R for Biostatistics

A One-Week Boot Camp for incoming graduate students

The rgtlab Curriculum Project

2026-04-29

GRADUATE BIOSTATISTICS SERIES

rgtlab Curriculum Project

R for Biostatistics

A One-Week Boot Camp for incoming students

First Edition · 2026

rgtlab

Welcome

This is the online version of **R for Biostatistics: A One-Week Boot Camp** by The rgtlab Curriculum Project, a short preparatory course for incoming graduate students in biostatistics and adjacent programmes.

The book covers everything an entering masters-level biostatistician needs to be functional in R on day one of the academic programme: installation, basic syntax, data manipulation with the tidyverse, visualisation with ggplot2, simple statistics, and the minimum viable reproducibility setup. It is designed for **five consecutive days** of work: one hour of lecture content each day, two hours of homework, no examinations.

The boot camp is the entry point to a five-volume graduate sequence:

- **R for Biostatistics: A One-Week Boot Camp** (this volume) — pre-program preparation.
- *Biostatistics Practicum* — workflow infrastructure (Git, Docker, renv, Quarto, CDISC).
- *Statistical Computing in the Age of AI* — introductory methods (linear models, GLM, mixed models, survival, Bayesian, bootstrap, simulation).
- *Advanced Statistical Computing in the Age of AI* — advanced computing (numerical stability, MCMC, HPC, high-dimensional methods, ML, software engineering).
- *Applied Generative AI for Health Sciences Research* — capability classes, RAG, agents, evaluation, regulation, deployment.

See the Preface for the design rationale and the Conventions page for visual cues.

License

R for Biostatistics

A One-Week Boot Camp for incoming graduate students.

Copyright

To cite this book, please use:

> Thomas, R. G. (2026). *R for Biostatistics: A One-Week Boot Camp*. Available at https://r-bootcamp.rgtlab.org.

Table of contents

Preface

This volume is a one-week boot camp for incoming graduate students in biostatistics who arrive with no or minimal prior exposure to R. It is the prerequisite that every subsequent volume in the sequence assumes; it covers exactly what is needed for a student to be functional in R on day one of an MS programme.

The design constraint is severe: five lectures, ten hours of homework, and the student arrives at the academic programme productive. The constraint is met by ruthless prioritisation. A great deal that R can do is omitted deliberately; the omitted material lives in the four companion volumes that follow.

What this book covers

Five days, one chapter each:

1. **Setup and first steps.** Install R and RStudio, the console and the script, vectors and arithmetic, data types (numeric, character, logical, factor), indexing, missing values.
2. **Data: tidy data, import, manipulate.** Read CSV and Excel, tibbles, the six `dplyr` verbs (`filter`, `select`, `mutate`, `summarise`, `arrange`, `group_by`), the native pipe `|>`, joining tables.
3. **Visualisation with ggplot2.** Grammar of graphics, geoms, aesthetics, scales, facets, themes, saving figures.
4. **Functions, control flow, applied statistics.** Writing functions, `if`/`else`, `purrr::map` over loops, summary statistics, `t.test`, `chisq.test`, simple linear regression with `lm`, interpreting output.
5. **Reproducibility and AI assistance.** Quarto basics for reports, RStudio projects, basic Git through the IDE, using AI assistance

(ChatGPT, Claude) responsibly with verification, pointers to the four companion volumes for everything beyond.

Each chapter is approximately 1 hour of reading and worked examples plus 2 hours of homework problems with worked solutions provided.

What this book does not cover

The book deliberately omits — and points elsewhere for — nearly everything beyond the entry-level basics:

- Statistical methodology beyond simple tests and `lm`. See *Statistical Computing in the Age of AI*.
- Reproducibility infrastructure beyond Quarto and basic Git. See *Biostatistics Practicum*.
- Advanced numerical methods, Bayesian computation, high-dimensional analysis, machine learning. See *Advanced Statistical Computing in the Age of AI*.
- Generative AI integration, agents, evaluation. See *Applied Generative AI for Public Health and Biostatistics*.
- Causal inference, longitudinal analysis, clinical trial design, missing-data depth. See *Applied Statistical Methods for Health Sciences Research*.

The boot camp is intentionally narrow. The student finishes Day 5 and proceeds to the methods courses with the R-side mechanics in place, leaving the methods courses free to teach methods.

How this book is meant to be used

The five-day cadence is the spine. A student starting the week with no R can finish the week ready for a graduate biostatistics programme. The cadence assumes roughly 3 hours of focused work per day:

- **Hour 1:** Read the chapter, work through the inline examples in your own R session.

- **Hours 2-3:** Complete the homework problems. Solutions are at the end of each chapter; check yourself only after attempting each problem.

The book also functions as a reference. After the boot camp, returning students often look up specific patterns (joining tables, customising a ggplot, writing a function) in their original chapters.

Prerequisites

The book assumes:

- A working laptop running macOS, Windows, or Linux.
- Basic familiarity with using a computer (opening applications, creating files, navigating the filesystem).
- Comfort with high-school algebra. No prior programming experience is assumed.

Acknowledgements

This boot camp consolidates patterns developed over two decades of teaching R to incoming biostatistics students. The Posit team's *R for Data Science* and the conventions of the modern tidyverse provide the technical scaffolding; the boot-camp pedagogy reflects experience with what incoming students actually need to know in the first week, as distinct from what an instructor might want them to know.

Conventions

This page summarises the visual conventions used throughout the book.

Code

R code appears in syntax-highlighted blocks. Output is prefixed with `#>` to make the boundary between input and output explicit:

```
mean(c(1, 2, 3, 4, 5))
#> [1] 3
```

Inline code is in `monospace`. Function calls always include parentheses (`mean()` rather than `mean`) so that they are unambiguously functions. Package-qualified calls (`dplyr::filter`) appear when the function is not universally known, when there is name-collision risk, or when the chapter is teaching package usage.

Callouts

Three callout types appear:

> **Tip**
>
> A small practical recommendation.

Check your understanding: example

A short question testing comprehension of the just-read material. Click to expand the answer.

Warning

A pitfall the reader may otherwise hit.

Cross-references

Within this book, sections, figures, and tables are referenced by their Quarto label (`@sec-monte-carlo-human`, `@fig-mcmc-trace`, `@tbl-comparison`). These resolve to clickable links in HTML and proper figure/table numbers in PDF.

References to the companion volumes *Statistical Computing in the Age of AI* and *Biostatistics Practicum* use **prose pointers** rather than Quarto cross-references, because cross-references do not resolve across separate books. For example: 'see the Optimisation chapter of the companion *Statistical Computing in the Age of AI* volume'.

Mathematical notation

Conventional notation throughout. Vectors are bold lower-case ($\mathbf{x}$); matrices are bold upper-case ($\mathbf{X}$); scalars and parameters are non-bold. Estimators carry hats ($\widehat{\theta}$). Sample size is n; parameter dimension is p.

Chapter structure

Every content chapter follows the same template:

1. **Learning objectives.** What you will be able to do after reading.
2. **Orientation.** A short prose framing.

3. **The statistician's contribution.** What no tool can automate. The judgements at the centre of the chapter.
4. **Content sections** with **Check-your-understanding** callouts at natural pauses.
5. **Collaborating with an LLM on the chapter topic.** Prompt / Watch for / Verification triples for AI assistance.
6. **Exercises.** The work.
7. **Further reading.** Where to go next on the topic.

The pattern repeats deliberately. By the third chapter you know where to find each component.

How to use this book

This short orientation chapter explains the daily cadence, the chapter template, and the conventions used throughout.

The daily cadence

Each day has the same shape:

- **Lecture content (~1 hour).** Read the chapter, working through the examples in your own R session as you encounter them. Do not skip the examples; the chapter is designed to be read with R open.
- **Homework (~2 hours).** Five to eight problems, with worked solutions at the end of the chapter. Attempt each problem before checking the solution.

The cadence assumes one chapter per day for five consecutive days. The boot camp can be compressed (two chapters per day across two-and-a-half days) or expanded (one chapter per week across a month) depending on the student's other commitments.

Chapter template

Each chapter follows a five-section template:

1. **Learning objectives.** A bulleted list of what the student should be able to do by the end of the day.
2. **Lecture.** The substantive content for the day, with inline examples to run.

3. **Worked example.** A small but realistic biomedical example that uses the day's content end to end.
4. **Homework.** Problems organised from easier to harder.
5. **Solutions.** Worked solutions to all homework problems. Read only after attempting the problem.

There are no quizzes, no examinations, and no tests in this book. Self-assessment is via the homework solutions.

Conventions

Visual cues used throughout the book are described on the Conventions page. The book is R-first; the style follows tidyverse conventions:

- The native pipe `|>` is used throughout (not `%>%`).
- Assignment uses `<-` (not `=`).
- Variable names use `snake_case`.
- Code lines are wrapped at 78 characters.

Companion volumes

After the boot camp, the four follow-on volumes are at:

- *Biostatistics Practicum* — https://rgtlab.org/practicum
- *Statistical Computing in the Age of AI* — https://scai.rgtlab.org
- *Advanced Statistical Computing in the Age of AI* — https://scai-advanced.rgtlab.org
- *Applied Generative AI for Public Health and Biostatistics* — https://applied-genai.rgtlab.org

The end of Chapter 5 lists which companion volume picks up which thread.

Part I.

Five Days

1. Day 1: Setup and First Steps in R

1.1. Learning objectives

By the end of this day you should be able to:

- Install R and RStudio on macOS, Windows, or Linux.
- Distinguish the console (interactive) from a script (saved code).
- Create and assign variables; perform arithmetic.
- Recognise and work with R's core data types: numeric, integer, character, logical, factor.
- Index vectors by position, by name, and by logical condition.
- Recognise missing values (`NA`) and avoid the common pitfalls.

1.2. Lecture

R is a programming language built specifically for statistical computing. You will spend the next few years of your graduate career running R in some form every day. This first hour gets you to the point where R is installed, you can type expressions and have them evaluated, and you understand what R is doing with the characters you type.

1.2.1. Installing R and RStudio

Two pieces of software, in this order.

R itself is the language and the engine. Download from https://cran.r-project.org/ and run the installer for your operating system. R 4.5+ is current as of writing; any 4.x release is fine for this book.

RStudio is the integrated development environment (IDE). It is the application you actually open every day; it talks to R behind the scenes. Download from https://posit.co/download/rstudio-desktop/ and install.

When you open RStudio for the first time you see four panes:

- **Source** (top-left): your code, saved in `.R` files.
- **Console** (bottom-left): the live R session.
- **Environment / History** (top-right): the objects you have created, and a log of recent commands.
- **Files / Plots / Help / Packages** (bottom-right): filesystem, figures you have produced, help pages, installed packages.

You will use the source pane (writing scripts) and the console pane (running them) most of the time.

1.2.2. The console: arithmetic and assignment

Click in the **Console** pane (bottom-left). You see a prompt (`>`). Type:

```
2 + 2
#> [1] 4
```

R evaluated `2 + 2` and printed `4`. The `[1]` means 'the first element of the result is printed here' — which matters when results have many elements.

The arithmetic operators are what you expect: `+`, `-`, `*`, `/`, `^` for exponentiation, `%%` for modulo (remainder).

```
17 %% 5
#> [1] 2
```

To save a value for later, **assign** it to a name. The R convention is to use `<-` (left-arrow):

```
x <- 17
x
#> [1] 17
x * 2
#> [1] 34
```

Names can use letters, digits, dot, and underscore, but must start with a letter or dot. The convention in modern R is `snake_case`:

```
patient_age <- 47
mean_blood_pressure <- 128.5
```

1.2.3. Vectors

The fundamental data structure in R is the **vector**: an ordered collection of values of the same type. Create one with `c()` (combine):

```
ages <- c(24, 47, 31, 62, 19, 55)
ages
#> [1] 24 47 31 62 19 55
length(ages)
#> [1] 6
```

Arithmetic on a vector applies element-wise:

```
ages_in_months <- ages * 12
ages_in_months
#> [1] 288 564 372 744 228 660
```

Many R functions take a vector and return either a vector of the same length or a single summary value:

```
mean(ages)
#> [1] 39.66667
sd(ages)
```

```
#> [1] 17.36856
range(ages)
#> [1] 19 62
```

This vectorised behaviour is the single most important idiom in R. You almost never write loops to apply an operation element-by-element; R does it for you.

1.2.4. Data types

Every value in R has a **type**. The five you will meet this week:

```
typeof(42L)         # integer (note the L)
#> [1] "integer"
typeof(3.14)        # double-precision floating-point
#> [1] "double"
typeof("hello")     # character (string)
#> [1] "character"
typeof(TRUE)        # logical (boolean)
#> [1] "logical"
typeof(factor(c("low", "high")))   # factor
#> [1] "integer"
```

A **factor** is R's way of representing a categorical variable. Internally it is an integer with a labelling. You will encounter factors immediately when reading data; treat them as categorical for now.

R coerces between types automatically when it can:

```
TRUE + 1
#> [1] 2          # TRUE is treated as 1
"5" + 1
#> Error: non-numeric argument to binary operator
as.numeric("5") + 1
#> [1] 6
```

Coercion is the source of many beginner bugs. When in doubt, check the type with `typeof()` or `class()`.

1.2.5. Indexing vectors

Three ways to extract elements from a vector.

By position (integer index, starting at 1):

```
ages <- c(24, 47, 31, 62, 19, 55)
ages[1]
#> [1] 24
ages[c(1, 3, 5)]
#> [1] 24 31 19
ages[-1]                # all but the first
#> [1] 47 31 62 19 55
```

By logical (a vector of `TRUE`/`FALSE` of the same length):

```
adults <- ages >= 18
adults
#> [1] TRUE TRUE TRUE TRUE TRUE TRUE
ages[ages >= 50]
#> [1] 62 55
```

By name (when the vector has names):

```
biomarkers <- c(hba1c = 7.2, ldl = 145, sbp = 132)
biomarkers["ldl"]
#> ldl
#> 145
```

Logical indexing is the workhorse of data manipulation in R. Read it as 'keep the elements where the condition is TRUE'.

1.2.6. Missing values

Real biomedical data has missing values. R represents them as `NA` (Not Available):

```
labs <- c(7.2, NA, 6.8, 8.1, NA)
labs
#> [1] 7.2  NA 6.8 8.1  NA
mean(labs)
#> [1] NA
mean(labs, na.rm = TRUE)
#> [1] 7.366667
```

`mean()` returns `NA` by default when any input is `NA`. Adding `na.rm = TRUE` removes the missing values before computing. Most R functions follow this convention; it is deliberate, to force you to think about how you want to handle the missingness rather than silently ignoring it.

Test for missingness with `is.na()`:

```
is.na(labs)
#> [1] FALSE  TRUE FALSE FALSE  TRUE
labs[!is.na(labs)]
#> [1] 7.2 6.8 8.1
```

Do **not** test for `NA` with `==`:

```
labs == NA
#> [1]  NA  NA  NA  NA  NA
```

`NA == NA` is `NA` (because the result of comparing an unknown to an unknown is itself unknown).

1.3. Worked example: a small biomedical dataset

You see ten patients in a clinic. Their ages and systolic blood pressures (SBP) are recorded.

```
patient_id <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
ages       <- c(34, 67, 52, 41, 58, 23, 71, 49, NA, 38)
sbp        <- c(122, 148, 135, 128, 142, 118, 156, 130,
                139, 124)
```

Sample size, mean age (handling the missing), proportion hypertensive (SBP >= 140):

```
length(patient_id)
#> [1] 10
mean(ages, na.rm = TRUE)
#> [1] 48.11111
mean(sbp >= 140)
#> [1] 0.3
```

Identify the hypertensive patients and report their IDs and ages:

```
hypertensive <- sbp >= 140
hypertensive
#>  [1] FALSE  TRUE FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE
patient_id[hypertensive]
#> [1] 2 5 7
ages[hypertensive]
#> [1] 67 58 71
```

In three lines you have done what an analyst might otherwise do by hand: counted the cohort, computed summary statistics with explicit missing-data handling, and identified the subset of clinical interest. This is the core rhythm of R for biostatistics.

1.4. Homework

Each problem is solvable with what was covered in the lecture. Attempt each one in your R session before checking the solution.

1. **Reproduce.** In your R session, type all of the examples from the 'Worked example' section. Confirm you get the same output.
2. **Five small expressions.** Write R expressions that compute each of these:
   (a) The cube root of 27.

   (b) The remainder when 1000 is divided by 7.
   (c) The mean of the integers 1 through 100.
   (d) The number of values in `c(2, 4, 6, 8, 10)` that are greater than 5.
   (e) The maximum minus the minimum of `c(11, 4, 19, 7, 2)`.

3. **Inspect three suspicious expressions.** For each, run it in R and explain what happens. If it errors, explain why; if it runs, explain whether the result is what a careful R user would write.

   (a) `mean(c(1, 2, 3, "4"))`
   (b) `c(1, 2, 3) + c(10, 20)`
   (c) `x = 5; x ** 2`

4. **Index by group.** Given `ages <- c(8, 22, 47, 65, 13, 51, 73, 29, 17, 55)`, compute the mean age in three groups: children (under 18), working-age adults (18-64 inclusive), seniors (65+). Use logical indexing.

5. **Missing values.** Given `labs <- c(7.2, NA, 6.8, 8.1, NA, 5.9, 7.4, NA)`:

   (a) How many values are missing?
   (b) What is the mean of the non-missing values?
   (c) Replace each `NA` with the mean of the non-missing values, producing a vector of length 8 with no `NA`s.

1.5. Solutions

Problem 1. Each line in the worked example should produce the output shown. If yours differs, common causes are: typo (check parentheses and commas), wrong type (e.g., quotes around numbers), or a left-over object from earlier in the session. Restart R if the session is in a strange state.

Problem 2.

```
# (a)
27 ^ (1/3)
#> [1] 3
# (b)
1000 %% 7
#> [1] 6
# (c)
mean(1:100)
#> [1] 50.5
# (d)
sum(c(2, 4, 6, 8, 10) > 5)
#> [1] 3
# (e)
max(c(11, 4, 19, 7, 2)) - min(c(11, 4, 19, 7, 2))
#> [1] 17
# (or, more cleanly:)
diff(range(c(11, 4, 19, 7, 2)))
#> [1] 17
```

Problem 3.

(a) `mean(c(1, 2, 3, "4"))`. The `"4"` (in quotes) is a character; `c()` coerces the whole vector to character because mixed types must collapse to one type. `mean()` on a character vector errors. Fix:

```
mean(c(1, 2, 3, 4))
#> [1] 2.5
```

(b) `c(1, 2, 3) + c(10, 20)`. R recycles the shorter vector, but lengths that do not divide evenly produce a warning. Fix to matching lengths:

```
c(1, 2, 3) + c(10, 20, 30)
#> [1] 11 22 33
```

(c) `x = 5; x ** 2`. The `=` is allowed for assignment but the convention in R is `<-`. R's parser actually silently rewrites `**` to `^` (see `?Arithmetic`), so this expression returns 25. The expression is not

strictly broken, but ** is not idiomatic in R and relies on a parser quirk that is easy to forget; prefer ^ explicitly:

```
x <- 5
x ^ 2
#> [1] 25
```

Problem 4.

```
ages <- c(8, 22, 47, 65, 13, 51, 73, 29, 17, 55)
mean(ages[ages < 18])           # children
#> [1] 12.66667
mean(ages[ages >= 18 & ages < 65])   # working-age
#> [1] 40.8
mean(ages[ages >= 65])          # seniors
#> [1] 69
```

Problem 5.

```
labs <- c(7.2, NA, 6.8, 8.1, NA, 5.9, 7.4, NA)
# (a)
sum(is.na(labs))
#> [1] 3
# (b)
mean(labs, na.rm = TRUE)
#> [1] 7.08
# (c)
labs[is.na(labs)] <- mean(labs, na.rm = TRUE)
labs
#> [1] 7.20 7.08 6.80 8.10 7.08 5.90 7.40 7.08
```

Note: imputing missing values with the mean is sometimes acceptable for exploratory work but can bias inferences in a real analysis. Day 4 introduces statistical tests; the *Practicum* and *Applied Methods* volumes treat missing-data handling at the level you will need for publishable work.

1.6. What's next

Day 2 covers reading real data into R and manipulating it with the `dplyr` verbs. Bring your laptop with R and RStudio installed and tested.

2. Day 2: Data Import and Manipulation

2.1. Learning objectives

By the end of this day you should be able to:

- Read a CSV or Excel file into R as a tibble.
- Distinguish a tibble from a base-R data frame and know when the difference matters.
- Apply the six core `dplyr` verbs: `filter`, `select`, `mutate`, `summarise`, `arrange`, `group_by`.
- Chain operations using the native pipe `|>`.
- Join two tables on a shared key (`left_join`, `inner_join`).

2.2. Lecture

Real biostatistical work starts with someone handing you a spreadsheet, a CSV from the EHR, or an export from REDCap. The first hour of an analysis is usually loading that data into R, looking at it, and getting it into a shape that your analysis can consume. The `tidyverse` is the modern toolkit for that work; the six core `dplyr` verbs are 90% of what you need.

2.2.1. Installing the tidyverse

Once, in your R session:

```
install.packages("tidyverse")
```

This installs `dplyr`, `tidyr`, `readr`, `ggplot2`, and a handful of supporting packages. After installation, load them at the start of every script:

```
library(tidyverse)
```

You will see a startup message listing the packages and flagging any conflicts with base R. The conflicts to know about: `filter()` and `lag()` are masked from `stats`. In practice this matters only when working with time series.

2.2.2. Tidy data

The convention the tidyverse rests on:

- One row per **observation**.
- One column per **variable**.
- One value per **cell**.

A tidy dataset is what most statistical functions expect. Real data is often untidy (multiple variables in one column, multiple observations in one row). Day 2 assumes the data arrives roughly tidy; Day 4 has a homework problem on tidying.

2.2.3. Reading data

CSV files are the most common format:

```
library(tidyverse)
patients <- read_csv("data/patients.csv")
```

`read_csv()` returns a **tibble** — a modern data frame. It prints nicely (only the first 10 rows; columns flagged by type), and it does not convert characters to factors. (Base R's `read.csv()` had `stringsAsFactors = TRUE` as its default until R 4.0.0 in 2020; the modern default is `FALSE`, but the tibble convention is still cleaner.)

Excel files:

```
library(readxl)
patients <- read_excel("data/patients.xlsx", sheet = 1)
```

For SAS, SPSS, Stata, REDCap, and other formats, the `haven`, `REDCapR`, and adjacent packages handle the import. The pattern is the same: `read_*()` returns a tibble.

After reading, look at the data:

```
patients
#> # A tibble: 200 × 5
#>      id   age sex     bmi sbp
#>   <dbl> <dbl> <chr> <dbl> <dbl>
#> 1     1    47 F      28.4   132
#> 2     2    62 M      31.1   148
#> ...

glimpse(patients)
#> Rows: 200
#> Columns: 5
#> $ id  <dbl> 1, 2, 3, 4, ...
#> $ age <dbl> 47, 62, 35, ...
#> $ sex <chr> "F", "M", "F", ...
```

`glimpse()` is the right first look at every dataset. `View(patients)` opens a spreadsheet-style viewer in RStudio.

2.2.4. The six dplyr verbs

Each verb takes a tibble as input and returns a tibble.

`filter()` keeps rows that match a condition:

```
adults <- filter(patients, age >= 18)
hypertensive <- filter(patients, sbp >= 140)
adult_women <- filter(patients, age >= 18, sex == "F")
```

Multiple conditions in one call are combined with AND. For OR, use `|`:

```
high_risk <- filter(patients, sbp >= 140 | bmi >= 30)
```

`select()` keeps columns:

```
demo <- select(patients, id, age, sex)
no_id <- select(patients, -id)
```

`mutate()` adds or modifies columns:

```
patients <- mutate(patients,
                   age_group = case_when(
                     age < 30 ~ "young",
                     age < 65 ~ "middle",
                     TRUE     ~ "older"
                   ),
                   bmi_category = case_when(
                     bmi < 25 ~ "normal",
                     bmi < 30 ~ "overweight",
                     TRUE     ~ "obese"
                   ))
```

`case_when()` is the multi-branch conditional. The right- hand side after `~` is the value when the left-hand condition is `TRUE`. Conditions are evaluated top to bottom; the `TRUE` at the end is the catch-all.

`summarise()` collapses a tibble to one or more summary values:

```
summarise(patients,
          n = n(),
          mean_age = mean(age, na.rm = TRUE),
          mean_sbp = mean(sbp, na.rm = TRUE))
#> # A tibble: 1 × 3
#>       n mean_age mean_sbp
#>   <int>    <dbl>    <dbl>
#> 1   200     46.3    128.5
```

`arrange()` sorts:

```
arrange(patients, age)
arrange(patients, desc(sbp))
```

`group_by()` sets a grouping for subsequent verbs. Combined with `summarise()`, it produces per-group summaries:

```
patients |>
  group_by(sex) |>
  summarise(n = n(),
            mean_age = mean(age),
            mean_sbp = mean(sbp))
#> # A tibble: 2 × 4
#>   sex       n mean_age mean_sbp
#>   <chr> <int>    <dbl>    <dbl>
#> 1 F       110     45.2    126.1
#> 2 M        90     47.8    131.4
```

Note the `|>` (the **native pipe**, R 4.1+). It takes the value on the left and passes it as the first argument to the function on the right. Chained pipes read top-to- bottom and are the standard tidyverse idiom:

```
patients |>
  filter(age >= 40) |>
  mutate(map = sbp * 2/3 + sbp_dia * 1/3) |>
  group_by(sex, age_group) |>
  summarise(n = n(),
            mean_map = mean(map, na.rm = TRUE),
            .groups = "drop")
```

The `.groups = "drop"` removes the grouping after the summary; without it, downstream operations remain grouped, which is occasionally a source of surprise.

2.2.5. Joining tables

Real data lives in multiple tables. Demographics in one file, lab results in another, medications in a third. The join verbs combine them.

```
patients
#> # A tibble: 200 × 5
#>      id   age sex     bmi sbp
labs
#> # A tibble: 410 × 3
#>      id   visit_date   hba1c
```

Combine with `left_join()` (keep all rows from the left table, add matching rows from the right):

```
combined <- patients |>
  left_join(labs, by = "id")
```

Each patient in `patients` may have multiple lab visits in `labs`; the result has one row per (patient, visit) pair. If you want one row per patient, summarise the labs before joining:

```
labs_per_patient <- labs |>
  group_by(id) |>
  summarise(latest_hba1c = last(hba1c, order_by = visit_date),
            n_visits = n())

combined <- patients |>
  left_join(labs_per_patient, by = "id")
```

Other joins:

- `inner_join()` keeps only IDs present in both tables.
- `right_join()` is `left_join()` with arguments swapped; rarely used.
- `full_join()` keeps all IDs from either table.
- `anti_join()` keeps left-table rows that have no match in the right table — useful for finding data-quality problems.

Always check the row count before and after a join. A join that unexpectedly multiplies rows means the join key is not unique on one side; investigate before proceeding.

> **Note**
>
> A common pattern in clinical data: the join key is a patient ID plus a visit date. Use `by = c("id", "visit_date")` to join on multiple columns.

2.3. Worked example: NHANES-style cohort wrangling

The data: a synthetic NHANES-style file with 500 adults, columns `id`, `age`, `sex`, `race`, `bmi`, `sbp`, `fasting_glucose`. The objective: characterise the cohort, identify the diabetic subgroup, compare blood pressure across age groups stratified by sex.

```
library(tidyverse)

# read
nhanes <- read_csv("data/nhanes-mock.csv")

# look at the data first
glimpse(nhanes)
#> Rows: 500
#> Columns: 7
#> $ id              <dbl> 1, 2, 3, ...
#> $ age             <dbl> 47, 62, 35, ...
#> $ sex             <chr> "F", "M", ...
#> $ race            <chr> "White", "Black", ...
#> $ bmi             <dbl> 28.4, 31.1, ...
#> $ sbp             <dbl> 132, 148, ...
#> $ fasting_glucose <dbl> 98, 154, 95, ...

# add derived variables
```

```
nhanes <- nhanes |>
  mutate(diabetic = fasting_glucose >= 126,
         age_group = case_when(
           age < 40 ~ "<40",
           age < 60 ~ "40-59",
           TRUE     ~ "60+"
         ))

# overall summary
nhanes |>
  summarise(n = n(),
            mean_age = mean(age),
            mean_bmi = mean(bmi),
            prop_diabetic = mean(diabetic))
#> # A tibble: 1 × 4
#>       n mean_age mean_bmi prop_diabetic
#>   <int>    <dbl>    <dbl>         <dbl>
#> 1   500     49.4     27.8         0.118

# stratified by age group and sex
nhanes |>
  group_by(age_group, sex) |>
  summarise(n = n(),
            mean_sbp = mean(sbp),
            prop_diabetic = mean(diabetic),
            .groups = "drop") |>
  arrange(age_group, sex)
#> # A tibble: 6 × 5
#>   age_group sex       n mean_sbp prop_diabetic
#>   <chr>     <chr> <int>    <dbl>         <dbl>
#> 1 <40       F        86     117.         0.023
#> 2 <40       M        72     119.         0.014
#> 3 40-59     F        99     124.         0.131
#> 4 40-59     M        88     128.         0.114
#> 5 60+       F        85     134.         0.176
#> 6 60+       M        70     138.         0.243
```

The pipeline is six lines of code that produces a publishable descriptive

table. The pattern — read, glimpse, mutate-derived, group, summarise — is the rhythm you will use every day.

2.4. Homework

The data file for the homework lives at the URL given on the course page (a synthetic public-health dataset of about 500 records). Download it and save it locally as `cohort.csv` before starting.

1. **Load and inspect.** Read the dataset into R as `cohort`. Use `glimpse()` to report the variables and types. How many rows? How many columns?

2. **Filter to a sub-cohort.** Filter to adults aged 40 and over with `bmi >= 25`. Report the sample size.

3. **Stratified summary.** Compute the mean of all numeric variables, grouped by the sex variable.

4. **Two-table join.** A second file (`labs.csv`) gives recent lab results, with multiple visits per patient. For each patient, retain only the most recent visit (use `slice_max(visit_date, n = 1)` after grouping by ID). Join the result to `cohort` so each cohort row gets that patient's most recent lab values. Verify that the joined table has the same number of rows as the original `cohort`.

5. **Identify anomalies.** Find three rows in `cohort` that have implausible values (e.g., negative ages, BMI above 70, or other clearly wrong numbers). Explain why each is implausible and propose how you would handle it (drop, set to `NA`, contact data source).

2.5. Solutions

Problem 1.

```
library(tidyverse)
cohort <- read_csv("data/cohort.csv")
glimpse(cohort)
#> Rows: 500
#> Columns: 8 (etc.)
nrow(cohort); ncol(cohort)
```

Problem 2.

```
sub <- cohort |>
  filter(age >= 40, bmi >= 25)
nrow(sub)
```

Problem 3.

```
cohort |>
  group_by(sex) |>
  summarise(across(where(is.numeric), \(x) mean(x, na.rm = TRUE)))
```

`across()` plus `where(is.numeric)` applies the same summary function to every numeric column. The `\(x) mean(x, na.rm = TRUE)` lambda is R's anonymous function syntax (also written `function(x) ...`).

Problem 4.

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

latest_labs <- labs |>
  group_by(id) |>
  slice_max(visit_date, n = 1, with_ties = FALSE) |>
  ungroup()

combined <- cohort |>
  left_join(latest_labs, by = "id")

nrow(combined) == nrow(cohort)
#> [1] TRUE
```

The `with_ties = FALSE` argument resolves ties (two visits on the same day for the same patient) by taking the first; in practice you might want a more deliberate rule.

Problem 5. Implausible values for the synthetic dataset. For each, two reasonable handling decisions:

- A negative age is a data-entry or sentinel value (e.g., -99 used to encode missing). Set to `NA`. Document the decision.
- BMI above 70 is biologically possible but rare; verify with the source. If verification is impossible, treat as `NA` for any analysis whose validity depends on reasonable BMI distributions, with a sensitivity analysis including the value.
- Systolic blood pressure of 0 is impossible (the patient would not be alive). Set to `NA`. Document.

The general lesson: data-quality issues are the analyst's to identify and to handle deliberately. Silent imputation, dropping, or inclusion of clearly wrong data all bias the result; documented handling does not.

2.6. What's next

Day 3 covers visualisation with `ggplot2`. We will use the cohort dataset from Day 2's worked example, so save your processed `cohort` object at the end of Day 2.

3. Day 3: Visualisation with ggplot2

3.1. Learning objectives

By the end of this day you should be able to:

- Build a `ggplot2` figure from scratch using the grammar of graphics: data, aesthetics, geoms.
- Choose the right geom for the data (`geom_point`, `geom_line`, `geom_bar`, `geom_histogram`, `geom_boxplot`).
- Customise scales, labels, and themes.
- Use `facet_wrap` to produce small multiples.
- Save figures as PNG and PDF at appropriate dimensions and resolution.

3.2. Lecture

`ggplot2` is the standard graphics package for the R ecosystem. It is built on the **grammar of graphics**: a deliberate vocabulary that lets you compose any figure from a small number of orthogonal pieces. Once you know the grammar, every plot is a recombination.

3.2.1. The grammar in one paragraph

A `ggplot2` figure has three required pieces and several optional ones:

- **data**: a tibble or data frame.
- **aesthetic mapping**: which columns map to which visual channels (x, y, colour, shape, size).
- **geom**: the geometric form (point, line, bar, histogram, boxplot).

Plus optional:

- **scales**: how aesthetic values map to visual values (continuous to log, categorical to a colour palette).
- **facets**: a small-multiples grid by some categorical variable.
- **theme**: non-data ink (axes, grid, fonts).

A minimal plot:

```
library(tidyverse)

ggplot(cohort, aes(x = age, y = sbp)) +
  geom_point()
```

Read it as: 'Take the `cohort` tibble, map `age` to the x-axis and `sbp` to the y-axis, and draw points.' Each new layer is added with `+`.

3.2.2. Common geoms

`geom_point` for scatter:

```
ggplot(cohort, aes(x = age, y = sbp, colour = sex)) +
  geom_point(alpha = 0.5)
```

`alpha = 0.5` makes points semi-transparent, helpful for overplotting.

`geom_histogram` for the distribution of one continuous variable:

```
ggplot(cohort, aes(x = bmi)) +
  geom_histogram(binwidth = 1)
```

`binwidth` controls bin width in data units. The default is sometimes too coarse or too fine; iterate.

`geom_boxplot` for distributions across categories:

```
ggplot(cohort, aes(x = age_group, y = sbp)) +
  geom_boxplot()
```

geom_bar for counts:

```
ggplot(cohort, aes(x = race)) +
  geom_bar()
```

geom_line for time series or any x-y where x is ordered:

```
ggplot(daily_admits, aes(x = date, y = n_admissions)) +
  geom_line()
```

geom_smooth adds a smoother. The default method depends on sample size: LOESS for n $<$ 1000, `gam` (a generalised additive model) for larger samples. Use `method = "lm"` for a linear fit, or specify any other method explicitly:

```
ggplot(cohort, aes(x = age, y = sbp)) +
  geom_point(alpha = 0.4) +
  geom_smooth(method = "lm")
```

Layers compose. The points and the line above are drawn on top of one another in the order they are added.

3.2.3. Aesthetic mapping vs. fixed values

Inside `aes()`: column → visual channel. Outside `aes()`: a fixed value applied to every observation.

```
# Wrong (alpha is data-driven; it is not a column)
ggplot(cohort, aes(x = age, y = sbp, alpha = 0.5)) +
  geom_point()

# Right (alpha is a fixed value)
```

```
ggplot(cohort, aes(x = age, y = sbp)) +
  geom_point(alpha = 0.5)

# Right (colour mapped to the sex column)
ggplot(cohort, aes(x = age, y = sbp, colour = sex)) +
  geom_point()
```

When you see strange figures (a single hue ramp where you expected discrete colours, an alpha legend you did not ask for), it is usually `aes()`-vs.-not confusion.

3.2.4. Scales

Continuous scales:

```
ggplot(cohort, aes(x = bmi, y = fasting_glucose)) +
  geom_point() +
  scale_y_log10()                       # log y axis
```

For colour:

```
ggplot(cohort, aes(x = age, y = sbp, colour = bmi)) +
  geom_point() +
  scale_colour_viridis_c()              # continuous viridis
```

For categorical colour, the default is fine for a few levels; for more, `scale_colour_brewer()` and `scale_colour_manual()` give control.

3.2.5. Labels and themes

```
ggplot(cohort, aes(x = age, y = sbp, colour = sex)) +
  geom_point(alpha = 0.5) +
  geom_smooth(method = "lm") +
  labs(title = "Systolic blood pressure rises with age",
```

```
       subtitle = "NHANES-style synthetic cohort, n = 500",
       x = "Age (years)",
       y = "Systolic blood pressure (mmHg)",
       colour = "Sex",
       caption = "Source: synthetic data") +
  theme_minimal(base_size = 12)
```

`theme_minimal()` strips the default grey background. Other themes: `theme_bw()`, `theme_classic()`, `theme_void()`. The `base_size` sets the default font size.

3.2.6. Facets

Small multiples are the right way to compare a relationship across groups:

```
ggplot(cohort, aes(x = age, y = sbp)) +
  geom_point(alpha = 0.4) +
  geom_smooth(method = "lm") +
  facet_wrap(~ sex)
```

`facet_wrap(~ var)` makes one panel per level of `var` arranged in a grid. `facet_grid(rows ~ cols)` makes a two-way grid.

3.2.7. Saving figures

```
p <- ggplot(cohort, aes(x = age, y = sbp)) +
  geom_point() + geom_smooth()

ggsave("figures/sbp-by-age.png", p, width = 6, height = 4,
       dpi = 300)
ggsave("figures/sbp-by-age.pdf", p, width = 6, height = 4)
```

PNG for screen and most slides; PDF for print and publication. The `dpi = 300` for PNG keeps the figure crisp at print size; PDF is vector and has no dpi.

> **Note**
>
> A figure for a journal article typically wants a specific width (often 3.5 inches single-column or 7 inches two-column). Match the journal's specification at save time so the figure does not need rescaling.

3.3. Worked example: figures from yesterday's analysis

Take yesterday's `cohort` and produce three figures: a histogram of age, a coloured scatter of SBP vs. BMI, a faceted comparison of fasting glucose by age group.

```
library(tidyverse)
cohort <- read_csv("data/cohort.csv")

# 1. Histogram of age
p1 <- ggplot(cohort, aes(x = age)) +
  geom_histogram(binwidth = 5, fill = "steelblue",
                 colour = "white") +
  labs(title = "Age distribution",
       x = "Age (years)", y = "Count") +
  theme_minimal(base_size = 12)

# 2. Scatter of SBP vs. BMI, coloured by sex
p2 <- ggplot(cohort, aes(x = bmi, y = sbp, colour = sex)) +
  geom_point(alpha = 0.6) +
  geom_smooth(method = "lm", se = FALSE) +
  labs(x = "BMI (kg/m^2)", y = "SBP (mmHg)",
       colour = "Sex") +
  theme_minimal(base_size = 12)

# 3. Faceted boxplot of fasting glucose by age group
p3 <- ggplot(cohort, aes(x = age_group, y = fasting_glucose,
                         fill = sex)) +
  geom_boxplot() +
```

```
  labs(x = "Age group", y = "Fasting glucose (mg/dL)",
       fill = "Sex") +
  theme_minimal(base_size = 12)

# save
ggsave("figures/age-hist.png", p1, width = 5, height = 4,
       dpi = 300)
ggsave("figures/sbp-bmi.png",  p2, width = 6, height = 4,
       dpi = 300)
ggsave("figures/glucose-box.png", p3, width = 6, height = 4,
       dpi = 300)
```

These three figures together would make a respectable 'descriptive figures' panel in a journal article.

3.4. Homework

1. **Recreate four figures.** For each of the four specifications below, produce the figure from yesterday's `cohort` dataset.

   (a) Histogram of `bmi` with bin width 1 and the title 'BMI distribution'.

   (b) Scatter of `fasting_glucose` (y) vs. `bmi` (x), coloured by `diabetic`, with a linear smoother per group, log-scaled y-axis.

   (c) Boxplot of `sbp` by `age_group` faceted by `sex`.

   (d) Bar chart of count by `race`, sorted from most common to least common.

2. **Log scale.** Plot `fasting_glucose` (y) against `bmi` (x) with both natural and log y-axes (two figures, side by side or sequential). When does the log scale change what you see? When does it not?

3. **Journal theme.** Customise figure 1(b) with a journal-style theme: white background, grid only on the y-axis, larger axis font, a specified colour palette (e.g., `scale_colour_manual(values = c("FALSE" = "grey50", "TRUE" = "firebrick"))`).

4. **Faceted comparison.** Plot the relationship between `sbp` (y) and `bmi` (x) with `geom_point` plus `geom_smooth`, faceted by `age_group` (three panels). Comment in 1-2 sentences on what the facets reveal.
5. **Save.** Save figures (a) through (d) from problem 1 as both PNG (300 dpi) and PDF, with widths 5 inches for histogram and 6 inches for the others.

3.5. Solutions

Problem 1.

```
# (a)
ggplot(cohort, aes(x = bmi)) +
  geom_histogram(binwidth = 1, fill = "steelblue",
                 colour = "white") +
  labs(title = "BMI distribution",
       x = "BMI (kg/m^2)", y = "Count") +
  theme_minimal()

# (b)
ggplot(cohort, aes(x = bmi, y = fasting_glucose,
                   colour = diabetic)) +
  geom_point(alpha = 0.6) +
  geom_smooth(method = "lm", se = FALSE) +
  scale_y_log10() +
  labs(x = "BMI (kg/m^2)",
       y = "Fasting glucose (mg/dL, log scale)",
       colour = "Diabetic") +
  theme_minimal()

# (c)
ggplot(cohort, aes(x = age_group, y = sbp)) +
  geom_boxplot() +
  facet_wrap(~ sex) +
  labs(x = "Age group", y = "SBP (mmHg)") +
  theme_minimal()
```

```r
# (d)
cohort |>
  count(race) |>
  mutate(race = fct_reorder(race, n, .desc = TRUE)) |>
  ggplot(aes(x = race, y = n)) +
  geom_col(fill = "steelblue") +
  labs(x = "Race", y = "Count") +
  theme_minimal()
```

`fct_reorder()` reorders the factor levels by the count; `geom_col()` is `geom_bar(stat = "identity")` (use the y-values as-is rather than counting).

Problem 2. The log scale changes what you see when the underlying distribution is right-skewed (typical for fasting glucose and most lab values). The log axis linearises a multiplicative relationship; differences at the high end appear less compressed.

Problem 3.

```r
ggplot(cohort, aes(x = bmi, y = fasting_glucose,
                   colour = diabetic)) +
  geom_point(alpha = 0.6) +
  geom_smooth(method = "lm", se = FALSE) +
  scale_y_log10() +
  scale_colour_manual(values = c("FALSE" = "grey50",
                                 "TRUE"  = "firebrick"),
                      labels = c("No", "Yes")) +
  labs(x = "BMI (kg/m^2)",
       y = "Fasting glucose (mg/dL, log)",
       colour = "Diabetic") +
  theme_classic(base_size = 14) +
  theme(panel.grid.major.y = element_line(),
        panel.grid.minor.y = element_line())
```

`theme_classic()` is the closest to a typical journal look. The `theme()` modifier adds the y-axis grid back.

Problem 4.

```
ggplot(cohort, aes(x = bmi, y = sbp)) +
  geom_point(alpha = 0.4) +
  geom_smooth(method = "lm") +
  facet_wrap(~ age_group) +
  labs(x = "BMI", y = "SBP") +
  theme_minimal()
```

The relationship between BMI and SBP is positive in every age group; the slope appears similar across groups but the intercept is higher in the older group (because SBP rises with age independent of BMI).

Problem 5.

```
ggsave("figures/bmi-hist.png", p_a,  width = 5, height = 4, dpi = 300)
ggsave("figures/bmi-hist.pdf", p_a,  width = 5, height = 4)
ggsave("figures/glucose-bmi.png", p_b, width = 6, height = 4, dpi = 300)
ggsave("figures/glucose-bmi.pdf", p_b, width = 6, height = 4)
ggsave("figures/sbp-box.png", p_c,   width = 6, height = 4, dpi = 300)
ggsave("figures/sbp-box.pdf", p_c,   width = 6, height = 4)
ggsave("figures/race-bar.png", p_d,  width = 6, height = 4, dpi = 300)
ggsave("figures/race-bar.pdf", p_d,  width = 6, height = 4)
```

3.6. What's next

Day 4 covers writing functions, control flow, and basic applied statistics. We will revisit the `cohort` dataset from Days 2 and 3 and run statistical tests on it.

4. Day 4: Functions, Control Flow, Applied Statistics

4.1. Learning objectives

By the end of this day you should be able to:

- Write a small R function with named arguments, defaults, and a single return value.
- Use `if/else` and `purrr::map` (in preference to explicit loops) for control flow.
- Compute summary statistics with `summary()`, `mean()`, `sd()`, `median()`, `quantile()`.
- Run a two-sample `t.test()` and a `chisq.test()` and interpret the output.
- Fit a simple linear regression with `lm()` and interpret the coefficients, residuals, and the `summary()` output.

4.2. Lecture

Day 4 takes you from 'I can read and visualise data' to 'I can compute statistics on it'. The four ideas are: write a function for anything you do more than once; prefer `purrr::map` over explicit loops; know the basic summary statistics; know how to run a t-test, a chi-squared test, and a simple regression. With these in place you can do most of the descriptive and exploratory work an MS-level biostatistician produces.

4.2.1. Writing functions

A function is a named, reusable piece of code that takes inputs (arguments) and returns a single output:

```
cv <- function(x) {
  sd(x, na.rm = TRUE) / mean(x, na.rm = TRUE)
}

cv(c(2.1, 2.3, 1.9, 2.5, 2.0))
#> [1] 0.1131371
```

Anatomy:

- `cv` is the name.
- `function(x)` declares one argument named `x`.
- Inside `{ }` is the body.
- The last expression is automatically returned (no explicit `return()` needed).

Defaults:

```
summarise_lab <- function(x, na.rm = TRUE) {
  c(n      = sum(!is.na(x)),
    mean   = mean(x, na.rm = na.rm),
    sd     = sd(x,   na.rm = na.rm),
    median = median(x, na.rm = na.rm))
}

summarise_lab(c(7.2, 6.8, NA, 8.1, 7.4))
#>    n      mean        sd    median
#> 4.000 7.375000 0.5468079 7.300000
```

The `na.rm = TRUE` default means the function silently removes missing values unless the caller overrides. Be deliberate about defaults; they affect the answer.

> **Note**
>
> The rule of thumb: write a function the third time you copy-paste a piece of code. Earlier than that and you spend more time generalising than the duplication costs.

4.2.2. Control flow: `if/else` and `case_when`

Inside a function:

```
classify_bp <- function(sbp) {
  if (sbp < 120) {
    "normal"
  } else if (sbp < 130) {
    "elevated"
  } else if (sbp < 140) {
    "stage 1"
  } else {
    "stage 2"
  }
}
classify_bp(118)
#> [1] "normal"
classify_bp(145)
#> [1] "stage 2"
```

`if/else` is fine for one value at a time. For a vector, use `case_when()` (introduced in Day 2):

```
patients <- patients |>
  mutate(bp_category = case_when(
    sbp < 120 ~ "normal",
    sbp < 130 ~ "elevated",
    sbp < 140 ~ "stage 1",
    TRUE      ~ "stage 2"))
```

4.2.3. `purrr::map` over loops

R has explicit `for` loops, and they work, but the tidyverse idiom is to use `map()` to apply a function to each element of a list or vector:

```
library(purrr)

# compute mean by group with explicit map
groups <- list(
  young  = c(110, 115, 122, 118),
  middle = c(125, 130, 132, 128),
  older  = c(138, 145, 142, 140)
)

map_dbl(groups, mean)
#>  young middle  older
#> 116.25 128.75 141.25
```

`map_dbl()` applies `mean` to each element and returns a numeric vector (`map` returns a list; `map_dbl`, `map_int`, `map_chr`, `map_lgl` return typed vectors). `map_dfr` returns a row-bound tibble.

For most applied biostatistics, the equivalent operation inside `dplyr` is `group_by` + `summarise`. Reach for `map` when working with lists of model fits, lists of files, lists of countries — anything that is genuinely list-shaped rather than tabular.

4.2.4. Summary statistics

`summary()` is the first-look function:

```
summary(cohort$age)
#>    Min.  1st Qu.  Median    Mean   3rd Qu.    Max.
#>   18.00   34.00   47.00   46.32    59.00   89.00
```

Per-statistic functions:

```r
mean(cohort$bmi)
sd(cohort$bmi)
median(cohort$bmi)
quantile(cohort$bmi, c(0.25, 0.50, 0.75, 0.95))
IQR(cohort$bmi)
```

For a count of categorical:

```r
table(cohort$sex)
#> F   M
#> 110 90

prop.table(table(cohort$sex))
#>    F    M
#> 0.55 0.45
```

For two-way:

```r
table(cohort$sex, cohort$diabetic)
#>     FALSE TRUE
#>   F    97    13
#>   M    77    13
```

4.2.5. t-test

The two-sample t-test compares means between two groups:

```r
t.test(sbp ~ sex, data = cohort)
#>
#>     Welch Two Sample t-test
#>
#> data:  sbp by sex
#> t = -2.31, df = 188.4, p-value = 0.022
#> alternative hypothesis: true difference in means
#>     between group F and group M is not equal to 0
#> 95 percent confidence interval:
```

```
#>  -9.74  -0.78
#> sample estimates:
#> mean in group F  mean in group M
#>           126.1            131.4
```

R defaults to **Welch's** t-test (unequal variances), which is the right default. The ~ (tilde) is R's formula syntax: 'sbp explained by sex'.

What to read in the output:

- `t = -2.31`: the test statistic.
- `df = 188.4`: degrees of freedom (non-integer because Welch).
- `p-value = 0.022`: the p-value for the two-sided test.
- `95 percent confidence interval: -9.74 -0.78`: the 95% CI for the difference in means.
- The sample means in each group.

A one-sentence interpretation: women have lower SBP than men by about 5.3 mmHg on average; the difference is statistically significant at $= 0.05$; the 95% CI for the difference is (-9.7, -0.8).

> **Note**
>
> A statistically significant t-test does not by itself mean clinically important. A 5 mmHg difference in SBP across sex is real but small relative to the within- person variation. The chapter on causal inference in the *Applied Methods* volume develops this distinction at length.

4.2.6. chi-squared test

The chi-squared test of independence compares distributions across a contingency table:

```
chisq.test(table(cohort$sex, cohort$diabetic))
#>
#>     Pearson's Chi-squared test with Yates' continuity correction
#>
```

```
#> data:  table(cohort$sex, cohort$diabetic)
#> X-squared = 0.07, df = 1, p-value = 0.79
```

Read the test statistic, df, and p-value as before. For sparse tables (any expected cell count under 5), use Fisher's exact test instead:

```
fisher.test(table(cohort$sex, cohort$diabetic))
```

A common applied error is to use chi-squared on a sparse table; the p-value is unreliable. Check the expected counts first:

```
chisq.test(table(cohort$sex, cohort$diabetic))$expected
```

4.2.7. Simple linear regression

The fit:

```
fit <- lm(sbp ~ age, data = cohort)
fit
#>
#> Call:
#> lm(formula = sbp ~ age, data = cohort)
#>
#> Coefficients:
#> (Intercept)          age
#>     106.21         0.46
```

The intercept (106.21) is the predicted SBP at age 0; the slope (0.46) is the predicted change in SBP per one-year increase in age.

The summary:

```
summary(fit)
#>
#> Call:
#> lm(formula = sbp ~ age, data = cohort)
```

```
#>
#> Residuals:
#>      Min       1Q   Median       3Q      Max
#> -28.45   -7.23     0.31     7.18     33.42
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept) 106.2147     2.0341    52.2  < 2e-16 ***
#> age           0.4623     0.0411    11.3  < 2e-16 ***
#>
#> Residual standard error: 11.4 on 498 degrees of freedom
#> Multiple R-squared:  0.203, Adjusted R-squared:  0.201
#> F-statistic: 126.7 on 1 and 498 DF,  p-value: < 2e-16
```

What to read:

- **Coefficients**: estimate, standard error, t, p-value. The slope on age is 0.46 (SE 0.04, p < 2e-16).
- **Residual standard error**: 11.4 mmHg. Interpret as the typical size of the prediction error.
- **R-squared**: 0.203. About 20% of the variance in SBP is explained by age alone.
- **F-statistic**: tests whether the model overall is better than the intercept-only model.

For multiple predictors:

```
fit2 <- lm(sbp ~ age + sex + bmi, data = cohort)
summary(fit2)
```

Each coefficient is interpreted holding the others constant. The interpretation gets harder once you include terms that are correlated; the *Applied Methods* volume's chapter on causal inference treats this carefully.

4.3. Worked example: an end-to-end exploratory analysis

Load the cohort, write a small function to summarise a lab value, run a t-test on a sex difference, fit a linear regression of SBP on age and BMI.

```
library(tidyverse)
cohort <- read_csv("data/cohort.csv") |>
  mutate(diabetic = fasting_glucose >= 126,
         age_group = case_when(
           age < 40 ~ "<40",
           age < 60 ~ "40-59",
           TRUE     ~ "60+"))

# function to summarise a numeric variable per group
group_summary <- function(data, group_var, value_var) {
  data |>
    group_by({{ group_var }}) |>
    summarise(n = n(),
              mean = mean({{ value_var }}, na.rm = TRUE),
              sd   = sd({{ value_var }},   na.rm = TRUE),
              .groups = "drop")
}

group_summary(cohort, sex, sbp)
#> # A tibble: 2 × 4
#>   sex       n  mean    sd
#>   <chr> <int> <dbl> <dbl>
#> 1 F       110  126.   12.3
#> 2 M        90  131.   13.1

# is the SBP difference between sexes significant?
t.test(sbp ~ sex, data = cohort)

# is the proportion of diabetics different between sexes?
chisq.test(table(cohort$sex, cohort$diabetic))
```

```r
# fit a regression of SBP on age and BMI
fit <- lm(sbp ~ age + bmi + sex, data = cohort)
summary(fit)
```

Reading the regression: SBP rises with age (= 0.42, SE 0.04, p < 0.001) and with BMI (= 0.85, SE 0.20, p < 0.001), and is lower in women (= -3.2, SE 1.1, p = 0.005) holding age and BMI constant. R^2 is around 0.27.

The `{ }` (curly-curly) syntax in the function above is how `dplyr` and friends pass column names as arguments into a function. You will use this idiom enough that it becomes muscle memory; for now, treat it as the recipe for 'pass a column name into a tidyverse function'.

4.4. Homework

1. **Three small functions.**
   (a) Write a function `cv()` that takes a numeric vector and returns its coefficient of variation (standard deviation divided by mean). Handle missing values with a `na.rm = TRUE` default.
   (b) Write a function `or_2x2()` that takes a 2×2 contingency table (a matrix or a `table` object) and returns the odds ratio. Use Haldane's correction (add 0.5 to each cell) if any cell count is zero.
   (c) Write a function `bonferroni()` that takes a vector of p-values and returns the Bonferroni-corrected p-values (each multiplied by the number of tests, capped at 1).

2. **Two-sample t-test.** Test whether mean BMI differs between diabetic and non-diabetic patients in the cohort. Report the test statistic, p-value, 95% CI for the difference, and group means. State the conclusion in one sentence.

3. **Chi-squared.** Test whether `age_group` and `diabetic` are independent. Report the test statistic, p-value, and the expected counts. Comment on whether Fisher's exact would have been more appropriate.

4. **Simple linear regression.** Fit `sbp ~ bmi` in the cohort. Report the slope estimate, its SE, t-statistic, and p-value. Interpret the slope in plain English using the data's units.
5. **Multiple regression.** Extend the model in problem 4 to `sbp ~ bmi + age + sex`. Compare the slope on `bmi` between the two models. Why does it change?

4.5. Solutions

Problem 1.

```
cv <- function(x, na.rm = TRUE) {
  sd(x, na.rm = na.rm) / mean(x, na.rm = na.rm)
}

or_2x2 <- function(tab) {
  # tab is a 2x2 with exposed/unexposed in rows, case/control in cols
  if (any(tab == 0)) tab <- tab + 0.5
  (tab[1,1] * tab[2,2]) / (tab[1,2] * tab[2,1])
}

bonferroni <- function(p) {
  pmin(p * length(p), 1)
}
```

Problem 2.

```
t.test(bmi ~ diabetic, data = cohort)
#> ...
#> t = -3.45, df = 67.2, p-value = 0.001
#> 95 CI: (-3.42, -0.93)
#> means: non-diabetic 27.4, diabetic 29.6
```

Diabetic patients have higher mean BMI than non- diabetic patients by about 2.2 kg/m² (95% CI 0.9-3.4, p = 0.001).

Problem 3.

```
chisq.test(table(cohort$age_group, cohort$diabetic))
#> X-squared = 22.1, df = 2, p-value = 1.6e-5
```

Expected counts (from `chisq.test(...)$expected`) are all above 5, so chi-squared is appropriate. Fisher's exact would not be necessary; it would give a similar answer but is computationally heavier.

Problem 4.

```
fit_simple <- lm(sbp ~ bmi, data = cohort)
summary(fit_simple)
#> bmi: estimate = 1.32, SE = 0.18, t = 7.3, p < 0.001
```

A 1 kg/m² increase in BMI is associated with a 1.32 mmHg increase in SBP.

Problem 5.

```
fit_multi <- lm(sbp ~ bmi + age + sex, data = cohort)
summary(fit_multi)
#> bmi: estimate = 0.85, SE = 0.20, t = 4.3, p < 0.001
```

The BMI slope drops from 1.32 to 0.85 when age and sex are added. The interpretation: in the simple model, the BMI coefficient absorbs some of the age-and-sex relationship with SBP (older and male patients tend to have higher BMI and higher SBP). Adjusting for age and sex isolates the BMI-SBP relationship from those confounders. The careful version of this story is the subject of the *Causal inference* chapters in *Applied Methods*.

4.6. What's next

Day 5 closes the boot camp with reproducibility (Quarto, basic Git through RStudio) and a primer on using AI assistance responsibly. After Day 5 the companion volumes pick up: each thread of statistical methodology (longitudinal, survival, causal, Bayesian, ML) has a home in one of the five companion books.

5. Day 5: Reproducibility and AI Assistance

5.1. Learning objectives

By the end of this day you should be able to:

- Build a one-page Quarto report combining prose, R code, and figures into a self-contained HTML or PDF.
- Use RStudio Projects to keep an analysis self-contained and portable.
- Initialise a Git repository, commit changes, and view the history through the RStudio IDE.
- Use AI assistance (ChatGPT, Claude) responsibly: recognise the failure modes, verify generated code before trusting it, and document the AI's involvement.
- Find the right companion volume for any topic that goes beyond the boot camp.

5.2. Lecture

The first four days have given you R as a calculator: you can read data, manipulate it, plot it, and compute statistics on it. Day 5 turns that into reproducible research output and introduces the AI tools you will already be using by the end of your first quarter. Nothing here is technically deep; everything here is the table-stakes you will be expected to have on day one.

5.2.1. Quarto

Quarto is the document format for combining prose, R code, and code output (tables, figures) into a single file. The same source file renders to HTML for the web, PDF for print, and Word for collaborators who insist on it.

A minimal Quarto document (`report.qmd`):

````
---
title: "My first analysis"
author: "Your name"
date: today
format: html
---

# Introduction

This report describes the cohort I assembled on Day 2.

# Methods

```{r}
#| label: setup
#| include: false
library(tidyverse)
cohort <- read_csv("data/cohort.csv")
```

The cohort has `r nrow(cohort)` patients across
`r length(unique(cohort$sex))` sex groups.

# Results

## Descriptive statistics

```{r}
cohort |>
  group_by(sex) |>
````
````

````
  summarise(n = n(),
            mean_age = mean(age),
            mean_sbp = mean(sbp))
```

```{r}
#| fig-cap: "Systolic blood pressure by sex."
ggplot(cohort, aes(x = sex, y = sbp)) +
  geom_boxplot() +
  theme_minimal()
```

# Conclusion

Mean SBP is higher in men than in women in this cohort.
````

Render with the **Render** button in RStudio (or `quarto render report.qmd` in the terminal). The output is a self-contained HTML file with prose, code, code output, tables, and figures interleaved.

The pieces:

- **YAML front matter** at the top (between `---` fences): metadata like title, author, output format.
- **Markdown** for prose: `#` for headings, `*italic*`, `**bold**`, etc.
- **Code chunks** in fenced blocks marked ```` ```{r} ````: R code that runs and (by default) shows both code and output.
- **Inline R** with `` `r ...` ``: R expression evaluated and substituted.
- **Chunk options** with `#|`: e.g., `#| include: false` hides the chunk from output; `#| echo: false` hides the code but shows the output; `#| fig-cap: "..."` adds a figure caption.

A single Quarto report replaces the typical split between analysis script + exported figures + Word document. The document is reproducible: render again later, get the same output.
````

5.2.2. RStudio Projects

A project is a folder with a `.Rproj` marker file. When you open the project, RStudio sets the working directory to the project root and remembers your open files and session state.

Create one: **File → New Project → New Directory → New Project**, give it a name and a location, click Create.

Inside the project, organise files by convention:

```
my-project/
├── my-project.Rproj
├── README.md
├── data/
│   ├── raw/
│   └── processed/
├── R/                      # scripts
├── analysis/
│   └── report.qmd
├── figures/
└── output/
```

The benefits of using projects:

- Paths are relative to the project root. `read_csv("data/cohort.csv")` works no matter where the project lives on your machine.
- The project is self-contained. Zip the folder, send to a collaborator, they unzip and open the `.Rproj` file and everything works.
- Each project has its own R session (no leftover variables from another analysis).

Always work inside a project. The `getwd()` function should return the project root.

5.2.3. Git through the IDE

Git tracks the history of changes to the files in your project. RStudio has a **Git** pane (top-right, between Environment and Files when Git is enabled) that shows all four operations a beginner needs.

To turn on Git for an existing project: **Tools → Project Options → Git/SVN → Version control system: Git**. RStudio prompts you to commit; do so.

The four operations:

1. **Status**: which files have changed since the last commit. Shown automatically in the Git pane: `M` (modified), `?` (untracked), `A` (added), `D` (deleted).
2. **Add (stage)**: select files in the pane and click the checkbox to stage. Staging is 'I want to include this in the next commit'.
3. **Commit**: click the Commit button, write a one-line message describing what changed, click Commit. The commit is a permanent snapshot.
4. **History**: click the History button (clock icon) to see all past commits with messages, authors, and diffs.

That covers single-developer Git. For collaboration, push and pull (working with a remote like GitHub) are the next two operations; they live in the same pane and are covered in the *Practicum* volume.

A working pattern for daily research:

- Start the day by writing one-line commits for the changes you made yesterday (if you forgot at the time).
- Commit at the end of each meaningful chunk of work ('cleaned the cohort', 'added SBP-by-age figure', 'wrote first draft of methods').
- Use descriptive commit messages. 'Updates' is useless six months later; 'fix typo in age-group cutoff' tells you what changed.

5.2.4. Using AI assistance responsibly

You are using ChatGPT, Claude, or a similar tool right now. So is every other student in your programme. The question is not whether to use them; it is how to use them well.

What AI is good for:

- **Boilerplate.** 'Write me a `dplyr` pipeline that filters to adults, groups by sex, and computes mean age and BMI.' This is tedious-to-type code that the AI generates correctly more often than not.
- **Translation.** 'Translate this `lm` formula to `glm` with a logistic family.' Format-conversion is cheap to verify (run it; check the output).
- **Explanation.** 'What does `na.rm = TRUE` do in `mean()`?' Documentation lookup is fine.
- **Debugging.** 'I get `Error: object "sbp" not found`, here is my code.' Often catches a typo or a missing column.

What AI is **not** good for:

- **Statistical judgement.** 'Should I use a t-test or a Wilcoxon?' The AI will pick one and justify it plausibly. It does not know your context.
- **Real-data verification.** AI-generated code frequently does what the AI thinks the data looks like, not what your data actually looks like. Run the code on your data; don't take its claims about output at face value.
- **Hallucinated functions.** AI will sometimes generate `dplyr::summarise_groups` (does not exist) or `tidyr::pivot_wider_with_progress()` (made up). Run the code; let R's 'function not found' error catch the hallucination.

The discipline:

1. **Read the code before you run it.** Understand each line. If you cannot, ask the AI to explain (and verify the explanation).
2. **Run on edge cases.** What does the code do on an empty input? On a vector with `NA`s? On a single-row tibble?
3. **Compare to a reference.** Cross-check against the package documentation (`?dplyr::summarise`).

4. **Document the AI involvement.** When you use AI in a real analysis, note which parts of the work were AI-assisted and how you verified them. The transparency is now an explicit reviewer expectation in many journals.

The deeper version of this material — context engineering, reasoning models, agents, evaluation harnesses — is the subject of *Applied Generative AI for Public Health and Biostatistics*. For the boot camp, the discipline above is enough.

5.3. Worked example: a one-page Quarto report

Build a one-page Quarto report from yesterday's analysis, commit it to a Git repository, render to HTML.

Step 1. Create the project. **File → New Project → New Directory → New Project**. Name it `cohort-analysis`.

Step 2. Save your data. Copy `cohort.csv` into a `data/` subfolder.

Step 3. Create `report.qmd`. Use the template at the top of the lecture, filling in the analysis pieces from Day 4 (the t-test, the regression, a figure).

Step 4. Render. Click the **Render** button. RStudio produces `report.html` and (if PDF is configured) `report.pdf`.

Step 5. Initialise Git. **Tools → Project Options → Git/SVN**. Stage all files and commit with the message 'Initial cohort analysis report'.

Step 6. Read your output. Open `report.html` in a browser. Confirm the prose, code, output, and figure all appear as expected.

Step 7. Iterate. Make a change to the report (add a new section, modify a figure). Re-render. Commit. Look at the History to see the two commits.

The whole flow takes 30-60 minutes the first time and about 5 minutes the second.

5.4. Homework

1. **Build a Quarto report.** Build a one-page Quarto report from your Day-4 analysis. Include: a brief prose introduction (3-4 sentences), the data import and cohort-construction code, a descriptive table by group (e.g., `summarise()` per `sex`), a figure (one from Day 3), a t-test or chi-squared test, and a 2-sentence conclusion. Render to HTML.

2. **Initialise Git.** Initialise a Git repository for your project. Commit the report and the data file (or a `.gitignore` that excludes the data; either is fine for this exercise). Add a README.md explaining what the analysis is. Commit again. Look at the history.

3. **AI assistance, with verification.** Open ChatGPT, Claude, or a similar tool. Ask it to suggest two improvements to your Day-4 analysis code (paste the code into the chat). Verify each suggestion by running the modified code. Document one suggestion that improved your code and one that was wrong, with a sentence explaining how you identified the wrong one.

4. **Companion-volume look-ahead.** Read the table-of-contents page for one of the four companion volumes (URLs at the end of this chapter). Identify two topics in the TOC that you expect to encounter in your first quarter of graduate study.

5. **Reflection.** Write a one-paragraph reflection on what you most need to practice in the first month of the programme. Be specific: not 'I should learn more R' but 'I need to get faster at writing pipelines that combine `filter`, `mutate`, `group_by`, and `summarise` without looking up the syntax each time'.

5.5. Solutions

Problem 1. A reasonable Quarto report follows the template at the top of the lecture and adds your own analysis. The render should produce a self-contained HTML file under 1 MB. If your render fails, the most common causes are: missing package (`install.packages()` the missing package),

wrong file path (the path is relative to the .qmd file), or a typo in YAML front matter (check colons and indentation).

Problem 2. A typical first commit history:

```
8a4f2e1  Add t-test and regression to report
3c9f1ad  Initial cohort analysis report
```

If your `data/cohort.csv` is large or sensitive, the right pattern is: - Add `data/` to `.gitignore`. - Document in the README how to obtain the data. - Commit the report (which references the data) but not the data itself.

Problem 3. Two examples of what 'verification' looks like:

- *Suggestion that improved the code.* The AI suggests replacing `cohort %>% filter(...)` with the native pipe `cohort |> filter(...)` for consistency. You verify by running the new pipeline; the output is identical. Adopt the suggestion.
- *Suggestion that was wrong.* The AI suggests `cohort |> dplyr::summarise_groups(mean(sbp))` to compute the per-group mean. You run it and get 'function not found'. The function does not exist; the AI hallucinated. The correct code is `cohort |> group_by(sex) |> summarise(mean(sbp))`.

The pattern: run the code. AI failures are usually loud (errors); the silent failures (subtly wrong output) require deeper checking.

Problems 4 and 5. Open-ended; the goal is to start forming the connection between today's bootcamp and your programme's actual curriculum.

5.6. What's next: the five companion volumes

The boot camp ends here. The next thread of the curriculum picks up in one of the companion volumes, depending on what your programme requires next:

- **Reproducibility infrastructure** (Git in depth, Docker, renv, Quarto book authoring, CDISC, SAS, AI-assisted coding): *Biostatistics Practicum* at https://rgtlab.org/practicum.
- **Statistical methods and computing** (linear models, GLM, mixed models, survival, Bayesian computation, simulation, bootstrap, ggplot2 advanced, Shiny, parallel R, packages): *Statistical Computing in the Age of AI* at https://scai.rgtlab.org.
- **Advanced computing** (numerical stability, MCMC in depth, HPC, high-dimensional methods, ML, software engineering): *Advanced Statistical Computing in the Age of AI* at https://scai-advanced.rgtlab.org.
- **Generative AI integration** (RAG, agents, evaluation, regulation, deployment): *Applied Generative AI for Public Health and Biostatistics* at https://applied-genai.rgtlab.org.
- **Applied methodological core** (causal inference, longitudinal at applied depth, survival applied, clinical trial design, missing data at depth, meta-analysis, advanced categorical): *Applied Statistical Methods for Public Health* at https://applied-methods.rgtlab.org.

The boot camp gave you the entry-level R competence the companion volumes assume. The companion volumes assume the rest of your graduate biostatistics training is underway in parallel; they build the methods curriculum on top of the R-side mechanics you now have.

Good luck with the programme.

References

Credits

The chapter list was constructed by surveying 12 publicly-available syllabi from major US biostatistics programmes (documented in `docs/syllabi-survey.md`) and adopting the topics that appeared in three or more of them.

Cover artwork generated procedurally by the Python script at `images/build-cover.py`. Watercolour palette: deep moss through forest and sage to lime and pale cream, anchored on a brand forest green `#2d6e3a`. Typography: Avenir family (Apple system font); body text in Source Serif 4.

Colophon

This book was produced with Quarto, typeset in Source Serif 4, with code blocks set in JetBrains Mono. Source code is in R 4.4 or later.

The book is hosted at https://scai-advanced.rgtlab.org on Netlify, with continuous deployment via GitHub Actions from the `rgt47/scai-advanced` repository on every push to `main`. The deployment recipe is in `HOSTING.md`.

The cover is generated procedurally by `images/build-cover.py` from a watercolour gradient anchored on `#7a2c4e` (the volume's brand burgundy) and overlaid with Avenir typography. Re-running the script regenerates the cover; the procedural approach makes the cover reproducible and editable in version control rather than dependent on a one-shot AI image-generation step.

Last rendered: see the build timestamp on the homepage.