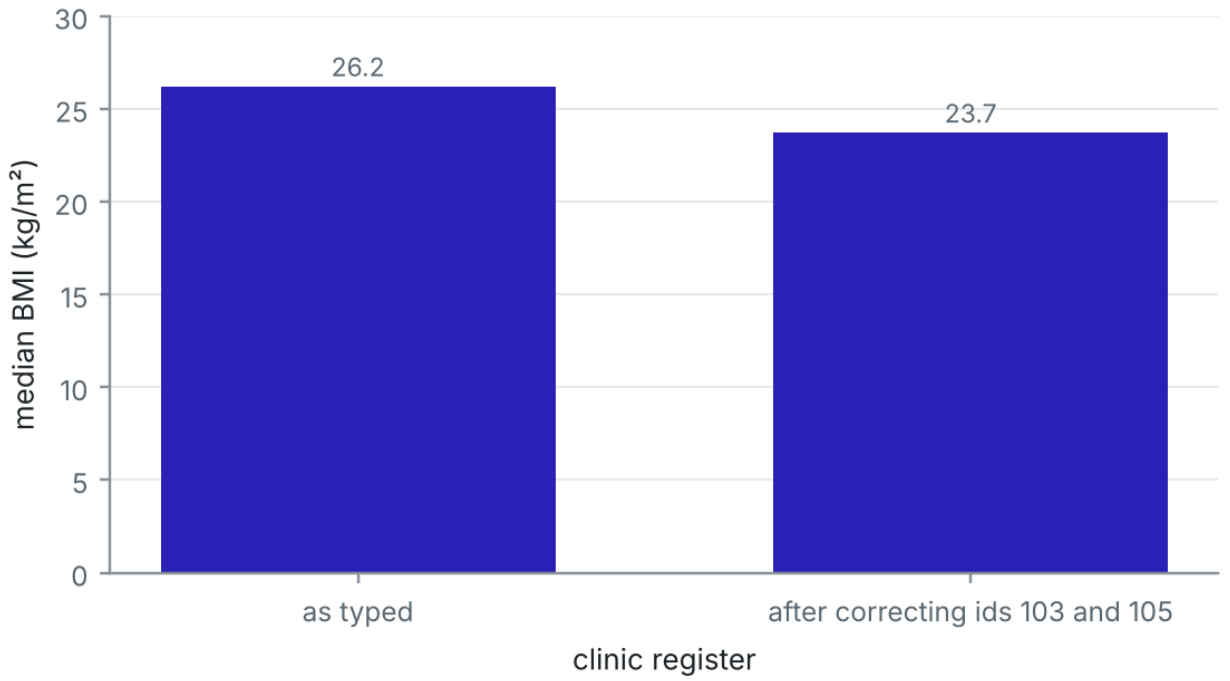
Three checks cover most of what goes wrong.

- **Range.** Every height lies between two limits you wrote down.
- **Key.** Every row has its own ID. Count the different IDs with `n_distinct()` and compare with the number of rows.
- **Row count.** Count the rows before a join or a filter and again after. Then compare.

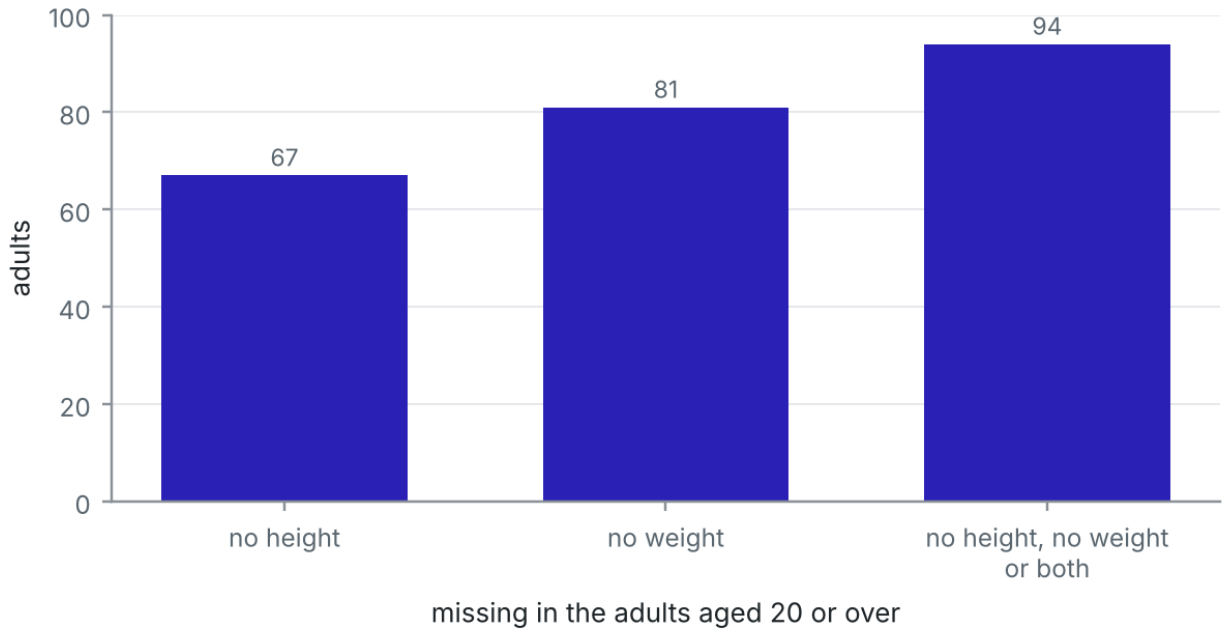
A check that fails is a question, not a verdict. Look at the rows it found. Go back to the paper register, the form or the person who measured. Then write the fix as a line of code, with the date and the reason in a comment. Never fix it in the raw file.



Rows after each step of the NHANES 2021–2023 checking script: 8860 in the body measures file as read, 8860 after joining the demographics file on SEQN, 6064 aged 20 or over, and 5970 with both height and weight recorded. The 94 adults missing a height, a weight or both are the difference between the last two bars. These are counts of rows in the files, unweighted, and estimate nothing about the United States.



Median BMI of the six adults in the made-up clinic register, before and after the range checks. As typed, with patient 103's height as 1.6 and patient 105's weight as 620, the median is 26.2 kg/m^2; with both corrected in code from the paper register, it is 23.7 kg/m^2. Both bars look like an ordinary clinic; only the check told them apart.



Missing measures among the 6064 NHANES adults aged 20 or over, counted after the height range check failed on them: 67 have no height, 81 have no weight, and 94 have no height, no weight or both (6064 - 5970). The bars overlap: 67 + 81 - 94 = 54 adults lack both. A report states these counts beside any result from the 5970.

Illustration 1. Start with an analysis that looks finished. Here is a made-up clinic register of six adults, typed from paper. Heights are in centimetres and weights in kilograms. Two values were typed wrong, and nothing in the code below knows it.

```
library(dplyr, warn.conflicts = FALSE)

clinic <- tibble(
  id = c(101, 102, 103, 104, 105, 106),
  height_cm = c(152, 160, 1.6, 171, 158, 165),
  weight_kg = c(41, 64, 58, 80.2, 620, 58)
)
```

Compute each BMI from height in metres, as the section on making new columns did, and report the median:

```
clinic |>
  mutate(bmi = weight_kg / (height_cm / 100)^2) |>
  summarise(n = n(), median_bmi = median(bmi))
```

```
# A tibble: 1 × 2
      n median_bmi
  <int>      <dbl>
1     6       26.2
```

The code runs with no warning. A median BMI of 26.2 kg/m^2 in a clinic of six adults looks ordinary. You could put it in a report, and nobody reading the report would object.

The median hides the damage. Patient 103's height was typed as 1.6, a height in metres, and patient 105's weight as 620 for 62. Their BMIs are absurd, but the median looks only at the middle of

the ordered values, so the median still looks ordinary. It is wrong all the same: the two absurd values went to the top and pushed two ordinary ones into the middle. With six rows you might have spotted them by eye. With six thousand you would not.

Now write down what you believed about the data, as code. Every height lies between 50 and 250 cm. Every weight lies between 20 and 300 kg. These limits are a decision, not a fact about people. They are set wide on purpose, so that a value outside them is one you must look at before you use it. Write them where anyone reading the script can see them and argue with them, as Book 0 asked of every guess. `all()` turns many `TRUE` and `FALSE` values into one, and it is where `na.rm = TRUE` goes.

```
stopifnot(
  "every height is between 50 and 250 cm" =
    all(clinic$height_cm >= 50 & clinic$height_cm <= 250),
  "every weight is between 20 and 300 kg" =
    all(clinic$weight_kg >= 20 & clinic$weight_kg <= 300)
)
```

```
Error: every height is between 50 and 250 cm
```

The script stops, and the message is the belief that failed. It stopped at the first failure, so the weight check never ran. To see every row that breaks either limit, filter for them:

```
clinic |>
  filter(
    height_cm < 50 | height_cm > 250 |
      weight_kg < 20 | weight_kg > 300
  )
```

```
# A tibble: 2 × 3
     id height_cm weight_kg
  <dbl>     <dbl>     <dbl>
1   103       1.6        58
2   105     158         620
```

Two rows: patient 103 and patient 105. Now go back to the paper register; in this made-up story it reads 160 cm for patient 103 and 62 kg for patient 105. Correct both in code. Pick out each row by its ID, never by its position. Put the corrected values in new columns, so the typed values stay beside them. Say in a comment when you made the change and where the right value came from.

```
clinic_checked <- clinic |>
  mutate(
    # 2026-10-02: register page for id 103 reads 160 cm; it was typed
    # as 1.6, a height in metres. Corrected here, raw file not edited.
    height_cm_checked = if_else(id == 103, 160, height_cm),
    # 2026-10-02: register page for id 105 reads 62 kg, typed as 620.
    weight_kg_checked = if_else(id == 105, 62, weight_kg)
  )

stopifnot(
  "every height is between 50 and 250 cm" =
    all(clinic_checked$height_cm_checked >= 50 &
      clinic_checked$height_cm_checked <= 250),
  "every weight is between 20 and 300 kg" =
    all(clinic_checked$weight_kg_checked >= 20 &
      clinic_checked$weight_kg_checked <= 300)
)
```

This time `stopifnot()` prints nothing. Silence is the answer you wanted: every test was `TRUE`. Now the median:

```
clinic_checked |>
  mutate(
    bmi = weight_kg_checked / (height_cm_checked / 100)^2
  ) |>
  summarise(n = n(), median_bmi = median(bmi))
```

```
# A tibble: 1 × 2
      n median_bmi
  <int>      <dbl>
1     6       23.7
```

The median moved from 26.2 to 23.7 kg/m^2. Both numbers looked reasonable. Only one was right, and only the check could tell them apart.

WHERE THIS PICTURE BREAKS. The six patients, the two slips and the paper register are made up. The code is the point, not the BMIs.

The limits catch a slip that makes a value far too large or far too small: a height in metres, a weight with an extra zero. They never catch a slip that stays inside them. A height of 126 cm typed for 162 cm passes both checks, and so would a weight of 58 kg typed for 85 kg. Like Book 0's order-of-magnitude check, a range check finds the impossible, not the wrong.

Illustration 2. Now run the same kind of checks on real data. The National Health and Nutrition Examination Survey (NHANES) of the United States, 2021–2023 cycle, gives its body measures in one file and its demographics in another, as the section on reshaping and joining used them. Each row of each file is one participant, with an ID in the column `SEQN`.

```
library(haven)
library(dplyr, warn.conflicts = FALSE)

bmx <- read_xpt(here::here("data-raw", "BMX_L.xpt"))
demo <- read_xpt(here::here("data-raw", "DEMO_L.xpt"))
```

First the key check. The join assumes that `SEQN` picks out one row in each file. Write that down:

```
stopifnot(
  "SEQN is unique in BMX_L" = n_distinct(bmx$SEQN) == nrow(bmx),
  "SEQN is unique in DEMO_L" = n_distinct(demo$SEQN) == nrow(demo)
)
```

Nothing printed, so both are true. Next the row-count check. Store the number of rows before the join, join, and compare:

```
n_before <- nrow(bmx)

joined <- bmx |>
  left_join(demo, by = "SEQN")

stopifnot("the join kept every row" = nrow(joined) == n_before)

nrow(joined)
```

```
[1] 8860
```

Still 8860 rows, as before the join. The section on reshaping and joining wrote the same promise into the join itself, with `relationship = "one-to-one"`. The row-count check does the same job after any step, not only a join. Now keep the adults, aged 20 or over:

```
adults <- joined |>
  filter(RIDAGEYR >= 20)

nrow(adults)
```

```
[1] 6064
```

6064 adults. Now the range checks, with the same limits as before. `BMXHT` is standing height in centimetres and `BMXWT` is weight in kilograms.

```
stopifnot(
  "adult heights are between 50 and 250 cm" =
    all(adults$BMXHT >= 50 & adults$BMXHT <= 250)
)
```

```
Error: adult heights are between 50 and 250 cm
```

The check fails. Before you hunt for a giant, count what made it fail:

```
sum(adults$BMXHT < 50 | adults$BMXHT > 250, na.rm = TRUE)
sum(is.na(adults$BMXHT))
```

```
[1] 0
[1] 67
```

No height lies outside the limits. But 67 adults have no height recorded, and a missing value is not `TRUE`, so `stopifnot()` fails on it. That is the right default: the check has made you notice the missing values. Now decide what you mean. The range check is about the heights that were

measured, so say so with `na.rm = TRUE` inside `all()`. The missing ones are counted and reported, as the section on missing values taught, not hidden.

```
stopifnot(
  "measured adult heights are between 50 and 250 cm" =
    all(adults$BMXHT >= 50 & adults$BMXHT <= 250, na.rm = TRUE),
  "measured adult weights are between 20 and 300 kg" =
    all(adults$BMXWT >= 20 & adults$BMXWT <= 300, na.rm = TRUE)
)

sum(is.na(adults$BMXWT))
```

```
[1] 81
```

Both pass, and 81 adults have no weight. Last, keep the adults with both measures, and count:

```
adults_measured <- adults |>
  filter(!is.na(BMXHT), !is.na(BMXWT))

nrow(adults_measured)
```

```
[1] 5970
```

Put the row counts side by side. Each line of the script that could change the number of rows now has a number beside it.

Step	Rows
BMX_L as read	8860
after the join	8860
aged 20 or over	6064
both measures recorded	5970

The last two rows differ by 94. That is the number of adults missing a height, a weight or both, and a report must say it.

One more thing to notice. The survey's own codebook prints a "Range of Values" for height of 79.1 to 200.7 cm. That is the smallest and largest height in the file, across all ages measured. It describes the file. A check with those numbers as its limits would pass on this file by construction, and so it could never catch an error in it.

Figures quoted in this illustration are from [178].

WHERE THIS PICTURE BREAKS. These are counts of rows in a file. The survey's documentation says its sample weights should be used to analyse the body measures data, and nothing here uses them. So no count in this example says anything about adults in the United States. Survey weights are taught in the subject on India's data infrastructure and complex survey analysis.

The public NHANES file has already been cleaned by the people who released it, so every range check on the recorded values passes. A register typed by hand in a clinic will rarely be so kind. The checks earn their place on the day they fail.

MUST KNOW POINTS FOR YOU.

- The error is to think that a script which runs without an error has checked the data. It has checked nothing you did not write as a check.
- Write each belief about the data as a `stopifnot()` line with a name that says the belief in words.
- A missing value fails `stopifnot()`. Do not delete the check to make the script run. Count the missing values and report them, then check the recorded values with `na.rm = TRUE` inside `all()`.
- Never set a range check's limits from the smallest and largest values in the same data. Limits taken from the data always pass on that data.
- Count the rows before and after every join and every filter, and compare the two in code. An ID repeated in the table you join to adds rows with no warning.
- Fix a wrong value with a line of code. Pick the row by its ID and write the right value into a new column. If no record gives the right value, set it to `NA` and report it.
- Never fix a row by its position, as in "row 3". Sorting or filtering earlier in the script moves the rows, and the fix lands on someone else without any error.
- A value outside your limits may still be a real person, and a repeated ID may be a second visit. Look before you correct.
- A range check catches a value outside its limits and never a wrong value inside them. A height of 126 cm typed for 162 cm passes.

EXERCISE 1 (critique). A colleague found a weight of 620 kg in their thesis spreadsheet. They looked up the paper form, which said 62, typed 62 into the cell, saved the file and reran their analysis. Say what is wrong with this, and what they should have done.

EXERCISE 2 (design). Your thesis dataset, made up for this exercise, is a CSV file of school children aged 6 to 14 from one district. It has one row per child. Its columns are `child_id`, `school_code`, `age_years`, `sex` (written `F` or `M`), `height_cm` and `weight_kg`. Write the checks you would put at the top of your script, straight after reading the file. For each limit, say where it comes from.

EXERCISE 3 (retrieval). Without looking back, write the three kinds of check this section uses, one line of code for each. Then say what `stopifnot()` does with a missing value.

EXERCISE 4 (teaching). A first-year resident has ten minutes and a whiteboard. They say: "I always look at my data before I analyse it, so I don't need checks in code." Teach them what a check in code does that looking does not.

PRACTICE. Thirteen problems on this technique, easiest first. Work them on paper before turning to the answers, which are in the appendix at the back under this section's name.

1. Before you run them, say what each line prints.

```
stopifnot(2 + 2 == 4)
stopifnot(c(5, 12, 7) > 0)
stopifnot(c(5, -12, 7) > 0)
```

2. Five made-up heights, in centimetres:

```
heights <- c(152, 1.6, 171, 158, 16.5)
```

Write one line that counts how many lie outside 50 to 250 cm.

3. Five made-up patient IDs:

```
library(dplyr, warn.conflicts = FALSE)

ids <- c(101, 102, 103, 103, 104)
```

Write a check that stops if any ID appears twice. Say what it prints.

4. Three made-up weights, in kilograms, one missing:

```
weights <- c(58, NA, 72)
```

Say what `stopifnot(all(weights > 20))` prints. Then write a version that checks only the recorded weights, and a line that counts the missing ones.

5. Read the penguin file:

```
library(readr)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)
```

Write a check that stops the script if any recorded body mass, in grams, lies outside 1000 to 10,000 g. Those limits are a decision made for practice, not a fact about penguins. Then count the rows with no body mass.

6. Read the penguin file:

```
library(readr)
library(dplyr, warn.conflicts = FALSE)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)
```

The section on the data dictionary found that species together with sample number picks out one row each. Write two checks: that the file has the 344 rows its help page states, and that species with sample number is a key. Then try the same key check on `Individual ID` alone, and say what happens.

7. Read the NHANES body measures file:

```
library(haven)

bmx <- read_xpt(here::here("data-raw", "BMX_L.xpt"))
```

The file has a BMI column in kg/m^2, named `BMXBMI`. Beside it are weight `BMXWT` in kilograms and height `BMXHT` in centimetres. Check in code that the BMI column agrees with weight divided by height in metres squared, on every row where all three are recorded. The file prints BMI to one decimal place, so allow a difference of up to 0.05.

8. A colleague checks the heights in a made-up register:

```
register <- data.frame(
  id = c(1, 2, 3),
  height_cm = c(158, 1.62, 171)
)

stopifnot(
  "heights are between 50 and 250 cm" =
    all(register$height_cm >= 50 | register$height_cm <= 250)
)
```

Nothing printed, and they write in their notes: “heights checked, all within range.” Find what broke.

9. A colleague corrects a height in a made-up register. The paper register gives 160 cm for patient 103.

```
library(dplyr, warn.conflicts = FALSE)

clinic <- tibble(
  id = c(101, 102, 103),
  height_cm = c(171, 160, 1.6)
)

clinic_sorted <- clinic |>
  arrange(height_cm)

# register page for id 103 reads 160 cm
clinic_sorted$height_cm[3] <- 160

clinic_sorted
```

```
# A tibble: 3 × 2
     id height_cm
  <dbl>     <dbl>
1   103       1.6
2   102     160
3   101     160
```

All three heights now look sensible. Find what broke.

10. A colleague writes a range check on made-up weights:

```
weights <- c(58, 64, 620, 71, 49)

lower <- min(weights, na.rm = TRUE)
upper <- max(weights, na.rm = TRUE)

stopifnot(
  "weights are within range" =
    all(weights >= lower & weights <= upper, na.rm = TRUE)
)
```

The check passes, and they report “all weights within range”. Find what broke.

11. A colleague joins made-up visit records to a made-up table of people:

```
library(dplyr, warn.conflicts = FALSE)

visits <- tibble(id = c(1, 2, 3), weight_kg = c(61, 74, 55))
people <- tibble(
  id = c(1, 2, 2, 3),
  sex = c("female", "male", "male", "female")
)

n_before <- nrow(visits)

joined <- visits |>
  left_join(people, by = "id")
```

```
stopifnot("the join kept every row" = nrow(joined) == n_before)
```

```
Error: the join kept every row
```

They say: “The check is too strict. I deleted it, the script runs, and the mean weight is 66 kg.” Find what broke.

12. A draft thesis on the NHANES 2021–2023 files says: “Height and weight were checked for all 6064 adults, and every value was plausible.” The adults are those aged 20 or over. Turn the sentence into code, with limits of 50 to 250 cm and 20 to 300 kg. Run it, and say what the result does not establish.

13. A colleague’s script reads the penguin file and computes the mean body mass of each species. In a made-up meeting they say: “The data are clean. The script ran without a single error.” Decide what you would have to compute before you could call the penguin file clean. Write it, run it, and say what it still does not establish.

20 · Does shared code rerun? What studies found

DEFINITION. A result is reproducible when the same data, analysed again with the same code, give the same result. A result is replicated when independent investigators collect new data on the same question and reach the same answer. Reproducing a result does not show that it is correct. Peng (2011) calls replication the ultimate standard for judging scientific claims. Between full replication and none he places a spectrum: a study is more or less reproducible depending on what data and code are made available.

Trisovic, Lau, Pasquier and Crosas (2022) measured re-execution at scale. They took 2109 replication packages, the data and code that authors deposit with a published paper, from the Harvard Dataverse, a website where researchers deposit such files. The packages were published from 2010 to July 2020 and held 9078 R files.

Each was run once as deposited and once after an automatic code cleaning. Files that ran out of time were excluded from the success rate.

Their abstract reports that 74% of the R files failed to complete without error on first execution, and 56% after code cleaning. Their results table gives success rates of 25% without cleaning, 40% with cleaning and 56% for the better of the two runs. Errors that remained included packages missing or of an incompatible version, wrong paths and output folders, and objects that did not exist. Journals with the strictest data-sharing policies had the highest re-execution rates. Most packages came from the social sciences, and all from one website. [331, 332, 333, 334, 335, 336, 337, 338, 339, 340, 341, 342, 343, 344, 345, 346, 347, 348, 349, 350, 351, 352]

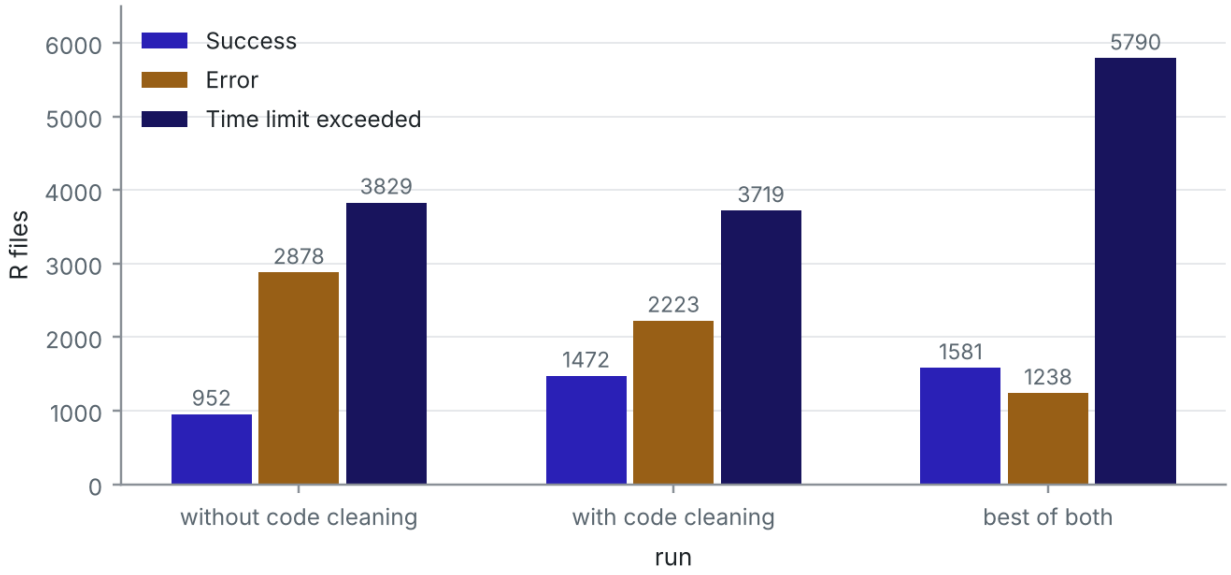
IN PLAIN TERMS. **Reproduce** means: take the same data, run the same code, get the same numbers.

Replicate means: collect new data on the same question and see whether the answer holds.

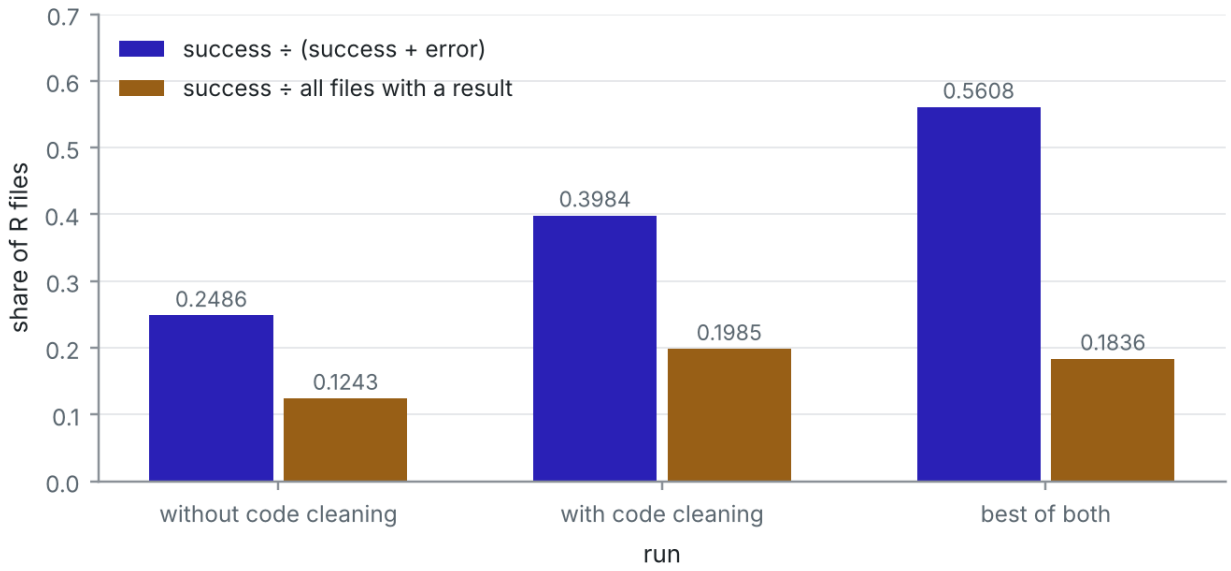
Reproducing is the smaller test. It cannot tell you whether the analysis was right, only whether it was done as described.

The first step of reproducing is that the code runs at all. One large study tested that. It took thousands of R files that researchers had shared with their published papers, and tried to run each on a clean computer. Most did not run.

A script that runs is still not a result reproduced. It may run and print different numbers.



Trisovic and colleagues' results table for R files shared on Harvard Dataverse, 2010 to July 2020: files that ran without error, stopped with an error, or exceeded the time limit. Without code cleaning: 952, 2878 and 3829 of 7659 files. With cleaning: 1472, 2223 and 3719 of 7414. Best of both runs: 1581, 1238 and 5790 of 8609. The table's success rates, 25%, 40% and 56%, leave out the files that ran out of time.



The same success counts from Trisovic and colleagues' table as shares on two bases. Over successes and errors only, the table's basis, they are 0.2486, 0.3984 and 0.5608: the table's 25%, 40% and 56%. Over every file with a result, time-outs included, they are 952 ÷ 7659 = 0.1243, 1472 ÷ 7414 = 0.1985 and 1581 ÷ 8609 = 0.1836. On that basis the best of both runs is no better than the run with cleaning, because 5790 of its files ran out of time. Per cent of what decides the number.

Illustration 1. Start with a script that worked, on the computer it was written on. Here are its first lines, made up but typical, and shown, not run:

```
setwd("C:/Users/resident/Desktop/thesis final")
library(janitor)

survey <- read.csv("data_clean_v3.csv")
```

Now move it to a colleague's computer and run it. The first line asks for a folder that exists only on the author's computer, and R stops. Suppose the colleague deletes it. The second line asks for a package their computer does not have, and R stops again. Suppose they install it. The third line asks for a file the author made by hand somewhere along the way and never shared, and R stops a third time. Nothing in the script was wrong on the day it was written. It depended on things that were not in the project folder.

Trisovic and colleagues, whose study the definition describes, measured how often this happens. The team tried to run each file on a clean computer under three versions of R. They gave each file up to one hour, and each package up to five. Not every file got a result: when earlier files used up a package's five hours, the rest never ran. Then they tried again after an automatic cleaning step. It removed absolute paths, the `C:/Users/...` kind, fixed how the text of each file was stored, and added lines that install any package the code loads, if it is missing.

Here is their results table, as they give it. A file counts as a success if any of the three versions of R ran it to the end without an error. "Time limit exceeded" means the file was still running when its hour ran out. The paper does not spell out the "best of both" row. Its counts fit the paper's rule applied to both runs together, so a file that errored in one run and ran out of time in the other is left out.

Run	Success	Error	Time limit exceeded
without code cleaning	952	2878	3829
with code cleaning	1472	2223	3719
best of both	1581	1238	5790

The table gives success rates of 25% without cleaning, 40% with cleaning and 56% for the best of both. Now ask Book 0's question: per cent of what? Add each row. Without cleaning, 952 + 2878 + 3829 = 7659 files had a result. With cleaning, 1472 + 2223 + 3719 = 7414. For the best of both, 1581 + 1238 + 5790 = 8609. The table gives the same totals.

Now divide each success count by successes and errors only, leaving out the files that ran out of time:

```
952 ÷ (952 + 2878) = 0.2486
1472 ÷ (1472 + 2223) = 0.3984
1581 ÷ (1581 + 1238) = 0.5608
```

Those round to the table's 25%, 40% and 56%. So the success rate is a share of the files that finished, one way or the other. The files that ran out of time are left out, as the paper says, because they might have run with more time. They are not a small group: 3829 of the 7659 files in the first run.

The abstract gives its figures the other way round: 74% of R files "failed to complete without error" on first execution, and 56% after cleaning. Those two are the authors' summary. The table's rates are on a different basis, and its 56% is a success rate where the abstract's 56% is a failure rate. The paper does not show how the abstract's two figures were worked out. So quote the abstract's sentence as the abstract's, and the table's rates with the table's labels, and never mix the two.

Last, the reasons. The cleaning fixed every error caused by `setwd()`, the line that set the working directory to a folder on the author's computer. It also fixed many errors about packages, by installing them. What was left included packages that were missing or of a version that no longer worked, paths to folders that did not exist, output files that were never saved, and objects the code used before anything had made them.

Figures quoted in this illustration are from [353].

WHERE THIS PICTURE BREAKS. The three-line script is made up. Its three failures are the kinds the study reports, not a finding about any one script.

The study is of one website, the Harvard Dataverse, and most of its packages came from the social sciences. Its success rate counts only whether a file ran to the end without an error. It does not say whether the file printed the paper's numbers.

Illustration 2. Now the second half of the claim: a file that runs is not a result reproduced. Trisovic and colleagues tested that too, on a small scale. They found 127 replication packages in which every R file ran without an error. They picked three at random and compared what the code printed with what the papers reported.

In the first, two of the three logs came out almost identical to the authors' own. The third nearly matched, but some decimal digits differed. A package that the code used was not available for the version of R it ran under, which may explain the difference. In the second, the script printed its two correlation coefficients but could not save the plot it was meant to make. In the third, the numbers printed, and the plots, which were meant to pop up on a screen, did not appear.

Three packages are too few to give a rate. What they show is the gap between the two words. "Ran without an error" is a fact about the code. "Reproduced" is a fact about the result, and it needs someone to compare the output with the paper.

Now place your own thesis on Peng's spectrum. Where it sits depends, in his words, on "what data and code are made available". At the far end, someone collects new data in another district and asks the same question: that is replication, and no amount of tidy code gives it to you.

A project folder that reruns from raw data with one command makes both your data and your code available. That is the build this book asks for.

Figures quoted in this illustration are from [353].

WHERE THIS PICTURE BREAKS. The three packages were picked at random from 127, and three is a handful. The study reports what it found in them; it does not give a share of shared code that reproduces its paper.

The spectrum is Peng's description, not a scale with measured steps. Where a given study sits on it is a judgement about what was shared, not a number.

MUST KNOW POINTS FOR YOU.

- One study took 9078 R files shared with published papers on the Harvard Dataverse, deposited from 2010 to July 2020. Its authors report that 74% failed to run without an error on first execution. Quote it with that source, sample and date, never as "most research code is broken".
- The error is to think that sharing your code makes your work reproducible. Code that sets the working directory to your own folder stops on its first line on another computer.
- Before you quote a percentage from a study, find what it is a share of. The study's 25% success rate leaves out the files that ran out of time; its abstract's 74% "failed" is on another basis.
- Run your script on a clean computer, or in a fresh R session, before you share it.
- Code that runs is not a result reproduced.
- A reproducible analysis can still be wrong. Do not offer reproducibility as evidence that a finding is true.
- Journals with the strictest data-sharing policies had the highest re-execution rates in this study. That is a correlation across a few journals, not a measured effect of the policy.
- The finding is that most R files shared on Harvard Dataverse from 2010 to July 2020 did not rerun as deposited. Studies of other websites, fields and years that find most files running would overturn it.

EXERCISE 1 (critique). A classmate's thesis says, in its methods: "All code is available on request, so the analysis is fully reproducible." Say what is wrong with the sentence, using this section's evidence, and write a sentence that would be true.

EXERCISE 2 (interpretation). A blog post says: "A Harvard study found only 25% of research code works." Using the results table in this section, say what the 25% is a share of, what it leaves out, and what the sentence gets wrong.

EXERCISE 3 (retrieval). Without looking back, say what "reproduce" and "replicate" each mean. Then name three reasons the study in this section found for shared R code failing to run.

EXERCISE 4 (teaching). A journalist has three minutes and asks you: "Is it true that most scientists' code doesn't even run?" Answer without jargon, say how sure you are, and give them one sentence they could quote.

21 · Top to bottom in a fresh session, with one command

DEFINITION. Hidden state is anything a script uses that the script itself did not make.

A script is tested for hidden state by running all of it, from the first line to the last, in a fresh session. A fresh session is a new R process whose workspace is empty. It has attached no package beyond R's defaults. The one command for this is `Rscript code/analysis.R`, typed in a terminal whose working directory is the project root.

A script that passes this test also writes every result it makes into `output/` with code. It sets the random-number seed with `set.seed()` before any random step. It ends with `sessionInfo()`, which prints the version of R and of every attached package. [354, 355, 356, 357, 358, 359, 360, 361, 362, 363, 364, 365, 366, 367, 368, 369, 370, 371, 372]

IN PLAIN TERMS. Here is the trap with a script that works on your computer. While you work, R remembers. Every object you made, in the script or typed in the console, sits in the workspace. Every package you attached stays attached. Then a colleague runs it on their computer, where none of it exists, and it stops.

So test the script the way a stranger would run it. Start a new R with nothing in it, and run every line from the top. There are two ways to do that.

- In RStudio, restart R (the menu Session, then Restart R), then run the whole script.
- In a terminal, in the project folder, type one command: `Rscript code/analysis.R`.

The second way is the one to write in your README, because it is one line anyone can type.

Illustration 1. Start with a script that works for you and for nobody else. An hour ago, exploring the penguin file, you typed these lines in the console:

```
library(readr)
library(dplyr, warn.conflicts = FALSE)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)
```

Then you tried a few things, and saved the line worth keeping as your script, `code/analysis.R`:

```
library(dplyr, warn.conflicts = FALSE)

penguins_raw |>
  count(Island)
```

Select all of it in the editor and run it. It prints the rows per island, because `penguins_raw` is still in your session from the console.

It works. Now run it the way anyone else would. Open a terminal in the project folder and type the one command:

```
Rscript code/analysis.R
```

```
Error: object 'penguins_raw' not found
Execution halted
```

Read the error literally. The new R has no object called `penguins_raw`. Nothing in the script makes one. In your session it existed because you typed the reading line in the console an hour ago. The script was never complete. Your workspace was hiding the gap.

Trisovic and colleagues reran R files that researchers had shared alongside published studies, and 74% failed to complete without error the first time they were run, as the last section showed. "Object not found" was one of the errors they met, and they write that it is "often related to missing files or incomplete code". This one is incomplete code.

Fix it in the script, not in the session. Put the packages first, then the reading line:

```
library(readr)
library(dplyr, warn.conflicts = FALSE)
library(here)

penguins_raw <- read_csv(
  here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)

penguins_raw |>
  count(Island)
```

Run the same command again:

```
Rscript code/analysis.R
```

```
here() starts at ~/project
# A tibble: 3 × 2
  Island        n
  <chr>     <int>
1 Biscoe      168
2 Dream       124
3 Torgersen    52
```

The same three counts, now from a script that makes everything it uses. Notice that `Rscript` printed the table without being asked. It prints what each line returns, just as the console does.

WHERE THIS PICTURE BREAKS. The fresh run catches what the script takes from your session: objects, attached packages, a working directory you set by hand. It does not catch what the script takes from your computer. A package that is installed on your computer and not on your

colleague's still stops the script on theirs, at its `library()` line. A different version of a package may run without any error and give a different result.

`Rscript` also stops at the first error. A script with three problems shows you one at a time. Fix it and run it again until it reaches the last line.

Illustration 2. Now the second trap: clearing the workspace by hand looks like a fresh start and is not one. The function `rm()` removes objects. `rm(list = ls())` removes every object the workspace lists. Run it, then ask what is left:

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

```
character(0)
 [1] ".GlobalEnv"        "package:dplyr"     "package:readr"
 [4] "package:stats"     "package:graphics"  "package:grDevices"
 [7] "package:utils"     "package:datasets"  "package:methods"
[10] "Autoloads"         "package:base"
```

`ls()` prints `character(0)`, an empty list: no objects. But `search()`, which lists where R looks for a name, still shows `package:dplyr` and `package:readr`. They are still attached. A script that forgot `library(dplyr)` would still find `count()` here, and stop on your colleague's computer. Only a new R clears both. In RStudio that is Session, then Restart R, or Ctrl + Shift + F10, followed by Ctrl + Shift + S to run the whole script. In a terminal it is `Rscript`.

The same holds for `source()`. It reads a file and runs its lines in your current workspace, so a script run with `source()` can still lean on objects and packages you already have. Use it only straight after a restart.

Now make the one command produce everything. Two more things go into the script. The first is a random step. Suppose you want five rows picked at random, to check by hand against the field notebooks. The function `sample()` picks at random, and `sample(344, 5)` picks five of the whole numbers from 1 to 344. Each run picks different ones, unless you set the seed first. The seed is the starting point R's random numbers are worked out from. See what it does:

```
set.seed(2026)
sample(344, 5)
set.seed(2026)
sample(344, 5)
set.seed(7)
sample(344, 5)
```

```
[1] 221 121 294  45 111
[1] 221 121 294  45 111
[1] 298 103 194 271 218
```

The same seed gives the same five rows. A different seed gives different ones. The seed is a number you choose and write in the script, so the next person gets your five rows, not their own.

The second is `sessionInfo()` as the last line. Here is the whole script:

```
# analysis.R: penguin rows by island, and five rows to check by hand
# Run from the project folder: Rscript code/analysis.R

library(readr)
library(dplyr, warn.conflicts = FALSE)
library(here)

penguins_raw <- read_csv(
  here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)

dir.create(here("output"), showWarnings = FALSE)

island_counts <- penguins_raw |>
  count(Island)
write_csv(island_counts, here("output", "island_counts.csv"))

# Five row numbers, picked at random, to check against the
# field notebooks. The seed makes the same five come out every run.
set.seed(2026)
rows_to_check <- tibble(row = sample(nrow(penguins_raw), 5))
write_csv(rows_to_check, here("output", "rows_to_check.csv"))

sessionInfo()
```

`dir.create()` makes the `output` folder if it is not there, and `showWarnings = FALSE` keeps it quiet when it is. Without that line the script would stop on a fresh copy of the project, where `output/` does not exist yet. `write_csv()` writes a table to a CSV file, as in the section on summaries by group. `tibble()` makes a small table, here with one column, `row`.

Run it:

```
Rscript code/analysis.R
```

```
here() starts at ~/project
R version 4.3.3 (2024-02-29)
Platform: x86_64-pc-linux-gnu (64-bit)
Running under: Ubuntu 24.04.5 LTS

Matrix products: default
BLAS:   /usr/lib/x86_64-linux-gnu/blas/libblas.so.3.12.0
LAPACK: /usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.12.0

locale:
 [1] LC_CTYPE=C.utf8       LC_NUMERIC=C          LC_TIME=C.utf8
 [4] LC_COLLATE=C.utf8     LC_MONETARY=C.utf8    LC_MESSAGES=C.utf8
 [7] LC_PAPER=C.utf8       LC_NAME=C             LC_ADDRESS=C
[10] LC_TELEPHONE=C        LC_MEASUREMENT=C.utf8 LC_IDENTIFICATION=C

time zone: UTC
tzcode source: system (glibc)

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
[1] here_1.0.1  dplyr_1.1.4 readr_2.1.5

loaded via a namespace (and not attached):
 [1] utf8_1.2.4       R6_2.5.1         bit_4.0.5        tidyselect_1.2.0
 [5] tzdb_0.4.0       magrittr_2.0.3   glue_1.7.0       tibble_3.2.1
 [9] parallel_4.3.3   pkgconfig_2.0.3  bit64_4.0.5      generics_0.1.3
[13] lifecycle_1.0.4  cli_3.6.2        fansi_1.0.5      vctrs_0.6.5
[17] compiler_4.3.3   rprojroot_2.0.4  tools_4.3.3      hms_1.1.3
[21] pillar_1.9.0     crayon_1.5.2     rlang_1.1.3      vroom_1.6.5
```

Nothing else printed, because the script writes its tables to files. Look in `output/`, and read one back as plain text. In the terminal, `ls` lists a folder and `cat` prints a file:

```
ls output
cat output/rows_to_check.csv
```

```
island_counts.csv
rows_to_check.csv
row
221
121
294
45
111
```

The five row numbers in the file are the five from `set.seed(2026)` above. Now read the record at the bottom of the run. The first line names R 4.3.3. Under “other attached packages” are the three this script attached, each with its version: here 1.0.1, dplyr 1.1.4, readr 2.1.5. Under “loaded via a namespace (and not attached)” are packages that are loaded but not attached. “Running under” names the computer’s operating system. Anyone who gets a different answer from your script can compare their own `sessionInfo()` with yours, line by line.

WHERE THIS PICTURE BREAKS. `sessionInfo()` records the versions. It does not install them. A colleague with a newer dplyr can read that you used 1.1.4 and still have to find and install the old

version by hand. Pinning versions so that another computer gets the same ones is what renv does, two rungs up. Recording every change to the script itself is version control, the next rung's subject.

A seed makes a random step repeat on the same version of R. It does not make the five rows a good sample, and it does not promise the same five on a version of R whose random-number method has changed, as it did in R 3.6.0. That is why the seed and the R version are both recorded.

MUST KNOW POINTS FOR YOU.

- The error is to think that a script which runs on your computer is a script that runs. Before you send a script to anyone, run it with `Rscript code/analysis.R` from the project folder. Send it only when it reaches the last line.
- `rm(list = ls())` empties the workspace and leaves every package attached. `source()` runs a file in the workspace you already have. Neither is a fresh session. Restart R, or use `Rscript`.
- When a script stops with "object not found" on someone else's computer, look for the line that makes that object. Add it to the script, above the line that uses the object.
- Put `set.seed()` with a number you write down before any random step: a random sample of rows to check, a random allocation, a simulation.
- End every analysis script with `sessionInfo()`, and keep its printout with the outputs.
- A number in your thesis that you typed from the screen has no line of code behind it. Write every table to `output/` with code, and copy numbers into the text from those files.
- A trainee says "it runs, I checked". Ask how. If the answer is "I ran it in RStudio", ask them to restart R and run it again in front of you.
- One command in a fresh session tests the script against your computer only. It cannot show that a colleague with other package versions gets the same numbers, and it cannot show that the numbers are right.

EXERCISE 1 (retrieval). Without looking back, say what "hidden state" is. Name the one command that tests a script for it. Say why `rm(list = ls())` is not enough.

EXERCISE 2 (build). Take the script you have been building through this book. Restart R, or open a terminal in the project folder, and run it with `Rscript code/analysis.R`. Fix every error in the script, one at a time, until it reaches the last line. Then make sure that it writes its outputs to

`output/` with code, that any random step follows `set.seed()`, and that it ends with `sessionInfo()`. Delete `output/` and run the command again. Write the command into your README.

EXERCISE 3 (teaching). A journalist has three minutes. A research group has just been told that its published analysis "does not reproduce". The journalist asks what that usually means, and whether it means the result is false. Answer them.

PRACTICE. Ten problems on this technique, easiest first. Work them on paper before turning to the answers, which are in the appendix at the back under this section's name.

1. Predict whether these lines print the same three numbers twice, then run them.

```
set.seed(11)
sample(10, 3)
set.seed(11)
sample(10, 3)
```

2. Predict what the last line prints, then run all three. `exists()` asks whether an object of that name exists.

```
height_cm <- c(132, 141, 150)
rm(list = ls())
exists("height_cm")
```

3. Predict what the last line prints, then run all three. Does `rm(list = ls())` detach dplyr?

```
library(dplyr, warn.conflicts = FALSE)
rm(list = ls())
"package:dplyr" %in% search()
```

4. Write a script, `code/analysis.R`, that reads the penguin file from `data-raw/`, counts the rows of each species, and writes the counts to `output/species_counts.csv`. It must run on a fresh copy of the project, where `output/` does not exist. Run it with `Rscript`, then show the file it wrote.

5. This section's last run of `code/analysis.R` ended with `sessionInfo()`, and its printout named the version of dplyr under "other attached packages". Get the same version number with one line of code instead of reading it by eye.

6. You want ten penguin rows picked at random, for a colleague to check against the field notebooks. She must get the same ten when she runs your script. Write the lines that pick ten row numbers from the penguin file with seed 52. Show that running them twice gives the same ten.

7. Meera's script reports the mean body mass of the penguins. Earlier in the same session she had typed two lines in the console. Here is everything her session ran, and what it printed:

```
library(readr)
library(dplyr, warn.conflicts = FALSE)
penguins_raw <- read_csv(here::here("data-raw", "penguins_raw.csv"),
                         show_col_types = FALSE)
penguins_raw <- penguins_raw |>
  filter(Island == "Biscoe")
```

Her script, `code/analysis.R`, run line by line in that session:

```
library(dplyr, warn.conflicts = FALSE)
penguins_raw |>
  summarise(mean_mass_g = mean(`Body Mass (g)`, na.rm = TRUE))
```

```
# A tibble: 1 × 1
  mean_mass_g
        <dbl>
1       4716.
```

She writes “mean body mass 4716 g” in her draft. What broke, and what is the right number for the whole file?

8. Ravi checks his script before sending it. In his session the penguin file is already read. He runs the script with `source()`, sees no error, and tells his guide it is reproducible:

```
library(readr)
penguins_raw <- read_csv(here::here("data-raw", "penguins_raw.csv"),
                         show_col_types = FALSE)
```

```
n_rows <- nrow(penguins_raw)
```

```
source(here::here("code", "analysis.R"))
n_rows
```

```
[1] 344
```

What is wrong with his check?

9. A thesis’s methods chapter says: “All analyses were performed in R and are fully reproducible.” The candidate gives you the project folder. Decide what to run to test the sentence, write it down, and then say what a successful run would still not establish.

10. In a department meeting a colleague says: “My script gives the same table every time. I ran it twice yesterday in RStudio and got the same numbers.” His script reads the penguin file and writes the number of rows of each sex to `output/sex_counts.csv`. Decide what would actually test his claim, run it, and say what your result does not establish.

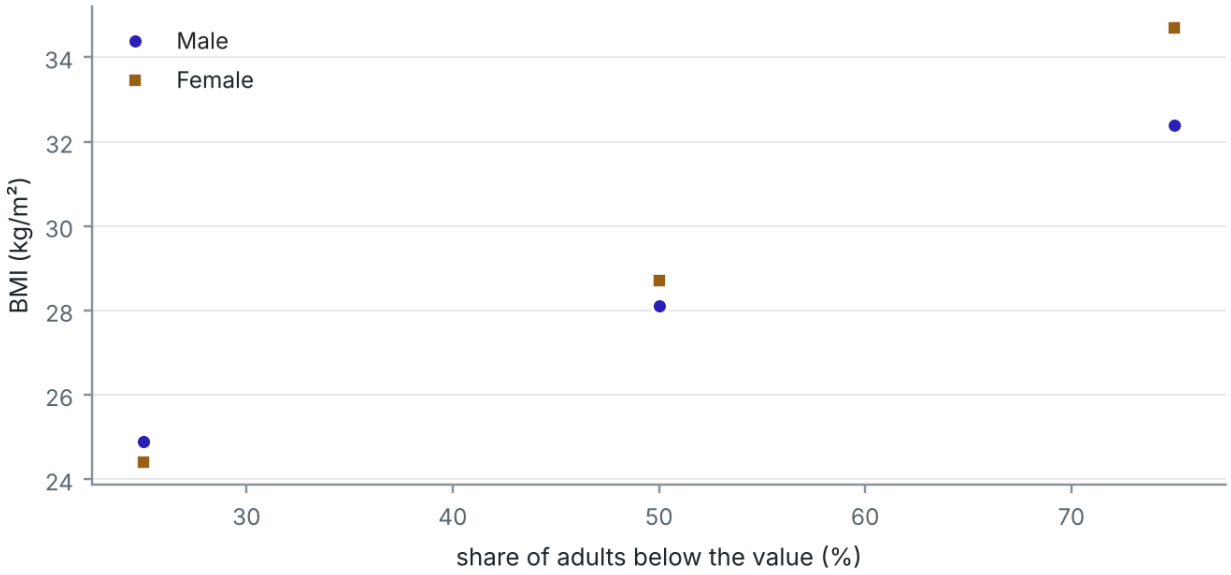
22 · Worked journey: One dataset, end to end

DEFINITION. An end-to-end analysis takes one question from raw data to its final table and figure in a single script that another person can run.

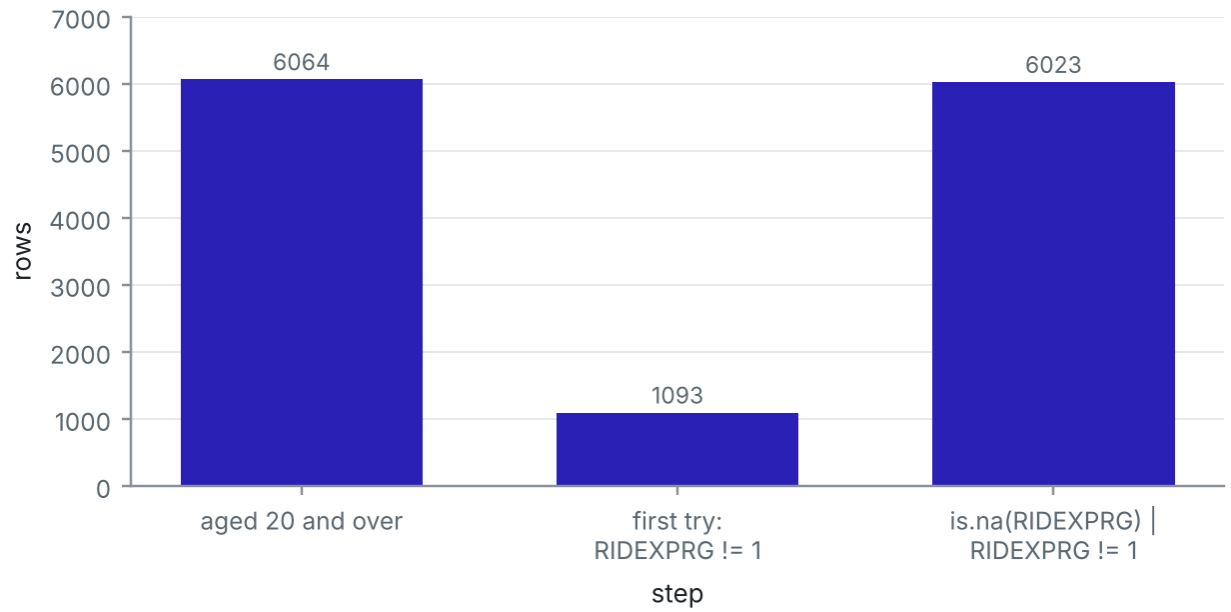
The analysis is reproducible in this book's sense when the outputs come back the same from the raw files with that one command. [354, 373, 325, 374, 375, 376, 377]

IN PLAIN TERMS. Eight steps, in the order real work runs them.

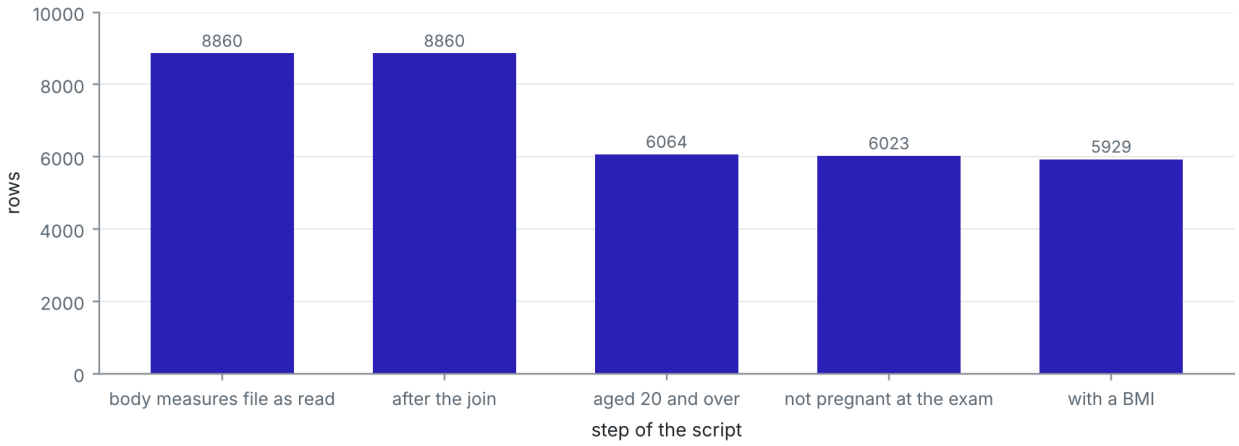
1. **The question.** One sentence, saying who, what is measured, and how it is summarised.
2. **The data.** Download the raw files. Read the terms that come with them. Record each file's checksum.
3. **The dictionary.** One row per variable you will use: its name in the file, its meaning, its unit, its codes, and where you found them.
4. **The project folder.** `data-raw/` for the files as they arrived, `code/` for the script, `output/` for what the script makes.
5. **The script.** Read, check, join, clean, work out new columns, summarise, plot, write the outputs, print the versions.
6. **The README.** What the project is, where the data came from, and the one command that reruns it.
7. **The rerun.** Delete the outputs and run the command from nothing. The same table and the same figure must come back.
8. **What it does not show.** Say it, next to the result.



The first quartile, median and third quartile of measured BMI in 2680 men and 3249 women aged 20 and over, not pregnant, examined in NHANES 2021–2023. The medians are close, 28.1 and 28.7 kg/m^2; the women's third quartile, 34.7 kg/m^2, is higher than the men's, 32.4 kg/m^2. Unweighted: these values describe the participants, not the United States.



Rows left by two ways of dropping the 41 adults pregnant at the exam from the 6064 aged 20 and over. The first try, filter(RIDEXPRG != 1), keeps 1093: the 1065 coded 2 and the 28 coded 3. It drops the 4930 rows where RIDEXPRG is missing, every man among them, with no warning. Keeping a row if RIDEXPRG is missing or not 1 leaves 6023, the 6064 minus 41 expected before the line was run.



Rows after each step of the end-to-end script: 8860 in the body measures file as read and after the join on SEQN, 6064 aged 20 and over, 6023 not pregnant at the exam, and 5929 with a BMI. The 41 pregnant adults and the 94 without a weight or a height are the two drops after the age filter. Each count is printed by the script, so a step that loses the wrong rows shows here first.

Illustration 1. Start with the question, written in one sentence before you touch the data:

> *Among adults aged 20 and over who were examined in NHANES between August 2021 and August 2023 and were not pregnant at the exam, how is measured body mass index (BMI) spread in men and in women?*

BMI is weight in kilograms divided by the square of height in metres, measured in kg/m^2. The question names who (adults examined in one survey cycle, not pregnant), what is measured (BMI from measured weight and height), and how it is summarised (its spread, by gender). It does not ask about the United States. That matters later.

Now the data. NHANES, the National Health and Nutrition Examination Survey, publishes each part of the survey as its own file. The measurements are in the body measures file, `BMX_L`. Age, gender and pregnancy are in the demographics file, `DEMO_L`. Both carry `SEQN`, the respondent sequence number, which is the key that joins them. You download both `.xpt` files from the survey's website and put them, unchanged, in `data-raw/`.

Before using them, read the terms. NHANES files come under the Data User Agreement of the National Center for Health Statistics, the American agency that runs the survey. Among other things, users agree to "Use the data in this dataset for statistical reporting and analysis only" and to "Make no attempt to learn the identity of any person or establishment included in these data". Neither the agreement nor the documentation states a copyright licence. Write the terms in the README.

Then record what arrived. Take each file's MD5 checksum, as in the section on reading raw files, and count its rows:

```
library(haven)
library(here)

unname(tools::md5sum(here("data-raw", c("BMX_L.xpt", "DEMO_L.xpt"))))

bmx <- read_xpt(here("data-raw", "BMX_L.xpt"))
demo <- read_xpt(here("data-raw", "DEMO_L.xpt"))
nrow(bmx)
nrow(demo)
```

```
here() starts at ~/project
[1] "ab25a36296e5b61881d42fef61c06b84"
[2] "14555b1f4e3c909172e900f63a2ecd03"
[1] 8860
[1] 11933
```

The body measures file has 8860 rows, one for each person examined. The demographics file has 11933, because it also holds people who were interviewed and never examined. The two checksums go in the README, so that anyone can confirm later that they have the same files.

Next, the dictionary. Write it from the survey's own documentation, the codebook pages for the two files, for the variables the question needs and no others:

Variable	Meaning, with unit	Codes	Source
SEQN	respondent sequence number, one per person	a number	both codebooks; checked unique in the file
RIAGENDR	gender	1 Male, 2 Female	DEMO_L codebook
RIDAGEYR	age in years at the screening interview	0 to 79; 80 means 80 and over	DEMO_L codebook
RIDEXPRG	pregnancy status at the exam	1 pregnant, 2 not pregnant, 3 cannot tell; missing for anyone not asked	DEMO_L codebook
BMXWT	weight, kg	measured value	BMX_L codebook
BMXHT	standing height, cm	measured value	BMX_L codebook
BMXBMI	BMI, kg/m^2, worked out from weight and height and rounded to one decimal	worked-out value	BMX_L codebook

Two entries will matter in the script. Ages of 80 and over are all written as 80, so no age in the file is above 80. And the pregnancy question was asked only of some women, so the column is missing for every man and for most women. Type the dictionary once, by hand, and save it as `data-raw/nhanes_dictionary.csv` beside the raw files.

The project folder is the one this book has used throughout: `data-raw/`, `code/` and `output/`. The README goes at the top. Write it now, with the outputs named, and treat its rerun line as a promise you are about to test:

> *BMI of adults in NHANES 2021–2023, by gender*
>
> *The spread of measured BMI in men and women aged 20 and over, not pregnant at the exam, examined in NHANES August 2021–August 2023. Every summary is unweighted and describes these participants only.*
>
> *Raw data:* `data-raw/BMX_L.xpt` *(MD5 ab25a36296e5b61881d42fef61c06b84) and* `data-raw/DEMO_L.xpt` *(MD5 14555b1f4e3c909172e900f63a2ecd03), downloaded unchanged from the NHANES website. Use is governed by the NCHS Data User Agreement: statistical reporting and analysis only, and no attempt to identify anyone.*
>
> *Data dictionary:* `data-raw/nhanes_dictionary.csv`.
>
> *To rerun: open a terminal in this folder and type* `Rscript code/analysis.R`. *The script reads* `data-raw/`*, never writes to it, and writes* `output/bmi_by_gender.csv` *(the table) and* `output/bmi_by_gender.png` *(the figure).*
>
> *Written with R 4.3.3; the versions of every package are printed at the end of each run. Contact: your name and email address.*

Figures quoted in this illustration are from [178].

WHERE THIS PICTURE BREAKS. These files came with complete documentation, a stated key and published terms. A thesis dataset exported from a form tool, or a register typed up from a clinic, often comes with none of the three. Then the dictionary is written from the questionnaire and from asking the people who collected the data, and the README says so.

The checksums prove that your files are the files you recorded. They cannot prove that those are the files the survey published, unless the survey also publishes checksums.

Illustration 2. Build the script a step at a time, checking the row count after each step. One step goes wrong.

Join the demographics onto the body measures by `SEQN`, as in the section on joining. A `left_join()` keeps every row of the left table, so the count must stay at 8860. `relationship = "one-to-one"` tells the join to stop with an error if a key turns up twice in either file:

```
library(dplyr, warn.conflicts = FALSE)

n_distinct(bmx$SEQN)
n_distinct(demo$SEQN)

nhanes <- bmx |>
  left_join(demo, by = "SEQN", relationship = "one-to-one")
nrow(nhanes)
```

```
[1] 8860
[1] 11933
[1] 8860
```

Each file has as many distinct `SEQN` values as rows, so the key is one per person in both. The join added no row and lost none. Now keep the adults:

```
adults <- nhanes |>
  filter(RIDAGEYR >= 20)
nrow(adults)

adults |>
  count(RIDEXPRG)
```

```
[1] 6064
# A tibble: 4 × 2
  RIDEXPRG     n
     <dbl> <int>
1        1    41
2        2  1065
3        3    28
4       NA  4930
```

6064 adults. Of them, 41 were pregnant at the exam (code 1). The question leaves them out, so you expect 6064 minus 41, which is 6023 rows, after the next step. Here is the first draft of that line:

```
first_try <- adults |>
  filter(RIDEXPRG != 1)
nrow(first_try)
```

```
[1] 1093
```

1093, not 6023. The line meant to drop 41 people dropped 4971. The reason is the 4930 rows where `RIDEXPRG` is missing: every man, and every woman who was not asked. For them `RIDEXPRG != 1` is neither true nor false. It is `NA`, and `filter()` keeps only the rows where the test is `TRUE`. The line ran without a warning and gave a plausible table of 1093 adults. All of them were women whose pregnancy status was recorded as code 2 or 3, and no man was left. Only the expected count caught it.

Say what you mean instead: keep a row if the value is missing, or if it is not 1.

```
adults <- adults |>
  filter(is.na(RIDEXPRG) | RIDEXPRG != 1)
nrow(adults)
sum(!is.na(adults$BMXBMI))
```

```
[1] 6023
[1] 5929
```

6023, as expected. Of these, 5929 have a BMI; the other 94 lack a weight or a height.

One more check before trusting the BMI column. The codebook says BMI was worked out as weight in kilograms divided by height in metres squared, then rounded to one decimal. Work it out again and find the largest gap between yours and the file's. `abs()` drops the sign of a difference.

```
bmi_check <- adults$BMXWT / (adults$BMXHT / 100)^2
max(abs(bmi_check - adults$BMXBMI), na.rm = TRUE)
```

```
[1] 0.04996478
```

The largest gap is under 0.05 kg/m^2, which is what rounding to one decimal allows. So the column is what the codebook says it is.

WHERE THIS PICTURE BREAKS. The expected count caught the pregnancy filter because `count()` gave the number of pregnant adults before the filter (41, the codebook's figure too), so you knew what to expect. A filter whose correct result you cannot predict has no such check. Count the rows it removes, and look at a few of them.

The BMI check compares the file with itself. It confirms the column was worked out as documented. It cannot catch a weight or height that was measured or recorded wrongly.

Illustration 3. Now put the steps that survived into the script, in order, with a comment for each. This is the whole of `code/analysis.R`:

```
# analysis.R
# BMI of adults aged 20 and over, by gender, in NHANES August
# 2021-August 2023 (files BMX_L and DEMO_L), leaving out anyone
# pregnant at the exam.
# Run from the project folder: Rscript code/analysis.R
# Reads data-raw/ and never writes to it; writes two files to
# output/. Every summary is unweighted: it describes these
# participants only.

library(haven)
library(dplyr, warn.conflicts = FALSE)
library(readr)
library(ggplot2)
library(here)

dir.create(here("output"), showWarnings = FALSE)

# 1. Read the two raw files
bmx <- read_xpt(here("data-raw", "BMX_L.xpt"))
demo <- read_xpt(here("data-raw", "DEMO_L.xpt"))
cat("Participants with body measures:", nrow(bmx), "\n")

# 2. Check the key: one row per participant in each file
stopifnot(
  "SEQN is unique in BMX_L" = n_distinct(bmx$SEQN) == nrow(bmx),
  "SEQN is unique in DEMO_L" = n_distinct(demo$SEQN) == nrow(demo)
)

# 3. Join demographics onto body measures; no row added or lost
nhanes <- bmx |>
  left_join(demo, by = "SEQN", relationship = "one-to-one")
stopifnot("the join kept every row" = nrow(nhanes) == nrow(bmx))

# 4. Keep adults not pregnant at the exam. RIDEXPRG is missing
#    for everyone not asked, so missing rows are kept on purpose.
adults <- nhanes |>
  filter(RIDAGEYR >= 20) |>
  filter(is.na(RIDEXPRG) | RIDEXPRG != 1) |>
  mutate(
    gender = factor(RIAGENDR, levels = c(1, 2),
                    labels = c("Male", "Female")),
    bmi_check = BMXWT / (BMXHT / 100)^2
  ) |>
  select(SEQN, gender, age = RIDAGEYR, weight_kg = BMXWT,
         height_cm = BMXHT, bmi = BMXBMI, bmi_check)
cat("Adults kept:", nrow(adults), "\n")
cat("Adults with a BMI:", sum(!is.na(adults$bmi)), "\n")

# 5. Check the values. Limits set before looking at the data;
#    a value outside them stops the script so that you look.
stopifnot(
  "measured adult heights are between 50 and 250 cm" =
    all(adults$height_cm >= 50 & adults$height_cm <= 250,
        na.rm = TRUE),
  "measured adult weights are between 20 and 300 kg" =
    all(adults$weight_kg >= 20 & adults$weight_kg <= 300,
        na.rm = TRUE),
  "BMI agrees with weight / height^2 to rounding" =
    all(abs(adults$bmi - adults$bmi_check) <= 0.05, na.rm = TRUE)
)
```

```
# 6. Summarise BMI by gender, with the n behind every number.
#    The mean is rounded for display, to BMI's one decimal.
bmi_by_gender <- adults |>
  group_by(gender) |>
  summarise(
    n_adults = n(),
    n_bmi = sum(!is.na(bmi)),
    mean_bmi = round(mean(bmi, na.rm = TRUE), 1),
    q1_bmi = quantile(bmi, probs = 0.25, na.rm = TRUE),
    median_bmi = median(bmi, na.rm = TRUE),
    q3_bmi = quantile(bmi, probs = 0.75, na.rm = TRUE)
  )
print(bmi_by_gender)
write_csv(bmi_by_gender, here("output", "bmi_by_gender.csv"))

# 7. Plot the spread of BMI by gender
bmi_plot <- adults |>
  filter(!is.na(bmi)) |>
  ggplot(aes(x = gender, y = bmi)) +
  geom_boxplot() +
  labs(
    x = NULL,
    y = "BMI (kg/m^2)",
    caption = "NHANES 2021-2023, aged 20+, not pregnant; unweighted"
  )
ggsave(here("output", "bmi_by_gender.png"), plot = bmi_plot,
       width = 5, height = 4, units = "in", dpi = 150)

# 8. Record the versions that made these outputs
sessionInfo()
```

Read it from the top and you can name the section of this book behind each step: the project folder and `here()`, reading a raw file, checks that stop the script, the join, choosing rows, new columns and factors, summaries by group, the plot, and the fresh run. `cat()` prints its arguments as one line of text, which gives the run a short log of counts, and `"\n"` ends the line. In step 4, `labels =` gives the names that replace the codes, in the order of `levels`. In step 7, the pipe hands the data to `ggplot()`; the layers after it still join with `+`. `NULL` leaves that axis without a title. `dpi` sets the dots per inch, the image's sharpness.

Open a terminal in the project folder and type the README's command:

```
Rscript code/analysis.R
```

```
here() starts at ~/project
Participants with body measures: 8860
Adults kept: 6023
Adults with a BMI: 5929
# A tibble: 2 × 7
  gender n_adults n_bmi mean_bmi q1_bmi median_bmi q3_bmi
  <fct>     <int> <int>    <dbl>  <dbl>      <dbl>  <dbl>
1 Male       2720  2680     29.3   24.9       28.1   32.4
2 Female     3303  3249     30.3   24.4       28.7   34.7
R version 4.3.3 (2024-02-29)
Platform: x86_64-pc-linux-gnu (64-bit)
Running under: Ubuntu 24.04.5 LTS

Matrix products: default
BLAS:   /usr/lib/x86_64-linux-gnu/blas/libblas.so.3.12.0
LAPACK: /usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.12.0

locale:
 [1] LC_CTYPE=C.utf8       LC_NUMERIC=C          LC_TIME=C.utf8
 [4] LC_COLLATE=C.utf8     LC_MONETARY=C.utf8    LC_MESSAGES=C.utf8
 [7] LC_PAPER=C.utf8       LC_NAME=C             LC_ADDRESS=C
[10] LC_TELEPHONE=C        LC_MEASUREMENT=C.utf8 LC_IDENTIFICATION=C

time zone: UTC
tzcode source: system (glibc)

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
[1] here_1.0.1    ggplot2_3.4.4 readr_2.1.5   dplyr_1.1.4   haven_2.5.4

loaded via a namespace (and not attached):
 [1] bit_4.0.5         gtable_0.3.4      compiler_4.3.3    crayon_1.5.2
 [5] tidyselect_1.2.0  parallel_4.3.3    systemfonts_1.0.5 scales_1.3.0
 [9] textshaping_0.3.7 R6_2.5.1          labeling_0.4.3    generics_0.1.3
[13] forcats_1.0.0     tibble_3.2.1      munsell_0.5.0     rprojroot_2.0.4
[17] pillar_1.9.0      tzdb_0.4.0        rlang_1.1.3       utf8_1.2.4
[21] bit64_4.0.5       cli_3.6.2         withr_2.5.0       magrittr_2.0.3
[25] grid_4.3.3        vroom_1.6.5       hms_1.1.3         lifecycle_1.0.4
[29] vctrs_0.6.5       glue_1.7.0        farver_2.1.1      ragg_1.2.7
[33] fansi_1.0.5       colorspace_2.1-0  tools_4.3.3       pkgconfig_2.0.3
```

Read the table as the dictionary and the question tell you to. There are 2680 men and 3249 women with a BMI, and 2680 + 3249 = 5929, the count in the log. Every check passed, or the script would have stopped at it. The versions printed below the table are the record of what produced it.

The three quartiles answer the question, as R's default (type 7) gives them: a quarter of the values lie below the first quartile, half below the median, three quarters below the third quartile.

The median BMI is 28.1 kg/m^2 in men and 28.7 kg/m^2 in women (unweighted: these participants only), close together. The spread is not. The middle half of the women runs from 24.4 to 34.7 kg/m^2, a width of 10.3 kg/m^2. The middle half of the men runs from 24.9 to 32.4 kg/m^2, a width of 7.5 kg/m^2. The women's upper quarter reaches higher. The means, 29.3 and 30.3 kg/m^2, sit above both medians, because a long tail of high values pulls a mean up and leaves a median where it is.

Figures quoted in this illustration are from [178].

WHERE THIS PICTURE BREAKS. The range limits in step 5 are the ones the section on checks used, and they are wide on purpose: they stop the script for a height of 17 cm or a weight of 700 kg, not for an unusual but real adult. They passed here partly because the survey's own team had already reviewed the values. Their documentation says values judged unrealistic "were deleted from the file". A file nobody has reviewed may stop at step 5, and that is the step doing its job.

The script handles these two files. If the survey releases a corrected file, its checksum will differ from the README's, and the counts in the log may change. Rerun, compare, and record the new checksum.

Illustration 4. The rerun is the test the whole book has been building to. Start from nothing: delete `output/`, run the command again, and compare the new table with the old one, byte for byte.

```
cp output/bmi_by_gender.csv first_run.csv
rm -r output
Rscript code/analysis.R > /dev/null 2>&1
ls output
cmp first_run.csv output/bmi_by_gender.csv && echo "identical"
cat output/bmi_by_gender.csv
```

```
bmi_by_gender.csv
bmi_by_gender.png
identical
gender,n_adults,n_bmi,mean_bmi,q1_bmi,median_bmi,q3_bmi
Male,2720,2680,29.3,24.9,28.1,32.4
Female,3303,3249,30.3,24.4,28.7,34.7
```

`cp` copies a file and `rm -r` deletes a folder. `> /dev/null 2>&1` throws away what the run prints. `cmp` compares two files and prints nothing when they are the same. `&&` runs the next command only if the last one succeeded, and `echo` prints its words.

Both outputs came back, and the table is identical to the one before. A colleague with the project folder runs the same command on their computer. If their table differs, the checksums and the version lines are where the two of you start looking. This is the gate to the next rung, the test you must pass before moving on to it: the analysis reproduces from the raw data with one command.

Now say what it does not show, in the README and next to any number you quote from it.

- It does not estimate anything about the United States. The survey's documentation says its sample weights "should be used to analyze the body measures data". This analysis uses none, so its summaries describe these 5929 participants and nobody else. Weights are taught in the subject on India's data infrastructure and complex surveys.
- It says nothing about India, or about adults anywhere else.
- BMI is worked out from weight and height. It is not a measurement of body fat.
- It is one examination of each person. It says nothing about change over time, or about what causes a higher or lower BMI.
- Its correctness rests on the checks. A mistake that passes every check reruns perfectly.

The next rung of this subject takes the same project further. A block of code repeated three times becomes a function you write, and you run it over many columns, files or groups at once. Git records every change to the script. A Quarto document replaces the step where you copy 28.1 into your text: the code writes the number into the document itself. DuckDB queries files too big to read into R at once. A form tool such as KoboToolbox, ODK or REDCap writes the raw file at the point where the data are collected.

WHERE THIS PICTURE BREAKS. The rerun on one computer shows the script is complete. It does not show that a computer with other package versions gets the same table; only a second person's run shows that, and pinning the versions is two rungs up.

The comparison here is of the CSV file. Compare a figure by looking at it: an image file can differ in its bytes when nothing you would see has changed.

MUST KNOW POINTS FOR YOU.

- Write the question before you open the data, and write who it is about.
- The error is to trust a filter because it ran without a warning. Before each filter, write down how many rows you expect it to keep, and check.
- Record each raw file's checksum in the README on the day it arrives, and the terms it came with.
- Put a printed row count after every step that can add or drop rows, and a `stopifnot()` where you know the right count.
- Any summary of an NHANES file without its sample weights describes the participants in the file. Write "unweighted" beside it, and never present it as the figure for the United States, still less for India.
- The gate is one command from raw data to the same outputs. Ask for it before you read a single result.
- Reproducible means the result can be checked, not that it is right. A wrong cut-point or a wrong filter reruns perfectly.

COMMON MISREADING. "NHANES shows the median BMI of American women is 28.7 kg/m^2." The table does not say that. It gives the median of 3249 women examined in one survey cycle. It uses none of the survey's sample weights, and the survey's documentation says those weights should be used. So 28.7 kg/m^2 describes these participants. Read it as "in this sample", never as "in America".

EXERCISE 1 (retrieval). Without looking back, list the steps of an end-to-end analysis in the order real work runs them. Then say what the gate to the next rung is.

EXERCISE 2 (build). Build the rung's target with a dataset of your own. It may be a thesis dataset, or a department register you have permission to use. It may be the NHANES files of this section, with a question of your own.

Write the question in one sentence. Put the raw files in `data-raw/` and record their checksums and terms. Write the data dictionary.

Write `code/analysis.R` so that it reads, checks, cleans, summarises and plots, writes one table and one figure to `output/`, and ends with `sessionInfo()`. Write the README. Then delete `output/`, run `Rscript code/analysis.R` in a fresh session, and give the project folder to a colleague to run the same command.

EXERCISE 3 (critique). A colleague sends you the project for her thesis chapter. The README says: "Run the scripts in code/ in order." `code/` holds `clean.R` and `analysis.R`. The first line of `clean.R` is `data <- read_excel("C:/Users/anita/Desktop/final data (2).xlsx")`. Its last line is `write_xlsx(data, "C:/Users/anita/Desktop/final data (2).xlsx")`. Her chapter reports "mean BMI 24.6 kg/m^2 (n = 412)". Find each thing that stops this project from passing the gate, and say what each one would cost her.

EXERCISE 4 (teaching). A health secretary asks why the department's analysts should hand over every analysis as a project folder that reruns with one command. Answer on one page. Put the answer first, then the reasons, with no methods section.

References

1. Ziemann, Mark and Eren, Yotam and El-Osta, Assam. Gene name errors are widespread in the scientific literature. PMID 27552985, PMC4994289. Open access under CC BY 4.0. Quoted from the PMC open-access text. Abstract: Excel with default settings. https://pmc.ncbi.nlm.nih.gov/articles/PMC4994289/ (accessed 2026-10-02).

2. Ziemann, Mark and Eren, Yotam and El-Osta, Assam. Gene name errors are widespread in the scientific literature. PMID 27552985, PMC4994289. Open access under CC BY 4.0. Quoted from the PMC open-access text. Background, first paragraph: the SEPT2 and MARCH1 examples. https://pmc.ncbi.nlm.nih.gov/articles/PMC4994289/ (accessed 2026-10-02).

3. Ziemann, Mark and Eren, Yotam and El-Osta, Assam. Gene name errors are widespread in the scientific literature. PMID 27552985, PMC4994289. Open access under CC BY 4.0. Quoted from the PMC open-access text. Methods: 18 journals, 2005 to 2015. https://pmc.ncbi.nlm.nih.gov/articles/PMC4994289/ (accessed 2026-10-02).

4. Ziemann, Mark and Eren, Yotam and El-Osta, Assam. Gene name errors are widespread in the scientific literature. PMID 27552985, PMC4994289. Open access under CC BY 4.0. Quoted from the PMC open-access text. Results, first paragraph: files screened and papers with gene lists. https://pmc.ncbi.nlm.nih.gov/articles/PMC4994289/ (accessed 2026-10-02).

5. Ziemann, Mark and Eren, Yotam and El-Osta, Assam. Gene name errors are widespread in the scientific literature. PMID 27552985, PMC4994289. Open access under CC BY 4.0. Quoted from the PMC open-access text. Results, first paragraph: confirmed errors. https://pmc.ncbi.nlm.nih.gov/articles/PMC4994289/ (accessed 2026-10-02).

6. Ziemann, Mark and Eren, Yotam and El-Osta, Assam. Gene name errors are widespread in the scientific literature. PMID 27552985, PMC4994289. Open access under CC BY 4.0. Quoted from the PMC open-access text. Results, first paragraph: the proportion affected. https://pmc.ncbi.nlm.nih.gov/articles/PMC4994289/ (accessed 2026-10-02).

7. Ziemann, Mark and Eren, Yotam and El-Osta, Assam. Gene name errors are widespread in the scientific literature. PMID 27552985, PMC4994289. Open access under CC BY 4.0. Quoted from the PMC open-access text. Discussion, first paragraph: no permanent switch. https://pmc.ncbi.nlm.nih.gov/articles/PMC4994289/ (accessed 2026-10-02).

8. Ziemann, Mark and Eren, Yotam and El-Osta, Assam. Gene name errors are widespread in the scientific literature. PMID 27552985, PMC4994289. Open access under CC BY 4.0. Quoted from the PMC open-access text. Discussion, second paragraph: files whose original names cannot be recovered. https://pmc.ncbi.nlm.nih.gov/articles/PMC4994289/ (accessed 2026-10-02).

9. Bruford, Elspeth A. and Braschi, Bryony and Denny, Paul and Jones, Tamsin E. M. and Seal, Ruth L. and Tweedie, Susan. Guidelines for Human Gene Nomenclature. 2020. p. 757, Box 3 “Scenarios that may merit a symbol change”: symbols that affect data handling. https://doi.org/10.1038/s41588-020-0669-3.

10. Bruford, Elspeth A. and Braschi, Bryony and Denny, Paul and Jones, Tamsin E. M. and Seal, Ruth L. and Tweedie, Susan. Guidelines for Human Gene Nomenclature. 2020. p. 754, introduction: the committee’s name. https://doi.org/10.1038/s41588-020-0669-3.

11. Herndon, Thomas and Ash, Michael and Pollin, Robert. Does High Public Debt Consistently Stifle Economic Growth? A Critique of Reinhart and Rogoff. 2014. p. 259, introduction: the three problems. https://doi.org/10.1093/cje/bet075.

12. Herndon, Thomas and Ash, Michael and Pollin, Robert. Does High Public Debt Consistently Stifle Economic Growth? A Critique of Reinhart and Rogoff. 2014. p. 261, section 3 Replication: the posted spreadsheets could not reproduce the results. https://doi.org/10.1093/cje/bet075.

13. Herndon, Thomas and Ash, Michael and Pollin, Robert. Does High Public Debt Consistently Stifle Economic Growth? A Critique of Reinhart and Rogoff. 2014. p. 263, section 3.2 Spreadsheet coding error. https://doi.org/10.1093/cje/bet075.

14. Herndon, Thomas and Ash, Michael and Pollin, Robert. Does High Public Debt Consistently Stifle Economic Growth? A Critique of Reinhart and Rogoff. 2014. p. 263, footnote 9: the rows the formula averaged. https://doi.org/10.1093/cje/bet075.

15. Herndon, Thomas and Ash, Michael and Pollin, Robert. Does High Public Debt Consistently Stifle Economic Growth? A Critique of Reinhart and Rogoff. 2014. p. 262, section 3.1: the exclusions not declared. https://doi.org/10.1093/cje/bet075.

16. Herndon, Thomas and Ash, Michael and Pollin, Robert. Does High Public Debt Consistently Stifle Economic Growth? A Critique of Reinhart and Rogoff. 2014. p. 264, section 3.3 and Table 2: country-years above 90%, correct total. https://doi.org/10.1093/cje/bet075.

17. Herndon, Thomas and Ash, Michael and Pollin, Robert. Does High Public Debt Consistently Stifle Economic Growth? A Critique of Reinhart and Rogoff. 2014. p. 264, section 3.3: after the chosen exclusions. https://doi.org/10.1093/cje/bet075.

18. Herndon, Thomas and Ash, Michael and Pollin, Robert. Does High Public Debt Consistently Stifle Economic Growth? A Critique of Reinhart and Rogoff. 2014. p. 264, section 3.3: after the exclusions and the spreadsheet errors. https://doi.org/10.1093/cje/bet075.

19. Herndon, Thomas and Ash, Michael and Pollin, Robert. Does High Public Debt Consistently Stifle Economic Growth? A Critique of Reinhart and Rogoff. 2014. p. 257, abstract: 2.2% against the published −0.1%. https://doi.org/10.1093/cje/bet075.

20. Herndon, Thomas and Ash, Michael and Pollin, Robert. Does High Public Debt Consistently Stifle Economic Growth? A Critique of Reinhart and Rogoff. 2014. p. 268, section 3.5: interactive effects. https://doi.org/10.1093/cje/bet075.

21. Herndon, Thomas and Ash, Michael and Pollin, Robert. Does High Public Debt Consistently Stifle Economic Growth? A Critique of Reinhart and Rogoff. 2014. p. 265, section 3.4: one number per country, whatever its years. https://doi.org/10.1093/cje/bet075.

22. Herndon, Thomas and Ash, Michael and Pollin, Robert. Does High Public Debt Consistently Stifle Economic Growth? A Critique of Reinhart and Rogoff. 2014. p. 269, Table 5, last two “Interactive effects” rows, >90% column: 0.0 with three problems, −0.1 with the transcription error added. https://doi.org/10.1093/cje/bet075.

23. Herndon, Thomas and Ash, Michael and Pollin, Robert. Does High Public Debt Consistently Stifle Economic Growth? A Critique of Reinhart and Rogoff. 2014. p. 269, Table 5, row “Spreadsheet error only”, last column (>90%). https://doi.org/10.1093/cje/bet075.

24. Herndon, Thomas and Ash, Michael and Pollin, Robert. Does High Public Debt Consistently Stifle Economic Growth? A Critique of Reinhart and Rogoff. 2014. p. 277, section 4 Conclusion: the errors acknowledged. https://doi.org/10.1093/cje/bet075.

25. Herndon, Thomas and Ash, Michael and Pollin, Robert. Does High Public Debt Consistently Stifle Economic Growth? A Critique of Reinhart and Rogoff. 2013. Working paper, p. 14, footnote 9: reproducible code against working spreadsheets. https://peri.umass.edu/publication/does-high-public-debt-consistently-stifle-economic-growth-a-critique-of-reinhart-and-rogoff/ (accessed 2026-10-02).

26. Public Health England. PHE statement on delayed reporting of COVID-19 cases. 2020. Statement of Michael Brodie, first paragraph: 15,841 cases. GOV.UK news story. https://www.gov.uk/government/news/phe-statement-on-delayed-reporting-of-covid-19-cases (accessed 2026-10-02).

27. Public Health England. PHE statement on delayed reporting of COVID-19 cases. 2020. Statement of Michael Brodie, first paragraph: when and where it was found. GOV.UK news story. https://www.gov.uk/government/news/phe-statement-on-delayed-reporting-of-covid-19-cases (accessed 2026-10-02).

28. Public Health England. PHE statement on delayed reporting of COVID-19 cases. 2020. Background information, fourth paragraph: the cause. GOV.UK news story. https://www.gov.uk/government/news/phe-statement-on-delayed-reporting-of-covid-19-cases (accessed 2026-10-02).

29. Public Health England. PHE statement on delayed reporting of COVID-19 cases. 2020. Background information, fourth paragraph: the fix. GOV.UK news story. https://www.gov.uk/government/news/phe-statement-on-delayed-reporting-of-covid-19-cases (accessed 2026-10-02).

30. Public Health England. PHE statement on delayed reporting of COVID-19 cases. 2020. Background information, fourth paragraph: checks already in place. GOV.UK news story. https://www.gov.uk/government/news/phe-statement-on-delayed-reporting-of-covid-19-cases (accessed 2026-10-02).

31. Herndon, Thomas and Ash, Michael and Pollin, Robert. Does High Public Debt Consistently Stifle Economic Growth? A Critique of Reinhart and Rogoff. 2014. p. 263, section 3.2: the omitted countries alphabetical. https://doi.org/10.1093/cje/bet075.

32. Herndon, Thomas and Ash, Michael and Pollin, Robert. Does High Public Debt Consistently Stifle Economic Growth? A Critique of Reinhart and Rogoff. 2014. p. 259, introduction: the 20 advanced economies. https://doi.org/10.1093/cje/bet075.

33. Public Health England. PHE statement on delayed reporting of COVID-19 cases. 2020. Statement of Michael Brodie, second paragraph: test results not affected. GOV.UK news story. https://www.gov.uk/government/news/phe-statement-on-delayed-reporting-of-covid-19-cases (accessed 2026-10-02).

34. Public Health England. PHE statement on delayed reporting of COVID-19 cases. 2020. Statement of Susan Hopkins, fourth paragraph: cases passed to contact tracing. GOV.UK news story. https://www.gov.uk/government/news/phe-statement-on-delayed-reporting-of-covid-19-cases (accessed 2026-10-02).

35. Public Health England. PHE statement on delayed reporting of COVID-19 cases. 2020. Background information, last paragraph before the table: the recorded dates. GOV.UK news story. https://www.gov.uk/government/news/phe-statement-on-delayed-reporting-of-covid-19-cases (accessed 2026-10-02).

36. Wilson, Greg and Bryan, Jennifer and Cranston, Karen and Kitzes, Justin and Nederbragt, Lex and Teal, Tracy K. Good enough practices in scientific computing. 2017. Data management, practice 1a “Save the raw data”. https://pmc.ncbi.nlm.nih.gov/articles/PMC5480810/ (accessed 2026-10-02).

37. Sandve, Geir Kjetil and Nekrutenko, Anton and Taylor, James and Hovig, Eivind. Ten simple rules for reproducible computational research. 2013. Rule 2: Avoid Manual Data Manipulation Steps. https://pmc.ncbi.nlm.nih.gov/articles/PMC3812051/ (accessed 2026-10-02).

38. Broman, Karl W and Woo, Kara H. Data organization in spreadsheets. 2018. p. 2, section 1 Introduction: analysis in a separate program. https://doi.org/10.1080/00031305.2017.1375989 (accessed 2026-10-02).

39. Ziemann, Mark and Eren, Yotam and El-Osta, Assam. Gene name errors are widespread in the scientific literature. PMID 27552985, PMC4994289. Open access under CC BY 4.0. Quoted from the PMC open-access text. figures quoted in the text. https://pmc.ncbi.nlm.nih.gov/articles/PMC4994289/ (accessed 2026-10-02).

40. Herndon, Thomas and Ash, Michael and Pollin, Robert. Does High Public Debt Consistently Stifle Economic Growth? A Critique of Reinhart and Rogoff. 2014. figures quoted in the text. https://doi.org/10.1093/cje/bet075.

41. Public Health England. PHE statement on delayed reporting of COVID-19 cases. 2020. figures quoted in the text. GOV.UK news story. https://www.gov.uk/government/news/phe-statement-on-delayed-reporting-of-covid-19-cases (accessed 2026-10-02).

42. The Carpentries. R for Reproducible Scientific Analysis. 2026. Summary and Setup, Prerequisites: install R, then RStudio. Software Carpentry lesson (r-novice-gapminder). https://swcarpentry.github.io/r-novice-gapminder/ (accessed 2026-10-02).

43. The Carpentries. R for Reproducible Scientific Analysis. 2026. Summary and Setup, Prerequisites: what RStudio is. Software Carpentry lesson (r-novice-gapminder). https://swcarpentry.github.io/r-novice-gapminder/ (accessed 2026-10-02).

44. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. Introduction, Prerequisites, “R”: where to download R. https://r4ds.hadley.nz/ (accessed 2026-10-02).

45. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. Introduction, Prerequisites, “RStudio”: where to download RStudio. https://r4ds.hadley.nz/ (accessed 2026-10-02).

46. The Carpentries. R for Reproducible Scientific Analysis. 2026. Episode 1, “Workflow within RStudio”, way 2: lines pushed to the console. Software Carpentry lesson (r-novice-gapminder). https://swcarpentry.github.io/r-novice-gapminder/ (accessed 2026-10-02).

47. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. Introduction, Prerequisites, “RStudio”. https://r4ds.hadley.nz/ (accessed 2026-10-02).

48. Venables, W. N. and Smith, D. M. and R Core Team. An Introduction to R: Notes on R, a Programming Environment for Data Analysis and Graphics. 2026. Section 1.8 “R commands, case sensitivity, etc.”: an expression is printed and lost. https://cran.r-project.org/doc/manuals/r-release/R-intro.html (accessed 2026-10-02).

49. The Carpentries. R for Reproducible Scientific Analysis. 2026. Episode 1, “Using R as a calculator”: the + prompt. Software Carpentry lesson (r-novice-gapminder). https://swcarpentry.github.io/r-novice-gapminder/ (accessed 2026-10-02).

50. The Carpentries. R for Reproducible Scientific Analysis. 2026. Episode 1, “R scripts”. Software Carpentry lesson (r-novice-gapminder). https://swcarpentry.github.io/r-novice-gapminder/ (accessed 2026-10-02).

51. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. Section 6.2.1 “What is the source of truth?”. https://r4ds.hadley.nz/ (accessed 2026-10-02).

52. The Carpentries. R for Reproducible Scientific Analysis. 2026. Episode 1, “Using R as a calculator”: order of operations. Software Carpentry lesson (r-novice-gapminder). https://swcarpentry.github.io/r-novice-gapminder/ (accessed 2026-10-02).

53. Venables, W. N. and Smith, D. M. and R Core Team. An Introduction to R: Notes on R, a Programming Environment for Data Analysis and Graphics. 2026. Section 2.2 “Vector arithmetic”: the arithmetic operators, ending with the sign for a power. https://cran.r-project.org/doc/manuals/r-release/R-intro.html (accessed 2026-10-02).

54. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. Section 2.2 “Comments”. https://r4ds.hadley.nz/ (accessed 2026-10-02).

55. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. Section 2.2 “Comments”: comments for why. https://r4ds.hadley.nz/ (accessed 2026-10-02).

56. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. Section 6.1.1 “Running code”: Ctrl + Enter. https://r4ds.hadley.nz/ (accessed 2026-10-02).

57. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. Section 6.1 “Scripts”: opening a new script. https://r4ds.hadley.nz/ (accessed 2026-10-02).

58. The Carpentries. R for Reproducible Scientific Analysis. 2026. Episode 1, tip “Running segments of your code”: running a block. Software Carpentry lesson (r-novice-gapminder). https://swcarpentry.github.io/r-novice-gapminder/ (accessed 2026-10-02).

59. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. Section 6.1.3 “Saving and naming”: informative names. https://r4ds.hadley.nz/ (accessed 2026-10-02).

60. The tidyverse team. The tidyverse style guide. Chapter 1 “Files”, 1.1 Names, rule 1. Online book. https://style.tidyverse.org/ (accessed 2026-10-02).

61. Venables, W. N. and Smith, D. M. and R Core Team. An Introduction to R: Notes on R, a Programming Environment for Data Analysis and Graphics. 2026. Section 1.8 “R commands, case sensitivity, etc.”: an assignment is not printed. https://cran.r-project.org/doc/manuals/r-release/R-intro.html (accessed 2026-10-02).

62. Venables, W. N. and Smith, D. M. and R Core Team. An Introduction to R: Notes on R, a Programming Environment for Data Analysis and Graphics. 2026. Section 1.10, objects: what R creates and manipulates. https://cran.r-project.org/doc/manuals/r-release/R-intro.html (accessed 2026-10-02).

63. The Carpentries. R for Reproducible Scientific Analysis. 2026. Episode 1, “Variables and assignment”: the right-hand side first. Software Carpentry lesson (r-novice-gapminder). https://swcarpentry.github.io/r-novice-gapminder/ (accessed 2026-10-02).

64. The Carpentries. R for Reproducible Scientific Analysis. 2026. Episode 1, “Variables and assignment”: reassignment. Software Carpentry lesson (r-novice-gapminder). https://swcarpentry.github.io/r-novice-gapminder/ (accessed 2026-10-02).

65. Venables, W. N. and Smith, D. M. and R Core Team. An Introduction to R: Notes on R, a Programming Environment for Data Analysis and Graphics. 2026. Section 1.8: case sensitivity. https://cran.r-project.org/doc/manuals/r-release/R-intro.html (accessed 2026-10-02).

66. The Carpentries. R for Reproducible Scientific Analysis. 2026. Episode 1, “Variables and assignment”: which names are allowed. Software Carpentry lesson (r-novice-gapminder). https://swcarpentry.github.io/r-novice-gapminder/ (accessed 2026-10-02).

67. Venables, W. N. and Smith, D. M. and R Core Team. An Introduction to R: Notes on R, a Programming Environment for Data Analysis and Graphics. 2026. Section 1.8: names starting with a dot. https://cran.r-project.org/doc/manuals/r-release/R-intro.html (accessed 2026-10-02).

68. Venables, W. N. and Smith, D. M. and R Core Team. An Introduction to R: Notes on R, a Programming Environment for Data Analysis and Graphics. 2026. Section 1.10: the workspace. https://cran.r-project.org/doc/manuals/r-release/R-intro.html (accessed 2026-10-02).

69. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. Section 2.4 “Calling functions”, last sentence: the environment tab. https://r4ds.hadley.nz/ (accessed 2026-10-02).

70. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. Section 6.2.1 “What is the source of truth?”: the environment against the script. https://r4ds.hadley.nz/ (accessed 2026-10-02).

71. The tidyverse team. The tidyverse style guide. 2.1 Object names. Online book. https://style.tidyverse.org/ (accessed 2026-10-02).

72. The tidyverse team. The tidyverse style guide. 2.8 Assignment. Online book. https://style.tidyverse.org/ (accessed 2026-10-02).

73. Venables, W. N. and Smith, D. M. and R Core Team. An Introduction to R: Notes on R, a Programming Environment for Data Analysis and Graphics. 2026. Section 2.1 “Vectors and assignment”: = as an alternative. https://cran.r-project.org/doc/manuals/r-release/R-intro.html (accessed 2026-10-02).

74. Venables, W. N. and Smith, D. M. and R Core Team. An Introduction to R: Notes on R, a Programming Environment for Data Analysis and Graphics. 2026. Section 2.1 “Vectors and assignment”: = as an alternative, continued. https://cran.r-project.org/doc/manuals/r-release/R-intro.html (accessed 2026-10-02).

75. The tidyverse team. The tidyverse style guide. 2.2.4 Infix operators: spaces round <-. Online book. https://style.tidyverse.org/ (accessed 2026-10-02).

76. Venables, W. N. and Smith, D. M. and R Core Team. An Introduction to R: Notes on R, a Programming Environment for Data Analysis and Graphics. 2026. [2.1 Vectors and assignment], opening paragraph: the numeric vector. https://cran.r-project.org/doc/manuals/r-release/R-intro.html (accessed 2026-10-02).

77. Venables, W. N. and Smith, D. M. and R Core Team. An Introduction to R: Notes on R, a Programming Environment for Data Analysis and Graphics. 2026. [3.1 Intrinsic attributes: mode and length], second paragraph: one mode per vector. https://cran.r-project.org/doc/manuals/r-release/R-intro.html (accessed 2026-10-02).

78. R Core Team. R 4.3.3, package base: help pages. 2024. Help page “c” (base 4.3.3), Description and Details: the common type and its hierarchy. R Foundation for Statistical Computing. https://www.R-project.org/ (accessed 2026-10-02).

79. The Carpentries. R for Reproducible Scientific Analysis. 2026. Ep. 04 Data Structures, “Coercion by combining vectors”. Software Carpentry lesson (r-novice-gapminder). https://swcarpentry.github.io/r-novice-gapminder/ (accessed 2026-10-02).

80. R Core Team. R 4.3.3, package base: help pages. 2024. Help page “mean” (base 4.3.3), Value: a vector that is not numeric or logical. R Foundation for Statistical Computing. https://www.R-project.org/ (accessed 2026-10-02).

81. Venables, W. N. and Smith, D. M. and R Core Team. An Introduction to R: Notes on R, a Programming Environment for Data Analysis and Graphics. 2026. [2.2 Vector arithmetic], first paragraph: element by element, and recycling. https://cran.r-project.org/doc/manuals/r-release/R-intro.html (accessed 2026-10-02).

82. Venables, W. N. and Smith, D. M. and R Core Team. An Introduction to R: Notes on R, a Programming Environment for Data Analysis and Graphics. 2026. [2.2 Vector arithmetic], first paragraph: a constant is repeated. https://cran.r-project.org/doc/manuals/r-release/R-intro.html (accessed 2026-10-02).

83. Venables, W. N. and Smith, D. M. and R Core Team. An Introduction to R: Notes on R, a Programming Environment for Data Analysis and Graphics. 2026. [2.4 Logical vectors]: comparisons make logical vectors. https://cran.r-project.org/doc/manuals/r-release/R-intro.html (accessed 2026-10-02).

84. Venables, W. N. and Smith, D. M. and R Core Team. An Introduction to R: Notes on R, a Programming Environment for Data Analysis and Graphics. 2026. [2.5 Missing values], first paragraph: why an operation on NA gives NA. https://cran.r-project.org/doc/manuals/r-release/R-intro.html (accessed 2026-10-02).

85. R Core Team. R 4.3.3, package base: help pages. 2024. Help page “NA” (base 4.3.3), Details: logical computations with NA. R Foundation for Statistical Computing. https://www.R-project.org/ (accessed 2026-10-02).

86. Venables, W. N. and Smith, D. M. and R Core Team. An Introduction to R: Notes on R, a Programming Environment for Data Analysis and Graphics. 2026. [2.5 Missing values], third paragraph: x == NA against is.na(x). https://cran.r-project.org/doc/manuals/r-release/R-intro.html (accessed 2026-10-02).

87. Venables, W. N. and Smith, D. M. and R Core Team. An Introduction to R: Notes on R, a Programming Environment for Data Analysis and Graphics. 2026. [2.7 Index vectors], opening paragraph: square brackets. https://cran.r-project.org/doc/manuals/r-release/R-intro.html (accessed 2026-10-02).

88. R Core Team. R 4.3.3, package base: help pages. 2024. Help page “Round” (base 4.3.3), Details: the represented number and the printed number. R Foundation for Statistical Computing. https://www.R-project.org/ (accessed 2026-10-02).

89. R Core Team. R 4.3.3, package base: help pages. 2024. Help page “Round” (base 4.3.3), Warning: decimal fractions in binary. R Foundation for Statistical Computing. https://www.R-project.org/ (accessed 2026-10-02).

90. Horst, Allison Marie and Hill, Alison Presmanes and Gorman, Kristen B. palmerpenguins: Palmer Archipelago (Antarctica) Penguin Data. 2020. figures quoted in the text. CRAN. https://allisonhorst.github.io/palmerpenguins/ (accessed 2026-10-02).

91. The Carpentries. R for Reproducible Scientific Analysis. 2026. Ep. 01, “Mathematical functions”: arguments are what goes inside the brackets. Software Carpentry lesson (r-novice-gapminder). https://swcarpentry.github.io/r-novice-gapminder/ (accessed 2026-10-02).

92. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. Ch. 2 Workflow: basics, 2.4 Calling functions: arguments by name and by position. https://r4ds.hadley.nz/ (accessed 2026-10-02).

93. The Carpentries. R for Reproducible Scientific Analysis. 2026. Ep. 03 Seeking Help, “Reading Help Files”: the sections of a help page, Usage. Software Carpentry lesson (r-novice-gapminder). https://swcarpentry.github.io/r-novice-gapminder/ (accessed 2026-10-02).

94. The Carpentries. R for Reproducible Scientific Analysis. 2026. Ep. 03 Seeking Help, “Reading Help Files”: the sections of a help page, Value. Software Carpentry lesson (r-novice-gapminder). https://swcarpentry.github.io/r-novice-gapminder/ (accessed 2026-10-02).

95. R Core Team. R 4.3.3, package base: help pages. 2024. Help page “mean” (base 4.3.3), Arguments: na.rm. R Foundation for Statistical Computing. https://www.R-project.org/ (accessed 2026-10-02).

96. R Core Team. R 4.3.3, package base: help pages. 2024. Help page “mean” (base 4.3.3), Arguments: trim, values outside 0 to 0.5. R Foundation for Statistical Computing. https://www.R-project.org/ (accessed 2026-10-02).

97. R Core Team. R 4.3.3, package base: help pages. 2024. Help page “mean” (base 4.3.3), Arguments: trim, what it does. R Foundation for Statistical Computing. https://www.R-project.org/ (accessed 2026-10-02).

98. R Core Team. R 4.3.3, package stats: help pages. 2024. Help page “median” (stats 4.3.3), Arguments: na.rm. R Foundation for Statistical Computing. https://www.R-project.org/ (accessed 2026-10-02).

99. R Core Team. R 4.3.3, package stats: help pages. 2024. Help page “sd” (stats 4.3.3), Usage and Arguments. R Foundation for Statistical Computing. https://www.R-project.org/ (accessed 2026-10-02).

100. R Core Team. R 4.3.3, package base: help pages. 2024. Help page “Round” (base 4.3.3), Description: round and its default. R Foundation for Statistical Computing. https://www.R-project.org/ (accessed 2026-10-02).

101. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. Ch. 3 Data transformation, 3.1.3 dplyr basics: what the pipe does. https://r4ds.hadley.nz/ (accessed 2026-10-02).

102. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. Ch. 3, 3.4 The pipe: the version of R that added it. https://r4ds.hadley.nz/ (accessed 2026-10-02).

103. The tidyverse team. The tidyverse style guide. Chapter 4 Pipes: layout. Online book. https://style.tidyverse.org/ (accessed 2026-10-02).

104. Venables, W. N. and Smith, D. M. and R Core Team. An Introduction to R: Notes on R, a Programming Environment for Data Analysis and Graphics. 2026. [1.7 Getting help with functions and features]: running a help page’s examples. https://cran.r-project.org/doc/manuals/r-release/R-intro.html (accessed 2026-10-02).

105. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. Introduction, Prerequisites, “The tidyverse”: what a package is. https://r4ds.hadley.nz/ (accessed 2026-10-02).

106. Venables, W. N. and Smith, D. M. and R Core Team. An Introduction to R: Notes on R, a Programming Environment for Data Analysis and Graphics. 2026. [13.1 Standard packages]: the base packages ship with R. https://cran.r-project.org/doc/manuals/r-release/R-intro.html (accessed 2026-10-02).

107. Venables, W. N. and Smith, D. M. and R Core Team. An Introduction to R: Notes on R, a Programming Environment for Data Analysis and Graphics. 2026. [13.2 Contributed packages and CRAN]: where packages come from. https://cran.r-project.org/doc/manuals/r-release/R-intro.html (accessed 2026-10-02).

108. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. Introduction, Prerequisites, “R”: what CRAN stands for. https://r4ds.hadley.nz/ (accessed 2026-10-02).

109. R Core Team. R 4.3.3, package utils: help pages. 2024. Help page “install.packages” (utils 4.3.3), Description. R Foundation for Statistical Computing. https://www.R-project.org/ (accessed 2026-10-02).

110. R Core Team. R 4.3.3, package utils: help pages. 2024. Help page “install.packages” (utils 4.3.3), Arguments: pkgs, the current version. R Foundation for Statistical Computing. https://www.R-project.org/ (accessed 2026-10-02).

111. Venables, W. N. and Smith, D. M. and R Core Team. An Introduction to R: Notes on R, a Programming Environment for Data Analysis and Graphics. 2026. [13 Packages], first paragraph: a package must be loaded. https://cran.r-project.org/doc/manuals/r-release/R-intro.html (accessed 2026-10-02).

112. R Core Team. R 4.3.3, package base: help pages. 2024. Help page “library” (base 4.3.3), Description. R Foundation for Statistical Computing. https://www.R-project.org/ (accessed 2026-10-02).

113. R Core Team. R 4.3.3, package base: help pages. 2024. Help page “library” (base 4.3.3), Arguments: package, with or without quotation marks. R Foundation for Statistical Computing. https://www.R-project.org/ (accessed 2026-10-02).

114. Venables, W. N. and Smith, D. M. and R Core Team. An Introduction to R: Notes on R, a Programming Environment for Data Analysis and Graphics. 2026. [1.10 Data permanency and removing objects]: a session has an end. https://cran.r-project.org/doc/manuals/r-release/R-intro.html (accessed 2026-10-02).

115. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. Ch. 6 Workflow: scripts and projects, 6.2.1: the menu route to a fresh session. https://r4ds.hadley.nz/ (accessed 2026-10-02).

116. R Core Team. R 4.3.3, package base: help pages. 2024. Help page “library” (base 4.3.3), Arguments: warn.conflicts, what a conflict is. R Foundation for Statistical Computing. https://www.R-project.org/ (accessed 2026-10-02).

117. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. Introduction, Prerequisites, “The tidyverse”: the conflicts message on loading. https://r4ds.hadley.nz/ (accessed 2026-10-02).

118. Venables, W. N. and Smith, D. M. and R Core Team. An Introduction to R: Notes on R, a Programming Environment for Data Analysis and Graphics. 2026. [13.3 Namespaces]: the double-colon operator. https://cran.r-project.org/doc/manuals/r-release/R-intro.html (accessed 2026-10-02).

119. Venables, W. N. and Smith, D. M. and R Core Team. An Introduction to R: Notes on R, a Programming Environment for Data Analysis and Graphics. 2026. [13.3 Namespaces], last paragraph: :: loads the package. https://cran.r-project.org/doc/manuals/r-release/R-intro.html (accessed 2026-10-02).

120. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. Introduction, Prerequisites, “The tidyverse”: the nine core packages. https://r4ds.hadley.nz/ (accessed 2026-10-02).

121. R Core Team. R 4.3.3, package utils: help pages. 2024. Help page “packageDescription” (utils 4.3.3), Value: packageVersion. R Foundation for Statistical Computing. https://www.R-project.org/ (accessed 2026-10-02).

122. R Core Team. R 4.3.3, package utils: help pages. 2024. Help page “packageDescription” (utils 4.3.3), Details: comparing a version with text. R Foundation for Statistical Computing. https://www.R-project.org/ (accessed 2026-10-02).

123. Horst, Allison Marie and Hill, Alison Presmanes and Gorman, Kristen B. palmerpenguins: Palmer Archipelago (Antarctica) Penguin Data. 2020. Help page “penguins_raw”, Description and Format: what a row records. CRAN. https://allisonhorst.github.io/palmerpenguins/ (accessed 2026-10-02).

124. Venables, W. N. and Smith, D. M. and R Core Team. An Introduction to R: Notes on R, a Programming Environment for Data Analysis and Graphics. 2026. Version line at the head of the manual: a newer R than this book’s. https://cran.r-project.org/doc/manuals/r-release/R-intro.html (accessed 2026-10-02).

125. Tidyverse team. Changelogs of ggplot2, dplyr, readr and tidyr: entries after ggplot2 3.4.4, dplyr 1.1.4, readr 2.1.5 and tidyr 1.3.1. 2026. dplyr changelog, dplyr 1.2.0 heading: a newer dplyr than this book’s. tidyverse.org. https://ggplot2.tidyverse.org/news/ (accessed 2026-10-02).

126. Tidyverse team. Changelogs of ggplot2, dplyr, readr and tidyr: entries after ggplot2 3.4.4, dplyr 1.1.4, readr 2.1.5 and tidyr 1.3.1. 2026. dplyr changelog, dplyr 1.2.0, Minor improvements: the R version it requires. tidyverse.org. https://ggplot2.tidyverse.org/news/ (accessed 2026-10-02).

127. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. Ch. 6 “Workflow: scripts and projects”, 6.2.2 “Where does your analysis live?”: the working directory. https://r4ds.hadley.nz/ (accessed 2026-10-02).

128. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. Ch. 6, 6.2.4 “Relative and absolute paths”, first paragraph: what a relative path is relative to. https://r4ds.hadley.nz/ (accessed 2026-10-02).

129. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. Ch. 6, 6.2.4 “Relative and absolute paths”, second paragraph: why absolute paths are never used in a script. https://r4ds.hadley.nz/ (accessed 2026-10-02).

130. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. Ch. 6, 6.2.4, last paragraph: backslashes in Windows paths. https://r4ds.hadley.nz/ (accessed 2026-10-02).

131. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. Ch. 6, 6.2.2, on setwd(). https://r4ds.hadley.nz/ (accessed 2026-10-02).

132. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. Ch. 6, 6.1.3 “Saving and naming”, first of the three principles for file names. https://r4ds.hadley.nz/ (accessed 2026-10-02).

133. M"uller, Kirill. here: A Simpler Way to Find Your Files, version 1.0.1: help pages. 2020. Help page “here” (here 1.0.1), Description. R package, CRAN. https://CRAN.R-project.org/package=here (accessed 2026-10-02).

134. M"uller, Kirill. here: A Simpler Way to Find Your Files, version 1.0.1: help pages. 2020. Help page “here” (here 1.0.1), “Project root”: the upward search and its markers. R package, CRAN. https://CRAN.R-project.org/package=here (accessed 2026-10-02).

135. M"uller, Kirill. here: A Simpler Way to Find Your Files, version 1.0.1: help pages. 2020. Help page “here” (here 1.0.1), “Project root”, last sentence: the arguments become path components. R package, CRAN. https://CRAN.R-project.org/package=here (accessed 2026-10-02).

136. Wilson, Greg and Bryan, Jennifer and Cranston, Karen and Kitzes, Justin and Nederbragt, Lex and Teal, Tracy K. Good enough practices in scientific computing. 2017. Project organization, recommendation 4a. https://pmc.ncbi.nlm.nih.gov/articles/PMC5480810/ (accessed 2026-10-02).

137. Wilson, Greg and Bryan, Jennifer and Cranston, Karen and Kitzes, Justin and Nederbragt, Lex and Teal, Tracy K. Good enough practices in scientific computing. 2017. Project organization, recommendation 4c: raw data apart from generated files. https://pmc.ncbi.nlm.nih.gov/articles/PMC5480810/ (accessed 2026-10-02).

138. Wilson, Greg and Bryan, Jennifer and Cranston, Karen and Kitzes, Justin and Nederbragt, Lex and Teal, Tracy K. Good enough practices in scientific computing. 2017. Data management, recommendation 1a “Save the raw data”. https://pmc.ncbi.nlm.nih.gov/articles/PMC5480810/ (accessed 2026-10-02).

139. Wilson, Greg and Bryan, Jennifer and Cranston, Karen and Kitzes, Justin and Nederbragt, Lex and Teal, Tracy K. Good enough practices in scientific computing. 2017. Collaboration, recommendation 3a: the README. https://pmc.ncbi.nlm.nih.gov/articles/PMC5480810/ (accessed 2026-10-02).

140. The Carpentries. R for Reproducible Scientific Analysis. 2026. Ep. 02 “Project Management With RStudio”, “Treat data as read only”. Software Carpentry lesson (r-novice-gapminder). https://swcarpentry.github.io/r-novice-gapminder/ (accessed 2026-10-02).

141. The Carpentries. R for Reproducible Scientific Analysis. 2026. Ep. 02, “Treat generated output as disposable”. Software Carpentry lesson (r-novice-gapminder). https://swcarpentry.github.io/r-novice-gapminder/ (accessed 2026-10-02).

142. The Carpentries. R for Reproducible Scientific Analysis. 2026. Ep. 02, “Version Control”. Software Carpentry lesson (r-novice-gapminder). https://swcarpentry.github.io/r-novice-gapminder/ (accessed 2026-10-02).

143. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. Ch. 6, 6.2.3 “RStudio projects”, first paragraph: making a project. https://r4ds.hadley.nz/ (accessed 2026-10-02).

144. The Carpentries. R for Reproducible Scientific Analysis. 2026. Ep. 04 “Data Structures”, after str(cats2): what a data frame is made of. Software Carpentry lesson (r-novice-gapminder). https://swcarpentry.github.io/r-novice-gapminder/ (accessed 2026-10-02).

145. The Carpentries. R for Reproducible Scientific Analysis. 2026. Ep. 04 “Data Structures”: the columns are vectors. Software Carpentry lesson (r-novice-gapminder). https://swcarpentry.github.io/r-novice-gapminder/ (accessed 2026-10-02).

146. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. Ch. 1 “Data visualization”, 1.2.1 “The penguins data frame”: definition of a data frame. https://r4ds.hadley.nz/ (accessed 2026-10-02).

147. Wickham, Hadley. Tidy data. Article licensed CC BY (Creative Commons Attribution, per the journal’s article page). Quoted from the article PDF. Section 2.2 “Data semantics”, first paragraph: variable and observation. https://www.jstatsoft.org/article/view/v059i10 (accessed 2026-10-02).

148. Wickham, Hadley. Tidy data. Article licensed CC BY (Creative Commons Attribution, per the journal’s article page). Quoted from the article PDF. Section 2.3 “Tidy data”: the three rules. https://www.jstatsoft.org/article/view/v059i10 (accessed 2026-10-02).

149. Wickham, Hadley. Tidy data. Article licensed CC BY (Creative Commons Attribution, per the journal’s article page). Quoted from the article PDF. Section 2.3, after the three rules: any other arrangement. https://www.jstatsoft.org/article/view/v059i10 (accessed 2026-10-02).

150. Wickham, Hadley. Tidy data. Article licensed CC BY (Creative Commons Attribution, per the journal’s article page). Quoted from the article PDF. Section 3 “Tidying messy datasets”, the list of the five most common problems (first, second and fourth items). https://www.jstatsoft.org/article/view/v059i10 (accessed 2026-10-02).

151. Wickham, Hadley. Tidy data. Article licensed CC BY (Creative Commons Attribution, per the journal’s article page). Quoted from the article PDF. Section 2.2: several types of observation in one study (the allergy trial example). https://www.jstatsoft.org/article/view/v059i10 (accessed 2026-10-02).

152. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. Ch. 5 “Data tidying”, 5.2 “Tidy data”: the book’s own third rule, which differs from Wickham 2014. https://r4ds.hadley.nz/ (accessed 2026-10-02).

153. Broman, Karl W and Woo, Kara H. Data organization in spreadsheets. 2018. Section 6 “Put Just One Thing in a Cell”, first paragraph. https://doi.org/10.1080/00031305.2017.1375989 (accessed 2026-10-02).

154. Broman, Karl W and Woo, Kara H. Data organization in spreadsheets. 2018. Section 7 “Make it a Rectangle”, first paragraph. https://doi.org/10.1080/00031305.2017.1375989 (accessed 2026-10-02).

155. Broman, Karl W and Woo, Kara H. Data organization in spreadsheets. 2018. Section 10 “Do Not Use Font Color or Highlighting as Data”, last paragraph. https://doi.org/10.1080/00031305.2017.1375989 (accessed 2026-10-02).

156. Broman, Karl W and Woo, Kara H. Data organization in spreadsheets. 2018. Section 6, second paragraph: units out of the cell. https://doi.org/10.1080/00031305.2017.1375989 (accessed 2026-10-02).

157. Horst, Allison Marie and Hill, Alison Presmanes and Gorman, Kristen B. palmerpenguins: Palmer Archipelago (Antarctica) Penguin Data. 2020. Help page “penguins_raw” (palmerpenguins 0.1.1), Description and the Individual ID entry. CRAN. https://allisonhorst.github.io/palmerpenguins/ (accessed 2026-10-02).

158. Wickham, Hadley. Tidy data. Article licensed CC BY (Creative Commons Attribution, per the journal’s article page). Quoted from the article PDF. Section 2.2 “Data semantics”, second paragraph: defining variables in general. https://www.jstatsoft.org/article/view/v059i10 (accessed 2026-10-02).

159. Broman, Karl W and Woo, Kara H. Data organization in spreadsheets. 2018. Section 9 “No Calculations in the Raw Data Files”, first paragraph. https://doi.org/10.1080/00031305.2017.1375989 (accessed 2026-10-02).

160. Horst, Allison Marie and Hill, Alison Presmanes and Gorman, Kristen B. palmerpenguins: Palmer Archipelago (Antarctica) Penguin Data. 2020. Help page “penguins_raw” (palmerpenguins 0.1.1), Format, the Individual ID entry. CRAN. https://allisonhorst.github.io/palmerpenguins/ (accessed 2026-10-02).

161. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. Ch. 7 “Data import”, 7.2 “Reading data from a file”, first paragraph: what a CSV file is. https://r4ds.hadley.nz/ (accessed 2026-10-02).

162. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. Ch. 7, 7.3 “Controlling column types”, first paragraph. https://r4ds.hadley.nz/ (accessed 2026-10-02).

163. Broman, Karl W and Woo, Kara H. Data organization in spreadsheets. 2018. Section 13 “Save the Data in Plain Text Files”, second paragraph. https://doi.org/10.1080/00031305.2017.1375989 (accessed 2026-10-02).

164. Wickham, Hadley and Hester, Jim and Bryan, Jennifer. readr: Read Rectangular Text Data, version 2.1.5: help pages. 2024. Help page “read_delim” (readr 2.1.5), Arguments, col_types: types guessed from a sample of rows. R package, CRAN. https://CRAN.R-project.org/package=readr (accessed 2026-10-02).

165. Wickham, Hadley and Hester, Jim and Bryan, Jennifer. readr: Read Rectangular Text Data, version 2.1.5: help pages. 2024. Help page “read_delim” (readr 2.1.5), Arguments, col_types: the message printed when types are guessed. R package, CRAN. https://CRAN.R-project.org/package=readr (accessed 2026-10-02).

166. Wickham, Hadley and Hester, Jim and Bryan, Jennifer. readr: Read Rectangular Text Data, version 2.1.5: help pages. 2024. Help page “read_delim” (readr 2.1.5), Usage, the default of na. R package, CRAN. https://CRAN.R-project.org/package=readr (accessed 2026-10-02).

167. Wickham, Hadley and Hester, Jim and Bryan, Jennifer. readr: Read Rectangular Text Data, version 2.1.5: help pages. 2024. Help page “read_delim” (readr 2.1.5), Arguments, na. R package, CRAN. https://CRAN.R-project.org/package=readr (accessed 2026-10-02).

168. Wickham, Hadley and Hester, Jim and Bryan, Jennifer. readr: Read Rectangular Text Data, version 2.1.5: help pages. 2024. Help page “read_delim” (readr 2.1.5), Value. R package, CRAN. https://CRAN.R-project.org/package=readr (accessed 2026-10-02).

169. Wickham, Hadley and Hester, Jim and Bryan, Jennifer. readr: Read Rectangular Text Data, version 2.1.5: help pages. 2024. Help page “read_delim” (readr 2.1.5), Arguments, file: literal data. R package, CRAN. https://CRAN.R-project.org/package=readr (accessed 2026-10-02).

170. Wickham, Hadley and Bryan, Jennifer. readxl: Read Excel Files, version 1.4.3: help pages. 2023. Help page “read_excel” (readxl 1.4.3), Description and Arguments sheet and range. R package, CRAN. https://CRAN.R-project.org/package=readxl (accessed 2026-10-02).

171. Wickham, Hadley and Bryan, Jennifer. readxl: Read Excel Files, version 1.4.3: help pages. 2023. Help page “read_excel” (readxl 1.4.3), Arguments, range. R package, CRAN. https://CRAN.R-project.org/package=readxl (accessed 2026-10-02).

172. Wickham, Hadley and Miller, Evan and Smith, Danny. haven: Import and Export ‘SPSS’, ‘Stata’ and ‘SAS’ Files, version 2.5.4: help pages. 2023. Help page “read_spss” (haven 2.5.4), Description. R package, CRAN. https://CRAN.R-project.org/package=haven (accessed 2026-10-02).

173. Wickham, Hadley and Miller, Evan and Smith, Danny. haven: Import and Export ‘SPSS’, ‘Stata’ and ‘SAS’ Files, version 2.5.4: help pages. 2023. Help page “read_spss” (haven 2.5.4), Arguments, user_na. R package, CRAN. https://CRAN.R-project.org/package=haven (accessed 2026-10-02).

174. Wickham, Hadley and Miller, Evan and Smith, Danny. haven: Import and Export ‘SPSS’, ‘Stata’ and ‘SAS’ Files, version 2.5.4: help pages. 2023. Help page “read_xpt” (haven 2.5.4), title, and Value: variable labels. R package, CRAN. https://CRAN.R-project.org/package=haven (accessed 2026-10-02).

175. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. Ch. 20 “Spreadsheets”, 20.2.3, after the reading examples: look at a copy, read the original. https://r4ds.hadley.nz/ (accessed 2026-10-02).

176. National Center for Health Statistics. National Health and Nutrition Examination Survey, August 2021–August 2023: Body Measures (BMX_L) and Demographic Variables and Sample Weights (DEMO_L), Data Files and Documentation. 2024. BMX_L codebook (block 3), “Body Mass Index (BMXBMI)”. NCHS. https://wwwn.cdc.gov/nchs/nhanes/search/datapage.aspx?Component=Examination&Cycle=2021-2023 (accessed 2026-10-02).

177. National Center for Health Statistics. National Health and Nutrition Examination Survey, August 2021–August 2023: Body Measures (BMX_L) and Demographic Variables and Sample Weights (DEMO_L), Data Files and Documentation. 2024. BMX_L codebook (block 3), “Sample weights”. NCHS. https://wwwn.cdc.gov/nchs/nhanes/search/datapage.aspx?Component=Examination&Cycle=2021-2023 (accessed 2026-10-02).

178. National Center for Health Statistics. National Health and Nutrition Examination Survey, August 2021–August 2023: Body Measures (BMX_L) and Demographic Variables and Sample Weights (DEMO_L), Data Files and Documentation. 2024. figures quoted in the text. NCHS. https://wwwn.cdc.gov/nchs/nhanes/search/datapage.aspx?Component=Examination&Cycle=2021-2023 (accessed 2026-10-02).

179. Broman, Karl W and Woo, Kara H. Data organization in spreadsheets. 2018. section 8 “Create a Data Dictionary”, first paragraph. https://doi.org/10.1080/00031305.2017.1375989 (accessed 2026-10-02).

180. Broman, Karl W and Woo, Kara H. Data organization in spreadsheets. 2018. section 8 “Create a Data Dictionary”: the exact name, and the units. https://doi.org/10.1080/00031305.2017.1375989 (accessed 2026-10-02).

181. Broman, Karl W and Woo, Kara H. Data organization in spreadsheets. 2018. section 8 “Create a Data Dictionary”, last paragraph: allowed values. https://doi.org/10.1080/00031305.2017.1375989 (accessed 2026-10-02).

182. Broman, Karl W and Woo, Kara H. Data organization in spreadsheets. 2018. section 8 “Create a Data Dictionary”: metadata and the ReadMe file. https://doi.org/10.1080/00031305.2017.1375989 (accessed 2026-10-02).

183. Wilson, Greg and Bryan, Jennifer and Cranston, Karen and Kitzes, Justin and Nederbragt, Lex and Teal, Tracy K. Good enough practices in scientific computing. 2017. Collaboration, 3a “Create an overview of your project”: what a README holds. https://pmc.ncbi.nlm.nih.gov/articles/PMC5480810/ (accessed 2026-10-02).

184. Wilson, Greg and Bryan, Jennifer and Cranston, Karen and Kitzes, Justin and Nederbragt, Lex and Teal, Tracy K. Good enough practices in scientific computing. 2017. Data management, 1a “Save the raw data”: record how the raw data were obtained. https://pmc.ncbi.nlm.nih.gov/articles/PMC5480810/ (accessed 2026-10-02).

185. Wilson, Greg and Bryan, Jennifer and Cranston, Karen and Kitzes, Justin and Nederbragt, Lex and Teal, Tracy K. Good enough practices in scientific computing. 2017. Project organization, 4c: raw data and metadata in a data directory. https://pmc.ncbi.nlm.nih.gov/articles/PMC5480810/ (accessed 2026-10-02).

186. Wilson, Greg and Bryan, Jennifer and Cranston, Karen and Kitzes, Justin and Nederbragt, Lex and Teal, Tracy K. Good enough practices in scientific computing. 2017. Collaboration, 3d “Make the license explicit”. https://pmc.ncbi.nlm.nih.gov/articles/PMC5480810/ (accessed 2026-10-02).

187. Wilson, Greg and Bryan, Jennifer and Cranston, Karen and Kitzes, Justin and Nederbragt, Lex and Teal, Tracy K. Good enough practices in scientific computing. 2017. Collaboration, 3d: the licences recommended, CC-0 among them. https://pmc.ncbi.nlm.nih.gov/articles/PMC5480810/ (accessed 2026-10-02).

188. Wilson, Greg and Bryan, Jennifer and Cranston, Karen and Kitzes, Justin and Nederbragt, Lex and Teal, Tracy K. Good enough practices in scientific computing. 2017. Collaboration, 3e “Make the project citable”. https://pmc.ncbi.nlm.nih.gov/articles/PMC5480810/ (accessed 2026-10-02).

189. Wilson, Greg and Bryan, Jennifer and Cranston, Karen and Kitzes, Justin and Nederbragt, Lex and Teal, Tracy K. Good enough practices in scientific computing. 2017. Data management, 1c “Create the data you wish to see in the world”: variable names and codes. https://pmc.ncbi.nlm.nih.gov/articles/PMC5480810/ (accessed 2026-10-02).

190. Sandve, Geir Kjetil and Nekrutenko, Anton and Taylor, James and Hovig, Eivind. Ten simple rules for reproducible computational research. 2013. Rule 1 “For Every Result, Keep Track of How It Was Produced”, minimum. https://pmc.ncbi.nlm.nih.gov/articles/PMC3812051/ (accessed 2026-10-02).

191. Horst, Allison Marie and Hill, Alison Presmanes and Gorman, Kristen B. palmerpenguins: Palmer Archipelago (Antarctica) Penguin Data. 2020. help page “penguins_raw” (palmerpenguins 0.1.1), Format: the 17 variables. CRAN. https://allisonhorst.github.io/palmerpenguins/ (accessed 2026-10-02).

192. Horst, Allison Marie and Hill, Alison Presmanes and Gorman, Kristen B. palmerpenguins: Palmer Archipelago (Antarctica) Penguin Data. 2020. help page “penguins_raw”, Format: Individual ID. CRAN. https://allisonhorst.github.io/palmerpenguins/ (accessed 2026-10-02).

193. Horst, Allison Marie and Hill, Alison Presmanes and Gorman, Kristen B. palmerpenguins: Palmer Archipelago (Antarctica) Penguin Data. 2020. help page “penguins_raw”, Format: Culmen Length and Body Mass, with their units. CRAN. https://allisonhorst.github.io/palmerpenguins/ (accessed 2026-10-02).

194. Horst, Allison Marie and Hill, Alison Presmanes and Gorman, Kristen B. palmerpenguins: Palmer Archipelago (Antarctica) Penguin Data. 2020. DESCRIPTION file (palmerpenguins 0.1.1): licence. CRAN. https://allisonhorst.github.io/palmerpenguins/ (accessed 2026-10-02).

195. Horst, Allison Marie and Hill, Alison Presmanes and Gorman, Kristen B. palmerpenguins: Palmer Archipelago (Antarctica) Penguin Data. 2020. help page “penguins_raw”, Source: who collected the data, and the first publication. CRAN. https://allisonhorst.github.io/palmerpenguins/ (accessed 2026-10-02).

196. Wickham, Hadley and Hester, Jim and Bryan, Jennifer. readr: Read Rectangular Text Data, version 2.1.5: help pages. 2024. vignette “readr” (readr 2.1.5), column specification: spec(). R package, CRAN. https://CRAN.R-project.org/package=readr (accessed 2026-10-02).

197. Wickham, Hadley and François, Romain and Henry, Lionel and Müller, Kirill and Vaughan, Davis. dplyr: A Grammar of Data Manipulation, version 1.1.4: help pages. 2023. help page “n_distinct” (dplyr 1.1.4), Description. R package, CRAN. https://CRAN.R-project.org/package=dplyr (accessed 2026-10-02).

198. Wickham, Hadley and François, Romain and Henry, Lionel and Müller, Kirill and Vaughan, Davis. dplyr: A Grammar of Data Manipulation, version 1.1.4: help pages. 2023. help page “count” (dplyr 1.1.4), Description. R package, CRAN. https://CRAN.R-project.org/package=dplyr (accessed 2026-10-02).

199. Wickham, Hadley and François, Romain and Henry, Lionel and Müller, Kirill and Vaughan, Davis. dplyr: A Grammar of Data Manipulation, version 1.1.4: help pages. 2023. help page “filter” (dplyr 1.1.4), Arguments, “...”: columns named as variables of the data. R package, CRAN. https://CRAN.R-project.org/package=dplyr (accessed 2026-10-02).

200. R Core Team. R 4.3.3, package base: help pages. 2024. help page “library” (base 4.3.3), Arguments: warn.conflicts. R Foundation for Statistical Computing. https://www.R-project.org/ (accessed 2026-10-02).

201. Wickham, Hadley and Hester, Jim and Bryan, Jennifer. readr: Read Rectangular Text Data, version 2.1.5: help pages. 2024. vignette “readr” (readr 2.1.5), Available column specifications, and Date/times: the default date format. R package, CRAN. https://CRAN.R-project.org/package=readr (accessed 2026-10-02).

202. Wickham, Hadley and Hester, Jim and Bryan, Jennifer. readr: Read Rectangular Text Data, version 2.1.5: help pages. 2024. help page “read_delim” (readr 2.1.5), Arguments: quote. R package, CRAN. https://CRAN.R-project.org/package=readr (accessed 2026-10-02).

203. Horst, Allison Marie and Hill, Alison Presmanes and Gorman, Kristen B. palmerpenguins: Palmer Archipelago (Antarctica) Penguin Data. 2020. DESCRIPTION file (palmerpenguins 0.1.1), Description: LTER expanded. CRAN. https://allisonhorst.github.io/palmerpenguins/ (accessed 2026-10-02).

204. Wickham, Hadley and François, Romain and Henry, Lionel and Müller, Kirill and Vaughan, Davis. dplyr: A Grammar of Data Manipulation, version 1.1.4: help pages. 2023. help page “filter” (dplyr 1.1.4), Description: rows kept, and NA dropped. R package, CRAN. https://CRAN.R-project.org/package=dplyr (accessed 2026-10-02).

205. Wickham, Hadley and François, Romain and Henry, Lionel and Müller, Kirill and Vaughan, Davis. dplyr: A Grammar of Data Manipulation, version 1.1.4: help pages. 2023. help page “filter” (dplyr 1.1.4), Arguments, “...”: several conditions combined with &. R package, CRAN. https://CRAN.R-project.org/package=dplyr (accessed 2026-10-02).

206. Wickham, Hadley and François, Romain and Henry, Lionel and Müller, Kirill and Vaughan, Davis. dplyr: A Grammar of Data Manipulation, version 1.1.4: help pages. 2023. help page “filter” (dplyr 1.1.4), “Useful filter functions”. R package, CRAN. https://CRAN.R-project.org/package=dplyr (accessed 2026-10-02).

207. Wickham, Hadley and François, Romain and Henry, Lionel and Müller, Kirill and Vaughan, Davis. dplyr: A Grammar of Data Manipulation, version 1.1.4: help pages. 2023. help page “select” (dplyr 1.1.4), Value: rows unaffected, renaming inside select. R package, CRAN. https://CRAN.R-project.org/package=dplyr (accessed 2026-10-02).

208. Wickham, Hadley and François, Romain and Henry, Lionel and Müller, Kirill and Vaughan, Davis. dplyr: A Grammar of Data Manipulation, version 1.1.4: help pages. 2023. help page “select” (dplyr 1.1.4), Overview of selection features: starts_with(). R package, CRAN. https://CRAN.R-project.org/package=dplyr (accessed 2026-10-02).

209. Wickham, Hadley and François, Romain and Henry, Lionel and Müller, Kirill and Vaughan, Davis. dplyr: A Grammar of Data Manipulation, version 1.1.4: help pages. 2023. help page “rename” (dplyr 1.1.4), Description and Value. R package, CRAN. https://CRAN.R-project.org/package=dplyr (accessed 2026-10-02).

210. Wickham, Hadley and François, Romain and Henry, Lionel and Müller, Kirill and Vaughan, Davis. dplyr: A Grammar of Data Manipulation, version 1.1.4: help pages. 2023. help page “arrange” (dplyr 1.1.4), Details, Missing values. R package, CRAN. https://CRAN.R-project.org/package=dplyr (accessed 2026-10-02).

211. Wickham, Hadley and François, Romain and Henry, Lionel and Müller, Kirill and Vaughan, Davis. dplyr: A Grammar of Data Manipulation, version 1.1.4: help pages. 2023. help page “desc” (dplyr 1.1.4), Description. R package, CRAN. https://CRAN.R-project.org/package=dplyr (accessed 2026-10-02).

212. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. ch. 3 “Data transformation”, 3.1.3 dplyr basics: what the verbs have in common. https://r4ds.hadley.nz/ (accessed 2026-10-02).

213. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. ch. 3 “Data transformation”, 3.2.1 filter(): the input is not changed. https://r4ds.hadley.nz/ (accessed 2026-10-02).

214. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. ch. 3 “Data transformation”, 3.2.1 filter(): %in%. https://r4ds.hadley.nz/ (accessed 2026-10-02).

215. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. ch. 3 “Data transformation”, 3.1.3 dplyr basics: the pipe read as “then”. https://r4ds.hadley.nz/ (accessed 2026-10-02).

216. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. ch. 7 “Data import”, 7.2: column names with spaces need backticks. https://r4ds.hadley.nz/ (accessed 2026-10-02).

217. The tidyverse team. The tidyverse style guide. 2.1 Object names: snake case. Online book. https://style.tidyverse.org/ (accessed 2026-10-02).

218. The tidyverse team. The tidyverse style guide. 4 Pipes: one step per line. Online book. https://style.tidyverse.org/ (accessed 2026-10-02).

219. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. ch. 3 “Data transformation”, 3.1.3 dplyr basics: the functions called verbs. https://r4ds.hadley.nz/ (accessed 2026-10-02).

220. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. ch. 3 “Data transformation”, 3.2.2 Common mistakes: “or” written as in English. https://r4ds.hadley.nz/ (accessed 2026-10-02).

221. Tidyverse team. Changelogs of ggplot2, dplyr, readr and tidyr: entries after ggplot2 3.4.4, dplyr 1.1.4, readr 2.1.5 and tidyr 1.3.1. 2026. dplyr 1.2.0 (CRAN release 2026-02-03), New features: filter_out(). tidyverse.org. https://ggplot2.tidyverse.org/news/ (accessed 2026-10-02).

222. Wickham, Hadley and François, Romain and Henry, Lionel and Müller, Kirill and Vaughan, Davis. dplyr: A Grammar of Data Manipulation, version 1.1.4: help pages. 2023. help page “mutate” (dplyr 1.1.4), Description. R package, CRAN. https://CRAN.R-project.org/package=dplyr (accessed 2026-10-02).

223. Wickham, Hadley and François, Romain and Henry, Lionel and Müller, Kirill and Vaughan, Davis. dplyr: A Grammar of Data Manipulation, version 1.1.4: help pages. 2023. help page “mutate” (dplyr 1.1.4), Value: rows. R package, CRAN. https://CRAN.R-project.org/package=dplyr (accessed 2026-10-02).

224. Wickham, Hadley and François, Romain and Henry, Lionel and Müller, Kirill and Vaughan, Davis. dplyr: A Grammar of Data Manipulation, version 1.1.4: help pages. 2023. help page “mutate” (dplyr 1.1.4), Arguments, .before, .after: where new columns go. R package, CRAN. https://CRAN.R-project.org/package=dplyr (accessed 2026-10-02).

225. Wickham, Hadley and François, Romain and Henry, Lionel and Müller, Kirill and Vaughan, Davis. dplyr: A Grammar of Data Manipulation, version 1.1.4: help pages. 2023. help page “if_else” (dplyr 1.1.4), Value. R package, CRAN. https://CRAN.R-project.org/package=dplyr (accessed 2026-10-02).

226. Wickham, Hadley and François, Romain and Henry, Lionel and Müller, Kirill and Vaughan, Davis. dplyr: A Grammar of Data Manipulation, version 1.1.4: help pages. 2023. help page “case_when” (dplyr 1.1.4), Description: cases in order, first match. R package, CRAN. https://CRAN.R-project.org/package=dplyr (accessed 2026-10-02).

227. Wickham, Hadley and François, Romain and Henry, Lionel and Müller, Kirill and Vaughan, Davis. dplyr: A Grammar of Data Manipulation, version 1.1.4: help pages. 2023. help page “case_when” (dplyr 1.1.4), Arguments, .default: NA conditions, and no default. R package, CRAN. https://CRAN.R-project.org/package=dplyr (accessed 2026-10-02).

228. Wickham, Hadley and François, Romain and Henry, Lionel and Müller, Kirill and Vaughan, Davis. dplyr: A Grammar of Data Manipulation, version 1.1.4: help pages. 2023. help page “rename” (dplyr 1.1.4), Description: rename_with(). R package, CRAN. https://CRAN.R-project.org/package=dplyr (accessed 2026-10-02).

229. R Core Team. R 4.3.3, package base: help pages. 2024. help page “Round” (base 4.3.3), Description: round(). R Foundation for Statistical Computing. https://www.R-project.org/ (accessed 2026-10-02).

230. R Core Team. R 4.3.3, package base: help pages. 2024. help page “Round” (base 4.3.3), Details: rounding off a 5, and the represented number. R Foundation for Statistical Computing. https://www.R-project.org/ (accessed 2026-10-02).

231. Wickham, Hadley. stringr: Simple, Consistent Wrappers for Common String Operations, version 1.5.1: help pages. 2023. help page “str_trim” (stringr 1.5.1), Description. R package, CRAN. https://CRAN.R-project.org/package=stringr (accessed 2026-10-02).

232. Wickham, Hadley. stringr: Simple, Consistent Wrappers for Common String Operations, version 1.5.1: help pages. 2023. help page “case” (stringr 1.5.1), Description: str_to_lower(). R package, CRAN. https://CRAN.R-project.org/package=stringr (accessed 2026-10-02).

233. International Institute for Population Sciences and Ministry of Health and Family Welfare. National Family Health Survey (NFHS-5), 2019-21: India Fact Sheet. 2021. Key indicators, row 87: the lower BMI cut-point and its boundary. https://dhsprogram.com/pubs/pdf/OF43/India_National_Fact_Sheet.pdf (accessed 2026-09-24).

234. International Institute for Population Sciences and Ministry of Health and Family Welfare. National Family Health Survey (NFHS-5), 2019-21: India Fact Sheet. 2021. Key indicators, row 89: the upper BMI cut-point and its boundary. https://dhsprogram.com/pubs/pdf/OF43/India_National_Fact_Sheet.pdf (accessed 2026-09-24).

235. Wickham, Hadley and François, Romain and Henry, Lionel and Müller, Kirill and Vaughan, Davis. dplyr: A Grammar of Data Manipulation, version 1.1.4: help pages. 2023. help page “case_when” (dplyr 1.1.4), Usage: .default after the cases. R package, CRAN. https://CRAN.R-project.org/package=dplyr (accessed 2026-10-02).

236. Wickham, Hadley and François, Romain and Henry, Lionel and Müller, Kirill and Vaughan, Davis. dplyr: A Grammar of Data Manipulation, version 1.1.4: help pages. 2023. help page “rename” (dplyr 1.1.4), Arguments, .fn; Examples. R package, CRAN. https://CRAN.R-project.org/package=dplyr (accessed 2026-10-02).

237. R Core Team. R 4.3.3, package base: help pages. 2024. help page “NA” (base 4.3.3), Title and Description. R Foundation for Statistical Computing. https://www.R-project.org/ (accessed 2026-10-02).

238. R Core Team. R 4.3.3, package base: help pages. 2024. help page “NA” (base 4.3.3), Details: arithmetic on NA. R Foundation for Statistical Computing. https://www.R-project.org/ (accessed 2026-10-02).

239. R Core Team. R 4.3.3, package base: help pages. 2024. help page “NA” (base 4.3.3), Value: is.na() on a data frame. R Foundation for Statistical Computing. https://www.R-project.org/ (accessed 2026-10-02).

240. Venables, W. N. and Smith, D. M. and R Core Team. An Introduction to R: Notes on R, a Programming Environment for Data Analysis and Graphics. 2026. section 2.4 “Logical vectors” (heading lost in extraction), last paragraph: TRUE and FALSE in arithmetic. https://cran.r-project.org/doc/manuals/r-release/R-intro.html (accessed 2026-10-02).

241. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. ch. 18 “Missing values”, 18.2.2 Fixed values: special values that stand for missing. https://r4ds.hadley.nz/ (accessed 2026-10-02).

242. The Carpentries. Data Organization in Spreadsheets for Ecologists. 2026. ep. 02 “Formatting problems”, Using problematic null values. Data Carpentry lesson (spreadsheet-ecology-lesson). https://datacarpentry.github.io/spreadsheet-ecology-lesson/ (accessed 2026-10-02).

243. Wickham, Hadley and Vaughan, Davis and Girlich, Maximilian. tidyr: Tidy Messy Data, version 1.3.1: help pages. 2024. help page “drop_na” (tidyr 1.3.1), Description. R package, CRAN. https://CRAN.R-project.org/package=tidyr (accessed 2026-10-02).

244. Wickham, Hadley and Vaughan, Davis and Girlich, Maximilian. tidyr: Tidy Messy Data, version 1.3.1: help pages. 2024. help page “drop_na” (tidyr 1.3.1), Arguments, ...: no columns named means all columns. R package, CRAN. https://CRAN.R-project.org/package=tidyr (accessed 2026-10-02).

245. Wickham, Hadley and Vaughan, Davis and Girlich, Maximilian. tidyr: Tidy Messy Data, version 1.3.1: help pages. 2024. help page “drop_na” (tidyr 1.3.1), Details: complete rows. R package, CRAN. https://CRAN.R-project.org/package=tidyr (accessed 2026-10-02).

246. Wickham, Hadley and François, Romain and Henry, Lionel and Müller, Kirill and Vaughan, Davis. dplyr: A Grammar of Data Manipulation, version 1.1.4: help pages. 2023. help page “summarise” (dplyr 1.1.4), Description and Arguments, ...: one row, and sum(is.na(y)) as a summary. R package, CRAN. https://CRAN.R-project.org/package=dplyr (accessed 2026-10-02).

247. Wickham, Hadley and François, Romain and Henry, Lionel and Müller, Kirill and Vaughan, Davis. dplyr: A Grammar of Data Manipulation, version 1.1.4: help pages. 2023. help page “context” (dplyr 1.1.4), Description: n(). R package, CRAN. https://CRAN.R-project.org/package=dplyr (accessed 2026-10-02).

248. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. ch. 16 “Factors”, 16.1 Introduction: what a factor is for. https://r4ds.hadley.nz/ (accessed 2026-10-02).

249. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. ch. 16, 16.2 Factor basics: values not in the levels. https://r4ds.hadley.nz/ (accessed 2026-10-02).

250. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. ch. 16, 16.2 Factor basics: levels taken from the data. https://r4ds.hadley.nz/ (accessed 2026-10-02).

251. Wickham, Hadley. forcats: Tools for Working with Categorical Variables (Factors), version 1.0.0: help pages. 2023. help page “fct_recode” (forcats 1.0.0), Arguments: new name on the left, old on the right. R package, CRAN. https://CRAN.R-project.org/package=forcats (accessed 2026-10-02).

252. Wickham, Hadley. forcats: Tools for Working with Categorical Variables (Factors), version 1.0.0: help pages. 2023. help page “fct_relevel” (forcats 1.0.0), Arguments, ...: levels not mentioned. R package, CRAN. https://CRAN.R-project.org/package=forcats (accessed 2026-10-02).

253. Wickham, Hadley. forcats: Tools for Working with Categorical Variables (Factors), version 1.0.0: help pages. 2023. help page “fct_inorder” (forcats 1.0.0), Description: fct_infreq(). R package, CRAN. https://CRAN.R-project.org/package=forcats (accessed 2026-10-02).

254. Wickham, Hadley and François, Romain and Henry, Lionel and Müller, Kirill and Vaughan, Davis. dplyr: A Grammar of Data Manipulation, version 1.1.4: help pages. 2023. help page “count” (dplyr 1.1.4), Arguments, .drop. R package, CRAN. https://CRAN.R-project.org/package=dplyr (accessed 2026-10-02).

255. Broman, Karl W and Woo, Kara H. Data organization in spreadsheets. 2018. section 2 “Be Consistent”, consistent codes for categorical variables (p. 2). https://doi.org/10.1080/00031305.2017.1375989 (accessed 2026-10-02).

256. Spinu, Vitalie and Grolemund, Garrett and Wickham, Hadley. lubridate: Make Dealing with Dates a Little Easier, version 1.9.3: help pages. 2023. help page “ymd” (lubridate 1.9.3), Description: text to Date, any separator. R package, CRAN. https://CRAN.R-project.org/package=lubridate (accessed 2026-10-02).

257. Spinu, Vitalie and Grolemund, Garrett and Wickham, Hadley. lubridate: Make Dealing with Dates a Little Easier, version 1.9.3: help pages. 2023. help page “ymd” (lubridate 1.9.3), Value: a Date when no time zone is given. R package, CRAN. https://CRAN.R-project.org/package=lubridate (accessed 2026-10-02).

258. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. ch. 17 “Dates and times”, 17.2.2 From strings: the function name follows the order of the parts. https://r4ds.hadley.nz/ (accessed 2026-10-02).

259. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. ch. 17, 17.2.1 During import: readr recognises ISO 8601 dates. https://r4ds.hadley.nz/ (accessed 2026-10-02).

260. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. ch. 17, 17.2.4 From other types: offsets in days from 1970-01-01. https://r4ds.hadley.nz/ (accessed 2026-10-02).

261. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. ch. 17, 17.4.1 Durations: subtracting two dates. https://r4ds.hadley.nz/ (accessed 2026-10-02).

262. The Carpentries. Data Organization in Spreadsheets for Ecologists. 2026. ep. 03 “Dates as data”, Dates stored as integers: the serial number of 2 July 2014. Data Carpentry lesson (spreadsheet-ecology-lesson). https://datacarpentry.github.io/spreadsheet-ecology-lesson/ (accessed 2026-10-02).

263. The Carpentries. Data Organization in Spreadsheets for Ecologists. 2026. ep. 03 “Dates as data”, Preferred date format: the 1904 date system. Data Carpentry lesson (spreadsheet-ecology-lesson). https://datacarpentry.github.io/spreadsheet-ecology-lesson/ (accessed 2026-10-02).

264. Broman, Karl W and Woo, Kara H. Data organization in spreadsheets. 2018. section 4 “Write Dates as YYYY-MM-DD” (p. 3): the ISO 8601 recommendation. https://doi.org/10.1080/00031305.2017.1375989 (accessed 2026-10-02).

265. Broman, Karl W and Woo, Kara H. Data organization in spreadsheets. 2018. section 4 “Write Dates as YYYY-MM-DD” (p. 3): Excel stores dates as numbers. https://doi.org/10.1080/00031305.2017.1375989 (accessed 2026-10-02).

266. Ziemann, Mark and Eren, Yotam and El-Osta, Assam. Gene name errors are widespread in the scientific literature. PMID 27552985, PMC4994289. Open access under CC BY 4.0. Quoted from the PMC open-access text. Background, first paragraph: SEPT2 converted to a full date. https://pmc.ncbi.nlm.nih.gov/articles/PMC4994289/ (accessed 2026-10-02).

267. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. ch. 17, 17.4.1 Durations: the average year. https://r4ds.hadley.nz/ (accessed 2026-10-02).

268. R Core Team. R 4.3.3, package base: help pages. 2024. help page “Round” (base 4.3.3), Description: floor(). R Foundation for Statistical Computing. https://www.R-project.org/ (accessed 2026-10-02).

269. The Carpentries. Data Organization in Spreadsheets for Ecologists. 2026. figures quoted in the text. Data Carpentry lesson (spreadsheet-ecology-lesson). https://datacarpentry.github.io/spreadsheet-ecology-lesson/ (accessed 2026-10-02).

270. Wickham, Hadley and François, Romain and Henry, Lionel and Müller, Kirill and Vaughan, Davis. dplyr: A Grammar of Data Manipulation, version 1.1.4: help pages. 2023. help page “summarise” (dplyr 1.1.4), Description: one row per group. R package, CRAN. https://CRAN.R-project.org/package=dplyr (accessed 2026-10-02).

271. R Core Team. R 4.3.3, package stats: help pages. 2024. help page “sd” (stats 4.3.3), Details: the denominator. R Foundation for Statistical Computing. https://www.R-project.org/ (accessed 2026-10-02).

272. R Core Team. R 4.3.3, package stats: help pages. 2024. help page “quantile” (stats 4.3.3), Types: nine algorithms, and the default. R Foundation for Statistical Computing. https://www.R-project.org/ (accessed 2026-10-02).

273. R Core Team. R 4.3.3, package stats: help pages. 2024. help page “quantile” (stats 4.3.3), Types: type 6 and the software that uses it. R Foundation for Statistical Computing. https://www.R-project.org/ (accessed 2026-10-02).

274. R Core Team. R 4.3.3, package stats: help pages. 2024. help page “quantile” (stats 4.3.3), Arguments, x: missing values. R Foundation for Statistical Computing. https://www.R-project.org/ (accessed 2026-10-02).

275. R Core Team. R 4.3.3, package base: help pages. 2024. help page “mean” (base 4.3.3), Arguments, na.rm (median: help page “median” (stats 4.3.3), Value). R Foundation for Statistical Computing. https://www.R-project.org/ (accessed 2026-10-02).

276. R Core Team. R 4.3.3, package stats: help pages. 2024. help page “IQR” (stats 4.3.3), Details: quartiles from quantile(). R Foundation for Statistical Computing. https://www.R-project.org/ (accessed 2026-10-02).

277. Wickham, Hadley and François, Romain and Henry, Lionel and Müller, Kirill and Vaughan, Davis. dplyr: A Grammar of Data Manipulation, version 1.1.4: help pages. 2023. help page “group_by” (dplyr 1.1.4), Description. R package, CRAN. https://CRAN.R-project.org/package=dplyr (accessed 2026-10-02).

278. Wickham, Hadley and François, Romain and Henry, Lionel and Müller, Kirill and Vaughan, Davis. dplyr: A Grammar of Data Manipulation, version 1.1.4: help pages. 2023. help page “summarise” (dplyr 1.1.4), Arguments, .groups: what is dropped by default. R package, CRAN. https://CRAN.R-project.org/package=dplyr (accessed 2026-10-02).

279. Wickham, Hadley and François, Romain and Henry, Lionel and Müller, Kirill and Vaughan, Davis. dplyr: A Grammar of Data Manipulation, version 1.1.4: help pages. 2023. help page “summarise” (dplyr 1.1.4), Arguments, .by. R package, CRAN. https://CRAN.R-project.org/package=dplyr (accessed 2026-10-02).

280. Wickham, Hadley and François, Romain and Henry, Lionel and Müller, Kirill and Vaughan, Davis. dplyr: A Grammar of Data Manipulation, version 1.1.4: help pages. 2023. help page “group_by” (dplyr 1.1.4), Ordering: groups in ascending order. R package, CRAN. https://CRAN.R-project.org/package=dplyr (accessed 2026-10-02).

281. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. ch. 3, section 3.6 “Case study: aggregates and sample size”, first sentence. https://r4ds.hadley.nz/ (accessed 2026-10-02).

282. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. ch. 7, section 7.5 “Writing to a file”, first sentence: write_csv(). https://r4ds.hadley.nz/ (accessed 2026-10-02).

283. Sandve, Geir Kjetil and Nekrutenko, Anton and Taylor, James and Hovig, Eivind. Ten simple rules for reproducible computational research. 2013. Rule 9 “Connect Textual Statements to Underlying Results”, last paragraph. https://pmc.ncbi.nlm.nih.gov/articles/PMC3812051/ (accessed 2026-10-02).

284. Wickham, Hadley and Vaughan, Davis and Girlich, Maximilian. tidyr: Tidy Messy Data, version 1.3.1: help pages. 2024. help page “pivot_longer” (tidyr 1.3.1), Description. R package, CRAN. https://CRAN.R-project.org/package=tidyr (accessed 2026-10-02).

285. Wickham, Hadley and Vaughan, Davis and Girlich, Maximilian. tidyr: Tidy Messy Data, version 1.3.1: help pages. 2024. help page “pivot_wider” (tidyr 1.3.1), Description. R package, CRAN. https://CRAN.R-project.org/package=tidyr (accessed 2026-10-02).

286. Wickham, Hadley and Vaughan, Davis and Girlich, Maximilian. tidyr: Tidy Messy Data, version 1.3.1: help pages. 2024. help page “pivot_longer” (tidyr 1.3.1), Arguments, names_prefix. R package, CRAN. https://CRAN.R-project.org/package=tidyr (accessed 2026-10-02).

287. Wickham, Hadley and Vaughan, Davis and Girlich, Maximilian. tidyr: Tidy Messy Data, version 1.3.1: help pages. 2024. help page “pivot_wider” (tidyr 1.3.1), Arguments, values_fill. R package, CRAN. https://CRAN.R-project.org/package=tidyr (accessed 2026-10-02).

288. Wickham, Hadley and Vaughan, Davis and Girlich, Maximilian. tidyr: Tidy Messy Data, version 1.3.1: help pages. 2024. help page “pivot_wider” (tidyr 1.3.1), Arguments, names_from and values_from; id_cols. R package, CRAN. https://CRAN.R-project.org/package=tidyr (accessed 2026-10-02).

289. Wickham, Hadley and François, Romain and Henry, Lionel and Müller, Kirill and Vaughan, Davis. dplyr: A Grammar of Data Manipulation, version 1.1.4: help pages. 2023. help page “mutate-joins” (dplyr 1.1.4), Description: what a mutating join does, and left_join(). R package, CRAN. https://CRAN.R-project.org/package=dplyr (accessed 2026-10-02).

290. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. ch. 19, section 19.3.1 “Mutating joins”: no match gives missing values. https://r4ds.hadley.nz/ (accessed 2026-10-02).

291. Wickham, Hadley and François, Romain and Henry, Lionel and Müller, Kirill and Vaughan, Davis. dplyr: A Grammar of Data Manipulation, version 1.1.4: help pages. 2023. vignette “two-table” (dplyr 1.1.4), “Observations”: duplicate keys add rows. R package, CRAN. https://CRAN.R-project.org/package=dplyr (accessed 2026-10-02).

292. Wickham, Hadley and François, Romain and Henry, Lionel and Müller, Kirill and Vaughan, Davis. dplyr: A Grammar of Data Manipulation, version 1.1.4: help pages. 2023. help page “filter-joins” (dplyr 1.1.4), Description: anti_join(). R package, CRAN. https://CRAN.R-project.org/package=dplyr (accessed 2026-10-02).

293. Wickham, Hadley and François, Romain and Henry, Lionel and Müller, Kirill and Vaughan, Davis. dplyr: A Grammar of Data Manipulation, version 1.1.4: help pages. 2023. help page “mutate-joins” (dplyr 1.1.4), Arguments, by: the natural join. R package, CRAN. https://CRAN.R-project.org/package=dplyr (accessed 2026-10-02).

294. Wickham, Hadley and François, Romain and Henry, Lionel and Müller, Kirill and Vaughan, Davis. dplyr: A Grammar of Data Manipulation, version 1.1.4: help pages. 2023. help page “mutate-joins” (dplyr 1.1.4), Arguments, relationship. R package, CRAN. https://CRAN.R-project.org/package=dplyr (accessed 2026-10-02).

295. National Center for Health Statistics. National Health and Nutrition Examination Survey, August 2021–August 2023: Body Measures (BMX_L) and Demographic Variables and Sample Weights (DEMO_L), Data Files and Documentation. 2024. BMX_L documentation, “Sample weights” paragraph (printed codebook). NCHS. https://wwwn.cdc.gov/nchs/nhanes/search/datapage.aspx?Component=Examination&Cycle=2021-2023 (accessed 2026-10-02).

296. National Center for Health Statistics. National Health and Nutrition Examination Survey, August 2021–August 2023: Body Measures (BMX_L) and Demographic Variables and Sample Weights (DEMO_L), Data Files and Documentation. 2024. DEMO_L documentation, analytic notes on RIDSTATR (printed codebook). NCHS. https://wwwn.cdc.gov/nchs/nhanes/search/datapage.aspx?Component=Examination&Cycle=2021-2023 (accessed 2026-10-02).

297. National Center for Health Statistics. National Health and Nutrition Examination Survey, August 2021–August 2023: Body Measures (BMX_L) and Demographic Variables and Sample Weights (DEMO_L), Data Files and Documentation. 2024. DEMO_L codebook, RIAGENDR code table. NCHS. https://wwwn.cdc.gov/nchs/nhanes/search/datapage.aspx?Component=Examination&Cycle=2021-2023 (accessed 2026-10-02).

298. Wickham, Hadley and Chang, Winston and Henry, Lionel and Pedersen, Thomas Lin and Takahashi, Kohske and Wilke, Claus and Woo, Kara and Yutani, Hiroaki and Dunnington, Dewey. ggplot2: Create Elegant Data Visualisa-

tions Using the Grammar of Graphics, version 3.4.4: help pages. 2023. help page “ggplot” (ggplot2 3.4.4), Description. R package, CRAN. https://CRAN.R-project.org/package=ggplot2 (accessed 2026-10-02).

299. Wickham, Hadley and Chang, Winston and Henry, Lionel and Pedersen, Thomas Lin and Takahashi, Kohske and Wilke, Claus and Woo, Kara and Yutani, Hiroaki and Dunnington, Dewey. ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics, version 3.4.4: help pages. 2023. help page “ggplot” (ggplot2 3.4.4), Details: adding with a plus sign. R package, CRAN. https://CRAN.R-project.org/package=ggplot2 (accessed 2026-10-02).

300. Wickham, Hadley and Chang, Winston and Henry, Lionel and Pedersen, Thomas Lin and Takahashi, Kohske and Wilke, Claus and Woo, Kara and Yutani, Hiroaki and Dunnington, Dewey. ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics, version 3.4.4: help pages. 2023. help page “aes” (ggplot2 3.4.4), Description. R package, CRAN. https://CRAN.R-project.org/package=ggplot2 (accessed 2026-10-02).

301. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. ch. 1, section 1.2.3 “Creating a ggplot”: what a geom is. https://r4ds.hadley.nz/ (accessed 2026-10-02).

302. Wickham, Hadley and Chang, Winston and Henry, Lionel and Pedersen, Thomas Lin and Takahashi, Kohske and Wilke, Claus and Woo, Kara and Yutani, Hiroaki and Dunnington, Dewey. ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics, version 3.4.4: help pages. 2023. help page “geom_histogram” (ggplot2 3.4.4), Description. R package, CRAN. https://CRAN.R-project.org/package=ggplot2 (accessed 2026-10-02).

303. Wickham, Hadley and Chang, Winston and Henry, Lionel and Pedersen, Thomas Lin and Takahashi, Kohske and Wilke, Claus and Woo, Kara and Yutani, Hiroaki and Dunnington, Dewey. ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics, version 3.4.4: help pages. 2023. help page “geom_histogram” (ggplot2 3.4.4), Arguments, bins: the default. R package, CRAN. https://CRAN.R-project.org/package=ggplot2 (accessed 2026-10-02).

304. Wickham, Hadley and Chang, Winston and Henry, Lionel and Pedersen, Thomas Lin and Takahashi, Kohske and Wilke, Claus and Woo, Kara and Yutani, Hiroaki and Dunnington, Dewey. ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics, version 3.4.4: help pages. 2023. help page “geom_histogram” (ggplot2 3.4.4), Usage of stat_bin(): the default for closed. R package, CRAN. https://CRAN.R-project.org/package=ggplot2 (accessed 2026-10-02).

305. Wickham, Hadley and Chang, Winston and Henry, Lionel and Pedersen, Thomas Lin and Takahashi, Kohske and Wilke, Claus and Woo, Kara and Yutani, Hiroaki and Dunnington, Dewey. ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics, version 3.4.4: help pages. 2023. help page “geom_histogram” (ggplot2 3.4.4), Arguments, center, boundary. R package, CRAN. https://CRAN.R-project.org/package=ggplot2 (accessed 2026-10-02).

306. Wickham, Hadley and Chang, Winston and Henry, Lionel and Pedersen, Thomas Lin and Takahashi, Kohske and Wilke, Claus and Woo, Kara and Yutani, Hiroaki and Dunnington, Dewey. ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics, version 3.4.4: help pages. 2023. help page “geom_boxplot” (ggplot2 3.4.4), Summary statistics: hinges. R package, CRAN. https://CRAN.R-project.org/package=ggplot2 (accessed 2026-10-02).

307. Wickham, Hadley and Chang, Winston and Henry, Lionel and Pedersen, Thomas Lin and Takahashi, Kohske and Wilke, Claus and Woo, Kara and Yutani, Hiroaki and Dunnington, Dewey. ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics, version 3.4.4: help pages. 2023. help page “geom_boxplot” (ggplot2 3.4.4), Summary statistics: whiskers and outlying points. R package, CRAN. https://CRAN.R-project.org/package=ggplot2 (accessed 2026-10-02).

308. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. ch. 1, section 1.5.1, the note on terminology: mapping against setting. https://r4ds.hadley.nz/ (accessed 2026-10-02).

309. The Carpentries. R for Reproducible Scientific Analysis. 2026. Ep. 08 “Creating publication-quality graphics with ggplot2”, tip “Setting an aesthetic to a value instead of a mapping”. Software Carpentry lesson (r-novice-gapminder). https://swcarpentry.github.io/r-novice-gapminder/ (accessed 2026-10-02).

310. Wickham, Hadley and Chang, Winston and Henry, Lionel and Pedersen, Thomas Lin and Takahashi, Kohske and Wilke, Claus and Woo, Kara and Yutani, Hiroaki and Dunnington, Dewey. ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics, version 3.4.4: help pages. 2023. help page "facet_wrap" (ggplot2 3.4.4), Description. R package, CRAN. https://CRAN.R-project.org/package=ggplot2 (accessed 2026-10-02).

311. Wickham, Hadley and Chang, Winston and Henry, Lionel and Pedersen, Thomas Lin and Takahashi, Kohske and Wilke, Claus and Woo, Kara and Yutani, Hiroaki and Dunnington, Dewey. ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics, version 3.4.4: help pages. 2023. help page "ggsave" (ggplot2 3.4.4), Arguments, width, height, units. R package, CRAN. https://CRAN.R-project.org/package=ggplot2 (accessed 2026-10-02).

312. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. ch. 1, section 1.6 "Saving your plots": give the size. https://r4ds.hadley.nz/ (accessed 2026-10-02).

313. Tidyverse team. Changelogs of ggplot2, dplyr, readr and tidyr: entries after ggplot2 3.4.4, dplyr 1.1.4, readr 2.1.5 and tidyr 1.3.1. 2026. ggplot2 3.5.0, Bug fixes: ggsave() and directories. tidyverse.org. https://ggplot2.tidyverse.org/news/ (accessed 2026-10-02).

314. Tidyverse team. Changelogs of ggplot2, dplyr, readr and tidyr: entries after ggplot2 3.4.4, dplyr 1.1.4, readr 2.1.5 and tidyr 1.3.1. 2026. ggplot2 4.0.0, Breaking changes: default bin boundary. tidyverse.org. https://ggplot2.tidyverse.org/news/ (accessed 2026-10-02).

315. Tidyverse team. Changelogs of ggplot2, dplyr, readr and tidyr: entries after ggplot2 3.4.4, dplyr 1.1.4, readr 2.1.5 and tidyr 1.3.1. 2026. ggplot2 4.0.1, bug fix: stat_bin(boundary) was ignored. tidyverse.org. https://ggplot2.tidyverse.org/news/ (accessed 2026-10-02).

316. Tidyverse team. Changelogs of ggplot2, dplyr, readr and tidyr: entries after ggplot2 3.4.4, dplyr 1.1.4, readr 2.1.5 and tidyr 1.3.1. 2026. ggplot2 4.0.3: geom_boxplot() gains quantile.type, default 7. tidyverse.org. https://ggplot2.tidyverse.org/news/ (accessed 2026-10-02).

317. Sandve, Geir Kjetil and Nekrutenko, Anton and Taylor, James and Hovig, Eivind. Ten simple rules for reproducible computational research. 2013. Rule 7 "Always Store Raw Data behind Plots", on histograms. https://pmc.ncbi.nlm.nih.gov/articles/PMC3812051/ (accessed 2026-10-02).

318. Sandve, Geir Kjetil and Nekrutenko, Anton and Taylor, James and Hovig, Eivind. Ten simple rules for reproducible computational research. 2013. Rule 7 "Always Store Raw Data behind Plots", on keeping the plotting code. https://pmc.ncbi.nlm.nih.gov/articles/PMC3812051/ (accessed 2026-10-02).

319. R Core Team. R 4.3.3, package base: help pages. 2024. help page "stopifnot" (base 4.3.3), Description. R Foundation for Statistical Computing. https://www.R-project.org/ (accessed 2026-10-02).

320. R Core Team. R 4.3.3, package base: help pages. 2024. help page "stopifnot" (base 4.3.3), Details: evaluation stops at the first failure. R Foundation for Statistical Computing. https://www.R-project.org/ (accessed 2026-10-02).

321. R Core Team. R 4.3.3, package base: help pages. 2024. help page "stopifnot" (base 4.3.3), Details: the conceptual equivalent, in which a missing value fails. R Foundation for Statistical Computing. https://www.R-project.org/ (accessed 2026-10-02).

322. R Core Team. R 4.3.3, package base: help pages. 2024. help page "stopifnot" (base 4.3.3), Arguments, ...: names replace the message. R Foundation for Statistical Computing. https://www.R-project.org/ (accessed 2026-10-02).

323. R Core Team. R 4.3.3, package base: help pages. 2024. help page "stopifnot" (base 4.3.3), Value. R Foundation for Statistical Computing. https://www.R-project.org/ (accessed 2026-10-02).

324. Wickham, Hadley and François, Romain and Henry, Lionel and Müller, Kirill and Vaughan, Davis. dplyr: A Grammar of Data Manipulation, version 1.1.4: help pages. 2023. help page "mutate-joins" (dplyr 1.1.4), Arguments, relationship: the default warns only on many-to-many. R package, CRAN. https://CRAN.R-project.org/package=dplyr (accessed 2026-10-02).

325. Wilson, Greg and Bryan, Jennifer and Cranston, Karen and Kitzes, Justin and Nederbragt, Lex and Teal, Tracy K. Good enough practices in scientific computing. 2017. Data management, 1a “Save the raw data”. https://pmc.ncbi.nlm.nih.gov/articles/PMC5480810/ (accessed 2026-10-02).

326. Wilson, Greg and Bryan, Jennifer and Cranston, Karen and Kitzes, Justin and Nederbragt, Lex and Teal, Tracy K. Good enough practices in scientific computing. 2017. Data management, 1e “Record all the steps used to process data”. https://pmc.ncbi.nlm.nih.gov/articles/PMC5480810/ (accessed 2026-10-02).

327. Wilson, Greg and Bryan, Jennifer and Cranston, Karen and Kitzes, Justin and Nederbragt, Lex and Teal, Tracy K. Good enough practices in scientific computing. 2017. Data management, 1f “use a unique identifier for every record”. https://pmc.ncbi.nlm.nih.gov/articles/PMC5480810/ (accessed 2026-10-02).

328. Sandve, Geir Kjetil and Nekrutenko, Anton and Taylor, James and Hovig, Eivind. Ten simple rules for reproducible computational research. 2013. Rule 2 “Avoid Manual Data Manipulation Steps”, first paragraph. https://pmc.ncbi.nlm.nih.gov/articles/PMC3812051/ (accessed 2026-10-02).

329. Sandve, Geir Kjetil and Nekrutenko, Anton and Taylor, James and Hovig, Eivind. Ten simple rules for reproducible computational research. 2013. Rule 5 “Record All Intermediate Results”, first paragraph: looking at intermediate results. https://pmc.ncbi.nlm.nih.gov/articles/PMC3812051/ (accessed 2026-10-02).

330. The Carpentries. Data Organization in Spreadsheets for Ecologists. 2026. Episode 04 “Quality Assurance”, first paragraph. Data Carpentry lesson (spreadsheet-ecology-lesson). https://datacarpentry.github.io/spreadsheet-ecology-lesson/ (accessed 2026-10-02).

331. Peng, Roger D. Reproducible research in computational science. 2011. Body, second paragraph: replication as the ultimate standard. https://pmc.ncbi.nlm.nih.gov/articles/PMC3383002/ (accessed 2026-10-02).

332. Peng, Roger D. Reproducible research in computational science. 2011. Body, third paragraph: what reproducibility requires, and how it falls short of replication. https://pmc.ncbi.nlm.nih.gov/articles/PMC3383002/ (accessed 2026-10-02).

333. Peng, Roger D. Reproducible research in computational science. 2011. Body, third paragraph: the spectrum (Fig. 1). https://pmc.ncbi.nlm.nih.gov/articles/PMC3383002/ (accessed 2026-10-02).

334. Peng, Roger D. Reproducible research in computational science. 2011. Body, seventh paragraph: reproducible is not correct. https://pmc.ncbi.nlm.nih.gov/articles/PMC3383002/ (accessed 2026-10-02).

335. Trisovic, Ana and Lau, Matthew K and Pasquier, Thomas and Crosas, Mercè. A large-scale study on research code quality and execution. PMID 35190569, PMC8861064. Open access under CC BY 4.0. Quoted from the PMC open-access text. Implementation and Methods, first paragraph: the sample. https://pmc.ncbi.nlm.nih.gov/articles/PMC8861064/ (accessed 2026-10-02).

336. Trisovic, Ana and Lau, Matthew K and Pasquier, Thomas and Crosas, Mercè. A large-scale study on research code quality and execution. PMID 35190569, PMC8861064. Open access under CC BY 4.0. Quoted from the PMC open-access text. Background, second paragraph: what a replication package is. https://pmc.ncbi.nlm.nih.gov/articles/PMC8861064/ (accessed 2026-10-02).

337. Trisovic, Ana and Lau, Matthew K and Pasquier, Thomas and Crosas, Mercè. A large-scale study on research code quality and execution. PMID 35190569, PMC8861064. Open access under CC BY 4.0. Quoted from the PMC open-access text. Results, RQ 4: three R versions, clean environment. https://pmc.ncbi.nlm.nih.gov/articles/PMC8861064/ (accessed 2026-10-02).

338. Trisovic, Ana and Lau, Matthew K and Pasquier, Thomas and Crosas, Mercè. A large-scale study on research code quality and execution. PMID 35190569, PMC8861064. Open access under CC BY 4.0. Quoted from the PMC open-access text. Results, RQ 4: the time allowed. https://pmc.ncbi.nlm.nih.gov/articles/PMC8861064/ (accessed 2026-10-02).

339. Trisovic, Ana and Lau, Matthew K and Pasquier, Thomas and Crosas, Mercè. A large-scale study on research code quality and execution. PMID 35190569, PMC8861064. Open access under CC BY 4.0. Quoted from the PMC open-access text. Results, RQ 4: time-limit results excluded. https://pmc.ncbi.nlm.nih.gov/articles/PMC8861064/ (accessed 2026-10-02).

340. Trisovic, Ana and Lau, Matthew K and Pasquier, Thomas and Crosas, Mercè. A large-scale study on research code quality and execution. PMID 35190569, PMC8861064. Open access under CC BY 4.0. Quoted from the PMC open-access text. Results, RQ 4, combining the three versions: success if any version succeeds. https://pmc.ncbi.nlm.nih.gov/articles/PMC8861064/ (accessed 2026-10-02).

341. Trisovic, Ana and Lau, Matthew K and Pasquier, Thomas and Crosas, Mercè. A large-scale study on research code quality and execution. PMID 35190569, PMC8861064. Open access under CC BY 4.0. Quoted from the PMC open-access text. Implementation and Methods, Code cleaning: what the cleaning did. https://pmc.ncbi.nlm.nih.gov/articles/PMC8861064/ (accessed 2026-10-02).

342. Trisovic, Ana and Lau, Matthew K and Pasquier, Thomas and Crosas, Mercè. A large-scale study on research code quality and execution. PMID 35190569, PMC8861064. Open access under CC BY 4.0. Quoted from the PMC open-access text. Abstract: the two failure figures. https://pmc.ncbi.nlm.nih.gov/articles/PMC8861064/ (accessed 2026-10-02).

343. Trisovic, Ana and Lau, Matthew K and Pasquier, Thomas and Crosas, Mercè. A large-scale study on research code quality and execution. PMID 35190569, PMC8861064. Open access under CC BY 4.0. Quoted from the PMC open-access text. Results, RQ 4, results table, row “Success rate” (columns: without code cleaning, with code cleaning, best of both). https://pmc.ncbi.nlm.nih.gov/articles/PMC8861064/ (accessed 2026-10-02).

344. Trisovic, Ana and Lau, Matthew K and Pasquier, Thomas and Crosas, Mercè. A large-scale study on research code quality and execution. PMID 35190569, PMC8861064. Open access under CC BY 4.0. Quoted from the PMC open-access text. Implementation and Methods, Data collection workflow: cleaning aimed at common errors. https://pmc.ncbi.nlm.nih.gov/articles/PMC8861064/ (accessed 2026-10-02).

345. Trisovic, Ana and Lau, Matthew K and Pasquier, Thomas and Crosas, Mercè. A large-scale study on research code quality and execution. PMID 35190569, PMC8861064. Open access under CC BY 4.0. Quoted from the PMC open-access text. Results, RQ 5: cleaning fixed every setwd error. https://pmc.ncbi.nlm.nih.gov/articles/PMC8861064/ (accessed 2026-10-02).

346. Trisovic, Ana and Lau, Matthew K and Pasquier, Thomas and Crosas, Mercè. A large-scale study on research code quality and execution. PMID 35190569, PMC8861064. Open access under CC BY 4.0. Quoted from the PMC open-access text. Results, RQ 5: library errors. https://pmc.ncbi.nlm.nih.gov/articles/PMC8861064/ (accessed 2026-10-02).

347. Trisovic, Ana and Lau, Matthew K and Pasquier, Thomas and Crosas, Mercè. A large-scale study on research code quality and execution. PMID 35190569, PMC8861064. Open access under CC BY 4.0. Quoted from the PMC open-access text. Results, RQ 5: errors left after cleaning, versions. https://pmc.ncbi.nlm.nih.gov/articles/PMC8861064/ (accessed 2026-10-02).

348. Trisovic, Ana and Lau, Matthew K and Pasquier, Thomas and Crosas, Mercè. A large-scale study on research code quality and execution. PMID 35190569, PMC8861064. Open access under CC BY 4.0. Quoted from the PMC open-access text. Results, RQ 5: errors left after cleaning, paths and missing objects. https://pmc.ncbi.nlm.nih.gov/articles/PMC8861064/ (accessed 2026-10-02).

349. Trisovic, Ana and Lau, Matthew K and Pasquier, Thomas and Crosas, Mercè. A large-scale study on research code quality and execution. PMID 35190569, PMC8861064. Open access under CC BY 4.0. Quoted from the PMC open-access text. Limitations of the Study, first paragraph: re-execution necessary, not sufficient. https://pmc.ncbi.nlm.nih.gov/articles/PMC8861064/ (accessed 2026-10-02).

350. Trisovic, Ana and Lau, Matthew K and Pasquier, Thomas and Crosas, Mercè. A large-scale study on research code quality and execution. PMID 35190569, PMC8861064. Open access under CC BY 4.0. Quoted from the PMC open-access text. Results, RQ 7: journal policy and re-execution. https://pmc.ncbi.nlm.nih.gov/articles/PMC8861064/ (accessed 2026-10-02).

351. Trisovic, Ana and Lau, Matthew K and Pasquier, Thomas and Crosas, Mercè. A large-scale study on research code quality and execution. PMID 35190569, PMC8861064. Open access under CC BY 4.0. Quoted from the PMC open-access text. Results, RQ 9: most datasets labelled social science. https://pmc.ncbi.nlm.nih.gov/articles/PMC8861064/ (accessed 2026-10-02).

352. Trisovic, Ana and Lau, Matthew K and Pasquier, Thomas and Crosas, Mercè. A large-scale study on research code quality and execution. PMID 35190569, PMC8861064. Open access under CC BY 4.0. Quoted from the PMC open-access text. Limitations of the Study, last paragraph: one source. https://pmc.ncbi.nlm.nih.gov/articles/PMC8861064/ (accessed 2026-10-02).

353. Trisovic, Ana and Lau, Matthew K and Pasquier, Thomas and Crosas, Mercè. A large-scale study on research code quality and execution. PMID 35190569, PMC8861064. Open access under CC BY 4.0. Quoted from the PMC open-access text. figures quoted in the text. https://pmc.ncbi.nlm.nih.gov/articles/PMC8861064/ (accessed 2026-10-02).

354. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. Ch. 6 "Workflow: scripts and projects", 6.2.1 "What is the source of truth?", first paragraph. https://r4ds.hadley.nz/ (accessed 2026-10-02).

355. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. Ch. 6, 6.2.1, the pair of keyboard shortcuts: restart R, then re-run the script. https://r4ds.hadley.nz/ (accessed 2026-10-02).

356. Wickham, Hadley and Çetinkaya-Rundel, Mine and Grolemund, Garrett. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 2023. Ch. 6, 6.2.1, last paragraph: restarting regularly for a clean slate. https://r4ds.hadley.nz/ (accessed 2026-10-02).

357. R Core Team. R 4.3.3, package utils: help pages. 2024. Help page "Rscript" (utils 4.3.3), Details: the options always supplied. R Foundation for Statistical Computing. https://www.R-project.org/ (accessed 2026-10-02).

358. R Core Team. R 4.3.3, package base: help pages. 2024. Help page "source" (base 4.3.3), Arguments: local. R Foundation for Statistical Computing. https://www.R-project.org/ (accessed 2026-10-02).

359. R Core Team. R 4.3.3, package base: help pages. 2024. Help page "Random" (base 4.3.3), Description: set.seed. R Foundation for Statistical Computing. https://www.R-project.org/ (accessed 2026-10-02).

360. R Core Team. R 4.3.3, package base: help pages. 2024. Help page "Random" (base 4.3.3), Examples: the same seed gives the same numbers. R Foundation for Statistical Computing. https://www.R-project.org/ (accessed 2026-10-02).

361. R Core Team. R 4.3.3, package base: help pages. 2024. Help page "Random" (base 4.3.3), Details: sample.kind, the default changed in R 3.6.0. R Foundation for Statistical Computing. https://www.R-project.org/ (accessed 2026-10-02).

362. R Core Team. R 4.3.3, package utils: help pages. 2024. Help page "sessionInfo" (utils 4.3.3), Description. R Foundation for Statistical Computing. https://www.R-project.org/ (accessed 2026-10-02).

363. Sandve, Geir Kjetil and Nekrutenko, Anton and Taylor, James and Hovig, Eivind. Ten simple rules for reproducible computational research. 2013. Rule 6: For Analyses That Include Randomness, Note Underlying Random Seeds. https://pmc.ncbi.nlm.nih.gov/articles/PMC3812051/ (accessed 2026-10-02).

364. Sandve, Geir Kjetil and Nekrutenko, Anton and Taylor, James and Hovig, Eivind. Ten simple rules for reproducible computational research. 2013. Rule 3: Archive the Exact Versions of All External Programs Used, last sentence. https://pmc.ncbi.nlm.nih.gov/articles/PMC3812051/ (accessed 2026-10-02).

365. Sandve, Geir Kjetil and Nekrutenko, Anton and Taylor, James and Hovig, Eivind. Ten simple rules for reproducible computational research. 2013. Rule 1: For Every Result, Keep Track of How It Was Produced, second paragraph. https://pmc.ncbi.nlm.nih.gov/articles/PMC3812051/ (accessed 2026-10-02).

366. Sandve, Geir Kjetil and Nekrutenko, Anton and Taylor, James and Hovig, Eivind. Ten simple rules for reproducible computational research. 2013. Rule 9: Connect Textual Statements to Underlying Results, last paragraph. https://pmc.ncbi.nlm.nih.gov/articles/PMC3812051/ (accessed 2026-10-02).

367. Trisovic, Ana and Lau, Matthew K and Pasquier, Thomas and Crosas, Mercè. A large-scale study on research code quality and execution. PMID 35190569, PMC8861064. Open access under CC BY 4.0. Quoted from the PMC open-access text. Best Practices and Recommendations, first recommendation for researchers who use R. https://pmc.ncbi.nlm.nih.gov/articles/PMC8861064/ (accessed 2026-10-02).

368. Trisovic, Ana and Lau, Matthew K and Pasquier, Thomas and Crosas, Mercè. A large-scale study on research code quality and execution. PMID 35190569, PMC8861064. Open access under CC BY 4.0. Quoted from the PMC open-access text. Results, after Fig. 8: errors left after code cleaning. https://pmc.ncbi.nlm.nih.gov/articles/PMC8861064/ (accessed 2026-10-02).

369. Trisovic, Ana and Lau, Matthew K and Pasquier, Thomas and Crosas, Mercè. A large-scale study on research code quality and execution. PMID 35190569, PMC8861064. Open access under CC BY 4.0. Quoted from the PMC open-access text. Results, after Fig. 8: the “object not found” error. https://pmc.ncbi.nlm.nih.gov/articles/PMC8861064/ (accessed 2026-10-02).

370. Trisovic, Ana and Lau, Matthew K and Pasquier, Thomas and Crosas, Mercè. A large-scale study on research code quality and execution. PMID 35190569, PMC8861064. Open access under CC BY 4.0. Quoted from the PMC open-access text. Abstract: the share of R files that stopped with an error. https://pmc.ncbi.nlm.nih.gov/articles/PMC8861064/ (accessed 2026-10-02).

371. Wilson, Greg and Bryan, Jennifer and Cranston, Karen and Kitzes, Justin and Nederbragt, Lex and Teal, Tracy K. Good enough practices in scientific computing. 2017. Project organization, 4d: the controller or driver script. https://pmc.ncbi.nlm.nih.gov/articles/PMC5480810/ (accessed 2026-10-02).

372. Wilson, Greg and Bryan, Jennifer and Cranston, Karen and Kitzes, Justin and Nederbragt, Lex and Teal, Tracy K. Good enough practices in scientific computing. 2017. Keeping track of changes, second paragraph: version control. https://pmc.ncbi.nlm.nih.gov/articles/PMC5480810/ (accessed 2026-10-02).

373. Wilson, Greg and Bryan, Jennifer and Cranston, Karen and Kitzes, Justin and Nederbragt, Lex and Teal, Tracy K. Good enough practices in scientific computing. 2017. Data management, 1e “Record all the steps used to process data”, second paragraph. https://pmc.ncbi.nlm.nih.gov/articles/PMC5480810/ (accessed 2026-10-02).

374. Wilson, Greg and Bryan, Jennifer and Cranston, Karen and Kitzes, Justin and Nederbragt, Lex and Teal, Tracy K. Good enough practices in scientific computing. 2017. Project organization, 4d: a single controller script for a small project. https://pmc.ncbi.nlm.nih.gov/articles/PMC5480810/ (accessed 2026-10-02).

375. Wilson, Greg and Bryan, Jennifer and Cranston, Karen and Kitzes, Justin and Nederbragt, Lex and Teal, Tracy K. Good enough practices in scientific computing. 2017. Collaboration, 3a “Create an overview of your project”: the README. https://pmc.ncbi.nlm.nih.gov/articles/PMC5480810/ (accessed 2026-10-02).

376. Sandve, Geir Kjetil and Nekrutenko, Anton and Taylor, James and Hovig, Eivind. Ten simple rules for reproducible computational research. 2013. Rule 2: Avoid Manual Data Manipulation Steps, first sentence. https://pmc.ncbi.nlm.nih.gov/articles/PMC3812051/ (accessed 2026-10-02).

377. National Center for Health Statistics. National Health and Nutrition Examination Survey, August 2021–August 2023: Body Measures (BMX_L) and Demographic Variables and Sample Weights (DEMO_L), Data Files and Documentation. 2024. Block 3, BMX_L documentation, Analytic Notes: sample weights. NCHS. https://wwwn.cdc.gov/nchs/nhanes/search/datapage.aspx?Component=Examination&Cycle=2021-2023 (accessed 2026-10-02).

Appendix · Worked answers

1 · When data change silently: three documented failures

EXERCISE 1. Gene names turned into dates. Excel, with its default settings, turned names such as SEPT2 into dates. Ziemann and colleagues found this by checking published files with their own programs. They found errors in 704 of 3597 papers with Excel gene lists.

The Reinhart and Rogoff spreadsheet. A formula averaged rows 30 to 44, not 30 to 49. Some data were also left out without saying so. Herndon and colleagues found both when they rebuilt the results from the authors' own spreadsheet.

The missing COVID-19 results. In England, 15,841 positive test results did not reach the daily figures. Some files were too big for the system that loaded them. PHE found the problem on 2 October 2020 and said so two days later.

EXERCISE 2. Three things are lost.

The original values. Once a cell is overwritten, the value that was first recorded is gone. If a correction was itself wrong, there is nothing to go back to. Ziemann and colleagues found 166 files in which gene names, once turned into dates, could not be recovered from the file at all.

The record of what changed. Nobody can later see which cells were corrected, by whom, or why. In the Reinhart and Rogoff case, the replication could not start from the posted files. They did not record which data and methods had been used.

The chance to check. A reviewer cannot rerun a correction typed into a cell. They can only trust it.

The proposal: keep the file as it arrived from the data collection, and never save over it. Make every correction as a line of code. The code reads the raw file and fixes the value. A comment says why. The code then writes a new, cleaned file.

Count the rows before and after. The colleague's goal, a clean dataset, is still met. It is now made from the raw file every time the code runs.

EXERCISE 3. Check the reference as Book 0 taught: find the source, confirm it, find the passage, read it.

The number holds: PHE says 15,841 cases from 25 September to 2 October were not included in the reported daily figures.

"Deleted" does not hold. The cases were delayed, not deleted. PHE says everyone who tested positive got their result as normal, and the outstanding cases were passed to contact tracing.

"Excel" and "row limit" do not hold. The statement says some files "exceeded the maximum file size" of the process that loads them into central systems. It names no software and no row limit.

A supported version, in two sentences: "In England, 15,841 positive COVID-19 test results from 25 September to 2 October 2020 were left out of the reported daily figures. Public Health England gave the cause as files that exceeded the maximum size the loading system could take."

EXERCISE 4. Draw a column of gene names on the board: SEPT2, MARCH1, and a few others. Next to it, draw the same column as Excel showed it: 2-Sep, 1-Mar.

Ask: if you had typed the first column and later opened the file, would you notice? Most people say yes, for two rows. Then point out that the errors Ziemann and colleagues found sat in published papers, checked by authors, reviewers and editors. Being able to see every cell is not the same as looking at every cell.

Then make the real point. Size is not the issue. The issue is whether a second person can see what was done. If every change is a line of code, the change is there to read. If it was typed into a cell, it is invisible.

End with the rule they can use tomorrow: keep the raw file as it came, never save over it, and put every change in code.

2 · Installing R and an editor; the console and the script

EXERCISE 1. The console shows > when it is waiting, and + when the line you entered is unfinished, for example a bracket left open.

A line you want to keep goes in the script, a saved file ending in `.R`. The console runs a line once and keeps only a history of everything typed. The script keeps the lines you chose, with comments, and runs again from the top to give the same answers.

EXERCISE 2. One possible script, saved as `bmi_practice.R`:

```
# BMI of a made-up adult, worked from the units the register uses

# Height in metres: recorded in cm, and BMI needs m
152 / 100

# BMI in kg/m^2: weight in kg divided by height in m, squared
58 / (152 / 100)^2
```

```
[1] 1.52
[1] 25.10388
```

The height is 1.52 m. The BMI is about 25.1 kg/m^2.

Check your script against three things. The file name has no spaces. Each calculation has a comment saying why. Running every line from the top gives the same two answers each time.

EXERCISE 3. Draw two boxes. Label one “console” and one “script”.

In the console box, write three lines as the resident might have typed them: `72 / 160^2`, `72 / 1.6 * 1.6` and `72 / 1.6^2`. Ask which one their thesis used. They cannot tell from the box. Neither could their guide, an examiner, or they themselves in six months.

In the script box, write the one right line with a comment above it: why 160 was divided by 100.

Then make the point. The console runs a line once, prints the answer and moves on. Its history holds every attempt in the order typed, wrong ones included. The script holds only the version you chose, with the reasons, and runs again from the top whenever anyone asks.

Speed is not lost. They can still try ideas in the console. What changes is where the answer they will defend lives.

1.

```
45 + 17 * 2
```

```
[1] 79
```

The multiplication goes first: 17 times 2 is 34, and 45 plus 34 is 79. If you expected 124, you added first. R follows Book 0's order of operations.

2.

```
150 / (20 + 30)
```

```
[1] 3
```

The bracket makes R add first, giving 50, and 150 divided by 50 is 3. Without the bracket, `150 / 20 + 30` would divide first and give 37.5.

3.

```
2^3 * 2
2^(3 * 2)
```

```
[1] 16
[1] 64
```

In the first line the power goes first: 2 to the power 3 is 8, and 8 times 2 is 16. In the second line the bracket goes first: 3 times 2 is 6, and 2 to the power 6 is 64.

4.

```
# Body mass of the first penguin: the file records grams, 1000 g in a kilogram
3750 / 1000
```

```
[1] 3.75
```

The first penguin weighed 3.75 kg. The comment keeps the unit and the reason for dividing by 1000 next to the number.

5.

```
# Height in metres: the register gives cm
172 / 100

# BMI in kg/m^2: weight in kg divided by height in m, squared
81 / (172 / 100)^2
```

```
[1] 1.72
[1] 27.37966
```

The height is 1.72 m and the BMI is 27.37966, about 27.4 kg/m^2. R prints more digits than the measurements carry. The rounding to one decimal place is for the report; the script keeps the full value.

6.

```
# Average flipper length of the first three penguins: mm to cm is divided by 10
(181 + 186 + 195) / 3 / 10
```

```
[1] 18.73333
```

The bracket adds the three lengths first, 562 mm. Divided by 3 that is 187.3 mm, and divided by 10 it is about 18.7 cm. The order of the two divisions does not matter here. The bracket does.

7. R divided before it added. The line worked out 71.8 divided by 2, which is 35.9, and then added 71.4. The sum was never divided.

```
(71.4 + 71.8) / 2
```

```
[1] 71.6
```

The average is 71.6 kg.

Why the wrong answer looks reasonable: 107.3 kg is a weight a real adult can have, so it passes a glance. The check that catches it is the one Book 0 taught for averages. An average of two numbers lies between them. 107.3 is not between 71.4 and 71.8.

8. The third line is meant to be a comment, but it has no #. R tries to read "BMI of patient 2" as an instruction. It finds the word "of" straight after "BMI" with nothing joining them, and stops. That is what "unexpected symbol" means: a word where R did not expect one. The `3:5` says where: line 3, fifth character, which is where "of" begins.

The book's output runs nothing from a block that R cannot read, so it shows the error alone. In the RStudio console, run line by line, line 2 prints `[1] 1.65` first, and the message is worded a little differently.

```
# Height in metres
165 / 100
# BMI of patient 2
68 / (165 / 100)^2
```

```
[1] 1.65
[1] 24.97704
```

With the # in place, the line is skipped and the BMI is about 25.0 kg/m^2. The error message showed exactly where to look: it quoted the words "BMI of" and pointed at the second one.

9.

```
# Mean of the three pilot weights, in kg
(64.2 + 71.5 + 69.1) / 3
```

```
[1] 68.26667
```

The mean is 68.26667 kg, which rounds to 68.3 kg. The sentence matches the register.

What the check does not establish:

- That the weights in the register were measured or copied correctly. You checked the arithmetic on the numbers you were given, nothing before that.
- That three participants tell you anything about anyone else. A mean of three is a fact about those three.
- That the thesis can be checked by someone else. Your line can, if you keep it in a script. The thesis's own number came from wherever its author worked it out.

10.

```
# Mean body mass of the first three penguins, converted from g to kg
(3750 + 3800 + 3250) / 3 / 1000
```

```
[1] 3.6
```

The arithmetic holds: the mean of those three is 3.6 kg.

What it does not establish:

- Anything about "the average penguin". These are the first three rows of a file of 344 rows. Rows come in the order they were entered, not chosen to stand for the rest.
- Anything about penguins of other species. The first rows of this file are all one species, and the file holds three.
- The mean of the whole file. That needs every row, read from the file by code, which later sections teach.

3 · Objects and names

EXERCISE 1. `bmi` still holds the number worked out from the old weight. The name stores a value, not the calculation. To bring it up to date, run the BMI line again. Better still, correct the weight in the script, then run the whole script from the top.

An R name may not start with a digit, and may not contain a space. It may also not start with `_`.

EXERCISE 2.

```
weight_kg <- 66
height_cm <- 158
# BMI needs height in metres; the register records centimetres
height_m <- height_cm / 100
bmi <- weight_kg / height_m^2
bmi
```

```
[1] 26.43807
```

The answer is the same as the classmate's, about 26.4 kg/m^2. Running their four lines gives 26.43807 too. What changed is that each line can now be read on its own. The conversion from centimetres to metres has a name and a reason. The capital letters are gone, and `x` no longer hides what it holds.

EXERCISE 3. Write three lines on the board, as they would be in a script:

`height_m <- 1.52`, then `bmi <- weight_kg / height_m^2`, then `bmi`.

Underneath, write the line they typed in the console: `height_m <- 1.62`.

Ask: what is `bmi` now? Most residents say it uses 1.62. It does not. `bmi` was worked out when its line ran, from the height at that moment, and stored as one number. The console line changed the height and nothing else.

Then ask: what happens tomorrow, when the script runs from the top? The console line is not in the script. The height goes back to 1.52, and so does every BMI made from it.

The fix they can use today: put the correction in the script, with a comment saying where the right value came from. Then run the whole script from the top and look at the new result.

1.

```
a <- 5
a <- a + 2
a
```

```
[1] 7
```

The second line works out the right-hand side first, 5 plus 2, and then stores 7 under `a`, replacing the 5.

2. The three that work make an object without a message:

```
bmi_2024 <- 1
bmi.2024 <- 1
.bmi <- 1
```

The other three stop with an error:

```
2024_bmi <- 1
```

```
Error: <text>:1:5: unexpected input
1: 2024_
        ^
```

```
bmi 2024 <- 1
```

```
Error: <text>:1:5: unexpected numeric constant
1: bmi 2024
        ^
```

```
_bmi <- 1
```

```
Error: <text>:1:2: unexpected symbol
1: _bmi
     ^
```

A name must start with a letter or a dot, so `2024_bmi` and `_bmi` fail. A name cannot hold a space, so `bmi 2024` fails. Of the three that work, only `bmi_2024` follows this book's style.

3.

```
x <- 10
y <- x * 2
x <- 3
y
```

```
[1] 20
```

`y` was worked out when its line ran, from `x` at that moment, which was 10. Giving `x` a new value afterwards does not reach back into `y`.

4.

```
body_mass_g <- 3750
body_mass_kg <- body_mass_g / 1000
body_mass_kg
```

```
[1] 3.75
```

The penguin weighed 3.75 kg. Both names say their unit, so the division by 1000 reads as a conversion from grams to kilograms.

5.

```
weight_kg <- 81
height_cm <- 172
height_m <- height_cm / 100
bmi <- weight_kg / height_m^2
bmi
```

```
[1] 27.37966
```

The BMI is about 27.4 kg/m^2. The height is stored once as recorded, in centimetres, and the conversion is a line of its own.

6.

```
body_mass_2_g <- 3800
body_mass_3_g <- 3250
difference_kg <- (body_mass_2_g - body_mass_3_g) / 1000
difference_kg
```

```
[1] 0.55
```

The second penguin weighed 0.55 kg more than the third. The bracket makes R subtract before it divides, and each name says which penguin and which unit.

7. The second line reused the name `bmi`. Assigning to it replaced patient A's value with patient B's, without any warning. The last line printed patient B's BMI.

```
bmi_a <- 72 / 1.6^2
bmi_b <- 58 / 1.52^2
bmi_a
bmi_b
```

```
[1] 28.125
[1] 25.10388
```

Patient A's BMI is about 28.1 kg/m^2. Patient B's is about 25.1 kg/m^2.

Why the wrong answer looks reasonable: 25.1 kg/m^2 is an ordinary adult BMI, so nothing about it looks off. The comments even say "patient A". Comments describe what the writer meant, and R does not read them. Separate names would have kept the two values apart.

8. The object was made as `height_m`, with a small h. The third line asks for `Height_m`, with a capital H. R treats them as two different names, finds no object called `Height_m`, and stops. The error printed under the prompt's code says so: object 'Height_m' not found.

```
weight_kg / height_m^2
```

```
[1] 22.86237
```

With the name spelt as it was made, the BMI is about 22.9 kg/m^2.

9.

```
height_m <- 1.58
# Participant 12: register says 47 kg; the paper form says 74 kg (digits swapped)
weight_recorded_kg <- 47
weight_corrected_kg <- 74
bmi_recorded <- weight_recorded_kg / height_m^2
bmi_corrected <- weight_corrected_kg / height_m^2
bmi_recorded
bmi_corrected
```

```
[1] 18.82711
[1] 29.64269
```

The BMI moves from about 18.8 to about 29.6 kg/m^2. Both values stay in the script under their own names, with a comment saying where the corrected value came from.

What these lines do not establish:

- That 74 kg is right. The lines record the correction; they do not check it. Only the paper form, or a second measurement, can.
- That no other participant has the same kind of error. One swapped pair of digits found is a reason to check the rest, not a sign that the rest are clean.
- Anything about the thesis's own results. If the correction was typed into a console and not into the script, the next run from the top will lose it.

10. The sentence says heights were in centimetres, so the lines divide by 100:

```
weight_kg <- 64
height_cm <- 1.63    # as written in the register
height_m <- height_cm / 100
bmi <- weight_kg / height_m^2
bmi
```

```
[1] 240882.2
```

The code runs without a complaint and gives a BMI in the hundreds of thousands. Book 0's order-of-magnitude check says no adult has that. The height of 1.63 was written in metres, not centimetres, and dividing it by 100 made it 1.63 cm.

The name `height_cm` did not protect you. A name says what you believe the value is. R does not check it against the value.

After checking the paper form, record what you found and use it:

```
weight_kg <- 64
# Register shows 1.63; the form confirms metres (163 cm), unlike the other entries
height_cm <- 163
height_m <- height_cm / 100
bmi <- weight_kg / height_m^2
bmi
```

```
[1] 24.08822
```

The BMI is about 24.1 kg/m^2.

What the methods sentence does not establish:

- That every height was recorded in centimetres. It describes the plan, not the register.
- That anyone checked. A conversion applied to every row runs just as smoothly on a value recorded in the wrong unit.
- That the results are right. A check you set yourself, such as flagging any adult height outside 1 m to 2 m, would catch an entry like this one.

4 · Vectors, types and NA

EXERCISE 1. R did not give an error. It gave `NA` and a warning. The appendix misreports what happened, so start by printing the vector:

```
hb_g_dl
```

```
[1] "11.2" "12.5" "9.8*" "13.1" "10.4"
```

Every value is in quotation marks. The entry "9.8*" is text, so the whole vector became text, and `mean()` cannot average text.

The asterisk was probably a mark copied from the record, such as a note that the value was rechecked. Only the person with the record can say. The analysis should go on by writing the decision as code, with a comment giving the reason, so a reader can see it. For example, if the record shows 9.8 g/dL is the value:

```
# 9.8* in the register: asterisk marks a recheck; the value is 9.8 g/dL
hb_g_dl <- c(11.2, 12.5, 9.8, 13.1, 10.4)
mean(hb_g_dl)
```

```
[1] 11.4
```

The mean of the five values is 11.4 g/dL. Suppose instead the asterisk meant the value was rejected. Then the line would put `NA` there, and the report would give the mean of four women.

EXERCISE 2. Numeric (numbers), character (text) and logical (`TRUE` or `FALSE`). When they are mixed, everything becomes the highest type in the order logical, numeric, character, so one piece of text makes the whole vector text. The missing values are found with `is.na(x)`, never with `x == NA`.

1.

```
height_cm <- c(98, 102.5, 110)
height_cm / 100
```

```
[1] 0.980 1.025 1.100
```

Each height is divided by 100, one at a time.

2.

```
c(1, TRUE, "no")
c(0, TRUE, FALSE)
```

```
[1] "1"    "TRUE" "no"
[1] 0 1 0
```

The first holds text, so everything becomes text: the number 1 becomes "1" and `TRUE` becomes "TRUE". The second holds a number and two logical values. Numeric is higher than logical, so `TRUE` becomes 1 and `FALSE` becomes 0.

```
typeof(c(1, TRUE, "no"))
typeof(c(0, TRUE, FALSE))
```

```
[1] "character"
[1] "double"
```

"double" is R's name for its usual kind of number.

3.

```
c(2, NA, 5) + 1
c(2, NA, 5) > 3
```

```
[1]  3 NA  6
[1] FALSE    NA  TRUE
```

Adding 1 works on each element. The missing value stays missing, because 1 more than an unknown number is still unknown. The comparison gives `FALSE`, `NA`, `TRUE`: R cannot say whether an unknown number is more than 3.

4.

```
flipper_mm <- c(181, 186, 195, NA, 193, 190, 181, 195, 193, 190)
flipper_mm / 10
flipper_mm > 190
```

```
 [1] 18.1 18.6 19.5   NA 19.3 19.0 18.1 19.5 19.3 19.0
 [1] FALSE FALSE  TRUE    NA  TRUE FALSE FALSE  TRUE  TRUE FALSE
```

There are 10 mm in a centimetre, so dividing by 10 gives centimetres: penguin 1's flipper is 18.1 cm. Penguins 3, 5, 8 and 9 have flippers longer than 190 mm. Penguins 6 and 10 measure exactly 190 mm, which is not longer, so they get `FALSE`. Penguin 4 gets `NA`: its length is not in the file.

5.

```
body_mass_g <- c(3750, 3800, 3250, NA, 3450, 3650, 3625, 4675, 3475, 4250)
body_mass_g[8] / 1000
body_mass_g[1:3]
```

```
[1] 4.675
[1] 3750 3800 3250
```

The eighth penguin weighs 4.675 kg. Inside square brackets, `1:3` means positions 1, 2 and 3, so the second line gives the first three body masses, in grams.

6.

```
body_mass_g <- c(3750, 3800, 3250, NA, 3450, 3650, 3625, 4675, 3475, 4250)
known_kg <- body_mass_g[!is.na(body_mass_g)] / 1000
known_kg
length(known_kg)
```

```
[1] 3.750 3.800 3.250 3.450 3.650 3.625 4.675 3.475 4.250
[1] 9
```

`is.na()` marks the missing value, `!` turns that round to mark the known ones, and the square brackets keep them. Nine of the ten body masses are known.

7. The heights were typed in quotation marks, so they are text, not numbers. Text is compared character by character, so "98" comes after "102" and "110".

```
typeof(height_cm)
height_cm <- c(98, 102, 110)
max(height_cm)
```

```
[1] "character"
[1] 110
```

The tallest child is 110 cm. The wrong answer looks reasonable because 98 cm is a perfectly possible height, and nothing in the output says an error happened. The quotation marks around "98" in the output are the only clue.

8. `bmi == NA` asks whether each value equals an unknown value. The answer is always unknown, so every element is `NA`. The colleague read "no `TRUE`" as "nothing missing". The test that can answer is `is.na()`:

```
is.na(bmi)
bmi[!is.na(bmi)]
```

```
[1] FALSE FALSE  TRUE FALSE  TRUE FALSE
[1] 22.4 31.0 27.9 19.6
```

The third and fifth values are missing, so four of the six BMI values are known. The wrong check looks reasonable because it runs without an error and prints something for every value.

9. The missing BMI was typed as "NA" inside quotation marks. That is the text N, A, not R's missing value. One text value made the whole vector text, so `>=` turned 25 into the text "25" and compared character by character. In this book's setup that puts "NA" above "25".

```
typeof(bmi)
bmi <- c(22.1, 27.4, NA, 30.2, 25.8)
bmi >= 25
```

```
[1] "character"
[1] FALSE  TRUE    NA  TRUE  TRUE
```

Now the third adult gets `NA`. Three of the four adults with a recorded BMI have a BMI of 25 or more, and one adult is unknown. The wrong answer looks reasonable because three of its five answers are right, and the wrong one says `TRUE` with no warning.

10.

```
body_mass_g <- c(3750, 3800, 3250, NA, 3450, 3650, 3625, 4675, 3475, 4250)
is.na(body_mass_g)
length(body_mass_g[is.na(body_mass_g)])
```

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

One of the ten is missing, the fourth, so the sentence is wrong. A corrected sentence: "Body mass was recorded for 9 of the first 10 rows."

What the check does not tell you:

- Why the fourth body mass is missing. The file records only that it is not there.
- Anything about the other 334 rows. Across the whole column, the file has 2 missing body masses.
- Whether the nine recorded values are correct. A value can be present and wrong.

11. A first try picks the body masses above 4000 g:

```
body_mass_g <- c(3750, 3800, 3250, NA, 3450, 3650, 3625, 4675, 3475, 4250)
body_mass_g[body_mass_g > 4000]
```

```
[1]   NA 4675 4250
```

Three values come back, and one is `NA`, the comparison's answer for penguin 4. So count only the known values that pass the test. The symbol `&` means "and": both conditions must be `TRUE`.

```
heavy_g <- body_mass_g[!is.na(body_mass_g) & body_mass_g > 4000]
heavy_g
length(heavy_g)
```

```
[1] 4675 4250
[1] 2
```

Rows 8 and 10 have body masses above 4000 g. The claim holds for the nine rows whose body mass is known.

What it does not establish:

- That exactly two of the ten body masses are above 4000 g. Penguin 4's body mass is unknown, so the true count is two or three. "Two of the ten" hides that. "Two of the nine with a recorded body mass" does not.
- Anything about penguins in the rest of the file, or penguins in general.

5 · Functions and arguments

EXERCISE 1. Ask them to type `?round` and look at the page with you. Point to three parts in turn.

Description says, in one line, that `round` rounds "to the specified number of decimal places (default 0)".

Usage shows the call written out with its defaults: `round(x, digits = 0)`. Ask them what `digits = 0` means. It means that if they give no `digits`, R uses 0.

Arguments says what `x` and `digits` are. Ask them to predict `round(3.14159)` from that. No `digits` is given, so the default 0 applies, and the answer is a whole number, 3. Then let them run it:

```
round(3.14159)
round(3.14159, digits = 2)
```

```
[1] 3
[1] 3.14
```

Finish by asking them to run one line from the Examples section at the bottom of the page. The habit to leave them with: Usage for the defaults, Arguments for what each one does, Examples to try it.

EXERCISE 2. It removes the missing values before the mean is computed. The default is `na.rm = FALSE`, which gives `NA` whenever a value is missing. Because the mean is then of the values present, report how many there were, as in "3.77 kg (n = 9)".

1.

```
round(24.83702, digits = 1)
round(24.83702)
```

```
[1] 24.8
[1] 25
```

With no `digits`, `round()` uses its default, 0, and gives a whole number.

2.

```
mean(c(4, 8, NA, 6))
mean(c(4, 8, NA, 6), na.rm = TRUE)
```

```
[1] NA
[1] 6
```

The default gives `NA`. With `na.rm = TRUE`, R averages 4, 8 and 6, which sum to 18, and 18 divided by 3 is 6.

3.

```
round(mean(c(61.2, 58.4, 70.9)), digits = 1)

c(61.2, 58.4, 70.9) |>
  mean() |>
  round(digits = 1)
```

```
[1] 63.5
[1] 63.5
```

The innermost call, `mean()`, comes first in the pipe. Each pipe hands the result on as the first argument of the next call.

4.

```
flipper_mm <- c(181, 186, 195, NA, 193, 190, 181, 195, 193, 190)
flipper_cm <- flipper_mm / 10
round(mean(flipper_cm, na.rm = TRUE), digits = 1)
round(median(flipper_cm, na.rm = TRUE), digits = 1)
sum(!is.na(flipper_cm))
```

```
[1] 18.9
[1] 19
[1] 9
```

The mean is 18.9 cm and the median is 19 cm, each from the 9 rows whose flipper length is recorded. R prints 19 rather than 19.0: a single value is printed without a trailing zero.

5. The Usage section of `?sd` reads `sd(x, na.rm = FALSE)`, and Arguments says `na.rm` decides whether missing values are removed. The default is `FALSE`, so set it to `TRUE`:

```
body_mass_g <- c(3750, 3800, 3250, NA, 3450, 3650, 3625, 4675, 3475, 4250)
round(sd(body_mass_g, na.rm = TRUE))
```

```
[1] 440
```

The standard deviation of the 9 known body masses is 440 g. In Book 0, a standard deviation from a sample divides by one less than the count. The help page says `sd()` does the same: “Like ‘var’ this uses denominator n - 1.”

6.

```
body_mass_g <- c(3750, 3800, 3250, NA, 3450, 3650, 3625, 4675, 3475, 4250)
known_g <- body_mass_g[!is.na(body_mass_g)]
sum(known_g) / length(known_g)
mean(body_mass_g, na.rm = TRUE)
```

```
[1] 3769.444
[1] 3769.444
```

Both give 3769.444 g. Removing the missing value first and then dividing by the count of what is left is exactly what `na.rm = TRUE` does.

7. The three numbers are not inside `c()`, so they are three separate arguments. R matched them by position: 3750 became `x`, 3800 became `trim`, and 3250 went into `...`. A `trim` above 0.5 is taken as 0.5. `x` is the single value 3750, and any mean of one value, trimmed or not, is that value.

```
mean(c(3750, 3800, 3250))
```

```
[1] 3600
```

The mean is 3600 g. The wrong answer looks reasonable because 3750 g is one of the masses. It lies inside the range of the data, and R gave no warning.

8. R’s logical values are written in capitals, `TRUE` and `FALSE`. Written in small letters, `true` is read as the name of an object, and there is no object called `true`.

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

```
[1] 3562.5
```

The mean of the four known masses is 3562.5 g. The error message gave the answer: it names the word R could not find.

9. The masses were rounded to whole kilograms before the mean was taken. Each value lost up to half a kilogram first, and only then were the values averaged. Look at what went into the mean:

```
round(body_mass_g / 1000)
```

```
 [1]  4  4  3 NA  3  4  4  5  3  4
```

Round once, at the end:

```
(body_mass_g / 1000) |>
  mean(na.rm = TRUE) |>
  round(digits = 2)
```

```
[1] 3.77
```

The mean of the 9 known masses is 3.77 kg. The wrong answer looks reasonable because it has the digits asked for and is only 0.01 kg away. It came from whole numbers, though, so its second decimal place means nothing. This is Book 0's rule again: keep full precision while you work and round once, at the end.

10. 3392.5 is lower than every one of the nine known masses except 3250 g, which is a sign that something pulled it down. Try dividing the total of the known masses by ten instead of nine:

```
body_mass_g <- c(3750, 3800, 3250, NA, 3450, 3650, 3625, 4675, 3475, 4250)
known_g <- body_mass_g[!is.na(body_mass_g)]
sum(known_g) / 10
mean(body_mass_g, na.rm = TRUE)
```

```
[1] 3392.5
[1] 3769.444
```

The report's number is the total of nine masses divided by ten: the missing weight was treated as zero. A corrected sentence: "The mean body mass of the 9 rows with a recorded weight was 3769.4 g."

What the corrected sentence does not establish:

- The mean of all ten. Penguin 4's weight is unknown, and it could lie anywhere.
- Anything about the penguins in the rest of the file, or about penguins in general.

11.

```
body_mass_g <- c(3750, 3800, 3250, NA, 3450, 3650, 3625, 4675, 3475, 4250)
median(body_mass_g, na.rm = TRUE)
sum(!is.na(body_mass_g))
```

```
[1] 3650
[1] 9
```

The median, 3650 g, is right. The n is wrong: the median was computed from 9 values, not 10, because `na.rm = TRUE` removed the missing one. The table should read "3650 g (n = 9)". The writer probably took n from `length(body_mass_g)`, which counts the missing value too.

What the checked result does not establish:

- The median of all ten values. With a tenth value, the median would be the mean of the two middle values, and that depends on the missing weight.
- Anything about penguins beyond these ten rows.

6 · Packages: where they come from and which version you have

EXERCISE 1. One version:

```
# Packages: installed once per computer with install.packages(), loaded here
library(readr)
library(dplyr, warn.conflicts = FALSE)  # hides the masking message shown above

# Versions this script ran on
R.version.string
packageVersion("readr")
packageVersion("dplyr")
```

```
[1] "R version 4.3.3 (2024-02-29)"
[1] '2.1.5'
[1] '1.1.4'
```

`install.packages()` is not in the script because installing is done once per computer, in the console. In a script it would download the package on every run, need an internet connection, and quietly change the version for anyone who ran it. The `library()` lines at the top tell a reader at once which packages the script needs. The version lines record what it ran on, so a later difference can be traced to a version and not to the code.

EXERCISE 2. `install.packages()` once per computer, typed in the console. `library()` in every session that uses the package, at the top of the script.

1. In the console, once per computer:

```
install.packages("readr")
```

At the top of the script:

```
library(readr)
```

readr prints nothing when it loads. Silence means it worked.

2. dplyr's `filter()` runs, because dplyr was attached after stats and masks it. The other is called as `stats::filter()`. The line that shows the masking is the `library()` call itself:

```
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
```

The message lists `filter` among the objects masked from `package:stats`.

3.

```
R.version.string
packageVersion("here")
```

```
[1] "R version 4.3.3 (2024-02-29)"
[1] '1.0.1'
```

This book's outputs were made with these versions. Yours may be newer.

4.

```
readr::read_csv(here::here("data-raw", "penguins_raw.csv"), show_col_types = FALSE) |>
  nrow()
```

```
[1] 344
```

`readr::` and `here::` call each function from its package without `library()`. The file has 344 rows. One row is one adult penguin's record on one sampling expedition; the same bird can appear in more than one row.

5. `read_csv()` belongs to readr, and the script never loads readr. Yesterday the colleague had loaded it by hand in the console, so the session knew the function. Today's fresh session does not. Add the `library()` line at the top:

```
library(readr)
raw <- here::here("data-raw", "penguins_raw.csv")
penguins_raw <- read_csv(raw, show_col_types = FALSE)
nrow(penguins_raw)
```

```
[1] 344
```

The script now reads the file's 344 rows. A script must load everything it uses. Whatever happened in the console earlier does not count.

6. `packageVersion()` takes the package's name as text, in quotation marks. Without them, R looks for an object called `dplyr`, and there is none.

```
packageVersion("dplyr")
```

```
[1] '1.1.4'
```

The mistake is easy to make because `library(dplyr)` works without quotation marks, so the two calls look as though they should take the same form. They do not: `library()` is written to accept a bare name, and most functions are not.

7.

```
R.version.string
packageVersion("readr")
packageVersion("dplyr")
```

```
[1] "R version 4.3.3 (2024-02-29)"
[1] ‘2.1.5’
[1] ‘1.1.4’
```

A fuller sentence: “All analyses were carried out in R version 4.3.3 with readr 2.1.5 and dplyr 1.1.4.”

What the fuller record does not establish:

- That the analysis gives the same numbers on newer versions. It records what was used, not what would happen with something else.
- That these are the only packages involved. Packages that load other packages, and the operating system, are not listed. A complete record comes later, in the section on running a script from top to bottom.
- That the code is correct. Recorded versions make a wrong analysis repeatable, not right.

7 · A project folder, and paths that work on someone else’s computer

EXERCISE 1. The line is `heights <- read_csv(here("data-raw", "heights.csv"))`. Load readr and here first. `here()` builds the address from the project root. It finds that root by itself, on whatever computer runs the script.

`setwd()` with a folder name moves R to a folder that exists only on the computer of the person who wrote the line. On any other computer the script stops there.

EXERCISE 2. One layout. The names inside are a choice. The separation is not.

Path	What it holds	Written by
school_camp_2026/	the project folder, named after the analysis	you, once
README.txt	what the project is, where the file came from, how to rerun it	you
data-raw/camp_export.csv	the file exactly as the camp team sent it	nobody
code/analysis.R	the one script that reads, checks, cleans and summarises	you
output/	cleaned file, summary tables, figures	the script only

The code writes only to `output/`. Nothing, including you, edits `data-raw/camp_export.csv`. If the camp team sends a corrected file, it goes in `data-raw/` under a new name with the date it arrived. The README says which file the analysis uses.

The reading line:

```
camp <- read_csv(here("data-raw", "camp_export.csv"))
```

It needs `library(readr)` and `library(here)` above it. It has no address from your own computer in it, so it runs on your guide's laptop as it does on yours.

EXERCISE 3. Write her failing line on the board, with its address: `read_csv("C:/Users/meera/Desktop/data.csv")`. Ask her where `C:/Users/meera` is. It is a folder on her laptop. On yours it does not exist.

Draw the project folder as a box with three boxes inside: `data-raw`, `code`, `output`. Put the data file in `data-raw`. Show that "go into data-raw, take data.csv" is true on any computer that has this box, while "go to Meera's desktop" is true on one.

Then write the fix: `read_csv(here("data-raw", "data.csv"))`. Say that `here()` finds the top of the box by itself, wherever R started.

End with the rule she can check by eye: no line of the script should contain the name of anyone's computer, user folder or desktop.

1. +———+———-+————————————————————-+ | Address | Kind | Why | +———+———-+ ————————————————————-+ | a | absolute | starts from a drive letter, C: | +———+———-+ ————————————————————-+ | b | relative | starts from a folder name, read from the working directory | +———+———-+————————————————————-+ | c | absolute | starts from the top of the disk, / | +———+ ———-+————————————————————-+ | d | relative | starts from a folder name | +———+———-+ ————————————————————-+ | e | absolute | starts from a drive letter, D:, and has a space in a folder name as well | +———+———-+————————————————————-+

The test is the first character or two. A drive letter or a leading slash means absolute.

2.

```
here::here("output", "figures", "bmi_histogram.png")
```

```
[1] "~/project/output/figures/bmi_histogram.png"
```

Each argument becomes one level of the address, in order. Nothing is created:

```
dir.exists(here::here("output", "figures"))
```

```
[1] FALSE
```

`dir.exists()` asks whether a folder exists, and answers FALSE. To make the folder you would call `dir.create()` on that address.

3.

```
library(readr)
library(here)
penguins_raw <- read_csv(here("data-raw", "penguins_raw.csv"), show_col_types = FALSE)
dim(penguins_raw)
```

```
here() starts at ~/project
[1] 344  17
```

`dim()` gives rows first, then columns: 344 rows and 17 columns.

4. +———————————+———————————-+ | File | Where | +———————————+———————————-+ | a. penguins_raw.csv as downloaded | data-raw/ | +———————————+———————————-+ | b. analysis.R | code/ | +

—————————————+—————————————-+ | c. species_counts.csv | output/ | +—————————————+—————————————-+ | d. mass_histogram.png | output/ | +—————————————+—————————————-+ | e. README.txt | the top of the project folder | +—————————————+—————————————-+ | f. hand-corrected copy | nowhere | +—————————————+—————————————-+

File f has no place. A correction made by hand leaves no record of what changed. Write the correction as a line of code that reads the raw file and writes the corrected version to `output/`. Then delete the hand-made copy.

5.

```
library(here)
dir.create(here("output"))
dir.exists(here("output"))
list.files(here())
```

```
here() starts at ~/project
[1] TRUE
[1] "data-raw" "output"
```

The project now has a `data-raw/` folder and an `output/` folder. A script that writes its tables with `here("output", ...)` has somewhere to put them.

6. The second line broke. `setwd()` tried to move R's working directory to a folder on Priya's laptop, and that folder does not exist here.

It looked reasonable because on Priya's laptop it works. There, the line moves R to her thesis folder, and the relative path on the next line then finds the file. Every time she ran it, it worked, so she had no reason to doubt it.

The repair removes the line and builds the address from the project root:

```
library(readr)
library(here)
penguins_raw <- read_csv(here("data-raw", "penguins_raw.csv"), show_col_types = FALSE)
nrow(penguins_raw)
```

```
here() starts at ~/project
[1] 344
```

7. The `write_csv()` line broke the project. It wrote the 300-row test file over the raw file, in `data-raw/`. The first run read all 344 rows. The second run read the overwritten file, so it found 300. The other 44 rows are gone from the project, and no line of code can bring them back.

Nothing looked wrong on the first run because the first run was correct. The damage shows only on the next run, when the number of rows changes with no change to the code.

The repair writes the test file to `output/` under its own name, and leaves `data-raw/` alone. It is shown here without being run, because the raw file in this session is already damaged:

```
library(readr)
library(here)
penguins_raw <- read_csv(here("data-raw", "penguins_raw.csv"), show_col_types = FALSE)
penguins_test <- head(penguins_raw, 300)
write_csv(penguins_test, here("output", "penguins_test_300.csv"))
```

With this version the raw file keeps its 344 rows however many times the script runs. The only way back from the broken version is a copy of the raw file kept somewhere else, which is one reason to keep one.

8. The nearest test you can run alone is to move the project somewhere else and run it there. Copy the folders the script needs, and the marker, into a new temporary folder, and run the script from inside the copy:

```
moved=$(mktemp -d)
cp -r data-raw code .here "$moved"
cd "$moved"
Rscript code/count_rows.R
```

```
[1] 344
```

`mktemp -d` makes a new empty folder with a made-up name. The script found its file from the new place, because `here()` built the address from wherever the project now sits. An absolute path would have pointed back to the old folder.

What the test does not establish:

- That it runs on your guide's laptop. Your copy still has your version of R and your packages. If your guide lacks readr or the here package, the first line stops.
- That the number is right. The script finding 344 rows says the file was read. It says nothing about whether the analysis that follows is correct.
- That the copy was complete. If the script read a file outside the project, the test would still have passed on your computer, where that file exists.

9. The two checksums are the same, so the remade file is, for practical purposes, identical to the one deleted. The cleaned file can be thrown away and made again at any time. So the result is the code and the raw file together, and the cleaned file is one product of them. Send the co-author the folder, not the file.

What this does not establish:

- That keeping only rows with a body mass was the right cleaning step. The check shows the step is repeatable, not that it is sensible.
- That the co-author could remake it on their computer. That needs their R and packages, which this check did not touch.
- That the raw file itself is the one that was collected. The checksum describes the cleaned file only.

8 · Tidy data and the data frame

EXERCISE 1. A data frame is a set of named columns side by side. Each column is a vector whose values share one type, and every column has the same length, so the values line up into rows.

One row of the penguin file is one nesting observation of one adult penguin, made on one expedition. It is not one penguin: the same ID appears on rows for birds of different species and islands.

EXERCISE 2. +————————-+————————————————————————+ ————————————————————-+ | Problem | Why it breaks a rule | Fix | +————————-+ ————————————————————————+————————————————————-+ | Class/Section | two variables in one column | separate columns class and section | +————————-+ ————————————————————————+————————————————————-+ | Ht 1 (cm), Ht 2 (cm) | a value (the measurement number) in the column names | one column height_cm with a column measurement_no, one row per measurement | +————————-+————————————————————————+ ————————————————————-+ | BMI | a calculation in the raw data | leave it out of the raw file; compute it in code | +————————-+————————————————————————+ ————————————————————-+ | red highlighting | colour used as data | a column referred, TRUE or

FALSE | +————————-+———————————————————————————+ ————————————————————————-+ | Name | not a tidy-rule break; a confidentiality point: it can repeat and carries personal information | a child_id column; names kept apart in a separate, protected file | +————————-+ ———————————————————————————+————————————————————————-+

The tidy result is two tables. One has a row per child: `child_id`, `class`, `section`, `referred`, `remarks`. The other has a row per measurement: `child_id`, `measurement_no`, `height_cm`. Weight was measured once, so it can sit in the child table as `weight_kg`.

The units in `Ht 1 (cm)` and `Wt (kg)` were in the column names, which is allowed. They stay there, as `height_cm` and `weight_kg`.

1.

```
small <- tibble::tibble(x = c(1, 2, 3), y = c("a", "b", "c"))
nrow(small)
ncol(small)
```

```
[1] 3
[1] 2
```

Three rows, because each column has three values. Two columns, `x` and `y`.

2. One answer. Your values will differ.

```
library(tibble)
children <- tibble(
  child_id = c("A01", "A02", "A03"),
  height_cm = c(121, 127, 118),
  weight_kg = c(22.5, 25.1, 20.9)
)
children
mean(children$weight_kg)
```

```
# A tibble: 3 × 3
  child_id height_cm weight_kg
  <chr>        <dbl>     <dbl>
1 A01            121      22.5
2 A02            127      25.1
3 A03            118      20.9
[1] 22.83333
```

```
22.5 + 25.1 + 20.9 = 68.5
68.5 / 3 = 22.83333
```

3.

```
library(readr)
library(here)
penguins_raw <- read_csv(here("data-raw", "penguins_raw.csv"), show_col_types = FALSE)
nrow(penguins_raw)
ncol(penguins_raw)
mean(penguins_raw$`Flipper Length (mm)`, na.rm = TRUE)
```

```
here() starts at ~/project
[1] 344
[1] 17
[1] 200.9152
```

344 rows and 17 columns. The mean flipper length is 200.9152 mm, over the rows that have a flipper length. The column name carries the unit, millimetres. The name needs backticks because it contains spaces and brackets.

4.

```
library(readr)
library(here)
penguins_raw <- read_csv(here("data-raw", "penguins_raw.csv"), show_col_types = FALSE)
unique(penguins_raw$Island)
```

```
here() starts at ~/project
[1] "Torgersen" "Biscoe"    "Dream"
```

Three islands: Torgersen, Biscoe and Dream. `unique()` gives each different value once, in the order it first appears.

5.

```
library(readr)
library(here)
penguins_raw <- read_csv(here("data-raw", "penguins_raw.csv"), show_col_types = FALSE)
table(penguins_raw$studyName)
```

```
here() starts at ~/project

PAL0708 PAL0809 PAL0910
    110     114     120
```

One row is one nesting observation of one adult penguin, made on one expedition. It is not one penguin, for two reasons the section showed. The same ID can appear on rows for different species and islands, so an ID does not name one bird. And nothing in the file says that two rows from different expeditions are not the same bird seen twice. The expedition counts add up to the 344 rows:

```
110 + 114 + 120 = 344
```

6. The last line broke. The column name `Body Mass (g)` has spaces and brackets, so R reads `penguins_raw$Body` and then meets the word `Mass`, which it cannot place. It stops before running anything on that line.

It looked right because the name is typed exactly as it appears in the file. The repair is backticks round the whole name:

```
mean(penguins_raw$`Body Mass (g)`, na.rm = TRUE)
```

```
[1] 4201.754
```

The mean is 4201.754 grams.

7. Nothing is missing. The `bp` column holds two numbers and a slash in each cell, so it is text, and `mean()` cannot average text. It warns, and returns `NA`. `NA` usually means "not available", so "data not available" reads as if R said so. The warning says something else: the argument is not numeric.

The repair is in the layout, not in the analysis. Store systolic and diastolic pressure in two columns:

```
camp <- tibble::tibble(
  child_id = c("K1", "K2", "K3", "K4"),
  systolic_mmhg = c(102, 108, 96, 110),
  diastolic_mmhg = c(64, 70, 60, 72)
)
mean(camp$systolic_mmhg)
mean(camp$diastolic_mmhg)
```

```
[1] 104
[1] 66.5
```

```
102 + 108 + 96 + 110 = 416
416 / 4 = 104
64 + 70 + 60 + 72 = 266
266 / 4 = 66.5
```

When the raw file arrives with "102/64" in it, the split is done in code, and the raw file is left as it came.

8. One row per child per visit:

```
library(tibble)
weights <- tibble(
  child_id = c("P1", "P1", "P1", "P2", "P2", "P2", "P3", "P3", "P3"),
  visit = c(1, 2, 3, 1, 2, 3, 1, 2, 3),
  weight_kg = c(18.2, 18.6, 19.1, 20.4, 20.9, 21.0, 17.5, 17.8, 18.4)
)
nrow(weights)
mean(weights$weight_kg)
```

```
[1] 9
[1] 19.1
```

Three children times three visits is nine observations, and the table has nine rows, so it matches the sentence.

```
3 * 3 = 9
18.2 + 18.6 + 19.1 + 20.4 + 20.9 + 21.0 + 17.5 + 17.8 + 18.4 = 171.9
171.9 / 9 = 19.1
```

What the table does not establish:

- That the visits were at the same times for each child. The visit number is an order, not a date.
- Anything about children in general. The three children are made up, and three real children would describe only themselves.
- That the mean of nine weights means anything. It mixes three children at three times. The tidy shape makes it easy to compute; it does not make it a sensible summary.

9. The claim treats one row as one penguin. Test that by counting rows, then counting what might identify a bird.

```
library(readr)
library(here)
penguins_raw <- read_csv(here("data-raw", "penguins_raw.csv"), show_col_types = FALSE)
nrow(penguins_raw)
length(unique(penguins_raw$`Individual ID`))
length(unique(paste(penguins_raw$Species, penguins_raw$Island, penguins_raw$`Individual
ID`)))
```

```
here() starts at ~/project
[1] 344
[1] 190
[1] 304
```

Rows against IDs is enough to refute the claim. The last line is a further check: `paste()` joins the three columns into one piece of text per row, and `unique()` keeps each different combination once.

There are 344 rows, 190 different IDs, and 304 different combinations of species, island and ID. So “344 penguins” is wrong if the same bird can appear on more than one expedition. “190 penguins” is wrong too, because one ID can name birds of two species.

What the answer does not establish: the number of birds. Suppose species, island and ID together always name one bird, and a bird keeps its ID from one expedition to the next. Then there were 304 birds. Neither is stated in the file or on its help page. Say “344 nesting observations”, which the file supports, and stop there.

9 · Reading a raw file without changing it

EXERCISE 1. `na = c("", "NA", "-99")`. Giving `na` replaces the default list, which is the empty string and the text “NA”, so both go back in beside -99.

`problems()`, called on the data frame that `read_csv()` returned. It gives the row, the column, the type expected and the value found.

EXERCISE 2.

```
library(readr)
library(here)
screening <- read_csv(
  here("data-raw", "screening_2026.csv"),
  na = c("", "NA", "999", "NR"),
  col_types = cols(
    village_code = col_character(),
    height_cm = col_double(),
    weight_kg = col_double()
  )
)
problems(screening)
```

- `here("data-raw", ...)`: the address works on any computer that has the project.
- `na`: 999 and “NR” become `NA`, so 999 cm never enters a summary. The empty string and “NA” stay in the list, because giving `na` replaces the default.
- `village_code = col_character()`: read as a number, 0042 would become 42. Read as text, it stays 0042, as the documentation writes it.

- `height_cm` and `weight_kg` as `col_double()`: any word typed in those columns is reported by `problems()` instead of turning the column into text.

Columns not named in `cols()` are still guessed. The block is shown, not run, because the file is made up.

EXERCISE 3. It tells you the file has 1250 rows and 6 columns, and that `weight_kg` was read as text. Weight should be a number. So at least one cell in that column is not a number. It may hold a unit ("62 kg"), a word ("absent"), or a comma used as a decimal mark.

It does not tell you which cells, how many, or whether the other numbers are right. It does not tell you whether 1250 rows means 1250 patients.

Next, read again with `col_types = cols(weight_kg = col_double())`. Run `problems()` to list the cells that are not numbers, and decide in code what each means. If they are a missing-value code, add it to `na`.

1.

```
library(readr)
read_csv(I("district,children,with_obesity
Raipur,410,23
Durg,385,19"))
```

```
Rows: 2 Columns: 3
── Column specification ────────────────────────────────────────────────────────
Delimiter: ","
chr (1): district
dbl (2): children, with_obesity

ℹ Use `spec()` to retrieve the full column specification for this data.
ℹ Specify the column types or set `show_col_types = FALSE` to quiet this message.
# A tibble: 2 × 3
  district children with_obesity
  <chr>       <dbl>        <dbl>
1 Raipur        410           23
2 Durg          385           19
```

Two rows and three columns. `district` is text; the two counts are numbers.

2.

```
library(readr)
read_csv(I("ward,beds,occupied
A,30,27
B,24,24"), col_types = "cdd")
```

```
# A tibble: 2 × 3
  ward   beds occupied
  <chr> <dbl>    <dbl>
1 A        30       27
2 B        24       24
```

The string `"cdd"` says: first column text, second and third numbers. No report is printed, because the types were given.

3.

```
library(readxl)
read_excel(readxl_example("deaths.xlsx"), range = "arts!A5:C8")
```

```
# A tibble: 3 × 3
  Name          Profession   Age
  <chr>         <chr>      <dbl>
1 David Bowie   musician      69
2 Carrie Fisher actor         60
3 Chuck Berry   musician      90
```

Three rows and three columns. Row 5 of the sheet became the header, and rows 6 to 8 are data. The sheet can be named inside the range, as `arts!`, instead of in `sheet`.

4.

```
library(readr)
library(here)
penguins_raw <- read_csv(here("data-raw", "penguins_raw.csv"))
```

```
here() starts at ~/project
Rows: 344 Columns: 17
── Column specification ────────────────────────────────────────────────────────
Delimiter: ","
chr  (9): studyName, Species, Region, Island, Stage, Individual ID, ...
dbl  (7): Sample Number, Culmen Length (mm), Culmen Depth (mm), Flip...
date (1): Date Egg

ℹ Use `spec()` to retrieve the full column specification for this data.
ℹ Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Nine as text (`chr`), seven as numbers (`dbl`), one as a date. The seven numeric columns include the four measurements whose names carry a unit, `(mm)` or `(g)`, and the two isotope ratios. The seventh is `Sample Number`, a running number, which is stored as a number but is not a measurement.

5.

```
library(haven)
library(here)
demo <- read_xpt(here("data-raw", "DEMO_L.xpt"))
dim(demo)
attr(demo$RIDAGEYR, "label")
```

```
here() starts at ~/project
[1] 11933    27
[1] "Age in years at screening"
```

11933 rows and 27 columns. `RIDAGEYR` is age in years at screening. The demographics file has more rows than the body measures file, because not everyone screened was examined.

6. The `col_types` line broke. `.default = col_double()` says every column is numbers, including `child_id`. "S01" is not a number, so every ID became `NA`. The IDs are in the file. The code threw them away.

Her note looks reasonable because the result shows `NA` in the ID column, and `NA` reads as "missing". The warning said what happened: values did not fit the type she asked for.

Name the ID column as text and let the rest default to numbers:

```
camp <- read_csv(I("child_id,height_cm,weight_kg
S01,128,26.0
S02,133,24.5
S03,131,27.5"), col_types = cols(child_id = col_character(), .default = col_double()))
camp
```

```
# A tibble: 3 × 3
  child_id height_cm weight_kg
  <chr>        <dbl>     <dbl>
1 S01            128      26
2 S02            133      24.5
3 S03            131      27.5
```

7. The `na = "-99"` argument broke it. It replaced the default list, so the text "NA" in the file is no longer read as missing. The 11 rows with no sex recorded now have the text "NA" as their sex, which is not `NA`, so the count is 0. The report gives it away: 15 columns are now `chr`, because "NA" in a number column is a word, not a number.

The answer looked believable because 0 is a possible count, and nothing stopped. Keep the defaults in the list:

```
penguins_raw <- read_csv(here("data-raw", "penguins_raw.csv"), na = c("", "NA", "-99"),
                         show_col_types = FALSE)
sum(is.na(penguins_raw$Sex))
```

```
[1] 11
```

Eleven rows have no sex recorded.

8.

```
library(haven)
library(here)
bmx <- read_xpt(here("data-raw", "BMX_L.xpt"))
nrow(bmx)
sum(!is.na(bmx$BMXWT))
mean(bmx$BMXWT, na.rm = TRUE)
min(bmx$BMXWT, na.rm = TRUE)
```

```
here() starts at ~/project
[1] 8860
[1] 8754
[1] 70.54904
[1] 2.7
```

The file has 8860 rows. Only 8754 of them have a weight, so the mean is over 8754 people, not 8860:

```
8860 - 106 = 8754
```

The mean of those weights is 70.54904 kg, which rounds to 70.5 kg. So both numbers can be reproduced from the file. The smallest weight is 2.7 kg, an infant: the file covers all ages.

What the sentence does not establish:

- The average weight of Americans. This mean is unweighted. The survey's documentation says its sample weights should be used, and they were not.
- The average weight of adults. Children and infants are in the 8754.

- Anything about India. It is one survey of the United States.

9. Compute the checksum of the file the analysis reads, and compare it with the one in the README:

```
library(readr)
library(here)
raw_path <- here("data-raw", "penguins_raw.csv")
unname(tools::md5sum(raw_path)) == "049da101568e078f9845c8b366481810"
```

```
here() starts at ~/project
[1] TRUE
```

TRUE: this copy is byte for byte the file that arrived. Now see what a single change does. Make a copy in `output/` and change one character on one line, the way a hand edit would. `readLines()` reads a file's lines into a vector, `sub()` replaces the first match in a piece of text, and `writeLines()` writes lines to a file:

```
lines <- readLines(raw_path)
lines[2] <- sub("3750", "3760", lines[2])
dir.create(here("output"), showWarnings = FALSE)
writeLines(lines, here("output", "penguins_edited.csv"))
unname(tools::md5sum(here("output", "penguins_edited.csv")))
```

```
[1] "ca70412ca7d8c1908ef8530279c1111f"
```

One digit changed, and the checksum is completely different. If your colleague's saved file gives the README's checksum, nothing changed. If it gives anything else, something did, even if the cells look the same on screen.

What the check does not establish:

- What changed, or whether it matters. A different checksum says the bytes differ. It does not say which value, or whether a spreadsheet only changed how a date was written.
- That the original was correct. The checksum only ties today's file to the day it arrived.
- Which file the analysis used, if the script reads some other copy. Check the path in the script as well.

10 · The data dictionary and the README

EXERCISE 1. One dictionary. Yours may word the meanings differently; check that every column is there, that names are exact, and that every entry says where it came from.

Variable	Meaning	Unit	Type	Allowed values or range	Missing written as	Source
child_id	the child's study number	none	text	one per child	none expected	protocol; check one per row in the file
school	school the child attends	none	text	the schools listed in the protocol	none expected	protocol
sex	sex of the child	none	number code	1 = boy, 2 = girl	none expected	protocol

dob	date of birth	none	date	before visit_date	none expected	protocol; check the date format in the file
visit_date	date of measurement	none	date	within the study period	none expected	protocol
height_cm	standing height, to 0.1 cm	cm	number	not stated	-99	protocol; missing code from the operator
weight_kg	weight, to 0.1 kg	kg	number	not stated	-99	protocol; missing code from the operator

Three things to notice. The codes 1 and 2 are written out, because nobody can recover "1 = boy" from the file. The -99 is recorded, so it can be declared as missing when the file is read, and never averaged. And the expected ranges say "not stated" rather than being filled from the data, because the protocol gives none.

Before cleaning, you would check each "one per child" and each code against the file, as the illustration did.

EXERCISE 2. Almost everything a README is for.

- **No title or purpose.** The next person cannot tell which thesis, or what question.
- **No source for the raw data.** "The department" names no file, no version, no date and no terms. Nobody can get a fresh copy, or say whether they may use it.
- **No rerun command.** "Run the scripts in order" leaves the order to be guessed, and says nothing about where to run them from. If the order matters, it belongs in the code: one script that runs the others, named in one line.
- **No outputs.** The next person cannot tell which file is the final table.
- **No pointer to a data dictionary**, so the meanings of columns and codes are lost.
- **No contact.** When it breaks, there is nobody to ask.

The cost is the same each time. The next person has to guess what one paragraph could have said. That person may be your colleague, a year later.

EXERCISE 3. For each variable, the data dictionary records seven things:

- its exact name as in the file;
- what it means;
- its unit;
- its type;
- its allowed values or expected range;
- how a missing value is written;
- where each of these facts came from.

The README says six things:

- the project's title and purpose;
- where the raw data came from, which version, on what terms, and its checksum;
- where the data dictionary is;
- the one command that reruns the analysis, and where to run it from;

- what each output is;
- whom to contact.

EXERCISE 4. Without a dictionary, the data cannot be read safely by anyone except the people who entered it.

A data file holds names and numbers. It does not say what "sex = 2" means, whether height is in centimetres or metres, or that -99 means "not measured". Those facts live in the heads of a few staff, or in a protocol nobody attaches. When those staff move on, the facts go with them.

The cost of not having one is concrete. A -99 read as a real value pulls an average down. A height read in the wrong unit makes every derived index wrong by a large factor. A code read the wrong way round swaps two groups. Each produces a confident, wrong number in a report.

A dictionary is one table, one row per column, written once when the data are collected. It costs an afternoon. Ask that every dataset the department releases or commissions comes with one, and that the dictionary is checked against the file before any analysis.

11 · Choosing rows and columns

EXERCISE 1. Write three values on the board: 11.2, NA, 9.8. Under each, write what `hb == NA` gives: NA, NA, NA. Comparing anything with an unknown value gives an unknown answer, even comparing NA with NA.

Then say what `filter()` does with the answers. It keeps a row only when the answer is `TRUE`. It drops `FALSE` and it drops `NA`. Every answer was `NA`, so every row was dropped. The empty result is a property of `==`, not of their survey.

Now write `is.na(hb)` under the same three values: FALSE, TRUE, FALSE. That is a real answer on every row, so `filter(is.na(hb))` keeps exactly the rows with no value.

Ask them to run it and report the number it gives. Then ask them to put that number in their results. Ask them to add how many rows the survey had, so a reader can see the denominator.

EXERCISE 2. `select()` keeps the columns you name, in that order, and keeps every row. The function `rename()` changes column names and keeps every row and every column. The function `filter()` keeps the rows whose condition is `TRUE`, dropping `FALSE` and `NA`, and keeps every column. The function `arrange()` keeps every row and column and changes only the order of the rows, with missing values last.

None of them changes the data frame you gave it. Each returns a new one, which is lost unless you assign it with `<-`.

EXERCISE 3. One answer. Your names may differ; check that each is lowercase, joined by _, and ends in its unit where it has one.

```
library(readr)
library(dplyr, warn.conflicts = FALSE)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)

penguins <- penguins_raw |>
  select(
    species = Species,
    island = Island,
    sex = Sex,
    flipper_length_mm = `Flipper Length (mm)`,
    body_mass_g = `Body Mass (g)`
  )

nrow(penguins)
ncol(penguins)
```

```
[1] 344
[1] 5
```

`select()` with `new_name = old_name` keeps and renames in one step. No row is lost, because `select()` never drops rows. The raw object `penguins_raw` still has all 17 columns.

1. (a)

```
clinic |>
  filter(sex == "F")
```

```
# A tibble: 2 × 3
     id sex   weight_kg
  <dbl> <chr>     <dbl>
1   101 F          61.2
2   103 F          NA
```

Two rows, ids 101 and 103. The test `sex == "F"` is `TRUE` on rows 1 and 3, so those stay. The missing weight on row 3 does not matter: the test was on `sex`, not on `weight_kg`.

1.

```
clinic |>
  select(id, weight_kg)
```

```
# A tibble: 3 × 2
     id weight_kg
  <dbl>     <dbl>
1   101      61.2
2   102      78.5
3   103      NA
```

Three rows and two columns. `select()` keeps every row and the named columns, in the order you named them.

1.

```
clinic |>
  arrange(desc(weight_kg))
```

```
# A tibble: 3 × 3
     id sex   weight_kg
  <dbl> <chr>     <dbl>
1   102 M          78.5
2   101 F          61.2
3   103 F          NA
```

102 (78.5 kg), then 101 (61.2 kg), then 103. The missing value goes last. `arrange()` puts missing values at the end whichever direction you sort, so `desc()` does not bring it to the top.

2.

```
penguins_raw |>
  filter(Species == "Chinstrap penguin (Pygoscelis antarctica)") |>
  nrow()
```

```
[1] 68
```

68 rows are Chinstrap penguins. The test must match the text exactly, letter for letter, including the capital C and the part in brackets.

3.

```
penguins_raw |>
  filter(Island == "Biscoe", `Flipper Length (mm)` >= 210) |>
  nrow()
```

```
[1] 109
```

109 rows from Biscoe have a flipper length of 210 mm or more. The comma joins the two tests, so both must be `TRUE`. `>=` includes a flipper of exactly 210 mm; `>` would leave it out.

4. Both missing: join the two tests with `&`.

```
penguins_raw |>
  filter(is.na(`Delta 15 N (o/oo)`) & is.na(`Delta 13 C (o/oo)`)) |>
  nrow()
```

```
[1] 13
```

At least one missing: join them with `|`.

```
penguins_raw |>
  filter(is.na(`Delta 15 N (o/oo)`) | is.na(`Delta 13 C (o/oo)`)) |>
  nrow()
```

```
[1] 14
```

13 rows are missing both isotope values and 14 are missing at least one. So one row has a carbon value and no nitrogen value. “And” can only give a count the same size as “or” or smaller, never larger.

5. The test does not match the text in the file. Look at the values that are there:

```
penguins_raw |>
  count(Species)
```

```
# A tibble: 3 × 2
  Species                                   n
  <chr>                                 <int>
1 Adelie Penguin (Pygoscelis adeliae)     152
2 Chinstrap penguin (Pygoscelis antarctica)  68
3 Gentoo penguin (Pygoscelis papua)       124
```

The file writes “Adelie Penguin”, with a capital P. The other two species have a small p. The comparison `==` works letter for letter. So “penguin” never equals “Penguin”, and the test is `FALSE` on every row. Copy the value exactly:

```
penguins_raw |>
  filter(Species == "Adelie Penguin (Pygoscelis adeliae)") |>
  nrow()
```

```
[1] 152
```

152 rows are Adelie penguins. (`count()`, met in section 10, gives the number of rows for each value of a column.)

Why the wrong answer looks reasonable: the code ran, the spelling looks right, and 0 is a real number. But a species the study was built around cannot have no rows. An answer of zero is the moment to print the values you tested against.

6. The test `Sex == NA` is `NA` on every row. A comparison with an unknown value is unknown. The function `filter()` drops every row whose test is `NA`, so it drops all of them. The test must be `is.na()`:

```
penguins_raw |>
  filter(is.na(Sex)) |>
  nrow()
```

```
[1] 11
```

11 rows have no sex recorded.

Why the wrong answer looks reasonable: zero missing is a common and welcome result, and nothing on the screen says the test was meaningless. The line reads correctly as English.

7. `Sex == "MALE" | "FEMALE"` is English, not R. R works out `Sex == "MALE"` first, which gives `TRUE` or `FALSE`. Then it tries `|` between that and the text “FEMALE”. The sign `|` needs `TRUE` or `FALSE` on both sides. A piece of text is neither, so R stops. (The error’s “complex” is a type of number this book does not use.)

Each side of `|` must be a whole test. Or use `%in%`, which says the same thing in one go:

```
penguins_raw |>
  filter(Island == "Dream", Sex %in% c("MALE", "FEMALE")) |>
  nrow()
```

```
[1] 123
```

123 rows on Dream have sex recorded as MALE or FEMALE.

Why the wrong line looks reasonable: it is how you would say it aloud. With numbers instead of text, the same mistake does not stop with an error; it quietly keeps the wrong rows, which is worse.

8. The result of `filter()` was printed and then thrown away. The function never changes `penguins_raw`. It hands back a new data frame, and nothing gave that one a name. So `mean()` was given the original column, missing values and all, and a mean with a missing value in it is missing.

Assign the filtered data frame, then take the mean of its column:

```
weighed <- penguins_raw |>
  filter(!is.na(`Body Mass (g)`))

nrow(weighed)
mean(weighed$`Body Mass (g)`)
```

```
[1] 342
[1] 4201.754
```

342 rows have a body mass, and their mean is about 4202 g (4201.754 as printed).

Why the wrong code looks reasonable: the filter printed the right number of rows, so it looked as if it had done its job. It had. It just did not do it to `penguins_raw`.

9. `arrange()` sorts smallest first. The top row is the lightest penguin, not the heaviest. Wrap the column in `desc()`:

```
penguins_raw |>
  arrange(desc(`Body Mass (g)`)) |>
  select(Species, `Body Mass (g)`) |>
  print(n = 1)
```

```
# A tibble: 344 × 2
  Species                            `Body Mass (g)`
  <chr>                                        <dbl>
1 Gentoo penguin (Pygoscelis papua)             6300
# ℹ 343 more rows
```

The heaviest penguin was a Gentoo at 6300 g.

Why the wrong answer looks reasonable: 2700 g is a real penguin's mass, and "top row" feels like "top value". Check a sort by looking at the last row as well as the first, or by asking whether the number is plausible as a maximum.

10. Read the file and keep the Adelie penguins on Torgersen. Then count each value of `Sex`.

```
library(readr)
library(dplyr, warn.conflicts = FALSE)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)

penguins_raw |>
  filter(
    Species == "Adelie Penguin (Pygoscelis adeliae)",
    Island == "Torgersen"
  ) |>
  count(Sex)
```

```
# A tibble: 3 × 2
  Sex        n
  <chr>  <int>
1 FEMALE    24
2 MALE      23
3 <NA>       5
```

52 rows are Adelie penguins on Torgersen: 24 FEMALE, 23 MALE, 5 with no sex recorded.

The claim depends on the denominator.

```
out of all 52: 24 ÷ 52 = 0.46, so 46%, less than half
out of the 47 with sex recorded: 24 ÷ 47 = 0.51, so 51%, more than half
```

So "more than half" is true only if the 5 unrecorded rows are left out of the bottom of the fraction. A report should say which denominator it used.

What it does not establish: a share of birds, because one bird can appear in more than one row (section 8). Nor anything about penguins on Torgersen outside this file, or about the 5 rows whose sex is unknown. Counted out of all 52, the FEMALE rows number somewhere from 24 to 29, depending on those 5. They are more than half only if at least 3 of the 5 are female. 26 is exactly half, and 27 ÷ 52 = 0.52.

11. 6 kg is 6000 g, the unit of the column. Keep Gentoo penguins heavier than 6000 g and look at their sex.

```
library(readr)
library(dplyr, warn.conflicts = FALSE)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)

penguins_raw |>
  filter(
    Species == "Gentoo penguin (Pygoscelis papua)",
    `Body Mass (g)` > 6000
  ) |>
  select(`Individual ID`, Sex, `Body Mass (g)`)
```

```
# A tibble: 2 × 3
  `Individual ID` Sex    `Body Mass (g)`
  <chr>           <chr>            <dbl>
1 N39A2           MALE              6300
2 N56A2           MALE              6050
```

Two Gentoo rows have a body mass over 6000 g, at 6300 g and 6050 g, and both are recorded as male. The sentence is true of this file.

What it does not establish:

- That heavy Gentoo penguins are male in general. Two rows are two rows.
- Anything about the Gentoo row with no body mass, N38A2. `filter()` dropped it, because its test was `NA`.
- Anything about a penguin of exactly 6000 g. "Over 6 kg" became `> 6000`, which leaves 6000 out. If the speaker meant "6 kg or more", the test is `>= 6000`. Run that too:

```
penguins_raw |>
  filter(
    Species == "Gentoo penguin (Pygoscelis papua)",
    `Body Mass (g)` >= 6000
  ) |>
  select(`Individual ID`, Sex, `Body Mass (g)`)
```

```
# A tibble: 4 × 3
  `Individual ID` Sex    `Body Mass (g)`
  <chr>           <chr>            <dbl>
1 N39A2           MALE              6300
2 N56A2           MALE              6050
3 N58A2           MALE              6000
4 N36A2           MALE              6000
```

Four rows, because two show exactly 6000 g, and all four are recorded as male. Here the two readings agree. They need not. Say which one you used.

12 · Making new columns

EXERCISE 1. Three things went wrong, and all three are about the raw file.

The raw file is no longer raw. A computed column now sits among the measured ones, and nothing in the file says which is which. Anyone reading it later may take the BMI as measured.

The rule is hidden. Whether the formula converted height to metres, and how it rounded, lives inside a cell nobody will open. If a weight is corrected later, the BMI may or may not update, depending on whether the formula survived the save.

There is no record of the step. Nothing shows when the column was added or by whom.

Instead: keep the file exactly as collected, and compute BMI in the script with `mutate()`, converting height to metres in its own column first. The rule is then one visible line, it reruns whenever the data change, and the raw file stays clean.

EXERCISE 2. Any four of these:

- A unit not converted first: BMI from height in centimetres is 10,000 times too small.
- Cases in the wrong order: an early, wide case catches rows meant for a later one.
- A `.default` that catches missing values and labels them as if measured.
- A value at a cut-point that prints as the cut-point but is stored a hair below or above it.
- New values written over an old column, so the column's name and unit no longer match its values.

EXERCISE 3.

```
library(readr)
library(dplyr, warn.conflicts = FALSE)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)

penguins <- penguins_raw |>
  mutate(
    flipper_length_cm = `Flipper Length (mm)` / 10,
    body_mass_kg = `Body Mass (g)` / 1000,
    mass_group = if_else(
      body_mass_kg >= 4.5,
      "4.5 kg or more",
      "under 4.5 kg"
    )
  )

penguins |>
  count(mass_group)
```

```
# A tibble: 3 × 2
  mass_group         n
  <chr>          <int>
1 4.5 kg or more   118
2 under 4.5 kg     224
3 <NA>               2
```

118 rows show 4.5 kg or more, 224 show less, and 2 have no body mass, so no label. A millimetre is a tenth of a centimetre, so dividing by 10 converts. The function `if_else()` gives `NA` where the test is `NA`. So the two rows with no body mass are not quietly counted as light.

1.

```
heights |>
  mutate(height_m = height_cm / 100)
```

```
# A tibble: 3 × 2
  height_cm height_m
      <dbl>    <dbl>
1       150      1.5
2       162.     1.62
3       178      1.78
```

A metre is 100 centimetres, so divide by 100: 1.5, 1.625 and 1.78 metres. The tibble prints 1.625 as 1.62, and 162.5 as `162.`, to fit its columns; the column holds the full values. The old column stays beside the new one.

2.

```
weights |>
  mutate(group = if_else(weight_kg >= 60, "60 or more", "under 60"))
```

```
# A tibble: 3 × 2
  weight_kg group
      <dbl> <chr>
1        72 60 or more
2        55 under 60
3        NA <NA>
```

The third row gets `NA`. Its test, `NA >= 60`, is `NA`, and `if_else()` gives `NA` where the test is `NA` unless you supply `missing`.

3.

```
round(c(0.5, 1.5, 2.5))
```

```
[1] 0 2 2
```

0, 2 and 2. A 5 at the cut goes to the even digit: 0.5 down to 0, 1.5 up to 2, 2.5 down to 2. School rounding would give 1, 2 and 3. Both are rules; R uses the first.

4.

```
penguins_raw |>
  mutate(flipper_length_cm = `Flipper Length (mm)` / 10) |>
  select(`Flipper Length (mm)`, flipper_length_cm) |>
  print(n = 4)
```

```
# A tibble: 344 × 2
  `Flipper Length (mm)` flipper_length_cm
                  <dbl>             <dbl>
1                   181              18.1
2                   186              18.6
3                   195              19.5
4                    NA              NA
# ℹ 340 more rows
```

The first penguin's flipper is 181 mm, which is 18.1 cm. There are 10 millimetres in a centimetre, so divide by 10. Row 4 is `NA` in both columns: no flipper was measured.

5.

```
penguins_raw |>
  mutate(
    body_mass_kg = `Body Mass (g)` / 1000,
    mass_group = case_when(
      body_mass_kg < 3.5 ~ "under 3.5 kg",
      body_mass_kg < 4.5 ~ "3.5 to under 4.5 kg",
      body_mass_kg >= 4.5 ~ "4.5 kg or more"
    )
  ) |>
  count(mass_group)
```

```
# A tibble: 4 × 2
  mass_group              n
  <chr>               <int>
1 3.5 to under 4.5 kg   153
2 4.5 kg or more        118
3 under 3.5 kg           71
4 <NA>                    2
```

153 rows from 3.5 to under 4.5 kg, 118 at 4.5 kg or more, 71 under 3.5 kg, and 2 with no body mass. The four counts add to 344, every row. `count()` sorts the labels alphabetically, which is why "3.5 to under 4.5 kg" comes first.

6. Count the spellings as typed:

```
register |>
  count(sex)
```

```
# A tibble: 5 × 2
  sex           n
  <chr>     <int>
1 " female"     1
2 "Female"      1
3 "MALE "       1
4 "Male"        1
5 "male"        1
```

Five rows, five different spellings: R treats " female" and "Female" as different values. Trim the spaces, then lower the case, in a new column:

```
register |>
  mutate(sex_clean = str_to_lower(str_trim(sex))) |>
  count(sex_clean)
```

```
# A tibble: 2 × 2
  sex_clean     n
  <chr>     <int>
1 female        2
2 male          3
```

Two values: 2 female and 3 male. Read the nested call from the inside out: trim first, then lower. The original `sex` column is kept, so the cleaning can be checked.

7. The height is in centimetres. BMI needs metres, and a height 100 times too large, squared, makes the BMI 10,000 times too small. Every BMI is about 0.002, which is below 18.5, so the test says `TRUE` for everyone. Convert first:

```
register |>
  mutate(
    height_m = height_cm / 100,
    bmi = weight_kg / height_m^2
  )
```

```
# A tibble: 3 × 4
  height_cm weight_kg height_m   bmi
      <dbl>     <dbl>    <dbl> <dbl>
1       158        62     1.58  24.8
2       172        85     1.72  28.7
3       149        47     1.49  21.2
```

The BMIs are about 24.8, 28.7 and 21.2 kg/m^2. None is below 18.5.

Why the wrong answer looks reasonable: the code ran without a warning, and the test that followed gave a tidy answer. Nobody looked at the BMI column itself, where 0.00248 would have given it away.

8. The cases are checked in order, and each row takes the first one it passes. A BMI of 26.4 passes `bmi >= 18.5`, the second case, so it is labelled "18.5 to below 25" and never reaches the third case. The second case must stop at 25:

```
register |>
  mutate(
    bmi_group = case_when(
      bmi < 18.5 ~ "below 18.5",
      bmi < 25 ~ "18.5 to below 25",
      bmi >= 25 ~ "25 or more"
    )
  )
```

```
# A tibble: 4 × 2
    bmi bmi_group
  <dbl> <chr>
1  17.2 below 18.5
2  23.9 18.5 to below 25
3  26.4 25 or more
4  31   25 or more
```

Two of the four, 26.4 and 31.0, are in "25 or more".

Why the wrong answer looks reasonable: each line, read alone, is true of its group. The label "18.5 to below 25" says "below 25", but the test beside it does not. The label is text; only the test decides.

9. The third patient has no BMI. Both tests are `NA` for that row, so no case matches, and the row gets the `.default`, "25 or more". A missing value has been counted as a high BMI.

Write the last case as a test, and leave `.default` out:

```
register |>
  mutate(
    bmi_group = case_when(
      bmi < 18.5 ~ "below 18.5",
      bmi < 25 ~ "18.5 to below 25",
      bmi >= 25 ~ "25 or more"
    )
  ) |>
  count(bmi_group)
```

```
# A tibble: 4 × 2
  bmi_group            n
  <chr>            <int>
1 18.5 to below 25     1
2 25 or more           1
3 below 18.5           1
4 <NA>                 1
```

One patient in each group and one `NA`: 1 of the 3 with a BMI is at 25 or more.

Why the wrong answer looks reasonable: "everything else is 25 or more" is true of every patient who has a BMI. The `.default` was written with measured values in mind, and the missing value slipped through.

10. The new values were written over the old column, under the old name. The column still says "(g)" but now holds kilograms. The arithmetic is right and the label is wrong, so the table reports a penguin of 4.2 grams.

Make a new column with a name that carries its unit:

```
penguins <- penguins_raw |>
  mutate(body_mass_kg = `Body Mass (g)` / 1000)

mean(penguins$body_mass_kg, na.rm = TRUE)
```

```
[1] 4.201754
```

The mean body mass is 4.2 kg (4.201754 as printed), from the 342 rows with a body mass.

Why the wrong answer looks reasonable: the number 4.2 is correct. Only its unit is wrong, and the unit came from a name the code no longer matched.

11. The first `mutate()` uses `height_m` before any line has made it. The pipe runs top to bottom, so when BMI is computed, the column `height_m` does not exist yet. Swap the order, or make both in one `mutate()`, height first:

```
register |>
  mutate(
    height_m = height_cm / 100,
    bmi = weight_kg / height_m^2
  )
```

```
# A tibble: 3 × 4
  height_cm weight_kg height_m   bmi
      <dbl>     <dbl>    <dbl> <dbl>
1       158        62     1.58  24.8
2       172        85     1.72  28.7
3       149        47     1.49  21.2
```

Why the wrong code looks reasonable: both lines are right. Only their order is wrong, and the order is the one thing a pipe never rearranges for you.

12. Compute BMI from height in metres. Then count the BMIs at 25 or more, first as stored, then rounded to one decimal, the precision of the cut-point.

```
audit |>
  mutate(
    height_m = height_cm / 100,
    bmi = weight_kg / height_m^2,
    bmi_1dp = round(bmi, 1),
    stored_25_plus = bmi >= 25,
    rounded_25_plus = bmi_1dp >= 25
  ) |>
  select(id, bmi_1dp, stored_25_plus, rounded_25_plus)
```

```
# A tibble: 5 × 4
     id bmi_1dp stored_25_plus rounded_25_plus
  <dbl>   <dbl> <lgl>          <lgl>
1     1    25   FALSE          TRUE
2     2    21.6 FALSE          FALSE
3     3    26   TRUE           TRUE
4     4    22.8 FALSE          FALSE
5     5    24   FALSE          FALSE
```

Patient 1's BMI is 25.0 on paper (64 divided by 1.6^2 is 25), but as stored it falls a hair below 25. Compared as stored, 1 patient is at 25 or more. Rounded to one decimal, 2 are: patients 1 and 3. So the claim is true under the rounding rule and false without it. The audit should say which rule it used.

What it does not establish: anything about the clinic's other patients, or about people outside the clinic. Five patients are five patients. And the BMI itself says nothing about any one patient's health without the rest of their assessment.

13. Convert to kilograms in a new column, keep the Gentoo penguins, and take the mean, leaving out missing values.

```
library(readr)
library(dplyr, warn.conflicts = FALSE)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)

gentoo <- penguins_raw |>
  mutate(body_mass_kg = `Body Mass (g)` / 1000) |>
  filter(Species == "Gentoo penguin (Pygoscelis papua)")

nrow(gentoo)
sum(is.na(gentoo$body_mass_kg))
mean(gentoo$body_mass_kg, na.rm = TRUE)
```

```
[1] 124
[1] 1
[1] 5.076016
```

5.076016 kg, which rounds to 5.08 kg. The sentence matches the file. But `na.rm = TRUE` left one row out: the mean is of 123 of the 124 Gentoo rows, and the sentence should say so.

What it does not establish:

- Anything about Gentoo penguins outside these three seasons of sampling.
- That the difference from another species' mean is real or large. A mean alone carries no comparison and no measure of spread.

- What the missing penguin weighed. Leaving it out is a choice, and the section on missing values says how to report it.

13 · Missing values: counting them and saying how many

EXERCISE 1.

```
library(readr)
library(dplyr, warn.conflicts = FALSE)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)

sum(is.na(penguins_raw$`Flipper Length (mm)`))

penguins_raw |>
  summarise(
    n = sum(!is.na(`Flipper Length (mm)`)),
    mean_flipper_mm = mean(`Flipper Length (mm)`, na.rm = TRUE)
  )
```

```
[1] 2
# A tibble: 1 × 2
      n mean_flipper_mm
  <int>           <dbl>
1   342            201.
```

Flipper length is missing in 2 rows, so the mean comes from 342 rows. The tibble prints it rounded to fit, as `201.`. In a table: “mean flipper length 201 mm (n = 342)”.

EXERCISE 2. Three things are missing.

Which columns. “Missing data” could mean a gap in any of the 28 columns, or only in height and weight. With `drop_na()` and no columns named, a gap in a column unrelated to BMI would remove the participant too.

How many. The reader cannot tell whether 400 participants were left or 40.

The n behind the mean. “Mean BMI was 26.1 kg/m^2” needs its count beside it.

Here is a rewrite. The counts are left blank, because the thesis did not give them: “Participants without a measured height or weight were excluded (n = ___ of 412). Mean BMI was 26.1 kg/m^2 (n = ___).” The script that made the table prints both counts, so they are copied from its output, never typed from memory.

EXERCISE 3. `sum(is.na(x))` for one column, because `TRUE` counts as 1. `colSums(is.na(d))` for every column. With no column named, `drop_na()` keeps only the rows with no missing value in any column.

1.

```
x <- c(3, NA, 7, NA, NA, 2)
sum(is.na(x))
```

```
[1] 3
```

Three. `is.na(x)` gives `FALSE TRUE FALSE TRUE TRUE FALSE`, and `sum()` counts each `TRUE` as 1.

2.

```
weights <- c(54, 61, NA, 70)
mean(weights)
mean(weights, na.rm = TRUE)
```

```
[1] NA
[1] 61.66667
```

The first is `NA`: one value is not known, so the sum is not known. The second leaves the missing value out. The sum is 54 plus 61 plus 70, which is 185, and 185 divided by 3 is 61.66667. It is the mean of 3 values, not 4.

3.

```
nrow(drop_na(d))
nrow(drop_na(d, height_cm))
```

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

`drop_na(d)` keeps 1 row: only the first has both values. `drop_na(d, height_cm)` looks at height alone and keeps 2 rows, the first and the third.

4.

```
sum(is.na(penguins_raw$Sex))
round(100 * sum(is.na(penguins_raw$Sex)) / nrow(penguins_raw), 1)
```

```
[1] 11
[1] 3.2
```

11 of the 344 rows have no recorded sex. 11 divided by 344 is 0.03198, which is 3.2 per cent. The denominator is every row, because the question is about the whole file.

5.

```
penguins_raw |>
  summarise(
    rows = n(),
    missing = sum(is.na(`Delta 15 N (o/oo)`)),
    mean_delta_15n = mean(`Delta 15 N (o/oo)`, na.rm = TRUE)
  )
```

```
# A tibble: 1 × 3
   rows missing mean_delta_15n
  <int>   <int>          <dbl>
1   344      14           8.73
```

344 rows, 14 missing, so the mean is of 330 values: 8.73 as printed. In a table: “mean delta 15 N 8.73 o/oo (n = 330; 14 of 344 not measured)”.

6.

```
kept <- drop_na(
  penguins_raw,
  `Culmen Length (mm)`,
  `Culmen Depth (mm)`,
  `Delta 15 N (o/oo)`,
  `Delta 13 C (o/oo)`
)

nrow(kept)
nrow(penguins_raw) - nrow(kept)
```

```
[1] 330
[1] 14
```

330 rows have all four, so 14 are left out. The four columns are missing in 2, 2, 14 and 13 rows, which add to 31, yet only 14 rows go. The gaps overlap: the rows missing a culmen measurement or a carbon value are among the 14 missing a nitrogen value. You cannot add up column counts to get the rows lost. Count the rows.

7. The code gives 40, as reported, and 40 is wrong. `!is.na()` removes nothing, because there is no `NA` in the vector: the two -9s are ordinary numbers, and they pull the mean down.

Turn the codes into `NA` first, then count and average:

```
weights_clean <- dplyr::if_else(weights == -9, NA, weights)
sum(is.na(weights_clean))
mean(weights_clean, na.rm = TRUE)
```

```
[1] 2
[1] 64.5
```

2 missing, and the mean of the other 4 is 64.5 kg. The report should say “64.5 kg (n = 4)”.

Why the wrong version looks reasonable: 40 kg is a possible adult weight, “removing missing values” was written, and the line does remove `NA`s. It had none to remove. `dplyr::if_else()` calls the function from dplyr without loading the whole package, so it works on a plain vector here.

8.

```
class(camp$height_cm)
```

```
[1] "character"
```

`class()` says what kind of object R holds, here text. The words “not recorded” are text, so `read_csv()` read the whole column as text, `chr`. A mean of text is not defined: R warns and returns `NA`. Only one height is missing, not all.

Declare the code when reading, keeping the default missing values beside it:

```
camp <- read_csv(I("id,height_cm
1,131
2,not recorded
3,128"), na = c("", "NA", "not recorded"), show_col_types = FALSE)

sum(is.na(camp$height_cm))
mean(camp$height_cm, na.rm = TRUE)
```

```
[1] 1
[1] 129.5
```

One missing value, and the mean of the other two heights is 129.5 cm (n = 2).

Why the wrong conclusion looks reasonable: R printed `NA`, which usually means missing. Here it meant "this is not a number", and the warning above it said so.

9. `drop_na()` with no columns named dropped every row with a gap anywhere, mostly in `Comments`. The mean is of 34 rows, not of every row of known sex. Name the columns:

```
known <- drop_na(penguins_raw, Sex, `Body Mass (g)`)

nrow(known)
mean(known$`Body Mass (g)`)
```

```
[1] 333
[1] 4207.057
```

333 rows of known sex have a body mass, and their mean is 4207.057 g. The first answer was 329.851 g too low. The 34 rows with a comment and no other gap are not a fair picture of the rest.

Why the wrong answer looks reasonable: the code ran, the number is a believable penguin weight, and no `NA` appeared. Only `nrow(known)` gives it away.

10.

```
library(readr)
library(dplyr, warn.conflicts = FALSE)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)

penguins_raw |>
  summarise(
    rows = n(),
    n = sum(!is.na(`Body Mass (g)`)),
    mean_mass_g = mean(`Body Mass (g)`, na.rm = TRUE)
  )
```

```
# A tibble: 1 × 3
   rows     n mean_mass_g
  <int> <int>       <dbl>
1   344   342       4202.
```

The mean prints as 4202 g, so the number is right. The count is wrong: the mean is of 342 rows, because 2 have no body mass. Rewrite: "Mean body mass was 4202 g (n = 342; 2 of 344 not weighed)."

What it does not establish:

- What the birds in the 2 rows weighed, or why they were not weighed.
- Anything about penguins outside these three seasons and three islands.
- That 344 rows are 344 penguins. The file records nesting observations, and an earlier section showed that its IDs repeat.

11.

```
library(readr)
library(dplyr, warn.conflicts = FALSE)
library(tidyr)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)

cc <- drop_na(penguins_raw, `Body Mass (g)`, `Flipper Length (mm)`, Sex)

nrow(penguins_raw)
nrow(cc)
```

```
[1] 344
[1] 333
```

The complete-case analysis kept 333 of 344 rows and left out 11. That much the code can show, and the methods should say it.

What the sentence claims that the data cannot show:

- "Not a problem" means the 11 rows left out are like the 333 kept. A count cannot show that. It needs a reason why the values are missing, which this file does not record.
- The analysis did not deal with the missing data. It chose to leave those rows out, and that choice is part of the result. Better: "Rows missing body mass, flipper length or sex were excluded (11 of 344)."
- How to judge whether leaving rows out could bias a result is taught in a later book on missing data.

14 · Categories as factors

EXERCISE 1.

```
library(readr)
library(dplyr, warn.conflicts = FALSE)
library(stringr)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)

penguins <- penguins_raw |>
  mutate(sex = factor(str_to_lower(Sex), levels = c("female", "male")))

penguins |>
  count(sex)
```

```
# A tibble: 3 × 2
  sex        n
  <fct>  <int>
1 female   165
2 male     168
3 <NA>      11
```

165 female, 168 male, and 11 with no recorded sex. The file writes FEMALE and MALE in capitals, so the values are lowered first. Without `str_to_lower()`, `factor()` would match nothing, and all 344 rows would become `NA`. The 11 `NA`s here were already missing in the file, as the section on missing values counted.

EXERCISE 2. It turns the value into `NA`, with no warning. Catch it by running `count()` on the new factor and looking for an `NA` row that was not there before.

EXERCISE 3. Write the three codes on the board and ask what each means. They will have to look in the data dictionary, which is the point. The numbers are labels, and only the dictionary says what they stand for. Suppose it says 0 never, 1 former, 2 current.

Ask what "0.8" means. It is not a kind of tobacco use. It is an average of labels, like averaging bed numbers. A different coding, 1 to 3 instead of 0 to 2, would give a different "mean" for the same people.

Then write the fix. Turn the code into a factor with the dictionary's labels: `fct_recode(as.character(tobacco), never = "0", former = "1", current = "2")`. Keep the code column beside it. Report a count of each category with its share, not a mean.

1.

```
factor(c("Dream", "Biscoe", "Dream", "Torgersen"))
```

```
[1] Dream     Biscoe    Dream     Torgersen
Levels: Biscoe Dream Torgersen
```

The levels are Biscoe, Dream, Torgersen: with no levels given, `factor()` takes them from the data in alphabetical order.

2.

```
library(forcats)

fct_infreq(c("rice", "roti", "rice", "millet", "rice", "roti"))
```

```
[1] rice   roti   rice   millet rice   roti
Levels: rice roti millet
```

rice, roti, millet. Rice appears three times, roti twice and millet once, and `fct_infreq()` puts the most frequent first. The values themselves do not move.

3.

```
library(forcats)

fct_recode(as.character(meal_code), breakfast = "1", lunch = "2", dinner = "3")
```

```
[1] lunch     dinner    dinner    breakfast lunch
Levels: breakfast lunch dinner
```

lunch, dinner, dinner, breakfast, lunch. `fct_recode()` takes text, so the numbers are turned into text first. Each new label sits on the left of its old code.

4.

```
penguins_raw |>
  count(`Clutch Completion`)
```

```
# A tibble: 2 × 2
  `Clutch Completion`     n
  <chr>               <int>
1 No                     36
2 Yes                   308
```

36 rows say “No” and 308 say “Yes”, which add to 344. The help page describes the column as whether the study nest was “observed with a full clutch, i.e., 2 eggs”. So “No” means the nest was not seen with two eggs.

5.

```
penguins_raw |>
  mutate(island = fct_relevel(Island, "Torgersen")) |>
  count(island)
```

```
# A tibble: 3 × 2
  island        n
  <fct>     <int>
1 Torgersen    52
2 Biscoe      168
3 Dream       124
```

Torgersen 52, Biscoe 168, Dream 124. `fct_relevel()` moves the named level to the front. It leaves the rest in their existing order, which for a text column is alphabetical.

6. Two “Female” values have a capital letter, and one also has a space at the end. Neither matches the level “female”, so `factor()` turned both into `NA` without a warning. Two females became missing, and the colleague read only the female row.

Clean the text first:

```
library(stringr)

register |>
  mutate(sex = factor(str_to_lower(str_trim(sex)), levels = c("female", "male"))) |>
  count(sex)
```

```
# A tibble: 2 × 2
  sex        n
  <fct>  <int>
1 female     4
2 male       2
```

4 of the 6 were female.

Why the wrong report looks reasonable: the count printed a female row and a male row, the two categories that were expected. The `NA` row was below them, and nobody asked why a register with no gaps had two missing values.

7. The old name for Chinstrap was typed with a capital P, “Penguin”. The file writes “penguin”. The name did not match, so `fct_recode()` warned “Unknown levels” and left the level as it was. The warning said what happened.

Copy the old names from a `count()` of the column, not from memory:

```
penguins_raw |>
  mutate(
    species = fct_recode(
      Species,
      Adelie = "Adelie Penguin (Pygoscelis adeliae)",
      Chinstrap = "Chinstrap penguin (Pygoscelis antarctica)",
      Gentoo = "Gentoo penguin (Pygoscelis papua)"
    )
  ) |>
  count(species)
```

```
# A tibble: 3 × 2
  species       n
  <fct>     <int>
1 Adelie      152
2 Chinstrap    68
3 Gentoo      124
```

Adelie 152, Chinstrap 68, Gentoo 124.

Why the wrong conclusion looks reasonable: the counts were right and only a label was long, so it looked like a cosmetic problem. It was a typing error that the warning had already named.

8. The labels are swapped. The dictionary says 1 is male, and the code says 1 is female. Every participant has the wrong label, and the code ran without a warning, because R has no way to know what the codes mean.

```
study |>
  mutate(sex = fct_recode(as.character(sex_code), male = "1", female = "2")) |>
  count(sex)
```

```
# A tibble: 2 × 2
  sex        n
  <fct>  <int>
1 male       2
2 female     5
```

5 of the 7 were female and 2 were male.

Why the wrong answer looks reasonable: the code is tidy and the counts add up. A swap of labels changes no total. Only a line-by-line check against the dictionary catches it, so copy the labels from the dictionary into a comment beside the line.

9.

```
library(readr)
library(dplyr, warn.conflicts = FALSE)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)

penguins_raw |>
  count(Sex)
```

```
# A tibble: 3 × 2
  Sex        n
  <chr>  <int>
1 FEMALE   165
2 MALE     168
3 <NA>      11
```

168 MALE and 165 FEMALE make 333 with sex recorded, and 11 more have none. The sentence matches the file's rows. 168 of 333 is 50.5 per cent.

What it does not establish:

- That the colony is about half male. The rows are nesting observations of birds that were sampled, not a count of every penguin.
- Anything about the birds in the 11 rows with no recorded sex.
- That each row is a different bird. The file's IDs repeat, as an earlier section showed.

10.

```
library(readr)
library(dplyr, warn.conflicts = FALSE)
library(forcats)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)

penguins_raw |>
  mutate(island = factor(Island)) |>
  filter(Species == "Gentoo penguin (Pygoscelis papua)") |>
  count(island, .drop = FALSE)
```

```
# A tibble: 3 × 2
  island        n
  <fct>     <int>
1 Biscoe      124
2 Dream         0
3 Torgersen     0
```

All 124 Gentoo rows are from Biscoe, with 0 from Dream and 0 from Torgersen. Making the island a factor first, with `.drop = FALSE`, keeps the zeros in view.

What it establishes: in this file, every Gentoo row was recorded on Biscoe.

What it does not establish:

- Where Gentoo penguins live. The file covers three islands near one research station, and only the nests the study chose to sample. A zero means none was recorded, not none exists.
- Why. Whether the study went looking for Gentoo on Dream and Torgersen, the file does not say.
- The claim in the meeting moved from "were recorded only on" to "live only on". That move is the error.

15 · Dates

EXERCISE 1.

```
library(readr)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)

class(penguins_raw$`Date Egg`)
sum(is.na(penguins_raw$`Date Egg`))
```

```
[1] "Date"
[1] 0
```

The column is already a `Date`, because the file writes it year-month-day. No date is missing, so `min()` and `max()` on it need no `na.rm`.

EXERCISE 2. `dmy()`, because the day comes first. `mdy()` would read "03/04/2021" as month 3, day 4: 4 March 2021. That is a real date, and the wrong one if the register meant 3 April.

EXERCISE 3. One possible answer.

Variable	Label	Type	Format	Missing value code
dob	date of birth	date	YYYY-MM-DD, for example 2014-06-21	empty cell
measure_date	date measured	date	YYYY-MM-DD	empty cell

The script reads the file with `read_csv()`, which recognises year-month-day dates by itself. It checks that both columns came in as dates, and counts any missing. Then it computes age at measurement from `measure_date - dob`, in completed years with `floor()`. Nobody types an age: the age is computed in code, so it can be checked.

Writing the format in the dictionary, with an example, is the step that stops a data-entry operator from typing 21/06/2014.

1.

```
library(lubridate, warn.conflicts = FALSE)

d <- ymd("2023-11-14")
d
class(d)
```

```
[1] "2023-11-14"
[1] "Date"
```

The year comes first, so `ymd()`. The result prints as 2023-11-14 and its class is `Date`.

2.

```
library(lubridate, warn.conflicts = FALSE)

dmy("05/06/2022")
mdy("05/06/2022")
```

```
[1] "2022-06-05"
[1] "2022-05-06"
```

`dmy()` gives 5 June 2022 and `mdy()` gives 6 May 2022. The same text, two dates a month apart, and both look right.

3.

```
library(lubridate, warn.conflicts = FALSE)

ymd("2024-03-01") - ymd("2024-02-01")
```

```
Time difference of 29 days
```

29 days. 2024 is a leap year, so February 2024 has 29 days. Date arithmetic counts the calendar, not 30 days a month.

4.

```
library(lubridate, warn.conflicts = FALSE)

ymd("2008-11-02") - min(penguins_raw$`Date Egg`)
```

```
Time difference of 359 days
```

The first nest in the file was observed on 2007-11-09, so 359 days separate the two. The second season's date is typed as text and parsed with `ymd()`; the file's own date is already a date.

5.

```
library(lubridate, warn.conflicts = FALSE)

penguins_raw |>
  filter(
    `Date Egg` >= ymd("2008-11-01"),
    `Date Egg` <= ymd("2008-11-30")
  ) |>
  nrow()
```

```
[1] 114
```

114 rows. "On or after 1 November and on or before 30 November" is two comparisons, as `filter()` would write for any number.

6.

```
children |>
  mutate(
    age_years = as.numeric(measured - dob) / 365.25,
    age_completed = floor(age_years)
  )
```

```
# A tibble: 2 × 5
  id    dob        measured   age_years age_completed
  <chr> <date>     <date>         <dbl>         <dbl>
1 C1    2014-06-21 2024-06-20      10.0             9
2 C2    2013-12-05 2024-06-20      10.5            10
```

C1 is 9 in completed years: measured on 20 June, a day before the 10th birthday. The printed 10.0 is rounded to fit; the stored age is a little under 10. C2 is 10. A report that subtracted years, 2024 minus 2014, would have made C1 10.

7. The dates are not missing. They were read in the wrong order. `ymd()` expects the year first, nothing fitted, and every value became `NA`. The warning says no format was found.

```
dmy(visits)
```

```
[1] "2022-01-12" "2022-01-19" "2022-01-26"
```

12, 19 and 26 January 2022.

Why the wrong report looks reasonable: R prints `NA`, which is R's word for missing. Here it meant "could not be read". The warning above it said which.

8. `mdy()` read the text month first. "02/03/2023" became 3 February 2023, a real date and the wrong one. "17/03/2023" failed, because there is no 17th month. The middle date, "11/03/2023", became 3 November 2023, also wrong. Nothing was missing.

```
dmy(screened)
```

```
[1] "2023-03-02" "2023-03-11" "2023-03-17"
```

2, 11 and 17 March 2023.

Why the wrong report looks reasonable: two of the three dates came out as dates. The one `NA` looked like a gap in the register. It was the only value whose day was above 12, and so the only one that could have shown the order was wrong.

9. The start date is a day late for this kind of file. The resident trusted a starting date instead of checking against the one date they knew. Start from 30 December 1899:

```
ymd("1899-12-30") + c(45292, 45306)
```

```
[1] "2024-01-01" "2024-01-15"
```

1 January and 15 January 2024. The first matches what the colleague remembers, so the second is right too.

Why the wrong answer looks reasonable: 2 January 2024 is a real date, near the right one, and nothing failed. Only the known date gave it away. Always keep one.

10.

```
clinic <- clinic |>
  mutate(days = as.numeric(visited - screened))

clinic$days
median(clinic$days)
```

```
[1] 11 18 14 28 14
[1] 14
```

The gaps are 11, 18, 14, 28 and 14 days. In order, 11, 14, 14, 18, 28; the middle value is 14. The claim matches the dates.

What it does not establish:

- That 14 days is typical for anyone else. Five patients are five patients.
- Anything about patients who were screened and never came. They have no visit date, so they are not in this table, and the median says nothing about them.
- Why the gaps differ. The longest, 28 days, may be a clinic closure or a patient's own choice; the dates cannot tell you.

11.

```
library(readr)
library(dplyr, warn.conflicts = FALSE)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)

season <- penguins_raw |>
  filter(studyName == "PAL0809")

nrow(season)
min(season$`Date Egg`)
max(season$`Date Egg`)
```

```
[1] 114
[1] "2008-11-02"
[1] "2008-11-25"
```

The 114 rows of that season run from 2008-11-02 to 2008-11-25. Every date is in November 2008.

What the file cannot support:

- That the egg was laid on that date. The help page describes `Date Egg` as the date the study nest was "observed with 1 egg". An egg may have been laid before the visit.
- "All penguins". Each row is one bird at a study nest, not every penguin in the colony.
- A fair rewrite: "In the 2008–09 season, all 114 rows are of study nests first observed with one egg between 2 and 25 November 2008."

16 · Summaries by group

EXERCISE 1.

```
library(readr)
library(dplyr, warn.conflicts = FALSE)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)

flipper_by_island <- penguins_raw |>
  summarise(
    n_rows = n(),
    n_flipper = sum(!is.na(`Flipper Length (mm)`)),
    mean_mm = mean(`Flipper Length (mm)`, na.rm = TRUE),
    median_mm = median(`Flipper Length (mm)`, na.rm = TRUE),
    .by = Island
  )

flipper_by_island

dir.create(here::here("output"), showWarnings = FALSE)
write_csv(flipper_by_island, here::here("output", "flipper_by_island.csv"))
```

```
# A tibble: 3 × 5
  Island    n_rows n_flipper mean_mm median_mm
  <chr>      <int>     <int>   <dbl>     <dbl>
1 Torgersen     52        51    191.       191
2 Biscoe       168       167    210.       214
3 Dream        124       124    193.       193
```

Three islands, three rows. Torgersen's mean rests on the fewest flipper lengths: 51 of its 52 rows have one. Biscoe and Torgersen each lost one penguin to a missing flipper length; Dream lost none. The file in `output/` holds the full numbers, and rerunning the script rewrites it.

EXERCISE 2. Two things can be wrong, and nobody can now tell whether they are.

The n may not be the n behind the mean. The function `nrow()` counts every woman in the data frame. `na.rm = TRUE` left out any woman with no haemoglobin result. If 20 results were missing, the mean came from 292 women and the table says 312.

The number has no trail. It was read off a screen and typed. If a value in the data is later corrected, or a typing slip turned 11.2 into 12.1, nothing connects the table to the code.

Instead, the script should compute the summary with `summarise()`: `n()` for the rows, `sum(!is.na(hb))` for the results used, and the mean and median. It should write that table to `output/` with `write_csv()`. The thesis table is then copied from that file, which the script rewrites every time it runs. A later book shows documents in which the code writes each number itself.

EXERCISE 3. Draw two made-up rows on the board. District A: mean BMI 24.0 kg/m^2. District B: mean BMI 27.0 kg/m^2. Ask her which district has the higher BMI. She will say B.

Now add a count to each row: A, 2,400 adults; B, 4 adults. Ask again. The difference is the same three units, but B's mean is four people. One person of high BMI who happened to come that day could have made it.

Then add a second count: values used. Say B screened 40 adults but only 4 had weight and height recorded. Now the mean describes the 4 who were measured, and nobody knows whether they are like the other 36.

Write the rule under the table. Every summary row carries the number of rows and the number of values used. In R that is `n()` and `sum(!is.na(x))`, inside the same `summarise()` as the mean. Then the counts can never drift away from the number they describe.

EXERCISE 4. Any three of these:

- The count printed is `n()`, the rows, while the mean used fewer values because `na.rm = TRUE` left the missing ones out.
- A percentage worked out after `group_by()` on two columns is a share within each group, not of the total, because the result was still grouped.
- A quartile differs from another tool's because the two use different rules.
- A group of `NA`, such as penguins with no sex recorded, is dropped from the report without a word, though the summary printed it.
- A number was copied off the screen and later typed wrongly, or not updated when the data changed.

1.

```
scores |>
  summarise(n = n(), mean_score = mean(score))
```

```
# A tibble: 1 × 2
      n mean_score
  <int>      <dbl>
1     3          6
```

One row: 3 rows, and a mean of 6, since 4 plus 9 plus 5 is 18, and 18 divided by 3 is 6.

2. Two clinics, so two rows.

```
visits |>
  summarise(total = sum(patients), .by = clinic)
```

```
# A tibble: 2 × 2
  clinic total
  <chr>  <dbl>
1 A         27
2 B         18
```

Clinic A: 12 plus 9 plus 6 is 27. Clinic B: 7 plus 11 is 18.

3. Book 0: the lower half is 2 and 4, so the first quartile is halfway between them, 3. The upper half is 6 and 8, so the third quartile is 7. The interquartile range is 7 minus 3, which is 4.

```
quantile(c(2, 4, 6, 8))
IQR(c(2, 4, 6, 8))
```

```
  0%  25%  50%  75% 100%
 2.0  3.5  5.0  6.5  8.0
[1] 3
```

R gives 3.5 and 6.5, and an interquartile range of 3. R's default, type 7, places a quartile by a different rule from Book 0's. Neither is a mistake. On four values the rules are far apart; on hundreds they land close together.

4.

```
penguins_raw |>
  summarise(
    n_rows = n(),
    n_species = n_distinct(Species),
    .by = Island
  )
```

```
# A tibble: 3 × 3
  Island    n_rows n_species
  <chr>      <int>     <int>
1 Torgersen     52         1
2 Biscoe       168         2
3 Dream        124         2
```

Torgersen: 52 rows, 1 species. Biscoe: 168 rows, 2 species. Dream: 124 rows, 2 species. The three counts of rows add up to 344, every row in the file. Only Adelie penguins were sampled on all three islands.

5.

```
penguins_raw |>
  summarise(
    n_rows = n(),
    n_mass = sum(!is.na(`Body Mass (g)`)),
    median_g = median(`Body Mass (g)`, na.rm = TRUE),
    .by = Sex
  )
```

```
# A tibble: 3 × 4
  Sex    n_rows n_mass median_g
  <chr>   <int>  <int>    <dbl>
1 MALE      168    168     4300
2 FEMALE    165    165     3650
3 <NA>       11      9     4100
```

Males: 168 rows, median 4300 g. Females: 165 rows, median 3650 g. The third row is `NA`: 11 penguins whose sex was not recorded, 9 of them with a body mass. A summary does not drop a missing category; it prints it as a group. Say how many penguins had no sex recorded, and do not quietly leave them out of the table.

6.

```
culmen_by_species <- penguins_raw |>
  summarise(
    n_culmen = sum(!is.na(`Culmen Length (mm)`)),
    mean_mm = mean(`Culmen Length (mm)`, na.rm = TRUE),
    sd_mm = sd(`Culmen Length (mm)`, na.rm = TRUE),
    .by = Species
  )

dir.create(here::here("output"), showWarnings = FALSE)
write_csv(culmen_by_species, here::here("output", "culmen_by_species.csv"))

file.exists(here::here("output", "culmen_by_species.csv"))
length(readLines(here::here("output", "culmen_by_species.csv")))
```

```
[1] TRUE
[1] 4
```

The file exists, with 4 lines: a header and one line per species. The function `sd()` divides by one less than the count, as Book 0 does for a sample. The numbers in the file are the ones a report should quote.

7. Nothing is wrong with the data. `mean()` returns `NA` if even one value in the group is missing, unless told to leave missing values out. Adelie and Gentoo each have one missing body mass; Chinstrap has none. Count, then summarise:

```
penguins_raw |>
  summarise(
    n_rows = n(),
    n_missing = sum(is.na(`Body Mass (g)`)),
    mean_g = mean(`Body Mass (g)`, na.rm = TRUE),
    .by = Species
  )
```

```
# A tibble: 3 × 4
  Species                                n_rows n_missing mean_g
  <chr>                                   <int>     <int>  <dbl>
1 Adelie Penguin (Pygoscelis adeliae)       152         1  3701.
2 Gentoo penguin (Pygoscelis papua)         124         1  5076.
3 Chinstrap penguin (Pygoscelis antarctica)  68         0  3733.
```

Adelie: 1 missing of 152. Gentoo: 1 missing of 124. Both have a mean.

Why the wrong reading looks reasonable: `NA` means "not available", and the note took that to mean the whole column was not available. It meant one value was.

8. The n counts rows, not body masses. One Gentoo penguin has no body mass, so the mean was worked out from 123 values, not 124.

```
penguins_raw |>
  filter(Species == "Gentoo penguin (Pygoscelis papua)") |>
  summarise(
    n_rows = n(),
    n_mass = sum(!is.na(`Body Mass (g)`)),
    mean_g = mean(`Body Mass (g)`, na.rm = TRUE)
  )
```

```
# A tibble: 1 × 3
  n_rows n_mass mean_g
   <int>  <int>  <dbl>
1    124    123  5076.
```

The table should read "n = 123" for the mean, and say that 1 of 124 had no body mass.

Why the wrong n looks reasonable: `n()` is the count every example shows beside a mean. With no missing values it is the right one. The error only appears in a group with a gap.

9. After `summarise()`, the result is still grouped by `Species`: the message says so. So `sum(n)` adds up within each species. 61 Gentoo males are 49.2% of 124 Gentoo penguins, not of all 344.

Count instead, which returns an ungrouped result:

```
penguins_raw |>
  count(Species, Sex) |>
  mutate(pct = 100 * n / sum(n)) |>
  filter(Sex == "MALE")
```

```
# A tibble: 3 × 4
  Species                                   Sex       n   pct
  <chr>                                     <chr> <int> <dbl>
1 Adelie Penguin (Pygoscelis adeliae)       MALE     73 21.2
2 Chinstrap penguin (Pygoscelis antarctica) MALE     34  9.88
3 Gentoo penguin (Pygoscelis papua)         MALE     61 17.7
```

Gentoo males are 17.7% of the 344 rows in the file. `count()` returns an ungrouped result, so `sum(n)` is 344.

Why the wrong answer looks reasonable: 49.2% is a sensible share for males in one species, so nothing about the number looks off. Only the denominator changed, and the denominator is not printed.

10. The file's column is `Species`, with a capital S. R tells capitals from small letters, so `.by = species` names a column that does not exist yet. The short names have to be made before the summary groups by them:

```
penguins_raw |>
  mutate(
    species = case_when(
      Species == "Adelie Penguin (Pygoscelis adeliae)" ~ "Adelie",
      Species == "Chinstrap penguin (Pygoscelis antarctica)" ~ "Chinstrap",
      Species == "Gentoo penguin (Pygoscelis papua)" ~ "Gentoo"
    )
  ) |>
  summarise(
    mean_g = mean(`Body Mass (g)`, na.rm = TRUE),
    .by = species
  )
```

```
# A tibble: 3 × 2
  species   mean_g
  <chr>      <dbl>
1 Adelie     3701.
2 Gentoo     5076.
3 Chinstrap  3733.
```

Three rows, one per short name. A summary keeps only the grouping columns and the summaries, so make every column you will group by before `summarise()`, not after.

Why the wrong code looks reasonable: the plan, "summarise, then rename", is fine in words. But after `summarise()` the long `Species` column would be the only one left to rename, and its name is not the one the code used.

11.

```
library(readr)
library(dplyr, warn.conflicts = FALSE)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)

penguins_raw |>
  filter(Species != "Chinstrap penguin (Pygoscelis antarctica)") |>
  summarise(
    n_rows = n(),
    n_mass = sum(!is.na(`Body Mass (g)`)),
    mean_g = mean(`Body Mass (g)`, na.rm = TRUE),
    median_g = median(`Body Mass (g)`, na.rm = TRUE),
    .by = Species
  )
```

```
# A tibble: 2 × 5
  Species                             n_rows n_mass mean_g median_g
  <chr>                                <int>  <int>  <dbl>    <dbl>
1 Adelie Penguin (Pygoscelis adeliae)    152    151  3701.     3700
2 Gentoo penguin (Pygoscelis papua)      124    123  5076.     5000
```

The means match, once rounded: 3701 g for Adelie and 5076 g for Gentoo. The counts do not describe the means. Each n is the number of rows; each mean used one value fewer, 151 and 123. The sentence should give those, or say that one body mass was missing in each species. The medians, 3700 g and 5000 g, tell the same story as the means.

What it does not establish:

- That Gentoo penguins in general are heavier. These are the penguins in one study, on certain islands in certain seasons.
- How large the difference is against the spread within each species, or whether it could be chance. That needs the standard deviations and a method from the statistics books.
- Why they differ. Species is not the only thing that differs between the two groups: they were also sampled on different islands.

12. Body mass is in grams, so 4.5 kg is 4500 g. Summarise by sex, with counts:

```
library(readr)
library(dplyr, warn.conflicts = FALSE)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)

penguins_raw |>
  summarise(
    n_mass = sum(!is.na(`Body Mass (g)`)),
    mean_g = mean(`Body Mass (g)`, na.rm = TRUE),
    .by = Sex
  )

penguins_raw |>
  filter(Sex == "MALE") |>
  summarise(
    n_mass = sum(!is.na(`Body Mass (g)`)),
    mean_g = mean(`Body Mass (g)`, na.rm = TRUE),
    .by = Species
  )
```

```
# A tibble: 3 × 3
  Sex    n_mass mean_g
  <chr>   <int>  <dbl>
1 MALE      168  4546.
2 FEMALE    165  3862.
3 <NA>        9  4006.
# A tibble: 3 × 3
  Species                                   n_mass mean_g
  <chr>                                      <int>  <dbl>
1 Adelie Penguin (Pygoscelis adeliae)           73  4043.
2 Gentoo penguin (Pygoscelis papua)             61  5485.
3 Chinstrap penguin (Pygoscelis antarctica)     34  3939.
```

The 168 males average 4546 g, about 4.5 kg, so the sentence matches the file. Now look inside it. Adelie males average 4043 g and Chinstrap males 3939 g; Gentoo males average 5485 g. No species has males that average about 4.5 kg. The pooled mean sits between a group of lighter birds and a group of heavier ones, and it describes almost no actual penguin.

What it does not establish:

- Anything about a typical male of any species. Grouping by sex alone mixed three species of quite different size.
- Anything about the 11 penguins with no sex recorded, who are in neither group.
- Anything about penguins outside this study.

17 · Reshaping and joining

EXERCISE 1.

```
library(dplyr, warn.conflicts = FALSE)

bmx <- haven::read_xpt(here::here("data-raw", "BMX_L.xpt"))
demo <- haven::read_xpt(here::here("data-raw", "DEMO_L.xpt"))

body <- bmx |>
  left_join(demo, by = "SEQN", relationship = "one-to-one")

nrow(bmx)
nrow(demo)
nrow(body)
nrow(anti_join(bmx, demo, by = "SEQN"))
nrow(filter(body, RIDAGEYR >= 20))
```

```
[1] 8860
[1] 11933
[1] 8860
[1] 0
[1] 6064
```

8860 rows of body measures and 11933 of demographics. The join has 8860 rows: none was added, and `relationship = "one-to-one"` confirms no key repeats. No body measures row lacks demographics. 6064 joined rows are adults aged 20 or over. Every count is of rows in the files; none is a survey estimate.

EXERCISE 2. Pasting side by side matches rows by position, not by ID. It is right only if both files hold exactly the same participants, each exactly once, sorted the same way. One participant missing from the laboratory file, or entered twice, shifts every row below by one. Every participant below the slip then carries someone else's results, and nothing in the file shows it.

The merged file also has no record of how it was made. Nobody can redo the step and check it.

Do it in code instead, in four steps.

1. Read both raw files, and count each ID in each with `count(id) |> filter(n > 1)`.
2. Join with `left_join(questionnaire, lab, by = "id", relationship = "one-to-one")`.
3. Compare the row counts before and after.
4. List the unmatched participants with `anti_join()`, both ways round.

The raw files stay as they arrived. The merge reruns whenever either file changes.

EXERCISE 3. What each returns:

- `pivot_longer()`: the same data with more rows and fewer columns; a set of columns becomes a column of their names and a column of their values.
- `pivot_wider()`: the reverse, with fewer rows and more columns.
- `left_join(x, y)`: every row of `x`, with the columns of `y` added where the keys match; `NA` where they do not; a row repeated when its key matches several rows of `y`.
- `anti_join(x, y)`: only the rows of `x` whose key has no match in `y`.

1.

```
screen |>
  pivot_longer(
    cols = starts_with("height_"),
    names_to = "year",
    names_prefix = "height_",
    values_to = "height_cm"
  )
```

```
# A tibble: 2 × 3
  child year  height_cm
  <chr> <chr>     <dbl>
1 A     2024        121
2 A     2025        127
```

Two rows: one child, two years. The `year` column holds text, "2024" and "2025", because it came from column names.

2.

```
long |>
  pivot_wider(names_from = round, values_from = screened)
```

```
# A tibble: 2 × 3
  district    r1    r2
  <chr>    <dbl> <dbl>
1 Durg       410   455
2 Korba      380   362
```

Two rows and three columns: Durg screened 410 and 455; Korba 380 and 362.

3.

```
left_join(x, y, by = "id")
```

```
# A tibble: 4 × 3
     id weight_kg ward
  <dbl>     <dbl> <chr>
1     1        60 <NA>
2     2        72 A
3     3        55 B
4     3        55 C
```

Four rows. Id 1 has no match, so it stays once with `NA` for `ward`. Id 2 matches once, ward A. Id 3 matches two rows of `y`, so it appears twice, with wards B and C. Id 4 is only in `y`, and a left join keeps only the rows of `x`. Three rows went in; four came out.

4.

```
bmx |>
  count(SEQN) |>
  filter(n > 1) |>
  nrow()

demo |>
  count(SEQN) |>
  filter(n > 1) |>
  nrow()
```

```
[1] 0
[1] 0
```

0 in each. No participant number repeats in either file, so a join on `SEQN` cannot add rows.

5.

```
penguins_raw |>
  count(Species, Island) |>
  pivot_wider(
    names_from = Island,
    values_from = n,
    values_fill = 0
  )
```

```
# A tibble: 3 × 4
  Species                                   Biscoe Dream Torgersen
  <chr>                                      <int> <int>     <int>
1 Adelie Penguin (Pygoscelis adeliae)           44    56        52
2 Chinstrap penguin (Pygoscelis antarctica)      0    68         0
3 Gentoo penguin (Pygoscelis papua)            124     0         0
```

Three species, three islands. Adelie penguins were found on all three; Chinstrap only on Dream; Gentoo only on Biscoe. Without `values_fill = 0`, an island where a species was never recorded would show `NA`. Here 0 is right: the count of rows for that pair is truly zero, not unknown. In a table of weights, filling a gap with 0 would be wrong, because a missing weight is not a weight of zero.

6.

```
body <- bmx |>
  left_join(
    demo |> select(SEQN, RIDAGEYR),
    by = "SEQN",
    relationship = "one-to-one"
  )

body |>
  filter(RIDAGEYR < 2) |>
  summarise(
    n_rows = n(),
    n_height = sum(!is.na(BMXHT))
  )
```

```
# A tibble: 1 × 2
  n_rows n_height
   <int>    <int>
1    249        0
```

The join keeps all 8860 rows. Of the 249 examined participants under 2, none has a standing height. The survey's documentation lists standing height for ages 2 years and older. Recumbent length, measured lying down, is taken from birth through 47 months, in its own column. So standing height is missing on purpose. A missing value can be part of how the survey works, not a failure to measure.

7. The three folded columns are three different variables, in two units: two lengths in millimetres and a mass in grams. The mean of the `value` column adds grams to millimetres. The number is meaningless, and the large body masses swamp the lengths.

Keep `measure` in the grouping, so each mean is of one variable in one unit:

```
penguins_raw |>
  pivot_longer(
    cols = c(`Culmen Length (mm)`, `Flipper Length (mm)`, `Body Mass (g)`),
    names_to = "measure",
    values_to = "value"
  ) |>
  summarise(mean_value = mean(value, na.rm = TRUE), .by = c(Species, measure)) |>
  filter(Species == "Gentoo penguin (Pygoscelis papua)")
```

```
# A tibble: 3 × 3
  Species                          measure             mean_value
  <chr>                            <chr>                    <dbl>
1 Gentoo penguin (Pygoscelis papua) Culmen Length (mm)        47.5
2 Gentoo penguin (Pygoscelis papua) Flipper Length (mm)      217.
3 Gentoo penguin (Pygoscelis papua) Body Mass (g)           5076.
```

Three means for Gentoo penguins, each with its unit in the measure's name.

Why the wrong answer looks reasonable: `pivot_longer()` ran without complaint and the result looked tidy. But a tidy column holds one variable. These three were never one variable, so folding them made the data less tidy, not more.

8. No `by` was given, so the join matched on every column the two tables share: `id` and `year`. The message says so: it joined by `id` and `year`. A visit in 2025 never matches a birth in 1980, so no row found a partner, and `sex` is `NA` everywhere.

Name the key:

```
visits |>
  left_join(register, by = "id")
```

```
# A tibble: 3 × 5
     id year.x   bmi year.y sex
  <dbl>  <dbl> <dbl>  <dbl> <chr>
1     1   2025  26.2   1980 F
2     2   2025  31     1972 M
3     3   2026  23.4   1991 F
```

Now each visit gets its patient's sex. The two `year` columns are both kept, and the join adds `.x` and `.y` to their names to tell them apart. Rename them to what they mean, such as `visit_year` and `birth_year`.

Why the wrong answer looks reasonable: "sex is not recorded" is a common problem in real registers, so the colleague believed it. The message naming both columns was the clue.

9. The key is a number in `register` and text in `lab`. R will not match `101` with `"101"`, because the two columns have different types. Convert in code, in a new column or by re-reading the file with its column types stated, never by editing the file:

```
lab_clean <- lab |>
  mutate(id = as.numeric(id), hb = as.numeric(hb))

register |>
  left_join(lab_clean, by = "id")
```

```
# A tibble: 3 × 3
     id sex      hb
  <dbl> <chr> <dbl>
1   101 F      12.1
2   102 M      14.3
3   103 F      11.8
```

Three rows, each with its haemoglobin. The stopping error was the helpful case. Two text columns that differ in another way, such as IDs with and without a leading zero typed as text, would join without error. They would fail to match, and nothing would say so.

Why the wrong code looks reasonable: the two IDs print the same. The type is only visible in the column heading, `<dbl>` against `<chr>`.

10.

```
library(dplyr, warn.conflicts = FALSE)

bmx <- haven::read_xpt(here::here("data-raw", "BMX_L.xpt"))
demo <- haven::read_xpt(here::here("data-raw", "DEMO_L.xpt"))

nrow(demo)
nrow(bmx)

bmx |>
  left_join(demo, by = "SEQN", relationship = "one-to-one") |>
  nrow()
```

```
[1] 11933
[1] 8860
[1] 8860
```

11933 is the number of rows in the demographics file. It is not the number with body measures. 8860 participants have a body measures record, and the merged file has 8860 rows. The other 3073 were interviewed and not examined. The sentence took the wrong file's count, which is what happens when the count is read off a screen rather than printed after the join.

What it does not establish:

- That the 8860 have every measurement. Height, weight and BMI each have missing values, and each analysis has its own n.
- Anything about the US population. The counts are rows of the files; estimates need the survey's sample weights.

11. Join, keep adults, and summarise BMI by `RIAGENDR`, where the codebook says 1 is male and 2 is female:

```
library(dplyr, warn.conflicts = FALSE)

bmx <- haven::read_xpt(here::here("data-raw", "BMX_L.xpt"))
demo <- haven::read_xpt(here::here("data-raw", "DEMO_L.xpt"))

bmx |>
  left_join(demo, by = "SEQN", relationship = "one-to-one") |>
  filter(RIDAGEYR >= 20) |>
  summarise(
    n_bmi = sum(!is.na(BMXBMI)),
    mean_bmi = mean(BMXBMI, na.rm = TRUE),
    median_bmi = median(BMXBMI, na.rm = TRUE),
    .by = RIAGENDR
  )
```

```
# A tibble: 2 × 4
  RIAGENDR n_bmi mean_bmi median_bmi
     <dbl> <int>    <dbl>      <dbl>
1        1  2680     29.3       28.1
2        2  3290     30.3       28.7
```

In these rows, the 3290 women have a mean BMI of 30.3 kg/m^2, and the 2680 men 29.3 kg/m^2. The medians are 28.7 and 28.1 kg/m^2. So the women examined have a higher mean and median BMI than the men examined.

What it does not establish:

- That women in the United States have a higher BMI than men. These summaries are unweighted. The survey's documentation says its sample weights should be used, and the weighted answer could differ.
- That a difference of about one unit matters, or is more than chance. That needs the statistics books.
- Anything about body fat. BMI is weight for height; it does not measure body fat.

18 · Plotting with the grammar of graphics

EXERCISE 1.

```
library(readr)
library(dplyr, warn.conflicts = FALSE)
library(ggplot2)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)

flipper_plot <- ggplot(penguins_raw, aes(x = `Flipper Length (mm)`)) +
  geom_histogram(binwidth = 10, boundary = 170, closed = "left") +
  labs(
    x = "Flipper length (mm)",
    y = "Penguins",
    caption = "Source: palmerpenguins raw data, as read from data-raw/"
  )

dir.create(here::here("output"), showWarnings = FALSE)
ggsave(
  here::here("output", "flipper_histogram.png"),
  plot = flipper_plot,
  width = 15,
  height = 10,
  units = "cm"
)

flipper_counts <- penguins_raw |>
  count(bin_start_mm = floor(`Flipper Length (mm)` / 10) * 10)

flipper_counts
write_csv(flipper_counts, here::here("output", "flipper_histogram_counts.csv"))
```

```
Warning message:
Removed 2 rows containing non-finite values (`stat_bin()`).
# A tibble: 8 × 2
  bin_start_mm     n
         <dbl> <int>
1          170     8
2          180    69
3          190   113
4          200    38
5          210    71
6          220    35
7          230     8
8           NA     2
```

The warning says 2 rows were removed, so the figure shows 342 rows. The counts per bin are in the CSV file, beside the figure they describe; the NA row holds the 2 with no flipper length.

EXERCISE 2. Nothing connects each saved image to the code and data that made it. The console keeps no record of which lines made figure 7, and the size she dragged to is written nowhere. After a correction she must remember, for each of eleven figures, which code made it and at what size, and get all of them right by hand.

Instead, put every figure in the analysis script. Make it with `ggplot()` and give it a name. Then save it with `ggsave()` into `output/`, with its file name, width, height and units in the code. After a correction she reruns the script once. All eleven figures are redrawn from the corrected data, at the same size. Keep the counts behind each figure in a file beside it, so a reader can check the numbers without measuring the picture.

EXERCISE 3. "A bar is read by its length, so the bars only tell the truth if they start at zero. This axis starts at 9,000, so the bottom of every bar has been cut off. A district with 10,000 adults screened and one with 9,200 would look like a tall bar and a stub, even though the numbers are close. Ask for the chart redrawn from zero, or simply ask for the numbers. If the press office made the chart in code, redrawing it is one line."

EXERCISE 4. The data, the aesthetic mappings made with `aes()`, and the geom.

The bin width and where the bins start (`binwidth` and `boundary`). Which edge each bin includes (`closed`) changes the bars too, when values sit exactly on edges.

1. The data are `clinic`. There are two mappings: `age_years` to position along x, and `bmi` to position along y. The geom is `geom_point()`, so each row becomes one point. The line draws a scatter plot of BMI against age.

2. In the first, colour is mapped to the column `sex`, inside `aes()`. Each value of `sex` gets its own colour, and a legend says which is which.

In the second, colour is set, outside `aes()`. Every point is dark green, and there is no legend, because nothing varies.

3. By default a bin includes its right edge. The first bin is "above 1, up to 2", and the others follow the same pattern. The first holds 1.2 and 1.8: 2 values. The second holds 2.1, 2.9 and 3.0: 3 values. The third holds 3.4 alone: 1 value.

With `closed = "left"` the bins are "1 up to under 2", "2 up to under 3" and "3 up to 4". The first holds 1.2 and 1.8: 2 values. The second holds 2.1 and 2.9: 2 values. The value 3.0 moves to the third, with 3.4: 2 values.

Only the value sitting exactly on an edge moved. In real data, measurements are often rounded to whole numbers, so many values sit on edges, and the choice changes the bars.

4.

```
ggplot(penguins_raw, aes(x = Island, y = `Body Mass (g)`)) +
  geom_boxplot() +
  labs(x = "Island", y = "Body mass (g)")

penguins_raw |>
  summarise(
    n_mass = sum(!is.na(`Body Mass (g)`)),
    q1_g = quantile(`Body Mass (g)`, probs = 0.25, na.rm = TRUE),
    median_g = median(`Body Mass (g)`, na.rm = TRUE),
    q3_g = quantile(`Body Mass (g)`, probs = 0.75, na.rm = TRUE),
    .by = Island
  )
```

```
Warning message:
Removed 2 rows containing non-finite values (`stat_boxplot()`).
# A tibble: 3 × 5
  Island    n_mass  q1_g median_g  q3_g
  <chr>      <int> <dbl>    <dbl> <dbl>
1 Torgersen     51 3338.    3700  4000
2 Biscoe       167 4200     4775  5325
3 Dream        124 3400     3688. 3956.
```

The warning says 2 rows were removed: two penguins with no body mass. Biscoe's box sits highest, with a median of 4775 g from 167 body masses. The boxes use the same quartiles as `quantile()`. Biscoe is the only island with Gentoo penguins, the heaviest species, so the island comparison is partly a species comparison.

5.

```
culmen_plot <- ggplot(
  penguins_raw,
  aes(x = `Culmen Length (mm)`, y = `Culmen Depth (mm)`)
) +
  geom_point() +
  facet_wrap(~Species) +
  labs(
    x = "Culmen length (mm)",
    y = "Culmen depth (mm)",
    caption = "Source: palmerpenguins raw data, as read from data-raw/"
  )

dir.create(here::here("output"), showWarnings = FALSE)
ggsave(
  here::here("output", "culmen.png"),
  plot = culmen_plot,
  width = 18,
  height = 8,
  units = "cm"
)

file.exists(here::here("output", "culmen.png"))
```

```
Warning message:
Removed 2 rows containing missing values (`geom_point()`).
[1] TRUE
```

The file exists. The plot was never shown on the screen, but `ggsave()` drew it to make the file. Its warning says 2 rows were removed: the penguins with no culmen measurements are not in the saved figure. Say so in the caption if the figure goes in a report.

6.

```
ggplot(penguins_raw, aes(x = Sex)) +
  geom_bar() +
  labs(x = "Sex as recorded", y = "Rows in the file")

penguins_raw |>
  count(Sex)
```

```
# A tibble: 3 × 2
  Sex        n
  <chr>  <int>
1 FEMALE   165
2 MALE     168
3 <NA>      11
```

165 female, 168 male, and 11 with no sex recorded. The `NA` rows get a bar of their own, just as `count()` gives them a row of their own. Keep it: the reader should see that 11 penguins could not be put in either group. The bars start at zero, so their lengths compare fairly.

7. The colour is inside `aes()`, so it is a mapping. ggplot2 treated "darkred" as a value in the data, a column holding that one word, and gave it the first colour of its own default set. The legend explains that value. R can draw dark red; it was never asked to.

Move the colour out of `aes()`, into the geom, to set it:

```
ggplot(penguins_raw, aes(x = `Flipper Length (mm)`, y = `Body Mass (g)`)) +
  geom_point(colour = "darkred")
```

Why the wrong conclusion looks reasonable: the code contains the word "darkred" and produced a colour close to it. So the gap looks like a limit of the software. The legend is the clue: a legend appears only for something mapped.

8. The data are more likely the same, and the bins different. ggplot2's notes for version 4.0.0 say the default bin boundary changed, and that this "may affect plots that use default binning". With no `binwidth` or `boundary` given, each version placed its 30 bins its own way.

Check the data first, cheaply: compare row counts and a checksum of the raw file, as the section on raw files taught. Then make the bins part of the code:

```
ggplot(penguins_raw, aes(x = `Body Mass (g)`)) +
  geom_histogram(binwidth = 500, boundary = 2500, closed = "left")
```

With the width, the boundary and the edge rule written down, both versions draw the same bins. Why the wrong conclusion looks reasonable: the code and the file are the same, so a different picture seems to mean different data. The third ingredient, the package version, is invisible unless you print it.

9. Layers are added to a plot with `+`, not passed along with `|>`. The pipe handed the whole plot to `geom_point()` as its first argument, where `geom_point()` expects a mapping. The error message even asks whether you used `|>` instead of `+`.

```
ggplot(penguins_raw, aes(x = `Flipper Length (mm)`, y = `Body Mass (g)`)) +
  geom_point()
```

```
Warning message:
Removed 2 rows containing missing values (`geom_point()`).
```

Now the plot draws, with the usual warning about 2 rows removed.

Why the wrong code looks reasonable: in every section before this one, each step of the work was joined to the next with `|>`. ggplot2 is older than that habit and builds a plot by adding layers.

10. R printed a warning: 2 rows were removed. The boxes are drawn from 342 body masses, not 344. Two penguins have no body mass, and `geom_boxplot()` left them out. The caption should say 342, and the methods should say why 2 are missing.

Count them to be sure:

```
sum(is.na(penguins_raw$`Body Mass (g)`))
```

```
[1] 2
```

Why the wrong caption looks reasonable: 344 is the number of rows in the file, which everyone remembers. A warning in the console is easy to scroll past, and it is the only place the difference is shown.

11. Draw the histogram with bins 500 g wide starting at 2500 g, so one bar is exactly 3500 to under 4000 g, and count:

```
library(readr)
library(dplyr, warn.conflicts = FALSE)
library(ggplot2)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)

ggplot(penguins_raw, aes(x = `Body Mass (g)`)) +
  geom_histogram(binwidth = 500, boundary = 2500, closed = "left")

penguins_raw |>
  filter(!is.na(`Body Mass (g)`)) |>
  summarise(
    n_mass = n(),
    n_3500_to_4000 = sum(`Body Mass (g)` >= 3500 & `Body Mass (g)` < 4000),
    pct = 100 * n_3500_to_4000 / n_mass
  )
```

```
Warning message:
Removed 2 rows containing non-finite values (`stat_bin()`).
# A tibble: 1 × 3
  n_mass n_3500_to_4000   pct
   <int>          <int> <dbl>
1    342             94  27.5
```

94 of the 342 penguins with a body mass, 27.5%, weigh from 3500 g to under 4000 g. That bar is the tallest, so 3.5 to 4 kg is the commonest band. But “most” means more than half, and this is about a quarter. The sentence should say “the commonest band was 3.5 to 4 kg, holding 94 of 342 penguins (27.5%)”.

What the figure does not establish:

- That the commonest band would be the same with other bins. With 250 g bins the tallest bar could move.
- Anything about a typical penguin of any one species. The histogram mixes three species of different size.
- Anything about penguins outside this study.

12. Join the files and keep the adults. Label the codes: the codebook says `RIAGENDR` 1 is male and 2 is female. Draw boxes rather than bars, since BMI is a distribution and does not start at zero:

```
library(dplyr, warn.conflicts = FALSE)
library(ggplot2)

bmx <- haven::read_xpt(here::here("data-raw", "BMX_L.xpt"))
demo <- haven::read_xpt(here::here("data-raw", "DEMO_L.xpt"))

adults <- bmx |>
  left_join(demo, by = "SEQN", relationship = "one-to-one") |>
  filter(RIDAGEYR >= 20, !is.na(BMXBMI)) |>
  mutate(sex = if_else(RIAGENDR == 1, "Men", "Women"))

count(adults, sex)

bmi_plot <- ggplot(adults, aes(x = sex, y = BMXBMI)) +
  geom_boxplot() +
  labs(
    x = NULL,
    y = "BMI (kg/m^2)",
    caption = "NHANES 2021-2023, examined adults aged 20 or over; unweighted, not a US
estimate"
  )

dir.create(here::here("output"), showWarnings = FALSE)
ggsave(
  here::here("output", "nhanes_bmi_by_sex.png"),
  plot = bmi_plot,
  width = 12,
  height = 10,
  units = "cm"
)
```

```
# A tibble: 2 × 2
  sex       n
  <chr> <int>
1 Men    2680
2 Women  3290
```

The figure has two boxes, from 2680 men and 3290 women with a BMI. The rows with no BMI were removed before plotting, on purpose, so the counts printed are the counts drawn. `NULL` leaves that axis without a title. The caption says what the boxes are, and what they are not.

What it does not establish:

- Anything about adults in the United States. The boxes are unweighted; the survey's documentation says its sample weights should be used.
- Whether the difference between the boxes is more than chance, or large enough to matter.
- Anything about adults in the officer's district. A figure from one country's survey is not a figure for another country's people.
- A clean comparison of BMI. It includes women who were pregnant at the exam, whose weight does not measure the same thing.

19 · Checking the data in code

EXERCISE 1. The correction is right. The way it was made is not.

The raw file is no longer the file that was collected. Nothing in it shows that one cell was changed, when, by whom, or why. A second reader who compares the file with the forms finds no trace of the slip that was fixed.

The step cannot be rerun. If the raw file is ever replaced by a fresh export from the data entry system, the 620 comes back. The fix lived in one person's hands, not in the script.

Nothing will catch the next one. A weight of 620 was found by luck. The script still contains no line that would stop on a weight of 700.

Instead: leave the raw file as it is. In the script, add a range check on weight, so that the script stops on 620. Then add a dated line that sets a new column to 62 for that one ID, with a comment saying the paper form gave 62. Rerun. The check now passes, and the script shows the change to anyone who reads it.

EXERCISE 2. One possible set. The file does not exist, so the code is shown and not run.

```
library(readr)
library(dplyr, warn.conflicts = FALSE)

children <- read_csv(
  here::here("data-raw", "school_children.csv"),
  show_col_types = FALSE
)

stopifnot(
  # one row per child, as the protocol says
  "child_id is unique" = n_distinct(children$child_id) == nrow(children),
  # the protocol enrols ages 6 to 14 only
  "every age is 6 to 14" =
    all(children$age_years >= 6 & children$age_years <= 14, na.rm = TRUE),
  # the data dictionary allows F, M or missing
  "sex is F, M or missing" =
    all(children$sex %in% c("F", "M") | is.na(children$sex)),
  # decision, set wide: a height outside 80 to 200 cm is looked at
  "heights are 80 to 200 cm" =
    all(children$height_cm >= 80 & children$height_cm <= 200,
      na.rm = TRUE),
  # decision, set wide: a weight outside 10 to 120 kg is looked at
  "weights are 10 to 120 kg" =
    all(children$weight_kg >= 10 & children$weight_kg <= 120,
      na.rm = TRUE)
)

sum(is.na(children$height_cm))
sum(is.na(children$weight_kg))
```

Where each limit comes from matters as much as the code. The age limits and the one-row-per-child rule come from the protocol. The allowed values of `sex` come from the data dictionary. The height and weight limits are your decision. Set them wide, so they catch a slip of units or an extra digit, and write that they are a decision. None of them comes from the file's own smallest and largest values.

Two more checks belong further down the script. One is a row count before and after any join, for example to a table of schools by `school_code`. The other is a row count after each filter.

EXERCISE 3. A range check: `all(x >= lower & x <= upper, na.rm = TRUE)`, with the limits from a source or a stated decision.

A key check: `n_distinct(data$id) == nrow(data)`.

A row-count check: store `n_before <- nrow(data)` before a join or filter, then test `nrow(result) == n_before`.

Each goes inside `stopifnot()`, with a name that says the belief in words. A missing value makes a test `NA`, and `stopifnot()` treats `NA` as a failure and stops.

EXERCISE 4. Agree first. Looking is good, and a check does not replace it.

Then draw two columns on the board: "looking" and "a check in code".

Looking happens once, on the day you look. A check runs every time the script runs. When a new batch of forms arrives next month, nobody looks again; the check does.

Looking at 6000 rows means looking at the first twenty. A check reads every row.

What you looked for is in your head. A check writes it down: "every height is between 50 and 250 cm". Your guide can read it and argue with the limits. A future you can read it too.

Then show the made-up six-patient register from this section. The median BMI with two slips in it was 26.2 kg/m^2, a normal-looking number. Looking at the result told you nothing. One `stopifnot()` line stopped the script and named the problem.

End on the limit. A check catches only what it tests, and a wrong value inside the limits passes. So keep looking, and write down what you look for.

1.

```
stopifnot(2 + 2 == 4)
stopifnot(c(5, 12, 7) > 0)
stopifnot(c(5, -12, 7) > 0)
```

```
Error: c(5, -12, 7) > 0 are not all TRUE
```

The first two print nothing: each test is `TRUE` in every element, and `stopifnot()` is silent when it passes. The third stops. The comparison gives `TRUE FALSE TRUE`, and one `FALSE` is enough. The message repeats the expression that failed, because it has no name.

2.

```
sum(heights < 50 | heights > 250)
```

```
[1] 2
```

Two: 1.6 and 16.5. Each comparison gives `TRUE` or `FALSE` for every element. `|` (or) is `TRUE` when either side is, and `sum()` counts each `TRUE` as 1. The 1.6 looks like a height in metres, and the 16.5 like a slipped decimal point. The count tells you only that two need looking at.

3.

```
stopifnot("every ID is different" = n_distinct(ids) == length(ids))
```

```
Error: every ID is different
```

It stops. `n_distinct(ids)` is 4, because 103 appears twice, and `length(ids)` is 5. Four is not five, so the test is `FALSE`, and the error prints the name you gave it.

4.

```
stopifnot(all(weights > 20))
```

```
Error: all(weights > 20) is not TRUE
```

It stops. `weights > 20` gives `TRUE NA TRUE`. So `all()` cannot say that all are `TRUE`, and gives `NA`. `stopifnot()` treats `NA` as a failure.

```
stopifnot(all(weights > 20, na.rm = TRUE))
sum(is.na(weights))
```

```
[1] 1
```

With `na.rm = TRUE`, `all()` looks only at the recorded values, and the check passes silently. The second line prints 1: one weight is missing, and that number belongs in your report.

5.

```
stopifnot(
  "recorded body masses are 1000 to 10000 g" =
    all(penguins_raw$`Body Mass (g)` >= 1000 &
      penguins_raw$`Body Mass (g)` <= 10000, na.rm = TRUE)
)

sum(is.na(penguins_raw$`Body Mass (g)`))
```

```
[1] 2
```

The check prints nothing, so every recorded body mass lies within the limits. Two rows have no body mass. Keep the unit in the check's name. A limit of 10 would be right in kilograms and wrong in grams, and the name tells the next reader which was meant.

6.

```
stopifnot(
  "the file has 344 rows" = nrow(penguins_raw) == 344,
  "Species with Sample Number is a key" =
    n_distinct(penguins_raw$Species, penguins_raw$`Sample Number`) ==
      nrow(penguins_raw)
)
```

Both pass. The first check is worth having for its own sake. If someone hands you a different version of the file, the script stops on its first lines.

```
stopifnot(
  "Individual ID is a key" =
    n_distinct(penguins_raw$`Individual ID`) == nrow(penguins_raw)
)
```

```
Error: Individual ID is a key
```

It stops: there are 190 different IDs in 344 rows. The help page calls this column the unique ID for each individual, and the check shows it does not pick out one row each. That is not an error to correct. It is a question about what one row stands for, and the section on the data dictionary records the answer you found rather than guessing.

7.

```
bmi_calc <- bmx$BMXWT / (bmx$BMXHT / 100)^2
difference <- abs(bmi_calc - bmx$BMXBMI)

max(difference, na.rm = TRUE)

stopifnot(
  "BMXBMI agrees with weight / height^2 to 0.05" =
    all(difference <= 0.05, na.rm = TRUE)
)
```

```
[1] 0.04997855
```

The largest difference is 0.04997855 kg/m^2, just under 0.05, and the check passes. A number rounded to one decimal place is never more than 0.05 from the full value. So this is what you would see if the survey worked out BMI from these two columns and rounded it. The check does not prove that is how they did it. It shows that nothing in the file says otherwise.

A column that disagrees with its own formula is a cheap error to find. This line looks for it on every row where all three are recorded, at once. `abs()` gives the size of a difference without its sign.

8. The check uses `|` (or) where it needs `&` (and). Every number is either at least 50 or at most 250, and most are both. So the test is `TRUE` for every possible height, and the check can never fail. The height of 1.62 passed because 1.62 is at most 250.

```
stopifnot(
  "heights are between 50 and 250 cm" =
    all(register$height_cm >= 50 & register$height_cm <= 250)
)
```

```
Error: heights are between 50 and 250 cm
```

With `&` the check stops, as it should. The 1.62 looks like a height in metres.

Why the wrong answer looks reasonable: a check that passes looks exactly like a check that works. The only way to know a check can fail is to watch it fail once. Feed it a value you know is wrong before you trust its silence.

9. The fix picked the row by its position, and the position had changed. `arrange()` sorted the rows by height. Patient 103, with 1.6, moved to the top, and row 3 became patient 101. The line set patient 101's height of 171 cm to 160 cm. Patient 103 still has 1.6.

Pick the row by its ID, and put the corrected value in a new column:

```
clinic_checked <- clinic |>
  mutate(
    # register page for id 103 reads 160 cm; typed as 1.6
    height_cm_checked = if_else(id == 103, 160, height_cm)
  )

clinic_checked
```

```
# A tibble: 3 × 3
     id height_cm height_cm_checked
  <dbl>     <dbl>              <dbl>
1   101     171                  171
2   102     160                  160
3   103       1.6                160
```

Now only patient 103 changes, whatever order the rows are in, and the typed value stays beside the corrected one.

Why the wrong answer looks reasonable: after the fix every height is plausible, so a range check passes. The error moved from one patient to another. It did not leave.

10. The limits were taken from the data. `lower` is the smallest weight, 49, and `upper` is the largest, 620. Every value lies between the smallest and the largest value, so the check passes on any data at all. It cannot fail, and the 620 kg weight sets its own upper limit.

Take the limits from outside the data, a source or a stated decision, and write which:

```
# decision, set wide: an adult weight outside 20 to 300 kg is looked at
stopifnot(
  "weights are 20 to 300 kg" =
    all(weights >= 20 & weights <= 300, na.rm = TRUE)
)
```

```
Error: weights are 20 to 300 kg
```

Now the check stops on the 620.

Why the wrong answer looks reasonable: `min()` and `max()` feel like "the range", and the code looks careful. But a range taken from the data describes the data. It can never flag an error in the data.

11. The check was right, and deleting it hid the problem. Person 2 appears twice in `people`, so the join matched visit 2 to both rows and the result has four rows, not three:

```
nrow(joined)

people |>
  count(id) |>
  filter(n > 1)
```

```
[1] 4
# A tibble: 1 × 2
     id     n
  <dbl> <int>
1     2     2
```

The mean over the joined rows counts person 2's weight twice: (61 + 74 + 74 + 55) ÷ 4 = 66 kg. The mean of the three visits is (61 + 74 + 55) ÷ 3 = 63.3 kg.

```
mean(joined$weight_kg)
mean(visits$weight_kg)
```

```
[1] 66
[1] 63.33333
```

The fix is in the table of people. Find out why person 2 is there twice. If the two rows are the same entry typed twice, remove one in code, with a dated comment saying so, and keep the row-count check. If they are two different people sharing an ID, the ID needs fixing at its source.

Why the wrong answer looks reasonable: the script ran, and 66 kg is a believable mean. A check that stops a script feels like the problem. It is the messenger.

12. Read, join, keep the adults, then count the values outside the limits and the values missing.

```
library(haven)
library(dplyr, warn.conflicts = FALSE)

bmx <- read_xpt(here::here("data-raw", "BMX_L.xpt"))
demo <- read_xpt(here::here("data-raw", "DEMO_L.xpt"))

adults <- bmx |>
  left_join(demo, by = "SEQN") |>
  filter(RIDAGEYR >= 20)

nrow(adults)
sum(adults$BMXHT < 50 | adults$BMXHT > 250, na.rm = TRUE)
sum(adults$BMXWT < 20 | adults$BMXWT > 300, na.rm = TRUE)
sum(is.na(adults$BMXHT))
sum(is.na(adults$BMXWT))
```

```
[1] 6064
[1] 0
[1] 0
[1] 67
[1] 81
```

There are 6064 adults, and no recorded height or weight lies outside the limits. But 67 have no height and 81 no weight. So “checked for all 6064” is not true: 6064 − 67 = 5997 heights and 6064 − 81 = 5983 weights were checked. The sentence should say how many of each were recorded.

What the result does not establish:

- That the values are right. A wrong value inside the limits passes.
- That the limits are the right ones. They are a decision, set wide, and the thesis should state them.
- Anything about adults in the United States. These are counts of rows, and the survey’s sample weights were not used.

13. “Ran without an error” says only that no line failed. A line can fail only if it tests something. So write the beliefs the analysis rests on as checks. Is it the right file? Is there one row per sample? Are the body masses of a possible size? Does `Sex` hold only the values the data dictionary allows?

```
library(readr)
library(dplyr, warn.conflicts = FALSE)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)

stopifnot(
  "the file has 344 rows" = nrow(penguins_raw) == 344,
  "Species with Sample Number is a key" =
    n_distinct(penguins_raw$Species, penguins_raw$`Sample Number`) ==
      nrow(penguins_raw),
  # decision for practice: a body mass outside 1000 to 10000 g is looked at
  "recorded body masses are 1000 to 10000 g" =
    all(penguins_raw$`Body Mass (g)` >= 1000 &
      penguins_raw$`Body Mass (g)` <= 10000, na.rm = TRUE),
  "Sex is FEMALE, MALE or missing" =
    all(penguins_raw$Sex %in% c("FEMALE", "MALE") | is.na(penguins_raw$Sex))
)

sum(is.na(penguins_raw$`Body Mass (g)`))
sum(is.na(penguins_raw$Sex))
```

```
[1] 2
[1] 11
```

All four checks pass. Two rows have no body mass and 11 have no sex recorded; a mean by species leaves out the first two, and should say so.

What it still does not establish:

- That any value is correct. A body mass typed 3750 for 3570 passes every check here.
- That "clean" means anything without the list. Clean against which checks? Put the checks in the script, and the word can be replaced by the list.
- That the colleague's means are right. The checks are about the file. A mistake in the summarising code is a separate question.

20 · Does shared code rerun? What studies found

EXERCISE 1. The sentence makes two claims, and neither follows.

"Available on request" is not shared. On Peng's spectrum, a study is more or less reproducible depending on what data and code are made available. Code kept on the author's own computer has not been made available. A request may never be answered, and the computer may not last.

"So the analysis is fully reproducible" claims that the code would run on someone else's computer and print the same numbers. Nobody has tested that. One large study took 9078 R files deposited with published papers. Its authors report that 74% failed to run without an error the first time. Among the common causes were a working directory on the author's computer and packages that were missing.

And "reproducible" would not mean "right". Peng: being reproducible "does not guarantee the quality, correctness, or validity of the published results."

A true sentence describes what was done. For example: “The raw data and one script that produces every table and figure from them are deposited at [website and identifier]. The script was run from a fresh R session on a second computer, and gave the reported results.” Write the second sentence only if someone actually did it.

EXERCISE 2. The 25% is the table’s success rate for the run without code cleaning. Dividing 952 successes by 952 successes plus 2878 errors gives 0.2486, which rounds to 25%. So it is a share of the files that finished, with or without an error.

It leaves out the 3829 files that ran out of their hour. Some of those might have run with more time, which is why the authors left them out.

What the sentence gets wrong:

- “Research code” is too wide. The study is of R files on one website, the Harvard Dataverse, deposited from 2010 to July 2020, most from the social sciences.
- “Works” is too wide. A success means the file ran to the end without an error on a clean computer, under one of three versions of R. It does not mean the file printed the paper’s numbers.
- “Harvard” names the website and the institute that built it. It says nothing about how far the finding reaches.
- It leaves out the second run. After an automatic cleaning step, the table’s success rate rose to 40%.

EXERCISE 3. Reproduce: run the same code on the same data and get the same result. Replicate: collect new data on the same question, usually by other investigators, and see whether the answer holds.

Any three of these reasons:

- A line that sets the working directory to a folder on the author’s computer, with `setwd()`.
- A package that is not installed.
- A package of a version that no longer works with the code.
- A path to a folder or file that does not exist on the new computer, or an output file never saved.
- An object the code uses before anything has made it.

EXERCISE 4. For one large body of code, yes, as deposited. A team took over nine thousand files of code in the R language, shared by researchers alongside published papers on one big website. They tried to run each on a clean computer. In their own summary, about three in four did not run to the end without an error the first time. After a simple automatic fix-up they report a little over half still failing.

How sure? Fairly, for that website. It was one website, mostly social science, code shared from 2010 to 2020. It does not tell you about every field, or about code that was never shared.

Two things matter more than the figure. Many failures were boring: the code pointed at a folder on the author’s own computer, or needed a piece of software that was not there. And running is not the same as being right. Code can run perfectly and still give a wrong answer.

A quotable sentence: “One large study looked at R code shared on one research website. About three in four files did not run to the end without an error. Many of the reasons are simple to prevent.”

21 · Top to bottom in a fresh session, with one command

EXERCISE 1. Hidden state is anything a script uses that the script did not make. It may be an object typed in the console, a package attached earlier or a file made by hand.

The command is `Rscript code/analysis.R`, typed in a terminal in the project folder. It starts a new R and runs the file from the first line to the last.

`rm(list = ls())` removes the objects and leaves the packages attached, so a script that forgot a `library()` line would still run.

EXERCISE 2. There is no single right script. Check yours against these:

1. `Rscript code/analysis.R`, from the project folder, runs to the end with no error.
2. Every `library()` line the script needs is in the script, at the top.
3. The raw file is read in the script, with `here()`, from `data-raw/`.
4. The script makes `output/` if it is missing (`dir.create()`), and every table and figure is written there by a line of code.
5. A seed set with `set.seed()` comes before any random step.
6. The last line is `sessionInfo()`.
7. After you delete `output/` and run the command again, every output comes back.
8. The README's "To rerun" line is that exact command.

If any of these fails, the script still depends on something outside it.

EXERCISE 3. It can mean that someone took the group's data and code and could not get the code to run, or got a different number. One study looked at R code shared on one research website. About three in four files did not run to the end without an error the first time. Common causes are dull ones: a file kept in a folder only the author had, a missing step, a package that has since changed.

So "does not reproduce" says that the analysis cannot be checked as it stands. It does not say that the result is false. It says nobody can yet show that it is true.

The quotable sentence: "If an analysis cannot be rerun, you cannot tell whether it is wrong, and neither can its authors."

1.

```
set.seed(11)
sample(10, 3)
set.seed(11)
sample(10, 3)
```

```
[1] 10  2  8
[1] 10  2  8
```

The same three, because the seed is set to the same number before each pick.

2.

```
height_cm <- c(132, 141, 150)
rm(list = ls())
exists("height_cm")
```

```
[1] FALSE
```

`rm(list = ls())` removed every object, so `height_cm` no longer exists, and `exists()` answers FALSE.

3.

```
library(dplyr, warn.conflicts = FALSE)
rm(list = ls())
"package:dplyr" %in% search()
```

```
[1] TRUE
```

TRUE. dplyr is still on the list of places R searches. `rm()` removes objects, not packages. The operator `%in%` asks whether the text on its left is among the values on its right.

4.

```
library(readr)
library(dplyr, warn.conflicts = FALSE)
library(here)

penguins_raw <- read_csv(
  here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)

dir.create(here("output"), showWarnings = FALSE)

species_counts <- penguins_raw |>
  count(Species)
write_csv(species_counts, here("output", "species_counts.csv"))
```

```
Rscript code/analysis.R
cat output/species_counts.csv
```

```
here() starts at ~/project
Species,n
Adelie Penguin (Pygoscelis adeliae),152
Chinstrap penguin (Pygoscelis antarctica),68
Gentoo penguin (Pygoscelis papua),124
```

152, 68 and 124 rows, which add up to 344. These are rows of the file, one per nesting observation, not counts of birds. The `dir.create()` line is what lets the script run on a copy with no `output/` folder.

5.

```
packageVersion("dplyr")
```

```
[1] '1.1.4'
```

`packageVersion()`, from the section on packages, prints version 1.1.4. That is the same as `dplyr_1.1.4` in the printout. Your own computer may print a newer version. Then your outputs may differ from this book's, and the version line tells you why.

6.

```
library(readr)

penguins_raw <- read_csv(
  here::here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)

set.seed(52)
first <- sample(nrow(penguins_raw), 10)
set.seed(52)
second <- sample(nrow(penguins_raw), 10)

first
identical(first, second)
```

```
 [1]  82 179 339 317 133 142  55 220 273 117
[1] TRUE
```

`nrow(penguins_raw)` is 344, so the picks are row numbers from 1 to 344. `identical()` asks whether two objects are exactly the same, and answers TRUE. In the script, `set.seed(52)` goes once, directly above the pick. The same ten come out for anyone who runs it on the same version of R.

7. The script uses `penguins_raw` without making it. In her session that object was the Biscoe rows only, because of the second console line. So 4716 g is the mean for one island, and the script does not say so. It looks reasonable because it is a real mean of real penguins.

Save her script and run it as a stranger would:

```
library(dplyr, warn.conflicts = FALSE)
penguins_raw |>
  summarise(mean_mass_g = mean(`Body Mass (g)`, na.rm = TRUE))
```

```
Rscript code/analysis.R
```

```
Error: object 'penguins_raw' not found
Execution halted
```

The fresh run finds the gap at once. Add the packages and the reading line to the script:

```
library(readr)
library(dplyr, warn.conflicts = FALSE)
penguins_raw <- read_csv(here::here("data-raw", "penguins_raw.csv"),
                         show_col_types = FALSE)
penguins_raw |>
  summarise(mean_mass_g = mean(`Body Mass (g)`, na.rm = TRUE))
```

```
Rscript code/analysis.R
```

```
# A tibble: 1 × 1
  mean_mass_g
        <dbl>
1       4202.
```

About 4202 g for every row with a body mass, against 4716 g for Biscoe alone. If she wants Biscoe, the filter belongs in the script, where a reader can see it.

8. `source()` ran the script in Ravi's own workspace, where `penguins_raw` already existed. So the check could not find the missing reading line. It looks like a test because the whole file ran without an error. But it ran with Ravi's objects.

The test that counts starts from nothing:

```
n_rows <- nrow(penguins_raw)
```

```
Rscript code/analysis.R
```

```
Error: object 'penguins_raw' not found
Execution halted
```

The script needs `library(readr)` and the `read_csv()` line at its top. Then the same command runs to the end. `source()` is a fair test only straight after restarting R.

9. Run the README's command in a terminal, in the project folder, on your computer:

```
Rscript code/analysis.R
```

Then compare what lands in `output/` with the tables in the thesis, number by number. Read the `sessionInfo()` printout at the end of the run against the versions the candidate recorded.

If the run reaches the last line and every number matches, you have shown one thing. The script reruns from the raw files on a second computer and gives the thesis's numbers.

It does not establish:

- that the numbers are right. A script can rerun perfectly and still compute the wrong thing;
- that the raw files are the files that were collected. The run starts from whatever is in `data-raw/`;
- that it will rerun next year. Newer package versions may stop it or change a result, unless the versions are pinned, which is renv's job two rungs up;
- that every number in the text came from the script. A number typed into the chapter by hand has no line of code behind it.

So the sentence the candidate can defend is narrower: "The analysis reruns from the raw data with one command, `Rscript code/analysis.R`, on R 4.3.3."

10. Running twice in the same RStudio session tests nothing new: the second run had the first run's workspace. The test is two runs that each start from nothing, with `output/` deleted in between, and a comparison of the files. Here is his script:

```
library(readr)
library(dplyr, warn.conflicts = FALSE)
library(here)

penguins_raw <- read_csv(
  here("data-raw", "penguins_raw.csv"),
  show_col_types = FALSE
)

dir.create(here("output"), showWarnings = FALSE)

sex_counts <- penguins_raw |>
  count(Sex)
write_csv(sex_counts, here("output", "sex_counts.csv"))
```

Run it and keep a copy of the output. Delete `output/`, run it again, and compare. `cmp` compares two files and prints nothing when they are the same. `cp` copies a file, `rm -r` deletes a folder, `&&` runs the next command only if the last one succeeded, and `echo` prints its words:

```
Rscript code/analysis.R > /dev/null 2>&1
cp output/sex_counts.csv first_run.csv
rm -r output
Rscript code/analysis.R > /dev/null 2>&1
cmp first_run.csv output/sex_counts.csv && echo "identical"
cat output/sex_counts.csv
```

```
identical
Sex,n
FEMALE,165
MALE,168
NA,11
```

`> /dev/null 2>&1` throws away what each run prints, since only the files matter here. The two files are identical: 165, 168 and 11 rows, which add up to 344.

What this does not establish:

- that the script runs on his colleagues' computers, with their package versions;
- that the counts are right. They are the rows of the file, and the 11 rows with no sex recorded are counted as NA, not hidden;
- anything about penguins beyond this file.

He can now say "the script gives the same table from a fresh start on my computer". That is more than he could say before, and less than "it reproduces".

22 · One dataset, end to end

EXERCISE 1. The steps, in order:

1. The question.
2. The raw files, their terms and their checksums.
3. The data dictionary.

4. The project folder.
5. The script. It reads, checks, joins, cleans, works out new columns, summarises, plots, writes its outputs and prints the versions.
6. The README, with the one command.
7. A rerun from nothing.
8. What the result does not show.

The gate: the analysis reproduces from the raw data with one command, `Rscript code/analysis.R`, in a fresh session.

EXERCISE 2. The project passes when all of these hold.

1. The question is one sentence and says who the answer is about.
2. `data-raw/` holds the files exactly as they arrived. Their checksums in the README match `tools::md5sum()` on them now. No code writes to that folder.
3. The dictionary has a row for every variable the script uses. Each row gives the name exactly as in the file, its meaning, unit and codes, and where each came from.
4. The script reads every file with `here()`. It needs no object, package or file that it does not load or make itself.
5. It prints the row count after each step that can add or drop rows. It stops with `stopifnot()` where a count, a key or a range is wrong.
6. Every change to the data is a line of code with a comment. No cell was edited by hand.
7. Every summary prints its n. A survey summary says whether it is weighted.
8. The table and the figure are written to `output/` by code, and the script ends with `sessionInfo()`.
9. After `output/` is deleted, `Rscript code/analysis.R` brings back the same table, byte for byte.
10. Your colleague's run gives the same table. If it does not, the two `sessionInfo()` printouts are compared and the difference is written in the README.

Write, beside the result, what it does not show.

EXERCISE 3. Five things stop it from passing.

- The README gives no single command, and "in order" leaves the reader guessing. Fix: one script, or one command that runs both, written in the README.
- The path is absolute and names her desktop, so the script stops on every other computer. Fix: put the file in `data-raw/` and read it with `here()`.
- The last line writes over the file it read. After one run the raw data are gone, replaced by the cleaned version, and nothing can be rerun from the original. Fix: never write to the raw file; write cleaned data to `output/` if it is needed at all.
- The file name "final data (2)" says there was a "final data (1)". Nobody can tell which one the chapter used. Fix: record the raw file's checksum in the README.
- The number 24.6 kg/m^2 has no line of code that writes it to a file, so it was copied off a screen. Fix: write the table to `output/` with code and take the number from there.

The cost: as it stands, nobody, including Anita in a year, can show that 24.6 kg/m^2 comes from the data collected. Her examiner can ask, and she cannot answer.

EXERCISE 4. Answer: because then someone else can check any number in a departmental report in minutes. Without it, nobody can check it at all.

- **Mistakes are found before publication, not after.** A rerun by a second analyst catches a step that lived only on one laptop.
- **Work survives people.** When an analyst leaves, the next one reruns the folder instead of starting again.
- **Corrections are cheap.** When the raw data are corrected, the same command remakes every table.
- **The cost is small.** It is a habit and free software, not a purchase.

What it does not buy: a rerunnable analysis can still ask the wrong question. Someone still has to read it.

Glossary

Every term of art this book teaches, in the plain words it gets where it is first used, and the section that teaches it.

Term	Plain words	Section
absolute path	a path that starts from the top of one computer's disk, so it names a place that exists on one computer only	7
aesthetic mapping	a statement, made inside aes(), of which column of the data controls which visual property of the marks: position along x or y, colour, and others	18
argument	(1) reasons, called premises, together with a further statement they are meant to establish (F4); (2) of a function call, a value written inside its brackets, given by name, as digits = 1, or by position (5)	5
assignment	name ← value, read "name gets value": R works out the right-hand side completely, then stores the result under the name; it prints nothing	3
base R	the set of standard packages installed with R itself	6
check (in code)	a line of code that states something you believe about the data and stops the script with an error when it is false	19
checksum (MD5 checksum)	a 32-character code calculated from every byte of a file; change one character in the file and the code changes	7, 9
class (of an R object)	the kind of object it is, which class(x) names, such as Date	15
coercion	R converting combined values of different types to the highest type, in the order logical, numeric, character, without a warning; one text value turns a vector of numbers into text	4
column specification	the type read_csv() guessed for each column, which it prints; read it every time	9
comment (in code)	everything after a # on a line; R ignores it, people read it	2
complete-case analysis	analysing only the rows with no missing value in the columns used	13
console	the pane in which R shows a prompt, >, and waits for one instruction; the place to try things	2
CRAN	the Comprehensive R Archive Network, the set of websites from which R and most packages are downloaded	2, 6
CSV file	comma-separated values: a plain-text file whose first line holds the column names; nothing in it says what type each column is	9
data dictionary	a separate file, laid out as a table, with one row per variable of one data file: its exact name, meaning, unit, type, the values it may take, how a missing value is written, and where each fact came from	10
data frame	R's table: different columns may hold different types, but the values within one column share one type, and every column has the same length	8

Term	Plain words	Section
date (in R)	an object of class Date: a number of days since 1 January 1970, printed year-month-day	15
default (of an argument)	the value an argument takes when the call does not give one	5
end-to-end analysis	one question taken from raw data to its final table and figure in a single script that another person can run	22
error	(1) what a measurement says minus what is actually true, or minus a trusted reference value (VIM 2.16: "measured quantity value minus a reference quantity value") (D7); (2) in R, R stopping because it could not carry out an instruction, reported in an error message (2)	2
error message	R's statement that it could not carry out an instruction: what it could not do and, for a line it could not read, where on the line it stopped	2
factor	a column that stores each value together with its levels	14
fresh session	a new R process whose workspace is empty and which has attached no package beyond R's defaults	6, 21
function call	a function's name followed by round brackets holding its arguments; the call returns one result, which R prints or you assign to a name	5
geom	the kind of mark that represents each observation or group, such as points, bars or boxes	18
help page	the page opened with ?name that describes one function in fixed sections: Description, Usage, Arguments, Value and Examples	5
hidden state	anything a script uses that the script itself did not make	21
install (a package)	download it from a repository such as CRAN and store it on the computer, with install.packages(); once per computer	6
ISO 8601	writing a date year-month-day, such as 2021-04-03	15
join	combining two data frames by matching rows whose keys are equal	17
key	a column, or set of columns, whose values pick out the row they belong to	17, 19
key check	a check that the key has as many different values as the data have rows	19
label (of a column)	the description stored with a column in an SPSS or SAS file, which haven keeps as a label attribute, extra information stored with the column	9
layer (of a plot)	the marks one geom_ function adds; layers are joined to the plot with +	18
level (of a factor)	one of the allowed categories, in a set order; a value not among them becomes NA without an error	14
load (a package)	make its functions available in the current session, with library(); in every session	6
mapped, set (of a visual property)	mapped: written inside aes(), it varies with a column; set: written outside, as an argument of the geom, one fixed value for every mark	18
missing-value code	a value such as -99, 999 or "NR" that stands for "not recorded"; R reads it as a real value unless told to turn it into NA	9, 13

Term	Plain words	Section
NA	"not available": R's mark for a missing value; almost any arithmetic or comparison that involves it gives NA	4
object (in R)	a value R holds in memory under a name	3
observation (in tidy data)	what one row holds, such as one child at one visit; a second visit is a second row, not a second column	8
package	a bundle of R functions, with the help pages that explain them and sometimes data, that extends what base R can do	6
parsing (a date)	turning text into a date by reading it according to the order of its parts, as ymd(), dmy() and mdy() do	15
path	the address of a file	7
project folder	one folder for each analysis, holding data-raw/ (the data as it arrived, written by hand, never by code), code/ (the scripts) and output/ (what the scripts make), with a README at the top	7
project root	the top folder of the project, which here() finds and builds every file address from	7
quantile	the value below which a given share of the data lies; R places it between two values by one of nine rules, the seventh unless told otherwise	16
R	a program that reads instructions written in the R language and carries them out	2
range check	a check that every recorded value of a column lies between a lower and an upper limit written in the code, taken from a source or a stated decision, never from the same data	19
raw file (raw data)	the data as it first reached you; never edited, and never saved over	1, 9
README	a short plain-text file at the top of the project folder that says what the project is and where the data came from, and gives the one command that reruns the analysis and the folder to run it from	7, 10
relative path	a path that starts from the working directory, such as data-raw/ heights.csv	7
replicated (result)	independent investigators collected new data on the same question and reached the same answer	20
replication package	the data and code that authors deposit with a published paper	20
reproducible (result)	the same data, analysed again with the same code, give the same result; this does not show the result is correct	20
row-count check	storing the number of rows before a step that could change it, such as a join, and comparing it with the number after	19
RStudio	the window you work in; installed after R	2
script	a plain-text file of R code, saved with a name ending in .R; the console is for trying, the script is what you keep	2
seed (random-number seed)	the starting point R's random numbers are worked out from; set it with set.seed() before a random step and the same picks come back	21

Term	Plain words	Section
session (R session)	R from the moment it starts until it is closed or restarted	6
sheet (of a workbook)	one tab of an Excel workbook	9
silent change	a change to data, or a choice about which data to use, that leaves no instruction a second person can read and run again, and that nothing on the screen announces	1
snake case	names in small letters with _ between words, such as height_cm	3
terminal	the window where you type commands to the computer itself rather than to R	7
tibble	the tidyverse's kind of data frame; it prints its size and the type of each column	8
tidy data	data in which each variable forms a column, each observation forms a row, and each type of observational unit forms a table (Wickham 2014)	8
tidyverse	a family of packages; loading it loads nine, among them dplyr and readr	3, 6
type (of a value in R)	the one kind of value a vector holds; the first three met are numeric (numbers), character (text) and logical (TRUE or FALSE)	4
variable	(1) a letter written in place of a number (C1); (2) in R, another name for a named object (3); (3) in a table of data, what one column holds: one column, one variable (8)	3, 8
vector	(1) an ordered list of numbers, called its entries; one row of a data table is a vector; (2) in R, an ordered set of values of one type, held as one object under one name and built with c(); each column of a data frame is one	4
verb (dplyr)	a dplyr function that does one thing to a data frame and hands back a new one, leaving the one it was given unchanged	11
working directory	the folder R treats as home; a relative path starts from it	7
workspace	the objects that exist at a given moment, which ls() lists and RStudio's Environment pane shows; it is not your script	3

The series: all 196 books

In build order: Book 0 first, then every Introductory rung, then every Intermediate rung, and so on; within a level, a subject comes after the subjects it depends on. "done" means that book's version 1 is finished. This is Book 6.

GROUND FLOOR		
0	Ground floor	done
INTRODUCTORY		
1	Energy balance and dynamic weight-change modelling · Rung 1	done
2	Mathematics for modelling and inference · Rung 1	done
3	Qualitative and mixed methods · Rung 1	done
4	Food systems, value chains and Indian food policy · Rung 1	done
5	Policy process, framing and advocacy craft · Rung 1	done
6	**R, reproducibility engineering and data management · Rung 1 ◀ you are here**	done
7	Question selection and research strategy · Rung 1	done
8	Teaching, mentorship and coalition building · Rung 1	done
9	Scientific writing, visualisation and public communication · Rung 1	done
10	Epistemics: philosophy of science and calibration · Rung 1	
11	Adjacent specialist domains: recognise and route · Rung 1	
12	Probability and statistical inference · Rung 1	
13	Integrative metabolism, insulin resistance and energy expenditure regulation · Rung 1	
14	Commercial determinants and corporate political activity · Rung 1	
15	System dynamics and complexity concepts · Rung 1	
16	Indian regulatory architecture and the live policy agenda · Rung 1	
17	Comparative and international policy · Rung 1	
18	Python for simulation, scraping and tooling · Rung 1	
19	Regression modelling for health outcomes · Rung 1	
20	Potential outcomes, identification and target trial emulation · Rung 1	
21	Adipose tissue biology, ectopic fat and the South Asian phenotype · Rung 1	
22	Appetite, reward and the determinants of intake · Rung 1	
23	Nutrient science for obesity and cardiometabolic risk · Rung 1	
24	Physical activity and sedentary behaviour science · Rung 1	
25	Sleep science and obstructive sleep apnoea · Rung 1	
26	Participatory systems mapping and group model building · Rung 1	
27	Agent-based and network modelling · Rung 1	
28	Clustered, longitudinal, hierarchical and small-area models · Rung 1	
29	Evidence synthesis and meta-analysis · Rung 1	
30	Prediction modelling and machine learning · Rung 1	
31	Measurement error theory and correction · Rung 1	
32	Causal diagrams, structural bias and sensitivity analysis · Rung 1	
33	Study and trial design, power and missing data · Rung 1	
34	Clinical assessment, staging and secondary causes · Rung 1	
35	Indian dietary patterns, guidelines, food composition and diet cost · Rung 1	
36	Circadian biology and chrononutrition · Rung 1	

37	Behavioural theory, choice architecture and intervention design · Rung 1
38	Measurement instruments: body composition, expenditure, diet, activity, sleep · Rung 1
39	G-methods, time-varying treatment and mediation · Rung 1
40	Instrumental variables and Mendelian randomisation · Rung 1
41	Quasi-experimental policy evaluation · Rung 1
42	Genetics, developmental origins and life-course epidemiology · Rung 1
43	Social epidemiology, equity measurement and surveillance in India · Rung 1
44	Nutritional epidemiology and its pathologies · Rung 1
45	Lifestyle, dietary and behavioural treatment · Rung 1
46	Pharmacotherapy of obesity · Rung 1
47	Metabolic and bariatric surgery: selection and co-management · Rung 1
48	Eating behaviour, mental health and weight stigma · Rung 1
49	Food environment measurement, retail and digital delivery · Rung 1
50	Food demand, elasticity and the economics of externality · Rung 1
51	Implementation science and programme craft · Rung 1
52	Protocol, preregistration, reporting standards and research ethics · Rung 1
53	Life-stage obesity: paediatric, adolescent and pregnancy · Rung 1
54	Ultra-processed food and the effects of processing · Rung 1
55	Activity measurement and compositional 24-hour analysis · Rung 1
56	Fiscal instruments: taxation and subsidy design in India · Rung 1
57	Burden of disease, economic evaluation and decision-analytic modelling · Rung 1
58	The Indian health system and its financing mechanics · Rung 1
59	India's data infrastructure, access and complex survey analysis · Rung 1
60	Ethics of obesity intervention and industry engagement · Rung 1
61	Health financing, HTA and pharmaceutical access in India · Rung 1
INTERMEDIATE	
62	Energy balance and dynamic weight-change modelling · Rung 2
63	Mathematics for modelling and inference · Rung 2
64	Qualitative and mixed methods · Rung 2
65	Food systems, value chains and Indian food policy · Rung 2
66	Policy process, framing and advocacy craft · Rung 2
67	R, reproducibility engineering and data management · Rung 2
68	Question selection and research strategy · Rung 2
69	Teaching, mentorship and coalition building · Rung 2
70	Scientific writing, visualisation and public communication · Rung 2
71	Epistemics: philosophy of science and calibration · Rung 2
72	Probability and statistical inference · Rung 2
73	Integrative metabolism, insulin resistance and energy expenditure regulation · Rung 2
74	Commercial determinants and corporate political activity · Rung 2
75	System dynamics and complexity concepts · Rung 2
76	Indian regulatory architecture and the live policy agenda · Rung 2
77	Comparative and international policy · Rung 2
78	Python for simulation, scraping and tooling · Rung 2
79	Regression modelling for health outcomes · Rung 2
80	Potential outcomes, identification and target trial emulation · Rung 2
81	Adipose tissue biology, ectopic fat and the South Asian phenotype · Rung 2

82	Appetite, reward and the determinants of intake · Rung 2
83	Nutrient science for obesity and cardiometabolic risk · Rung 2
84	Physical activity and sedentary behaviour science · Rung 2
85	Sleep science and obstructive sleep apnoea · Rung 2
86	Participatory systems mapping and group model building · Rung 2
87	Agent-based and network modelling · Rung 2
88	Clustered, longitudinal, hierarchical and small-area models · Rung 2
89	Evidence synthesis and meta-analysis · Rung 2
90	Prediction modelling and machine learning · Rung 2
91	Measurement error theory and correction · Rung 2
92	Causal diagrams, structural bias and sensitivity analysis · Rung 2
93	Study and trial design, power and missing data · Rung 2
94	Clinical assessment, staging and secondary causes · Rung 2
95	Indian dietary patterns, guidelines, food composition and diet cost · Rung 2
96	Circadian biology and chrononutrition · Rung 2
97	Behavioural theory, choice architecture and intervention design · Rung 2
98	Measurement instruments: body composition, expenditure, diet, activity, sleep · Rung 2
99	G-methods, time-varying treatment and mediation · Rung 2
100	Instrumental variables and Mendelian randomisation · Rung 2
101	Quasi-experimental policy evaluation · Rung 2
102	Genetics, developmental origins and life-course epidemiology · Rung 2
103	Social epidemiology, equity measurement and surveillance in India · Rung 2
104	Nutritional epidemiology and its pathologies · Rung 2
105	Lifestyle, dietary and behavioural treatment · Rung 2
106	Pharmacotherapy of obesity · Rung 2
107	Metabolic and bariatric surgery: selection and co-management · Rung 2
108	Eating behaviour, mental health and weight stigma · Rung 2
109	Food environment measurement, retail and digital delivery · Rung 2
110	Food demand, elasticity and the economics of externality · Rung 2
111	Implementation science and programme craft · Rung 2
112	Protocol, preregistration, reporting standards and research ethics · Rung 2
113	Life-stage obesity: paediatric, adolescent and pregnancy · Rung 2
114	Ultra-processed food and the effects of processing · Rung 2
115	Activity measurement and compositional 24-hour analysis · Rung 2
116	Fiscal instruments: taxation and subsidy design in India · Rung 2
117	Burden of disease, economic evaluation and decision-analytic modelling · Rung 2
118	The Indian health system and its financing mechanics · Rung 2
119	India's data infrastructure, access and complex survey analysis · Rung 2
120	Ethics of obesity intervention and industry engagement · Rung 2
121	Health financing, HTA and pharmaceutical access in India · Rung 2
ADVANCED	
122	Energy balance and dynamic weight-change modelling · Rung 3
123	Mathematics for modelling and inference · Rung 3
124	Qualitative and mixed methods · Rung 3
125	Food systems, value chains and Indian food policy · Rung 3
126	Policy process, framing and advocacy craft · Rung 3

127	R, reproducibility engineering and data management · Rung 3
128	Question selection and research strategy · Rung 3
129	Teaching, mentorship and coalition building · Rung 3
130	Scientific writing, visualisation and public communication · Rung 3
131	Epistemics: philosophy of science and calibration · Rung 3
132	Probability and statistical inference · Rung 3
133	Integrative metabolism, insulin resistance and energy expenditure regulation · Rung 3
134	Commercial determinants and corporate political activity · Rung 3
135	System dynamics and complexity concepts · Rung 3
136	Indian regulatory architecture and the live policy agenda · Rung 3
137	Regression modelling for health outcomes · Rung 3
138	Potential outcomes, identification and target trial emulation · Rung 3
139	Adipose tissue biology, ectopic fat and the South Asian phenotype · Rung 3
140	Appetite, reward and the determinants of intake · Rung 3
141	Nutrient science for obesity and cardiometabolic risk · Rung 3
142	Physical activity and sedentary behaviour science · Rung 3
143	Sleep science and obstructive sleep apnoea · Rung 3
144	Participatory systems mapping and group model building · Rung 3
145	Clustered, longitudinal, hierarchical and small-area models · Rung 3
146	Evidence synthesis and meta-analysis · Rung 3
147	Measurement error theory and correction · Rung 3
148	Causal diagrams, structural bias and sensitivity analysis · Rung 3
149	Study and trial design, power and missing data · Rung 3
150	Clinical assessment, staging and secondary causes · Rung 3
151	Indian dietary patterns, guidelines, food composition and diet cost · Rung 3
152	Circadian biology and chrononutrition · Rung 3
153	Behavioural theory, choice architecture and intervention design · Rung 3
154	Measurement instruments: body composition, expenditure, diet, activity, sleep · Rung 3
155	G-methods, time-varying treatment and mediation · Rung 3
156	Instrumental variables and Mendelian randomisation · Rung 3
157	Quasi-experimental policy evaluation · Rung 3
158	Genetics, developmental origins and life-course epidemiology · Rung 3
159	Social epidemiology, equity measurement and surveillance in India · Rung 3
160	Nutritional epidemiology and its pathologies · Rung 3
161	Lifestyle, dietary and behavioural treatment · Rung 3
162	Pharmacotherapy of obesity · Rung 3
163	Metabolic and bariatric surgery: selection and co-management · Rung 3
164	Eating behaviour, mental health and weight stigma · Rung 3
165	Food environment measurement, retail and digital delivery · Rung 3
166	Food demand, elasticity and the economics of externality · Rung 3
167	Implementation science and programme craft · Rung 3
168	Protocol, preregistration, reporting standards and research ethics · Rung 3
169	Life-stage obesity: paediatric, adolescent and pregnancy · Rung 3
170	Ultra-processed food and the effects of processing · Rung 3
171	Activity measurement and compositional 24-hour analysis · Rung 3
172	Fiscal instruments: taxation and subsidy design in India · Rung 3
173	Burden of disease, economic evaluation and decision-analytic modelling · Rung 3

174	The Indian health system and its financing mechanics · Rung 3
175	India's data infrastructure, access and complex survey analysis · Rung 3
176	Ethics of obesity intervention and industry engagement · Rung 3

EXPERT

177	Energy balance and dynamic weight-change modelling · Rung 4
178	R, reproducibility engineering and data management · Rung 4
179	Question selection and research strategy · Rung 4
180	Probability and statistical inference · Rung 4
181	Indian regulatory architecture and the live policy agenda · Rung 4
182	Regression modelling for health outcomes · Rung 4
183	Potential outcomes, identification and target trial emulation · Rung 4
184	Adipose tissue biology, ectopic fat and the South Asian phenotype · Rung 4
185	Appetite, reward and the determinants of intake · Rung 4
186	Measurement error theory and correction · Rung 4
187	Causal diagrams, structural bias and sensitivity analysis · Rung 4
188	Circadian biology and chrononutrition · Rung 4
189	Measurement instruments: body composition, expenditure, diet, activity, sleep · Rung 4
190	Genetics, developmental origins and life-course epidemiology · Rung 4
191	Nutritional epidemiology and its pathologies · Rung 4
192	Eating behaviour, mental health and weight stigma · Rung 4
193	Activity measurement and compositional 24-hour analysis · Rung 4
194	India's data infrastructure, access and complex survey analysis · Rung 4
195	Ethics of obesity intervention and industry engagement · Rung 4

OBESITY EXPERTISE · BOOK 6

R, reproducibility engineering and data management · Rung 1

A short, practical book on starting data analysis the reproducible way. It covers R as a language, what a data frame is, the tidyverse verbs for importing, cleaning, summarising and plotting, why raw data is never edited and spreadsheets are never the place where analysis happens, and how to organise a project so that your whole analysis reruns from raw data with one command.

Book 6 of the series' 196 books, free to read at drharshmaheshwari.com.



Dr. Harsh Maheshwari
MD Community Medicine Resident, AIIMS Raipur
contact@drharshmaheshwari.com
ORCID 0000-0001-8848-1918
drharshmaheshwari.com
Version 1.0 · 3 October 2026

CC BY-NC-SA 4.0 · free to share, not to sell