Statistical Computing and Non-Parametric Inference Using R

3 Credits (L = 2, P = 1)

Dr. Shrikrishna Bhat K 

2026-07-23

Statistical Computing and Non-Parametric Inference Using R

Author
Dr. Shrikrishna Bhat K

Affiliation
Assistant Professor
Department of Applied Statistics and Data Science
Prasanna School of Public Health
Manipal Academy of Higher Education (MAHE)
Manipal, Karnataka, India

This book is an independently authored open educational resource developed by the author for teaching and learning purposes. The views expressed are those of the author and do not necessarily reflect the views of Manipal Academy of Higher Education or the Prasanna School of Public Health.

Interactive R Package: `remotes::install_github("kskbhat/scnpir")`
Online Book: https://kskbhat.github.io/Teaching/scnpir/

Contents

Course Overview

Welcome to **Statistical Computing and Non-Parametric Inference Using R (3 Credits: L = 2, P = 1)**.

Course Outcomes (COs)

On successful completion of this course, students will be able to:

- **CO1**: Demonstrate the ability to use the R programming environment for data handling, workspace management, and manipulation of different data structures.
- **CO2**: Compute and interpret descriptive statistical measures and perform basic statistical summaries using R.
- **CO3**: Apply programming constructs such as conditional statements, loops, and user-defined functions in R to automate statistical computations.
- **CO4**: Implement resampling techniques, contingency table analyses, and generate structured statistical summaries and reports using R.
- **CO5**: Select, apply, and interpret appropriate non-parametric statistical tests for different data situations using R.

Course Structure

Unit	Topic	Hours
Unit 1	Introduction to R and Data Structures	8
Unit 2	Descriptive Statistics and Programming Fundamentals in R	10
Unit 3	Confidence Interval Estimation and Resampling Methods	12
Unit 4	Analysis of Contingency Tables and Compact Reporting	15
Unit 5	Non-Parametric Tests and Applications Using R	15

Interactive Tutorials (`scnpir` R Package)

Interactive `learnr` tutorials for this course are provided via the **`scnpir`** R package. Install and run them in RStudio:

```
if (!requireNamespace("remotes", quietly = TRUE)) install.packages("remotes")
remotes::install_github("kskbhat/scnpir")

library(scnpir)
learnr::run_tutorial("unit1_r_basics", package = "scnpir")
```

References

1. R Core Team. *An introduction to R*. Vienna: R Foundation for Statistical Computing; 2023.
2. Wickham H, Grolemund G. *R for data science: import, tidy, transform, visualize, and model data*. Sebastopol (CA): O'Reilly Media; 2017.
3. Venables WN, Ripley BD. *Modern applied statistics with S*. 4th ed. New York: Springer; 2002.
4. Efron B, Tibshirani RJ. *An introduction to the bootstrap*. New York: Chapman & Hall/CRC; 1993.
5. Hollander M, Wolfe DA, Chicken E. *Nonparametric statistical methods*. 3rd ed. Hoboken (NJ): Wiley; 2014.
6. Conover WJ. *Practical nonparametric statistics*. 3rd ed. New York: Wiley; 1999.
7. Agresti A. *Categorical data analysis*. 3rd ed. Hoboken (NJ): Wiley; 2013.

Chapter 1

Introduction to R and Data Structures

By the end of this chapter, you will be able to:

- Navigate the RStudio interface and use all four panes effectively
- Write well-structured, commented R scripts with proper headers
- Create and manipulate vectors, matrices, arrays, data frames, factors, and lists
- Import and export data from CSV, Excel (.xlsx), and other file formats
- Install and load R packages, and understand the namespace system

> **Why this chapter?** R is the world's most widely used statistical programming language in academia and research. Before any analysis can happen, you need to understand the tools and building blocks. This chapter sets up your analytical workspace — by the end, you will have everything you need to load a real dataset and begin exploring it.

- **Computer literacy**: Basic familiarity with files, folders, and installing software
- **Mathematics**: Basic arithmetic (no calculus or advanced math needed here)
- **R/RStudio**: Must be installed — download from CRAN and RStudio

Welcome to your first chapter! By the end of this chapter, you'll be comfortable navigating RStudio, managing your workspace, importing data from several file formats, and working confidently with every core R data structure: vectors, matrices, arrays, data frames, factors, and lists.

1.1 The R and RStudio Environment

R is the *language*. RStudio is the *program* you use to write and run R code comfortably — think of R as the engine and RStudio as the car built around it.

When you open RStudio, you'll typically see four panes:

Pane	Location	What it's for
Source	Top-left	Where you write and save scripts (`.R` files) — your permanent, reusable code
Console	Bottom-left	Where code actually runs, one command at a time
Environment/History	Top-right	Shows every object currently in memory, and what commands you've run
Files/Plots/Packages/Help	Bottom-right	Browse files, view graphs, manage packages, read documentation

The **Source** pane is for code you want to keep. The **Console** is for code you want to try out once. A common beginner habit is typing everything directly into the Console — resist this. Write it in a script so you (and your future self) can re-run it later.

1.1.1 R Script Structure: Headers, Comments, and Timestamps

When writing code in R scripts (`.R` files) or R Markdown (`.Rmd`), maintaining a consistent, well-documented structure is essential for reproducibility and clarity.

1.1.1.1

1. Script Header Every R script should start with a header comment block describing the script's metadata (course, name, author, date, and description).

```
# ============================================================================
# Course: Statistical Computing and Non-Parametric Inference Using R
# Script: Unit 1 - R Basics & Data Structures
# Author: Shrikrishna Bhat K
# Date: 2026-07-13
# Description: Examples and exercises covering basic R commands and data structures.
# ============================================================================
```

1.1.1.2

2. Code Comments Use the `#` symbol in R to add comments. Comments are ignored by R during execution and are used to explain *why* the code does what it does.

```
x <- 5  # Assign the value 5 to object x
```

1.1.1.3

3. Execution Timestamps Adding a timestamp print statement at the beginning or end of your script execution is a best practice. It documents exactly when the calculations were run.

```
# Print current date and time
Sys.time()
```

```
#> [1] "2026-07-23 17:09:55 UTC"
```

```
# Print formatted date and time
cat("Execution Timestamp:", format(Sys.time(), "%Y-%m-%d %H:%M:%S"), "\n")
```

```
#> Execution Timestamp: 2026-07-23 17:09:55
```

1.1.2 Getting help

Every function and package in R has documentation. Here are the core ways to find help:

```r
# 1. Open documentation for a specific function
?mean
help(median)

# 2. Search all installed help pages for a keyword or phrase
??"confidence interval"
help.search("weighted mean")

# 3. Open the documentation index for a specific package
help(package = "readxl")
```

1.1.3 Using Packages & Namespaces

R functions are organized into **packages**. To use a package's functions, you must first install it once from CRAN, and then load it into your current R session:

```r
# Install once from CRAN
install.packages("dplyr")

# Load it in every new script
library(dplyr)
```

Sometimes you only want to use a single function from a package without loading the entire package into your workspace, or you want to prevent name conflicts (when two packages have functions with the same name). In R, you can access a function directly from its **namespace** using the double colon (`::`) operator:

```r
# Use the select() function from the dplyr package explicitly
dplyr::select(mtcars, mpg, hp)
```

This namespace notation is highly recommended for clean, reproducible code!

1.2 Workspace Management

Your **workspace** is the collection of every object (data, variables, functions) currently loaded in R's memory.

```r
alpha <- 5
b <- "hello"
scores <- c(88, 76, 92, 65)

ls()  # list everything currently in your workspace
```

```
#> [1] "alpha"  "b"      "scores" "x"
```

Removing objects:

```r
rm(b)
ls()
```

```
#> [1] "alpha"  "scores" "x"
```

Clearing everything:

```
rm(list = ls())   # wipes the entire workspace - use with care!
```

Saving and restoring your work:

```
save.image("my_session.RData")   # saves everything
load("my_session.RData")          # restores it later
```

Working directory — this is the folder R looks in by default when you read or write files:

```
getwd()                       # where am I?
setwd("~/Documents/RCourse") # change to alpha specific folder
```

`setwd()` with a hardcoded path (like `"C:/Users/Yourname/Desktop"`) will break the moment someone else — or you, on a different computer — tries to run your script. If you're working in an RStudio **Project** (`.Rproj` file), you never need `setwd()` at all; the working directory is set automatically. This habit will save you real pain later.

1. Create three objects: your age (numeric), your name (character), and a logical value for whether you've used R before.
2. List your workspace with `ls()`.
3. Remove just the logical value.
4. Confirm it's gone.

1.3 Importing Data

Real analysis starts with real data, and real data rarely arrives as R code — it arrives as files. R can read almost any format; here are the three you'll use constantly.

This section uses **`student_scores`**, a small dataset of 40 students with their group, exam score, and hours studied — provided in three formats (`.csv`, `.xlsx`, `.txt`) so you can practice importing all three. See the Appendix for its full column description.

1.3.1 CSV files (the workhorse format)

```
scores_csv <- read.csv("data/student_scores.csv")
head(scores_csv, 3)
```

```
#>   id       name group score hours_studied
#> 1  1 Student_01     A    63           4.7
#> 2  2 Student_02     A    66           5.3
#> 3  3 Student_03     A    49           4.6
```

1.3.2 Excel files

```
library(readxl)
scores_xlsx <- read_excel("data/student_scores.xlsx")
head(scores_xlsx, 3)
```

1.3.3 Tab-delimited text files

```
scores_txt <- read.table("data/student_scores.txt", header = TRUE, sep = "\t")
head(scores_txt, 3)
```

```
#>   id       name group score hours_studied
#> 1  1 Student_01     A    63           4.7
#> 2  2 Student_02     A    66           5.3
#> 3  3 Student_03     A    49           4.6
```

1.3.4 Inspecting what you just imported

Three functions you should run on *every* dataset the moment you load it:

```
dim(scores_csv)     # rows x columns
```

```
#> [1] 40  5
```

```
str(scores_csv)     # structure: column names, types, first few values
```

```
#> 'data.frame':    40 obs. of  5 variables:
#>  $ id           : int  1 2 3 4 5 6 7 8 9 10 ...
#>  $ name         : chr  "Student_01" "Student_02" "Student_03" "Student_04" ...
#>  $ group        : chr  "A" "A" "A" "B" ...
#>  $ score        : int  63 66 49 89 69 95 92 69 77 92 ...
#>  $ hours_studied: num  4.7 5.3 4.6 4.4 1.5 4.5 1.4 7.3 3 5.3 ...
```

```
summary(scores_csv) # quick numeric summary of every column
```

```
#>        id              name              group               score       hours_studied
#>  Min.   : 1.00   Length   :40   Length   :40   Min.   :49.00   Min.   :1.100
#>  1st Qu.:10.75   N.unique :40   N.unique : 2   1st Qu.:56.00   1st Qu.:3.075
#>  Median :20.50   N.blank  : 0   N.blank  : 0   Median :69.00   Median :4.650
#>  Mean   :20.50   Min.nchar:10   Min.nchar: 1   Mean   :68.47   Mean   :4.840
#>  3rd Qu.:30.25   Max.nchar:10   Max.nchar: 1   3rd Qu.:76.25   3rd Qu.:6.450
#>  Max.   :40.00                                 Max.   :95.00   Max.   :9.800
```

`str()` is arguably the single most useful function in this entire chapter. It instantly tells you: how many rows and columns you have, what type each column is (numeric, character, factor…), and a preview of the actual values. Make it a habit to run `str()` immediately after importing anything.

Notice that `read.csv()` and `read_excel()` can treat text columns differently — `read.csv()` gives you plain character vectors by default, while `read_excel()` also returns characters, but numeric columns from Excel can sometimes import with unexpected decimal points or date formats if the original spreadsheet was formatted oddly. Always run `str()` and check.

Import `data/student_scores.txt` yourself, then check: does `str()` show `group` as a character column or something else? What about in the `.csv` version? Are they the same?

1.3.5 Interactive Editing inside RStudio

While we typically import prepared files, R also provides a built-in spreadsheet-like GUI editor for making quick changes to a data frame or creating new data from scratch.

- `edit()`: Opens the editor. Crucially, **`edit()` returns the modified data frame**, so you must assign the result back to an object to save your changes (e.g., `my_data <- edit(my_data)`).

- `fix()`: A shortcut that modifies the data frame in-place without needing an assignment (e.g., `fix(my_data)`).

Never run `edit()` or `fix()` inside an R Markdown (`.Rmd`) file that you plan to compile/render. Because these functions open an interactive window, the compiler will hang indefinitely waiting for user input, causing the build to fail. Only use them interactively in the R Console (we demonstrate them using `eval=FALSE` in code chunks so they don't run during compile).

Here is how you can use them:

```
# INTERACTIVE ONLY - Never run inside an Rmd code block during render!
# Edit an existing dataset
scores_edited <- edit(scores_csv)

# INTERACTIVE ONLY
# Edit in-place
fix(scores_csv)
```

Historical Note: The `stringsAsFactors` Shift In older R versions (prior to version 4.0.0), any character string column imported via `read.csv()` or `read.table()` was automatically converted into a `factor` by default. To prevent this, developers had to write `stringsAsFactors = FALSE` on almost every import. Since R 4.0.0, the default behavior has changed to `stringsAsFactors = FALSE`. You will still see `stringsAsFactors = FALSE` in older textbooks, online code snippets, and legacy projects. In modern R, it is rarely needed unless you explicitly want to override a factor conversion!

Exercise: Create a new dataset using `edit()`

Since we cannot run interactive GUI functions during Bookdown compiling, try this in your RStudio Console:

1. Create a blank template with columns of different classes (character, integer, logical): `my_log <- data.frame(Name = character(), Age = integer(), Present = logical(), stringsAsFactors = FALSE)`
2. Run the editor: `my_log <- edit(my_log)`
3. A grid will pop up. Enter 3 rows of values. R will automatically enforce the column types you defined (e.g., entering letters in the `Age` column or `Present` column will warn you or coerce them).
4. Close the grid. Run `str(my_log)` to verify that your entries were saved and that the column classes remain correct.

1.4 Vectors

A **vector** is R's most basic data structure: an ordered collection of values, *all of the same type.*

Visual Mental Map of R Data Structures

Before dive-bombing into vectors, use this simple mental map to visualize R's primary data structures based on their **dimensionality** and **type constraints**:

	Type Constraints	
	Homogeneous	Heterogeneous
1D	Vector	List (Flexible)
2D	Matrix	Data Frame
N-D	Array	-

Note: A List is technically 1-dimensional (it is a vector of elements), but it is recursive and recursive lists can contain other lists, making it the most flexible container in R.

1.4.1 Creating vectors

```
ages <- c(23, 45, 31, 29, 52)
names_vec <- c("Asha", "Ravi", "Meera", "Kiran", "Divya")
passed <- c(TRUE, FALSE, TRUE, TRUE, FALSE)

ages
```

```
#> [1] 23 45 31 29 52
```

```
class(ages)
```

```
#> [1] "numeric"
```

```
class(names_vec)
```

```
#> [1] "character"
```

```
class(passed)
```

```
#> [1] "logical"
```

Other common ways to build vectors:

```
seq(1, 20, by = 4)        # alpha sequence
```

```
#> [1]  1  5  9 13 17
```

```
seq_len(6)                 # 1 through 6
```

```
#> [1] 1 2 3 4 5 6
```

```
rep(c("Low", "High"), times = 3)   # repeat alpha pattern
```

```
#> [1] "Low"  "High" "Low"  "High" "Low"  "High"
```

```
rep(1:3, each = 2)                  # repeat each element
```

```
#> [1] 1 1 2 2 3 3
```

1.4.2 Indexing and Subsetting Vectors

Subsetting means pulling out specific elements — and this is where a lot of real data wrangling happens. You can subset by position (using indices), by value (using logical criteria), or by name:

```
# 1. Subsetting by position
ages[1]             # first element
```

```
#> [1] 23
```

```
ages[c(1, 3)]       # first and third
```

```
#> [1] 23 31
```

```
ages[-1]            # everything EXCEPT the first
```

```
#> [1] 45 31 29 52
```

```
ages[2:4]           # elements two to four
```

```
#> [1] 45 31 29
```

```
ages[-(2:4)]        # all elements except two to four
```

```
#> [1] 23 52
```

```
# 2. Subsetting by value (logical indexing)
ages[ages > 30]     # only ages above 30
```

```
#> [1] 45 31 52
```

```
ages[ages %in% c(23, 52)] # only elements that are in the set {23, 52}
```

```
#> [1] 23 52
```

Logical indexing — `ages[ages > 30]` — is one of the most powerful ideas in R. `ages > 30` produces a vector of `TRUE`/`FALSE` values (one per element), and putting that inside `[ ]` keeps only the elements where it's `TRUE`. You will use this constantly, for the rest of the course and beyond.

Named Vectors

You can also assign names to the elements of a vector, allowing you to subset them by name (like lookup tables):

```
# Assign names to the ages vector
names(ages) <- c("Asha", "Ravi", "Meera", "Kiran", "Divya")
ages
```

```
#>  Asha  Ravi Meera Kiran Divya
#>    23    45    31    29    52
```

```
# Subset by name
ages["Meera"]
```

```
#> Meera
#>    31
```

```
ages[c("Asha", "Divya")]
```

```
#>  Asha Divya
#>    23    52
```

1.4.3 Essential Vector Functions

R provides several built-in functions to sort, summarize, and clean up vectors:

```
# 1. sort() - return vector sorted (default ascending)
sort(ages)
```

```
#>  Asha Kiran Meera  Ravi Divya
#>    23    29    31    45    52
```

```
sort(ages, decreasing = TRUE)
```

```
#> Divya  Ravi Meera Kiran  Asha
#>    52    45    31    29    23
```

```
# 2. rev() - return vector reversed
rev(ages)
```

```
#> Divya Kiran Meera  Ravi  Asha
#>    52    29    31    45    23
```

```
# 3. unique() - see unique values
unique(c(1, 1, 2, 3, 3, 3))
```

```
#> [1] 1 2 3
```

```
# 4. table() - count frequencies of values
table(c("Yes", "No", "Yes", "Yes", "No"))
```

```
#>
#>  No Yes
#>   2   3
```

1.4.4 Vectorized operations

R applies math to whole vectors at once — no loop required.

```
heights_cm <- c(150, 162, 158, 170, 145)
heights_m <- heights_cm / 100
heights_m
```

```
#> [1] 1.50 1.62 1.58 1.70 1.45
```

```
bonus <- c(2, 2, 2, 2, 2)
ages + bonus
```

```
#>  Asha  Ravi Meera Kiran Divya
#>    25    47    33    31    54
```

Example with a second dataset — `mtcars` (built into R):

```
data(mtcars)
head(mtcars[, c("mpg", "hp")], 4)
```

```
#>                   mpg  hp
#> Mazda RX4        21.0 110
#> Mazda RX4 Wag    21.0 110
#> Datsun 710       22.8  93
#> Hornet 4 Drive   21.4 110
```

```
# Convert miles-per-gallon to km-per-litre for every car at once
mtcars$kpl <- mtcars$mpg * 0.425
head(mtcars$kpl)
```

```
#> [1] 8.9250 8.9250 9.6900 9.0950 7.9475 7.6925
```

Using the `ages` vector above: (1) extract everyone under 30; (2) compute everyone's age in months; (3) count how many people are 30 or older using `sum(ages >= 30)` (this works because `TRUE` counts as 1 and `FALSE` as 0).

1.4.5 Converting Between Data Types (Coercion)

In R, you will frequently need to convert a variable from one data type to another. This process is called **coercion**. R provides a set of `as....()` functions for this purpose:

Function	Output Data Type	R Example	Output
`as.logical()`	Logical (`TRUE`/`FALSE`)	`as.logical(c(1, 0, 5))`	`TRUE FALSE TRUE`
`as.numeric()`	Numeric (Floating point/double)	`as.numeric(c(TRUE, FALSE))`	`1 0`
`as.character()`	Character (Text string)	`as.character(c(1.5, TRUE))`	`"1.5" "TRUE"`
`as.factor()`	Factor (Categorical with levels)	`as.factor(c("A", "B", "A"))`	`A B A` (Levels: A B)

Let's see coercion in action:

```
# 1. Convert logicals to numbers (useful for summing TRUEs)
logical_vec <- c(TRUE, FALSE, TRUE)
as.numeric(logical_vec)
```

```
#> [1] 1 0 1
```

```r
# 2. Convert character strings to numbers (must contain valid numbers!)
char_num <- c("24", "45", "30")
as.numeric(char_num)
```

```
#> [1] 24 45 30
```

```r
# 3. Warning: invalid conversions return NA (Not Available)
as.numeric(c("24", "Asha", "30"))
```

```
#> [1] 24 NA 30
```

R Coercion Hierarchy If you try to combine different data types inside a single vector using `c()`, R will automatically coerce them to the most flexible type to maintain homogeneity. The hierarchy from least flexible (strictest) to most flexible is:

$$\text{Logical} \longrightarrow \text{Integer} \longrightarrow \text{Numeric} \longrightarrow \text{Character}$$

For example, combining a number and a text string will result in R converting everything to characters: `c(10, "hello")` becomes `c("10", "hello")`.

1.5 Matrices

A **matrix** extends a vector to two dimensions — rows and columns — but like a vector, every element must be the same type.

```r
scores_mat <- matrix(c(85, 90, 78, 92, 88, 76, 95, 81, 89),
                     nrow = 3, ncol = 3, byrow = TRUE)
rownames(scores_mat) <- c("Test1", "Test2", "Test3")
colnames(scores_mat) <- c("StudentA", "StudentB", "StudentC")
scores_mat
```

```
#>       StudentA StudentB StudentC
#> Test1       85       90       78
#> Test2       92       88       76
#> Test3       95       81       89
```

Indexing a matrix uses `[row, column]`:

```r
scores_mat[1, ]        # entire first row
```

```
#> StudentA StudentB StudentC
#>       85       90       78
```

```r
scores_mat["Test2", ]  # same idea, using the row name
```

```
#> StudentA StudentB StudentC
#>       92       88       76
```

```r
scores_mat[, "StudentC"]  # entire column
```

```
#> Test1 Test2 Test3
#>    78    76    89
```

```
scores_mat[2, 3]        # single element
```

```
#> [1] 76
```

Row/column summaries:

```
rowMeans(scores_mat)
```

```
#>    Test1    Test2    Test3
#> 84.33333 85.33333 88.33333
```

```
colMeans(scores_mat)
```

```
#> StudentA StudentB StudentC
#> 90.66667 86.33333 81.00000
```

```
apply(scores_mat, 1, max)   # max of each row (margin = 1)
```

```
#> Test1 Test2 Test3
#>    90    92    95
```

```
apply(scores_mat, 2, min)   # min of each column (margin = 2)
```

```
#> StudentA StudentB StudentC
#>       85       81       76
```

Second example — a distance matrix from `mtcars`:

```
subset_cars <- mtcars[1:5, c("mpg", "hp", "wt")]
dist_mat <- as.matrix(dist(subset_cars))
round(dist_mat, 1)
```

```
#>                   Mazda RX4 Mazda RX4 Wag Datsun 710 Hornet 4 Drive
#> Mazda RX4               0.0           0.3       17.1            0.7
#> Mazda RX4 Wag           0.3           0.0       17.1            0.5
#> Datsun 710             17.1          17.1        0.0           17.1
#> Hornet 4 Drive          0.7           0.5       17.1            0.0
#> Hornet Sportabout      65.0          65.0       82.1           65.1
#>                   Hornet Sportabout
#> Mazda RX4                      65.0
#> Mazda RX4 Wag                  65.0
#> Datsun 710                     82.1
#> Hornet 4 Drive                 65.1
#> Hornet Sportabout               0.0
```

In `apply(X, MARGIN, FUN)`, `MARGIN = 1` means "do this for every **row**" and `MARGIN = 2` means "do this for every **column**." Students often mix these up — a helpful trick: 1 looks like a tall vertical stroke (rows go down), 2 has a flat bottom (columns go across).

1.6 Arrays

An **array** generalizes a matrix to three or more dimensions. Think of it as stacking several matrices of the same shape.

```
quarterly_sales <- array(
  data = 1:24,
  dim = c(2, 3, 4),   # 2 rows, 3 columns, 4 "layers" (quarters)
  dimnames = list(
    c("ProductA", "ProductB"),
    c("RegionN", "RegionS", "RegionE"),
    c("Q1", "Q2", "Q3", "Q4")
  )
)
quarterly_sales[, , "Q1"]     # the entire Q1 matrix
```

```
#>          RegionN RegionS RegionE
#> ProductA       1       3       5
#> ProductB       2       4       6
```

```
quarterly_sales["ProductA", "RegionS", ]  # ProductA in RegionS, across all quarters
```

```
#> Q1 Q2 Q3 Q4
#>  3  9 15 21
```

```
dim(quarterly_sales)
```

```
#> [1] 2 3 4
```

Using `quarterly_sales`: extract every value for `ProductB` in `RegionE`, across all four quarters. Then compute the total across all quarters for `ProductA` in `RegionN`.

1.7 Data Frames

A **data frame** is the structure you will use the most in this entire course. It looks like a spreadsheet: rows are observations, columns are variables, and — unlike a matrix — *different columns can hold different types.*

```
survey <- data.frame(
  id = 1:5,
  age = c(23, 45, 31, 29, 52),
  city = c("Chennai", "Pune", "Chennai", "Delhi", "Pune"),
  passed_exam = c(TRUE, FALSE, TRUE, TRUE, FALSE)
)
survey
```

```
#>   id age    city passed_exam
#> 1  1  23 Chennai        TRUE
#> 2  2  45    Pune       FALSE
#> 3  3  31 Chennai        TRUE
#> 4  4  29   Delhi        TRUE
#> 5  5  52    Pune       FALSE
```

```
str(survey)
```

```
#> 'data.frame':    5 obs. of  4 variables:
#>  $ id         : int  1 2 3 4 5
#>  $ age        : num  23 45 31 29 52
#>  $ city       : chr  "Chennai" "Pune" "Chennai" "Delhi" ...
#>  $ passed_exam: logi  TRUE FALSE TRUE TRUE FALSE
```

Selecting columns and rows:

```
survey$age              # alpha single column, by name
```

```
#> [1] 23 45 31 29 52
```

```
survey[, "city"]        # same thing, different syntax
```

```
#> [1] "Chennai" "Pune"    "Chennai" "Delhi"   "Pune"
```

```
survey[survey$age > 30, ]   # rows where age > 30 (logical indexing on alpha data
 ↪  frame!)
```

```
#>   id age    city passed_exam
#> 2  2  45    Pune       FALSE
#> 3  3  31 Chennai        TRUE
#> 5  5  52    Pune       FALSE
```

```
survey[1:2, c("id", "city")]  # specific rows AND columns
```

```
#>   id    city
#> 1  1 Chennai
#> 2  2    Pune
```

Adding a new column:

```
survey$age_group <- ifelse(survey$age < 35, "Younger", "Older")
survey
```

```
#>   id age    city passed_exam age_group
#> 1  1  23 Chennai        TRUE   Younger
#> 2  2  45    Pune       FALSE     Older
#> 3  3  31 Chennai        TRUE   Younger
#> 4  4  29   Delhi        TRUE   Younger
#> 5  5  52    Pune       FALSE     Older
```

Second example — built-in `iris` dataset:

```
data(iris)
str(iris)
```

```
#> 'data.frame':    150 obs. of  5 variables:
#>  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
#>  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
#>  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
#>  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
#>  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

```
# Average petal length per species
tapply(iris$Petal.Length, iris$Species, mean)
```

```
#>     setosa versicolor  virginica
#>      1.462      4.260      5.552
```

`tapply(X, INDEX, FUN)` applies a function to `X`, split by groups defined in `INDEX`. It's a compact way to answer "what's the average of *this*, broken down by *that group*?" — you'll use this pattern constantly in Chapter 2.

1.8 Factors

A **factor** represents categorical data with a fixed, known set of possible values (called *levels*). Factors look like character vectors but behave very differently under the hood.

```
severity <- factor(c("Low", "High", "Medium", "Low", "High"),
                   levels = c("Low", "Medium", "High"))
severity
```

```
#> [1] Low    High   Medium Low    High
#> Levels: Low Medium High
```

```
levels(severity)
```

```
#> [1] "Low"    "Medium" "High"
```

```
as.integer(severity)   # internally, factors are stored as integers!
```

```
#> [1] 1 3 2 1 3
```

Ordered factors carry ranking information:

```
severity_ordered <- factor(c("Low", "High", "Medium"),
                           levels = c("Low", "Medium", "High"),
                           ordered = TRUE)
severity_ordered
```

```
#> [1] Low    High   Medium
#> Levels: Low < Medium < High
```

```
severity_ordered[1] < severity_ordered[2]   # "Low" < "Medium" - TRUE, because it's
 ↪  ordered
```

```
#> [1] TRUE
```

A plain (unordered) factor doesn't support `<` or `>` comparisons in any meaningful way, even if it looks like it should. If your categories have a natural order (Low/Medium/High, Mild/Moderate/Severe), always set `ordered = TRUE` — Chapter 4 depends on this for trend tests.

Second example — using `PlantGrowth` (built into R):

```
data(PlantGrowth)
str(PlantGrowth)   # 'group' is already alpha factor with 3 levels
```

```
#> 'data.frame':    30 obs. of  2 variables:
#>  $ weight: num  4.17 5.58 5.18 6.11 4.5 4.61 5.17 4.53 5.33 5.14 ...
#>  $ group : Factor w/ 3 levels "ctrl","trt1",..: 1 1 1 1 1 1 1 1 1 1 ...
```

```
levels(PlantGrowth$group)
```

```
#> [1] "ctrl" "trt1" "trt2"
```

```r
table(PlantGrowth$group)   # how many observations per level
```

```
#>
#> ctrl trt1 trt2
#>   10   10   10
```

1.9 Lists

A **list** is the most flexible data structure in R. Unlike vectors, matrices, or data frames, a list does not enforce any structure or data type constraints. It is an ordered collection where *each element can be a completely different type or size* — a list can contain vectors, matrices, data frames, or even other lists.

Let's create a list to record a student's profile:

```r
student_record <- list(
  name = "Anjali Rao",
  scores = c(78, 85, 91, 88),
  passed = TRUE,
  metadata = data.frame(term = "Fall 2026", credits = 4)
)
print(student_record)
```

```
#> $name
#> [1] "Anjali Rao"
#>
#> $scores
#> [1] 78 85 91 88
#>
#> $passed
#> [1] TRUE
#>
#> $metadata
#>        term credits
#> 1 Fall 2026       4
```

1.9.1 The Big List Indexing "Gotcha" (Single vs. Double Brackets)

One of the most common syntax errors students face in R is the difference between single brackets `[ ]`, double brackets `[[ ]]`, and the dollar sign `$`. Let's break it down using a simple analogy:

> **The Cargo Train Analogy**: Think of a list as a train made of cargo cars. Each car contains some cargo inside it.
>
> - **`[` Single Brackets `]` gets you the train car(s)**. The result is always another list, even if you only select one element.
> - **`[[` Double Brackets `]]` opens the car and gets you the cargo inside**. The result is the actual object (vector, data frame, etc.) stored there.
> - **`$` Dollar Sign is a shortcut for named cargo**. It behaves exactly like `[[ ]]`, extracting the raw cargo by its name instead of its index.

Let's test this in code:

```r
# 1. Using Single Brackets [ ]
car_list <- student_record[1]
class(car_list)     # Still alpha list!
```

```
#> [1] "list"
```

```
print(car_list)
```

```
#> $name
#> [1] "Anjali Rao"
```

```
# 2. Using Double Brackets [[ ]]
cargo_val <- student_record[[1]]
class(cargo_val)    # A character vector!
```

```
#> [1] "character"
```

```
print(cargo_val)
```

```
#> [1] "Anjali Rao"
```

```
# 3. Using the Dollar Sign $
cargo_by_name <- student_record$name
class(cargo_by_name)
```

```
#> [1] "character"
```

```
print(cargo_by_name)
```

```
#> [1] "Anjali Rao"
```

1.9.2 Accessing Nested Elements

If you want to extract a sub-element inside a list (like the third score from the `scores` vector inside `student_record`), you must extract the vector first using `[[ ]]` or `$`, and then index it:

```
# Step 1: Get the vector
my_scores <- student_record$scores
# Step 2: Get the 3rd element
my_scores[3]
```

```
#> [1] 91
```

```
# Combine both steps in a single line:
student_record$scores[3]
```

```
#> [1] 91
```

1.9.3 Lists in Statistical Functions

Many statistical functions in R (such as `t.test()`, `chisq.test()`, `wilcox.test()`) calculate their results and bundle them into a list. Knowing how to extract elements from a list is crucial for automating statistical workflows.

```r
# Run a simple t-test comparing weight in two plant groups
test_result <- t.test(PlantGrowth$weight[PlantGrowth$group == "ctrl"],
                      PlantGrowth$weight[PlantGrowth$group == "trt1"])

class(test_result)     # It's alpha list!
```

```
#> [1] "htest"
```

```r
names(test_result)     # See what variables are in the list
```

```
#>  [1] "statistic"   "parameter"   "p.value"     "conf.int"    "estimate"
#>  [6] "null.value"  "stderr"      "alternative" "method"      "data.name"
```

```r
# Extract specific results:
p_val <- test_result$p.value
conf_interval <- test_result$conf.int

cat("P-value:", p_val, "\n")
```

```
#> P-value: 0.2503825
```

```r
print(conf_interval)
```

```
#> [1] -0.2875162  1.0295162
#> attr(,"conf.level")
#> [1] 0.95
```

Exercise: List Indexing Practice

1. Create a list of your favorite things: `my_favorites <- list(movies = c("Inception", "Interstellar"), rating = 9.5)`
2. Extract the second movie (`"Interstellar"`) in one line of code.
3. Run `class(my_favorites[1])` and `class(my_favorites[[1]])`. Explain in your own words why they are different.

Remember: almost every statistical test in R returns its results as a **list**. Knowing how to pull out `$p.value` or `$conf.int` from that list is a fundamental skill you will use throughout this course.

1.10 Comparing All Six Structures

Structure	Dimensions	Data Types	Typical Use
Vector	1D	Single type	A single variable's values
Matrix	2D	Single type	Numeric computation, algebra
Array	N-D	Single type	Multi-way numeric data
Data Frame	2D	Mixed (per column)	Real datasets, rows = observations
Factor	1D	Categorical labels	Grouping / categorical variables
List	Flexible	Mixed (any)	Bundling heterogeneous objects

For each of the following real-world situations, decide which structure fits best, and why:

1. The monthly temperature readings for one city over a year
2. A gradebook with student names, three test scores, and a pass/fail column
3. Sales figures for 5 products, across 4 regions, across 4 quarters

4. The blood type of 200 patients (A, B, AB, O)
5. The complete output of a clustering analysis (cluster assignments, centers, and a summary data frame)

- RStudio's four panes each have a distinct job — write reusable code in **Source**, not the Console.
- Always run `str()` immediately after importing any dataset.
- Vectors, matrices, and arrays require a single data type; data frames and lists allow mixed types.
- Logical indexing (`x[x > 30]`) is the single most useful indexing pattern you'll use all course.
- Factors store categorical data efficiently and — when `ordered = TRUE` — carry ranking information that later chapters depend on.
- Statistical test functions return **lists**; knowing how to extract `$p.value` and similar elements is essential going forward.

Next, in Chapter 2, we'll build on data frames and vectors to compute descriptive statistics and write our own R functions and loops.

Chapter 1 covered:

- **RStudio**: Source, Console, Environment, and Files/Plots panes; scripts vs. interactive commands
- **R data types**: numeric (double and integer), character, logical, complex — and automatic coercion rules
- **Core data structures**:
  - `vector`: 1D, homogeneous
  - `matrix` / `array`: 2D or nD, homogeneous
  - `data.frame`: 2D, heterogeneous — the workhorse of data analysis
  - `factor`: categorical variable with defined levels
  - `list`: heterogeneous container for any R objects
- **Data import**: `read.csv()`, `readxl::read_excel()`, `read.table()`
- **Package management**: `install.packages()` once; `library()` each session

Coming up: Chapter 2 applies these data structures to real data analysis — computing descriptive statistics, visualising distributions, and writing reusable functions and loops.

1.11 Lab 1: Building Your Analytical Workspace

Objective: Apply everything from Chapter 1 in one connected workflow.

Dataset: We will create a small dataset from scratch to simulate a course registration scenario.

1.11.1 Step 1: Write a Proper Script Header

Open a new R script (File → New File → R Script). Add this header:

```
# ==============================================================================
# Course: Statistical Computing and Non-Parametric Inference Using R
# Lab:    Chapter 1 - Building Your Analytical Workspace
# Author: [Your Name]
# Date:   [Today's Date]
# ==============================================================================
```

1.11.2 Step 2: Create and Manipulate Vectors

```r
# Student names (character vector)
students <- c("Alice", "Bob", "Cynthia", "David", "Eva",
              "Frank", "Grace", "Hani", "Ivan", "Jana")

# Mid-semester scores (numeric vector, some missing)
scores <- c(78, 85, NA, 92, 67, 88, NA, 74, 95, 81)

# Check: how many students have missing scores?
sum(is.na(scores))
```

```
#> [1] 2
```

```r
# Compute mean score, ignoring NA
mean(scores, na.rm = TRUE)
```

```
#> [1] 82.5
```

1.11.3 Step 3: Build a Data Frame

```r
course_df <- data.frame(
  student   = students,
  score     = scores,
  programme = factor(c("MPH", "MSc", "MPH", "MSc", "MSc",
                       "MPH", "MSc", "MPH", "MSc", "MPH")),
  stringsAsFactors = FALSE
)

# Inspect the structure
str(course_df)
```

```
#> 'data.frame':    10 obs. of  3 variables:
#>  $ student  : chr  "Alice" "Bob" "Cynthia" "David" ...
#>  $ score    : num  78 85 NA 92 67 88 NA 74 95 81
#>  $ programme: Factor w/ 2 levels "MPH","MSc": 1 2 1 2 2 1 2 1 2 1
```

```r
summary(course_df)
```

```
#>       student       score      programme
#>  Length   :10   Min.   :67.0   MPH:5
#>  N.unique :10   1st Qu.:77.0   MSc:5
#>  N.blank  : 0   Median :83.0
#>  Min.nchar: 3   Mean   :82.5
#>  Max.nchar: 7   3rd Qu.:89.0
#>                 Max.   :95.0
#>                 NAs    :2
```

1.11.4 Step 4: Subset and Transform

```r
# Students with scores above 80
course_df[!is.na(course_df$score) & course_df$score > 80, ]
```

```
#>    student score programme
#> 2      Bob    85       MSc
#> 4    David    92       MSc
#> 6    Frank    88       MPH
#> 9     Ivan    95       MSc
#> 10    Jana    81       MPH
```

```
# Add a Pass/Fail column
course_df$result <- ifelse(course_df$score >= 75, "Pass", "Fail")
course_df$result[is.na(course_df$score)] <- "Absent"
print(course_df)
```

```
#>    student score programme result
#> 1    Alice    78       MPH   Pass
#> 2      Bob    85       MSc   Pass
#> 3  Cynthia    NA       MPH Absent
#> 4    David    92       MSc   Pass
#> 5      Eva    67       MSc   Fail
#> 6    Frank    88       MPH   Pass
#> 7    Grace    NA       MSc Absent
#> 8     Hani    74       MPH   Fail
#> 9     Ivan    95       MSc   Pass
#> 10    Jana    81       MPH   Pass
```

1.11.5 Step 5: Save and Reload

```
write.csv(course_df, "lab1_output.csv", row.names = FALSE)
reloaded <- read.csv("lab1_output.csv")
identical(course_df$student, reloaded$student)  # Should be TRUE
```

```
#> [1] TRUE
```

1.11.6 Lab Exercises

1. Add a column `adjusted_score` that adds 5 bonus points to every non-missing score, capped at 100.
2. How many students in each programme have a score above 80? Use `table()` or manual subsetting.
3. Save the `MPH` students only to a separate CSV called `lab1_mph.csv`.

Chapter 2

Descriptive Statistics and Programming Fundamentals in R

By the end of this chapter, you will be able to:

- Compute and interpret measures of central tendency (mean, median, mode, trimmed mean) and dispersion (variance, SD, IQR, range)
- Produce and interpret histograms, boxplots, and density plots using base R and `ggplot2`
- Write reusable R functions with default arguments and proper documentation
- Use `for` loops, `while` loops, and `apply`-family functions to automate repetitive analysis
- Create publication-quality summary tables using `psych` and `gtsummary`

> **Why this chapter?** You have collected data from 200 participants in a health study — ages, pain scores, and treatment groups. Before fitting any model, you need to understand what you actually have. Are there outliers? Are the distributions skewed? Do the two treatment groups look similar at baseline? Descriptive statistics is not a preliminary formality — it is where you *discover* your data.

- **Chapter 1**: R data structures — vectors, data frames, factors
- **Basic statistics**: Concepts of mean, median, variance (definitions only — we compute them here)

In this chapter, you'll learn to summarize data numerically and start writing your own R code logic — conditionals, loops, and functions — rather than only using functions someone else wrote.

2.1 Central Tendency and Dispersion

This section uses `student_scores` (course dataset) and `mtcars` (built into R) so you see the same measures applied to two very different kinds of data.

```
scores <- read.csv("data/student_scores.csv")
head(scores, 3)
```

```
#>   id       name group score hours_studied
#> 1  1 Student_01     A    63           4.7
#> 2  2 Student_02     A    66           5.3
#> 3  3 Student_03     A    49           4.6
```

2.1.1 Measures of central tendency

```
mean(scores$score)
```

```
#> [1] 68.475
```

```
median(scores$score)
```

```
#> [1] 69
```

2.1.2 Measures of dispersion

```
sd(scores$score)
```

```
#> [1] 12.70774
```

```
var(scores$score)
```

```
#> [1] 161.4865
```

```
range(scores$score)
```

```
#> [1] 49 95
```

```
diff(range(scores$score))   # the width of the range, as alpha single number
```

```
#> [1] 46
```

The **mean** is pulled toward extreme values (outliers); the **median** is not. If a dataset has a few very unusual values, the median is often the more trustworthy "typical value." Always compute both and compare them — a large gap between mean and median is itself informative, and it's exactly the kind of signal that will motivate Chapters 3 and 5.

Second example — `mtcars`:

```
mean(mtcars$mpg)
```

```
#> [1] 20.09062
```

```
median(mtcars$mpg)
```

```
#> [1] 19.2
```

```
sd(mtcars$mpg)
```

```
#> [1] 6.026948
```

```r
# Compare a heavily-skewed variable in the same dataset
mean(mtcars$hp)
```

```
#> [1] 146.6875
```

```r
median(mtcars$hp)
```

```
#> [1] 123
```

Third example — grouped summaries with `tapply()`:

```r
tapply(scores$score, scores$group, mean)
```

```
#>        A        B
#> 67.94737 68.95238
```

```r
tapply(scores$score, scores$group, sd)
```

```
#>        A        B
#> 12.42969 13.24189
```

Using `mtcars`, compute the mean and median of `wt` (car weight) and of `qsec` (quarter-mile time). For which variable is the mean/median gap larger? What might explain that?

Useful Mathematical and Rounding Functions

Beyond basic summary metrics, statistical workflows frequently require transforming data or rounding numerical results. R provides several key functions:

Function	Description	R Example	Output
`log(x)`	Natural logarithm ($\ln(x)$)	`log(10)`	`2.302585`
`exp(x)`	Exponential (e^x)	`exp(1)`	`2.718282`
`min(x)`, `max(x)`	Minimum and maximum values	`min(c(5, 8, 2))`	`2`
`round(x, digits)`	Round to n decimal places	`round(3.14159, 2)`	`3.14`
`signif(x, digits)`	Round to n significant figures	`signif(0.01234, 2)`	`0.012`

Let's demonstrate using these on our data:

```r
# 1. Log-transforming skewed variables (common in clinical stats)
log_income <- log(c(12000, 150000, 45000))
log_income
```

```
#> [1]  9.392662 11.918391 10.714418
```

```r
# 2. Rounding the result for reporting
round(log_income, digits = 3)
```

```
#> [1]  9.393 11.918 10.714
```

```r
# 3. Rounding to 2 significant figures
signif(c(12345, 0.00567), digits = 2)
```

```
#> [1] 1.2e+04 5.7e-03
```

2.2 Quantiles, Percentiles, and Distribution Shape

```r
quantile(scores$score)                          # default: 0, 25, 50, 75, 100 percentiles
```

```
#>    0%   25%   50%   75%  100%
#> 49.00 56.00 69.00 76.25 95.00
```

```r
quantile(scores$score, probs = c(0.1, 0.9))  # custom percentiles
```

```
#>  10%  90%
#> 52.0 88.1
```

```r
IQR(scores$score)                             # interquartile range: Q3 - Q1
```

```
#> [1] 20.25
```

```r
summary(scores$score)                          # alpha quick all-in-one summary
```

```
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>   49.00   56.00   69.00   68.47   76.25   95.00
```

2.2.1 Skewness and kurtosis (built by hand)

It's worth seeing the formulas once so the concept of distribution shape is concrete. The formula for sample skewness (representing asymmetry) is:

$$Skewness = \frac{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^3}{\left(\sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2}\right)^3}$$

And the formula for sample excess kurtosis (representing tail weight compared to a normal distribution, where a normal distribution has a kurtosis of 0) is:

$$Excess\ Kurtosis = \frac{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^4}{\left(\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2\right)^2} - 3$$

Implemented manually in R, it looks like this:

```r
skewness_manual <- function(x) {
  n <- length(x)
  m <- mean(x)
  if (n < 3) return(NA)  # Handle edge cases
  (sum((x - m)^3) / n) / (sum((x - m)^2) / n)^(3/2)
}
```

```
kurtosis_manual <- function(x) {
  n <- length(x)
  m <- mean(x)
  if (n < 4) return(NA)  # Handle edge cases
  excess_kurtosis <- (sum((x - m)^4) / n) / (sum((x - m)^2) / n)^2 - 3
  return(excess_kurtosis)
}

skewness_manual(scores$score)
```

```
#> [1] 0.3423179
```

```
kurtosis_manual(mtcars$hp)
```

```
#> [1] 0.05223273
```

In practice, you do not need to build these calculations from scratch. You can use the professional functions from the **psych** package, which offer different computation types (Type 1 is the biased/raw sample value matching our manual formulas, while Type 3 is the sample-size corrected version used by default in software like SPSS and SAS):

```
library(psych)

# Calculate using package functions (Type 1 matches manual formulas exactly)
psych::skew(scores$score, type = 1)
```

```
#> [1] 0.3423179
```

```
psych::kurtosi(mtcars$hp, type = 1)
```

```
#> [1] 0.05223273
```

```
# Default package outputs (Type 3)
psych::skew(scores$score)
```

```
#> [1] 0.3295615
```

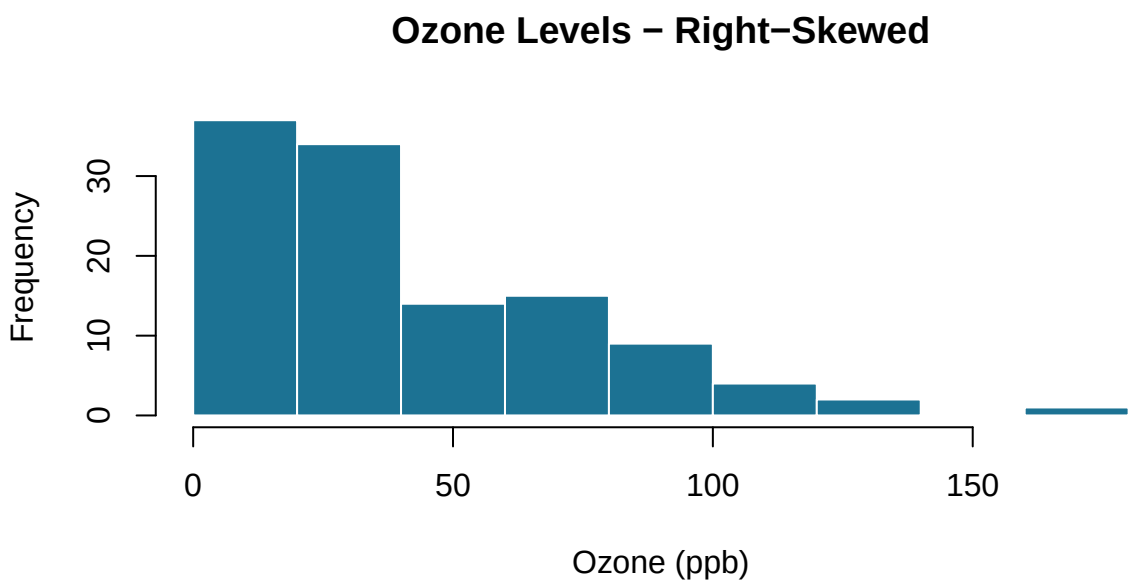
```
psych::kurtosi(mtcars$hp)
```

```
#> [1] -0.1355511
```

Skewness > 0 means a long tail toward high values (right-skewed); **skewness < 0** means a long tail toward low values (left-skewed). **Kurtosis > 3** means heavier tails than a normal distribution (more extreme outliers than you'd expect). You don't need to memorize the formulas — but understanding what the number *means* will help you decide, in Chapter 3 and Chapter 5, whether a parametric method is trustworthy.

Second example — visualizing shape with `airquality` (built into R):

```
data(airquality)
hist(airquality$Ozone, main = "Ozone Levels - Right-Skewed",
     xlab = "Ozone (ppb)", col = "#1C7293", border = "white")
```




```
skewness_manual(na.omit(airquality$Ozone))
```

```
#> [1] 1.225681
```

Compute the skewness of `chickwts$weight` (another built-in dataset — chick weights by feed type). Is it closer to symmetric or skewed? Confirm your answer with a histogram.

2.3 Theoretical Probability Distributions and the d/p/q/r Family

While descriptive statistics summarize our *observed* sample data, inferential statistics compare our sample data to *theoretical probability distributions*. R provides a highly unified system for working with theoretical distributions using a consistent naming convention: a prefix representing the function type, followed by the distribution's suffix.

Prefix Functions (The d/p/q/r Family)

There are four prefixes that you can combine with any distribution name:

Prefix	Meaning	Question it answers	R Example (Normal)
`d`	Density (probability mass/density)	"How tall is the curve at point x?"	`dnorm(x)`
`p`	Probability (cumulative distribution)	"What is the area under the curve to the left of x?"	`pnorm(x)`
`q`	Quantile (inverse cumulative)	"What x-value has exactly p area to its left?"	`qnorm(p)`
`r`	Random generation	"Give me n random draws from this distribution."	`rnorm(n)`

Distribution Suffixes

The same four prefixes apply to every standard probability distribution in R. You simply swap the suffix to target a different distribution:

Suffix	Distribution	Required Parameters	R Example (Cumulative Probability)
`norm`	Normal	`mean`, `sd` (default 0, 1)	`pnorm(1.96, mean = 0, sd = 1)`
`t`	Student's t	`df` (degrees of freedom)	`pt(2.13, df = 15)`
`chisq`	Chi-Square (χ^2)	`df`	`pchisq(3.84, df = 1)`
`f`	Fisher's F	`df1`, `df2`	`pf(4.10, df1 = 2, df2 = 20)`
`binom`	Binomial	`size` (trials), `prob`	`pbinom(3, size = 10, prob = 0.5)`

Code Demonstration

Let's test these functions in R using the standard normal distribution:

```
# 1. dnorm: height of density curve at Z = 0 (the peak)
dnorm(0)
```

```
#> [1] 0.3989423
```

```
# 2. pnorm: cumulative probability to the left of Z = 1.96 (~97.5%)
pnorm(1.96)
```

```
#> [1] 0.9750021
```

```
# 3. qnorm: find the Z-score for the 95th percentile (~1.64)
qnorm(0.95)
```

```
#> [1] 1.644854
```

```
# 4. rnorm: generate 5 random numbers from standard normal
set.seed(42)
rnorm(5)
```

```
#> [1]  1.3709584 -0.5646982  0.3631284  0.6328626  0.4042683
```

The General p-value Recipe

Understanding this `d/p/q/r` family is what unlocks manual statistical computing. Every hypothesis test calculates a *test statistic* (like t, χ^2, or W). Under the null hypothesis, this statistic follows a known theoretical distribution.

The p-value is simply the probability of getting a statistic at least as extreme as the one we calculated, assuming the null hypothesis is true. In R, we compute this "area under the curve" using a `p...()` function. For example, if we have a chi-square statistic of 5.4 with 1 degree of freedom, our p-value is: `pchisq(5.4, df = 1, lower.tail = FALSE)`

We will use this exact recipe to calculate p-values manually throughout Chapters 3, 4, and 5!

2.4 Conditional Statements

2.4.1 `if` / `else` — for single values

```
score_value <- 42
if (score_value >= 50) {
  result <- "Pass"
} else {
  result <- "Fail"
}
result
```

```
#> [1] "Fail"
```

You can chain multiple conditions with `else if`:

```
grade_letter <- function(score) {
  if (score >= 90) {
    "A"
  } else if (score >= 75) {
    "B"
  } else if (score >= 50) {
    "C"
  } else {
    "F"
  }
}
grade_letter(42)
```

```
#> [1] "F"
```

```
grade_letter(91)
```

```
#> [1] "A"
```

2.4.2 `ifelse()` — the vectorized version

`if`/`else` only works on a single value at a time. When you need to apply a condition to an *entire column at once*, use `ifelse()`:

```
scores$result <- ifelse(scores$score >= 50, "Pass", "Fail")
table(scores$result)
```

```
#>
#> Fail Pass
#>    1   39
```

Nested `ifelse()` for more than two categories:

```
scores$grade <- ifelse(scores$score >= 90, "A",
                  ifelse(scores$score >= 75, "B",
                  ifelse(scores$score >= 50, "C", "F")))
table(scores$grade)
```

```
#>
#>  A  B  C  F
#>  3  9 27  1
```

A very common mistake: using `if`/`else` directly on a whole vector or column. `if (scores$score >= 50)` will only look at the *first* element and (in newer R versions) throw an error for the rest. Whenever you're working on a full column, reach for `ifelse()`, not `if`.

Second example — with `PlantGrowth`:

```
data(PlantGrowth)
PlantGrowth$high_yield <- ifelse(PlantGrowth$weight > median(PlantGrowth$weight),
                                 "Above Median", "Below Median")
table(PlantGrowth$group, PlantGrowth$high_yield)
```

```
#>
#>        Above Median Below Median
#>   ctrl            5            5
#>   trt1            2            8
#>   trt2            8            2
```

Using `mtcars`, create a new column `efficiency` that labels each car `"Efficient"` if `mpg > 20` and `"Gas Guzzler"` otherwise, using `ifelse()`.

2.5 Looping Constructs

2.5.1 `for` loops

Use a `for` loop when you know exactly how many times you want to repeat something.

```
for (i in 1:5) {
  cat("The square of", i, "is", i^2, "\n")
}
```

```
#> The square of 1 is 1
#> The square of 2 is 4
#> The square of 3 is 9
#> The square of 4 is 16
#> The square of 5 is 25
```

A more realistic example — computing something for every column:

```
numeric_cols <- c("hours_studied", "score")
for (col in numeric_cols) {
  cat(col, ": mean =", round(mean(scores[[col]]), 2), "\n")
}
```

```
#> hours_studied : mean = 4.84
#> score : mean = 68.47
```

2.5.2 `while` loops

Use `while` when you don't know in advance how many iterations you'll need — you repeat *until* a condition becomes false.

```r
total <- 0
count <- 0
while (total < 100) {
  count <- count + 1
  total <- total + count
}
cat("It took", count, "steps to pass
100. Final total:", total, "\n")
```

```
#> It took 14 steps to pass
#> 100. Final total: 105
```

2.5.3 `repeat` loops

`repeat` runs forever until you explicitly `break` out.

```r
x <- 1
repeat {
  x <- x * 2
  if (x > 50) break
}
x
```

```
#> [1] 64
```

Comparison of Looping Structures

To help decide which looping structure to use in your statistical code, use this comparison matrix:

Loop Type	Evaluation Timing	Termination Criteria	Best Use Case	R Syntax Structure
`for`	At loop entry (iterates over a sequence)	When the end of the sequence is reached	When the exact number of iterations is known beforehand (e.g., column indices)	`for (i in seq) { ... }`
`while`	Before entering the loop body	When the condition becomes `FALSE`	When you do not know the exact count but have a condition to check first (e.g., convergence)	`while (cond) { ... }`
`repeat`	After executing the loop body (via `break`)	Explicitly checked inside the body using a conditional `break`	When the loop body must execute *at least once* before the condition is checked	`repeat { ...; if (cond) break }`

Efficiency note: R is optimized for vectorized operations, not loops. Compare these two ways of squaring every number from 1 to 1,000,000:

```r
system.time({
  result_loop <- numeric(1e6)
  for (i in 1:1e6) result_loop[i] <- i^2
})
```

```
#>    user  system elapsed
#>   0.055   0.003   0.056
```

```
system.time({
  result_vectorized <- (1:1e6)^2
})
```

```
#>    user  system elapsed
#>   0.004   0.000   0.004
```

The vectorized version is typically 10-50x faster. **Rule of thumb:** if you find yourself writing a loop to do simple arithmetic on every element of a vector, there's almost always a vectorized alternative.

Write a `for` loop that goes through `1:10` and prints `"even"` or `"odd"` for each number (combine what you just learned about `if`/`else` inside a loop).

2.6 User-Defined Functions

A function packages up a piece of logic so you can reuse it without retyping it.

```
summary_stats <- function(x) {
  c(mean = mean(x), median = median(x), sd = sd(x), IQR = IQR(x))
}

summary_stats(scores$score)
```

```
#>     mean   median       sd      IQR
#> 68.47500 69.00000 12.70774 20.25000
```

```
summary_stats(mtcars$mpg)
```

```
#>      mean    median        sd       IQR
#> 20.090625 19.200000  6.026948  7.375000
```

2.6.1 Combining functions with `sapply()` and `lapply()`

Once you have a function, you can apply it across many columns or many groups at once, without writing a loop by hand.

```
sapply(scores[, c("hours_studied", "score")], summary_stats)
```

```
#>        hours_studied    score
#> mean        4.840000 68.47500
#> median      4.650000 69.00000
#> sd          2.194842 12.70774
#> IQR         3.375000 20.25000
```

```
# Apply your function separately within each group
lapply(split(scores$score, scores$group), summary_stats)
```

```
#> $A
#>     mean   median       sd      IQR
#> 67.94737 66.00000 12.42969 13.50000
#>
#> $B
#>     mean   median       sd      IQR
#> 68.95238 71.00000 13.24189 22.00000
```

Second example — applying a custom function across an entire built-in dataset:

```
sapply(iris[, 1:4], summary_stats)
```

```
#>        Sepal.Length Sepal.Width Petal.Length Petal.Width
#> mean      5.8433333   3.0573333     3.758000   1.1993333
#> median    5.8000000   3.0000000     4.350000   1.3000000
#> sd        0.8280661   0.4358663     1.765298   0.7622377
#> IQR       1.3000000   0.5000000     3.500000   1.5000000
```

`sapply()` tries to simplify its result into a clean matrix or vector; `lapply()` always returns a list, even if every result looks similar. When you're not sure which to use: try `sapply()` first, and if the output looks messy or you need to keep each result completely separate, switch to `lapply()`.

2.6.2 A more advanced function: default arguments and multiple returns

```
describe_column <- function(x, digits = 2) {
  list(
    n = length(x),
    missing = sum(is.na(x)),
    mean = round(mean(x, na.rm = TRUE), digits),
    sd = round(sd(x, na.rm = TRUE), digits),
    range = round(range(x, na.rm = TRUE), digits)
  )
}

describe_column(airquality$Ozone)          # uses default digits = 2
```

```
#> $n
#> [1] 153
#>
#> $missing
#> [1] 37
#>
#> $mean
#> [1] 42.13
#>
#> $sd
#> [1] 32.99
#>
#> $range
#> [1]   1 168
```

```
describe_column(airquality$Ozone, digits = 4)  # overrides the default
```

```
#> $n
```

```
#> [1] 153
#>
#> $missing
#> [1] 37
#>
#> $mean
#> [1] 42.1293
#>
#> $sd
#> [1] 32.9879
#>
#> $range
#> [1]   1 168
```

Write a function called `pass_rate(x, threshold = 50)` that takes a numeric vector of scores and returns the *percentage* of values at or above `threshold` (default 50). Test it on `scores$score`, then test it again with `threshold = 75`.

2.6.3 Scoping Rules (Lexical Scoping)

An important aspect of writing user-defined functions is understanding R's **scoping rules**. Scoping determines how R finds a value associated with a variable name. R uses a system called **lexical scoping** (or static scoping), which means that the scoping rules are determined by where the function was *defined*, not where it is *called*.

The Box-in-a-Box (Container) Analogy Think of scoping like nested boxes:

- When a function runs, it creates a small, private box (the local environment). R looks inside this small box first for any variable.
- If it cannot find the variable locally, R steps out into the next larger box (the parent environment, which is typically the global workspace) to search.
- This lookup is one-way: code in the larger box (global environment) can never look inside the small box (local environment) to access private variables!

Let's see this in code:

```
x <- 10  # Defined in the global environment

scoping_demo <- function() {
  y <- 5  # Defined in the local environment
  # R finds y locally, and x globally by stepping out one level
  return(x + y)
}

scoping_demo()
```

```
#> [1] 15
```

If we try to access the local variable `y` in the global environment, R throws an error because the parent box cannot look inside the child box:

```
# This will result in an error: object 'y' not found
print(y)
```

2.6.4 Object-Oriented Programming (OOP) in R: The S3 System

While R is primarily a functional programming language, it also contains object-oriented programming (OOP) systems. The most common and simple OOP system in R is the **S3 Class System**. It is an informal system that allows you to attach a class attribute to a list, enabling generic functions (like `print()`, `plot()`, or `summary()`) to behave differently depending on the class of the object passed to them.

2.6.4.1

1. Creating S3 Objects An S3 object is usually a list with a `"class"` attribute. You can assign the class of an object using the `class()` function:

```
# 1. Create a basic list representing a medical patient
patient_a <- list(name = "John Doe", age = 45, heart_rate = 72)

# 2. Assign class "patient" to the list
class(patient_a) <- "patient"

# Verify the class
class(patient_a)
```

```
#> [1] "patient"
```

2.6.4.2

2. Writing S3 Methods for Existing Generics A **generic function** is a function that decides which method to call based on the class of its first argument. The naming convention for S3 methods is `generic_name.class_name()`. Let's write a custom method for the standard `print()` generic for our `"patient"` class:

```
# Define the print method for class "patient"
print.patient <- function(x, ...) {
  cat("=== Patient Record ===\n")
  cat("Name:       ", x$name, "\n")
  cat("Age:        ", x$age, " years\n")
  cat("Heart Rate:", x$heart_rate, " bpm\n")
}

# Now call print() on our patient object
print(patient_a)
```

```
#> === Patient Record ===
#> Name:       John Doe
#> Age:        45  years
#> Heart Rate: 72  bpm
```

2.6.4.3

3. Creating Custom Generics and Methods You can also define your own generic functions using `UseMethod()`. For example, let's create a generic named `is_unhealthy` and implement a method for the `"patient"` class:

```r
# Define a custom generic function
is_unhealthy <- function(x, ...) {
  UseMethod("is_unhealthy")
}

# Define the method for class "patient"
is_unhealthy.patient <- function(x, ...) {
  # Let's say heart rate > 100 or < 60 is unhealthy
  return(x$heart_rate > 100 || x$heart_rate < 60)
}

# Test our custom method
is_unhealthy(patient_a)
```

```
#> [1] FALSE
```

2.7 Modern R Workflows: dplyr and ggplot2 Basics

Up to now, we have written code in **Base R** (the core R language). However, in modern data science, most R users use packages from the **Tidyverse**—a collection of R packages designed for data science. The two most important packages in the Tidyverse are:

1. `dplyr`: For fast, intuitive data manipulation.
2. `ggplot2`: For building elegant and complex graphics using the "Grammar of Graphics."

2.7.1 The Pipe Operator (%>% or |>)

Before learning the functions, we must understand the **pipe operator** (`%>%` from the `magrittr` package, or R's built-in native pipe `|>`). Think of the pipe as reading *"and then"*. It takes the output of one function and passes it as the first parameter to the next function.

> **The Baking Analogy**: Imagine you are baking a cake.
>
> - **Nested (Base R) Style**: To write this in nested functions, it looks like this: `frost(bake(add_sugar(flour)))` You have to read this from the *inside-out* (start with flour, then sugar, then bake, then frost).
> - **Piped (Tidyverse) Style**: Chaining it with pipes looks like this: `flour %>% add_sugar() %>% bake() %>% frost()` This reads naturally from *left-to-right* (take flour, **and then** add sugar, **and then** bake, **and then** frost).

Let's compare in code:

```r
library(dplyr)

# Nested Base R way (hard to read):
head(select(scores, name, score), 3)
```

```
#>         name score
#> 1 Student_01    63
#> 2 Student_02    66
#> 3 Student_03    49
```

```r
# Piped dplyr way (reads naturally left-to-right):
scores %>%
  select(name, score) %>%
  head(3)
```

```
#>         name score
#> 1 Student_01    63
#> 2 Student_02    66
#> 3 Student_03    49
```

2.7.2 Step-by-Step Data Manipulation with `dplyr`

`dplyr` uses clean, verbs-based functions to manipulate data frames. Let's learn them one-by-one, progressing from minor examples to advanced applications.

2.7.2.1

1. Column Selection with `select()` `select()` allows you to subset columns from a data frame.

- **Minor (Select One Column)**:

```
scores %>%
  select(score) %>%
  head(2)
```

```
#>   score
#> 1    63
#> 2    66
```

- **Medium (Select Multiple Columns)**:

```
scores %>%
  select(name, group, score) %>%
  head(2)
```

```
#>         name group score
#> 1 Student_01     A    63
#> 2 Student_02     A    66
```

- **Advanced (Deselect/Exclude Columns)**: Use the minus sign `-` to keep everything *except* specific columns.

```
scores %>%
  select(-id, -hours_studied) %>%
  head(2)
```

```
#>         name group score result grade
#> 1 Student_01     A    63   Pass     C
#> 2 Student_02     A    66   Pass     C
```

2.7.2.2

2. Row Filtering with `filter()` `filter()` allows you to subset rows based on logical conditions.

- **Minor (Single Numeric Condition)**:

```
  scores %>%
    filter(score >= 85) %>%
    head(2)
```

```
#>   id       name group score hours_studied result grade
#> 1  4 Student_04     B    89           4.4   Pass     B
#> 2  6 Student_06     A    95           4.5   Pass     A
```

- **Medium (Character/Categorical Match)**:

```
  scores %>%
    filter(group == "B") %>%
    head(2)
```

```
#>   id       name group score hours_studied result grade
#> 1  4 Student_04     B    89           4.4   Pass     B
#> 2  8 Student_08     B    69           7.3   Pass     C
```

- **Advanced (Multiple Combined Conditions)**: Combine conditions using `&` (AND) or `|` (OR).

```
  # Filter students in Group A AND studied more than 6 hours
  scores %>%
    filter(group == "A" & hours_studied > 6)
```

```
#>   id       name group score hours_studied result grade
#> 1 17 Student_17     A    76           9.2   Pass     B
#> 2 18 Student_18     A    56           6.4   Pass     C
#> 3 29 Student_29     A    79           7.4   Pass     B
#> 4 32 Student_32     A    65           7.9   Pass     C
#> 5 34 Student_34     A    74           7.0   Pass     C
```

2.7.2.3

3. Creating/Modifying Columns with `mutate()` `mutate()` allows you to compute and add new columns, or overwrite existing ones.

- **Minor (Add Constant Column)**:

```
  scores %>%
    mutate(course_year = 2026) %>%
    head(2)
```

```
#>   id       name group score hours_studied result grade course_year
#> 1  1 Student_01     A    63           4.7   Pass     C        2026
#> 2  2 Student_02     A    66           5.3   Pass     C        2026
```

- **Medium (Arithmetic Transformation)**: Calculate score percentage out of a maximum of 100:

```
  scores %>%
    mutate(percentage = score / 100 * 100) %>%
    head(2)
```

```
#>   id       name group score hours_studied result grade percentage
#> 1  1 Student_01     A    63           4.7   Pass     C         63
#> 2  2 Student_02     A    66           5.3   Pass     C         66
```

- **Advanced (Conditional Columns using `ifelse`)**: Create a pass/fail indicator column based on scores:

```
scores %>%
  mutate(status = ifelse(score >= 50, "Pass", "Fail")) %>%
  head(3)
```

```
#>   id       name group score hours_studied result grade status
#> 1  1 Student_01     A    63           4.7   Pass     C   Pass
#> 2  2 Student_02     A    66           5.3   Pass     C   Pass
#> 3  3 Student_03     A    49           4.6   Fail     F   Fail
```

2.7.2.4

4. Grouping & Summarizing with `group_by()` and `summarise()` `group_by()` groups your dataset by a categorical variable, and `summarise()` calculates summary statistics for each group. They are almost always used together.

- **Minor (Overall Summary - No Grouping)**:

```
scores %>%
  summarise(overall_avg = mean(score))
```

```
#>   overall_avg
#> 1      68.475
```

- **Medium (Grouped Summary)**: Calculate the average score separately for each group:

```
scores %>%
  group_by(group) %>%
  summarise(mean_score = mean(score))
```

```
#> # A tibble: 2 x 2
#>   group mean_score
#>   <chr>      <dbl>
#> 1 A           67.9
#> 2 B           69.0
```

- **Advanced (Multiple Grouped Metrics)**: Calculate the mean, standard deviation, and count `n()` for multiple groups:

```
scores %>%
  group_by(group) %>%
  summarise(
    avg_score = mean(score),
    sd_score = sd(score),
    sample_size = n()
  )
```

```
#> # A tibble: 2 x 4
#>   group avg_score sd_score sample_size
#>   <chr>     <dbl>    <dbl>       <int>
#> 1 A          67.9     12.4          19
#> 2 B          69.0     13.2          21
```

2.7.3 Chaining It All Together (A Complete Pipeline)

Now, let's combine all these functions into a single pipeline. We want to take our raw `scores` data, filter out students who didn't study, create a pass/fail variable, group by their study group, and calculate the pass rate and average score.

- **Base R Nested Way (Very hard to read and debug)**:

```
sub_df <- scores[scores$hours_studied > 2, ]
sub_df$status <- ifelse(sub_df$score >= 50, 1, 0)
aggregate(cbind(score, status) ~ group, data = sub_df, FUN = mean)
```

```
#>   group    score    status
#> 1     A 66.47059 0.9411765
#> 2     B 70.27778 1.0000000
```

- **Modern `dplyr` Chained Pipeline**:

```
scores %>%
  filter(hours_studied > 2) %>%
  mutate(passed = ifelse(score >= 50, 1, 0)) %>%
  group_by(group) %>%
  summarise(
    average_score = mean(score),
    pass_rate = mean(passed) * 100,
    total_students = n()
  )
```

```
#> # A tibble: 2 x 4
#>   group average_score pass_rate total_students
#>   <chr>         <dbl>     <dbl>          <int>
#> 1 A              66.5      94.1             17
#> 2 B              70.3     100               18
```

Notice how the `dplyr` pipeline reads like a step-by-step description of your analysis plan, making it simple to write, read, and maintain.

2.7.4 Grammar of Graphics with `ggplot2`

`ggplot2` is built on the **Grammar of Graphics** philosophy: a plot is constructed by layering components (Data → Aesthetics → Geometries → Labels → Themes) on top of each other using the `+` operator.

Let's build a histogram step-by-step, showing how a plot grows from a blank canvas to a publication-ready figure.

2.7.4.1 Step 1: The Blank Canvas

Initialize the plot with the dataset and the aesthetic mappings `aes()` (telling R that the x-axis maps to `Ozone` levels).

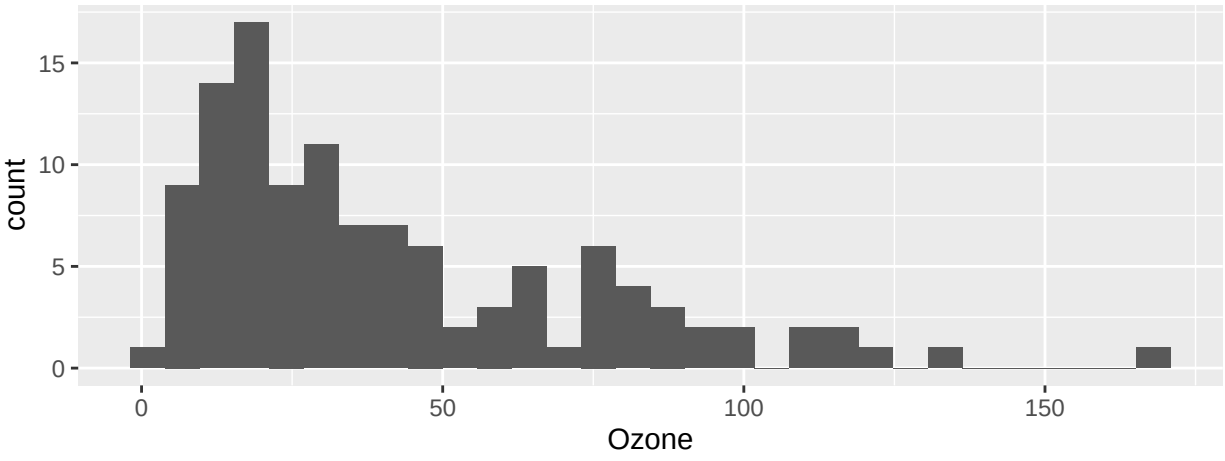
```
library(ggplot2)
ggplot(airquality, aes(x = Ozone))
```

Note: This creates a blank grey grid because we haven't told R what shape (geometry) to draw on the canvas yet.

2.7.4.2 Step 2: Adding a Geometry (`geom_*`)

We add a geometry layer using the `+` operator. For a single numeric variable, we use `geom_histogram()`.

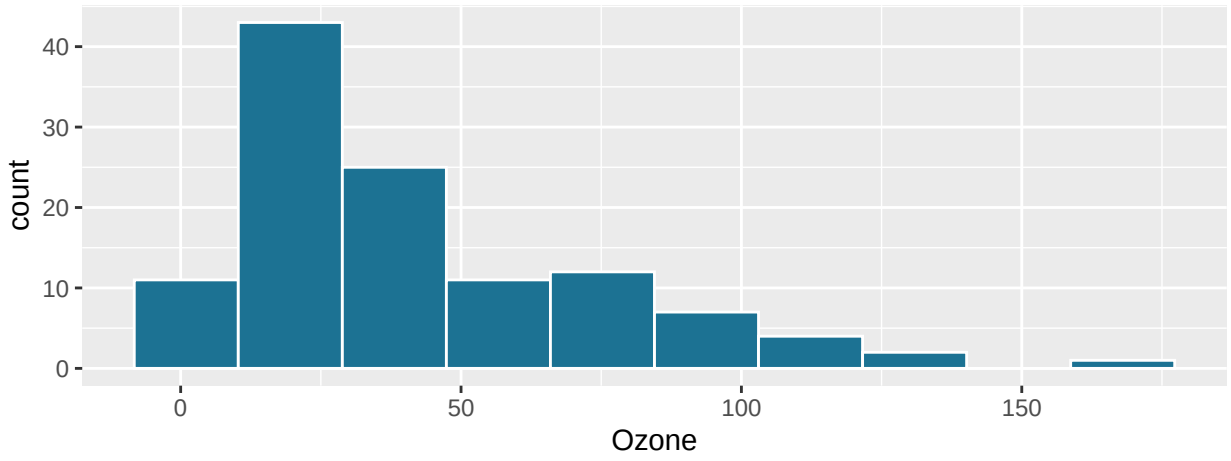
```
ggplot(airquality, aes(x = Ozone)) +
  geom_histogram(na.rm = TRUE)
```



2.7.4.3 Step 3: Customizing the Geometry

We can customize the shape outlines, fill colors, and the number of bins inside `geom_histogram()`.

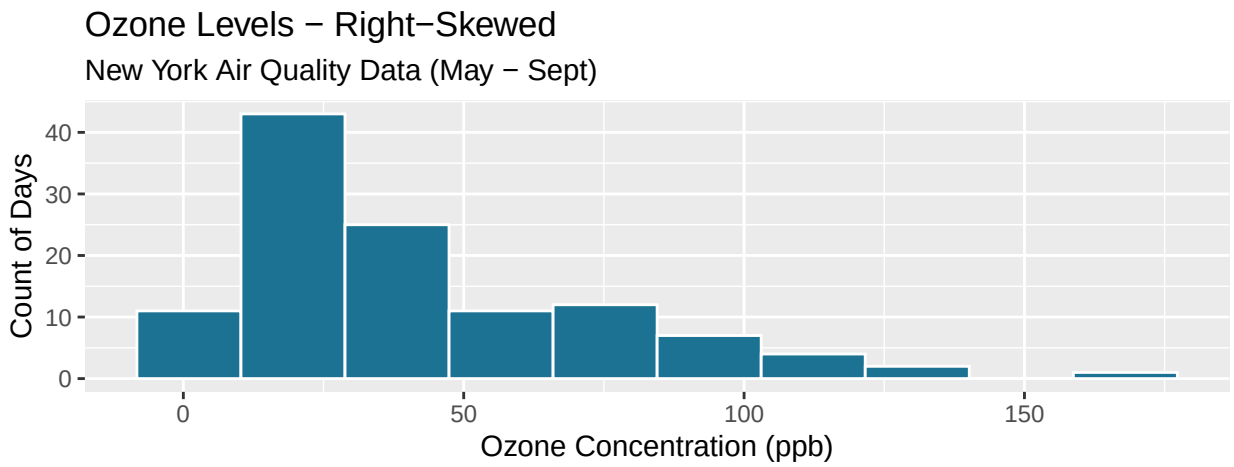
```
ggplot(airquality, aes(x = Ozone)) +
  geom_histogram(fill = "#1C7293", color = "white", bins = 10, na.rm = TRUE)
```



2.7.4.4 Step 4: Adding Titles & Axis Labels (`labs()`)

Add clear, professional titles and coordinate labels:

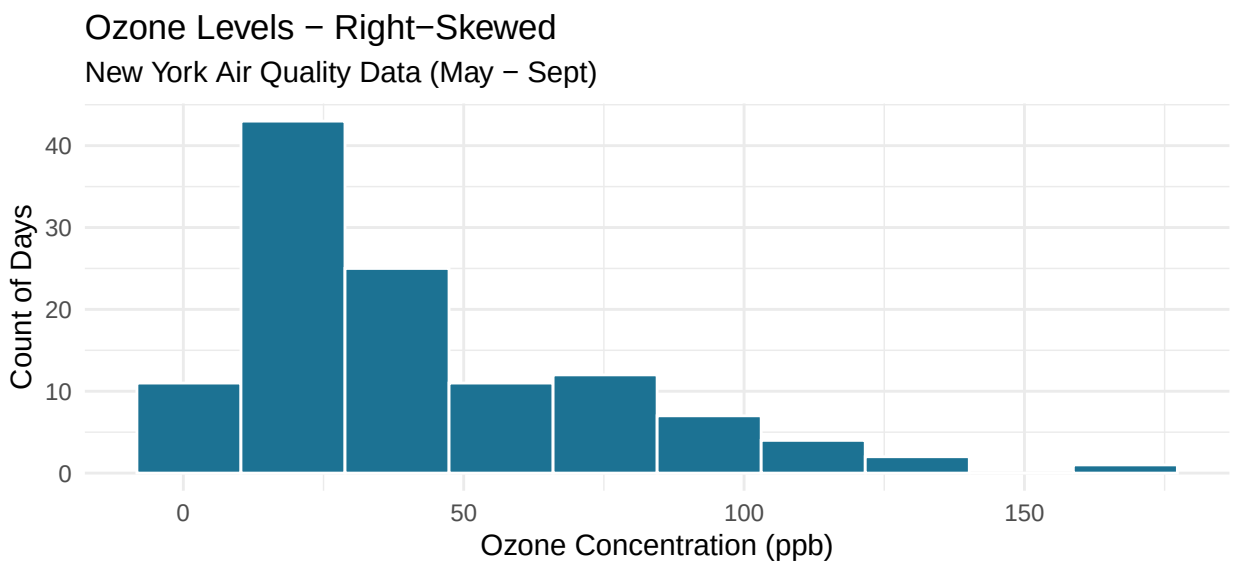
```
ggplot(airquality, aes(x = Ozone)) +
  geom_histogram(fill = "#1C7293", color = "white", bins = 10, na.rm = TRUE) +
  labs(
    title = "Ozone Levels - Right-Skewed",
    subtitle = "New York Air Quality Data (May - Sept)",
    x = "Ozone Concentration (ppb)",
    y = "Count of Days"
  )
```



2.7.4.5 Step 5: Applying a Theme (`theme_*()`)

Themes clean up the background grid and borders. `theme_minimal()` is a clean, modern option:

```
ggplot(airquality, aes(x = Ozone)) +
  geom_histogram(fill = "#1C7293", color = "white", bins = 10, na.rm = TRUE) +
  labs(
    title = "Ozone Levels - Right-Skewed",
    subtitle = "New York Air Quality Data (May - Sept)",
    x = "Ozone Concentration (ppb)",
    y = "Count of Days"
  ) +
  theme_minimal()
```

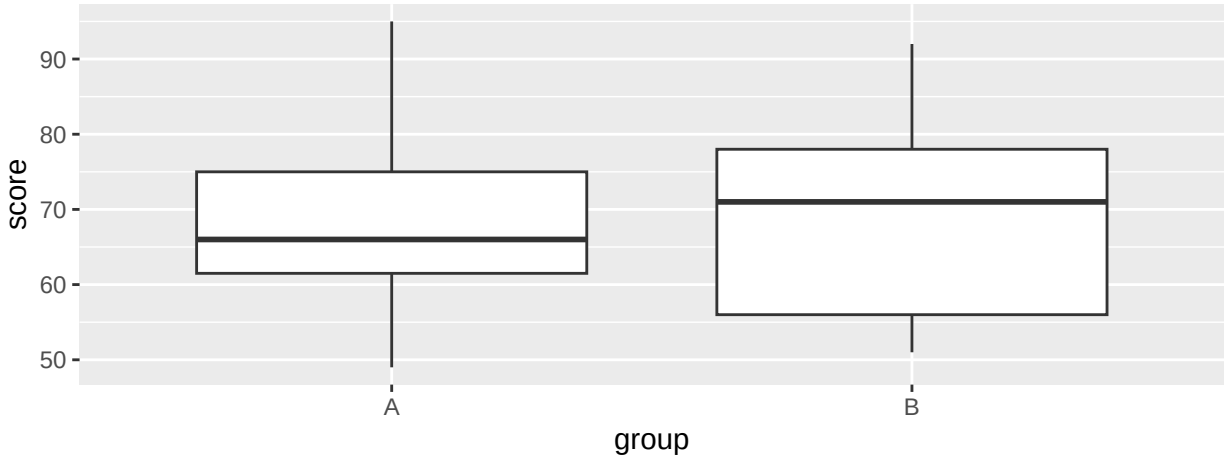


2.7.5 Visualizing Groups (Advanced ggplot2)

ggplot2 shines when comparing different groups of data. Let's build a boxplot step-by-step to compare test scores between groups.

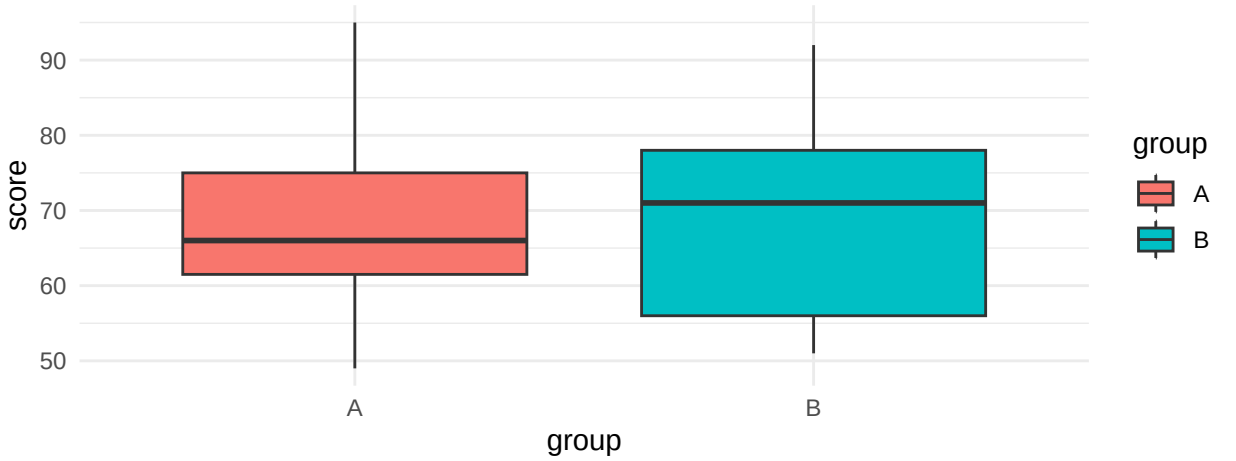
- **Step A: Basic Boxplot**: We map the x-axis to `group` (categorical) and the y-axis to `score` (numeric) using `geom_boxplot()`:

```
ggplot(scores, aes(x = group, y = score)) +
  geom_boxplot()
```



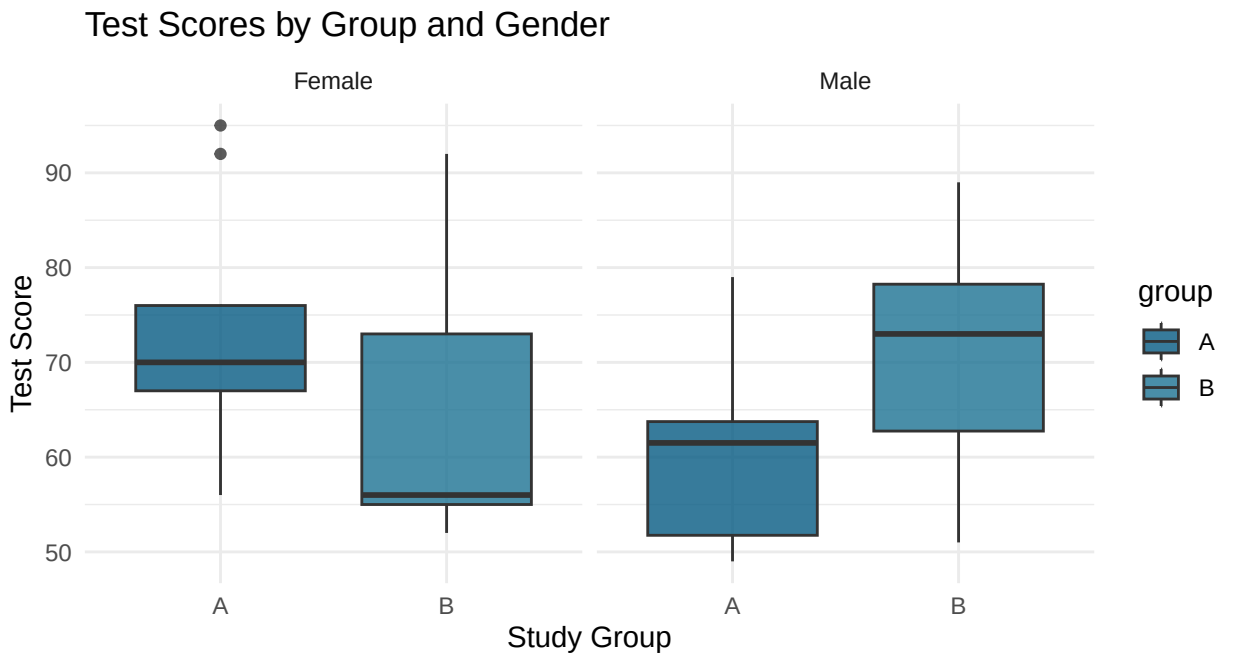
- **Step B: Add Group Coloring (`fill`)**: To color the boxes by group, we add `fill = group` inside the aesthetic mappings `aes()`:

```
ggplot(scores, aes(x = group, y = score, fill = group)) +
  geom_boxplot() +
  theme_minimal()
```



- **Step C: Custom Colors & Facet Wrap (Subplots)**: Let's manually define the colors using `scale_fill_manual()` and draw separate subplots for each gender using `facet_wrap()`:

```
# Simulate a sex variable for illustration:
set.seed(42)
scores_sex <- scores %>%
  mutate(sex = sample(c("Male", "Female"), n(), replace = TRUE))

ggplot(scores_sex, aes(x = group, y = score, fill = group)) +
  geom_boxplot(alpha = 0.8) +
  scale_fill_manual(values = c("A" = "#065A82", "B" = "#1C7293")) +
  facet_wrap(~sex) +
  theme_minimal() +
  labs(
    title = "Test Scores by Group and Gender",
    x = "Study Group",
    y = "Test Score"
  )
```



2.7.6 Visual Plot Builder: The `esquisse` Package

If you find `ggplot2` syntax intimidating at first, R offers an interactive helper package called **`esquisse`**. It provides a drag-and-drop Shiny GUI directly inside RStudio to build plots visually and automatically outputs the exact `ggplot2` code for you to copy.

To try it out, run this code in your interactive RStudio Console (not inside an Rmd code chunk):

```
# 1. Install the package
install.packages("esquisse")

# 2. Launch the visual builder with a dataset
esquisse::esquisser(airquality)
```

In the window that pops up:

1. Drag the `Ozone` variable box into the **X** axis slot.
2. The builder will automatically render a **Histogram**.
3. Go to the **Labels & Title** tab at the bottom to type your custom labels.
4. Click on the **Code** button in the bottom right corner, copy the generated code, and paste it into your script. It is a fantastic way to learn ggplot2!

Exercise: dplyr and ggplot2 Using R's built-in **`mtcars`** dataset:

1. Write a `dplyr` pipeline that:
   - Filters for cars with horsepower (`hp`) greater than 100.
   - Selects the columns `mpg` (miles per gallon), `cyl` (cylinders), and `hp`.
   - Groups the data by `cyl`.
   - Summarises the average `mpg` and average `hp` for each cylinder group.
2. Use `ggplot2` to draw a boxplot of `mpg` (y-axis) by cylinder group (x-axis). *(Hint: convert `cyl` to a factor inside the aesthetics mapping: `aes(x = as.factor(cyl), y = mpg)` so that R treats cylinders as a categorical group rather than a continuous number).*

- The mean is sensitive to outliers; the median is not — always compute both.
- Skewness and kurtosis give you a numeric handle on distribution shape, foreshadowing why Chapters 3 and 5 exist.
- Use `if/else` for single values, `ifelse()` for whole vectors/columns — mixing these up is one of the most common beginner bugs.
- `for`, `while`, and `repeat` each suit a different situation, but vectorized operations usually beat all three for simple arithmetic.
- Writing your own functions, then applying them with `sapply()/lapply()`, is how you scale a one-off calculation into a repeatable, reliable workflow.
- **`dplyr`** and **`ggplot2`** form the backbone of modern R workflows, allowing you to manipulate datasets using clean verbs and build professional, publication-grade plots layer-by-layer.

Next, in Chapter 3, we'll use these programming skills to build a bootstrap resampling procedure from scratch — the foundation of every modern non-parametric confidence interval.

Chapter 2 covered:

- **Central tendency**: Mean (sensitive to outliers), median (robust), mode (for categorical), trimmed mean (balanced)
- **Dispersion**: Variance $s^2 = \frac{1}{n-1}\sum(x_i - \bar{x})^2$, standard deviation s, IQR $= Q_3 - Q_1$, coefficient of variation $CV = s/\bar{x}$

- **Visualisation**: Histograms (distribution shape), boxplots (outliers and quartiles), density plots (smooth shape estimate)
- **Programming**: Functions with `function()`, `for`/`while` loops, `apply()` / `sapply()` / `lapply()` for column-wise operations
- **Summary tables**: `psych::describe()` for a comprehensive numeric summary; `gtsummary::tbl_summary()` for publication-ready tables

Coming up: Chapter 3 moves from *describing* samples to *inferring* population parameters — specifically, constructing confidence intervals using both parametric theory and the bootstrap.

2.8 Lab 2: Full Descriptive Profile of the Course Dataset

Objective: Conduct a complete descriptive analysis of the `course_dataset`, mirroring what you would do at the start of a real research project.

Dataset: `course_dataset.csv` (provided in the `data/` folder).

2.8.1 Step 1: Load and Inspect

```
course_df <- read.csv("data/course_dataset.csv")
str(course_df)
```

```
#> 'data.frame':    180 obs. of  14 variables:
#>  $ id             : chr  "P001" "P002" "P003" "P004" ...
#>  $ site           : chr  "Site C" "Site B" "Site A" "Site C" ...
#>  $ treatment      : chr  "New" "Standard" "New" "Standard" ...
#>  $ age            : int  59 68 33 53 58 49 42 40 67 49 ...
#>  $ sex            : chr  "Male" "Male" "Male" "Male" ...
#>  $ activity_level : chr  "Moderate" "High" "Moderate" "Low" ...
#>  $ baseline_pain  : num  8.7 6.7 7.6 5.5 6 4.1 6.8 6 5.8 6.6 ...
#>  $ week4_pain     : num  5.8 6.5 3.9 2.7 5.9 4.6 6.6 2.5 4.4 2.9 ...
#>  $ week8_pain     : num  5.1 5.6 0.1 2.4 5.1 4.6 4.2 2.4 3.5 1.3 ...
#>  $ response       : chr  "Responder" "Non-Responder" "Responder" "Responder" ...
#>  $ symptom_pre    : chr  "Present" "Absent" "Present" "Present" ...
#>  $ symptom_post   : chr  "Absent" "Present" "Present" "Absent" ...
#>  $ rater1_severity: chr  "Severe" "Mild" "Moderate" "Severe" ...
#>  $ rater2_severity: chr  "Severe" "Mild" "Moderate" "Severe" ...
```

```
dim(course_df)
```

```
#> [1] 180  14
```

```
head(course_df)
```

```
#>     id   site treatment age    sex activity_level baseline_pain week4_pain
#> 1 P001 Site C       New  59   Male       Moderate           8.7        5.8
#> 2 P002 Site B  Standard  68   Male           High           6.7        6.5
#> 3 P003 Site A       New  33   Male       Moderate           7.6        3.9
#> 4 P004 Site C  Standard  53   Male            Low           5.5        2.7
#> 5 P005 Site B  Standard  58   Male            Low           6.0        5.9
#> 6 P006 Site B  Standard  49 Female            Low           4.1        4.6
#>   week8_pain      response symptom_pre symptom_post rater1_severity
#> 1        5.1     Responder     Present       Absent          Severe
#> 2        5.6 Non-Responder      Absent      Present            Mild
```

```
#> 3        0.1     Responder     Present      Present        Moderate
#> 4        2.4     Responder     Present       Absent          Severe
#> 5        5.1 Non-Responder      Absent      Present            Mild
#> 6        4.6 Non-Responder      Absent      Present          Severe
#>   rater2_severity
#> 1          Severe
#> 2            Mild
#> 3        Moderate
#> 4          Severe
#> 5            Mild
#> 6          Severe
```

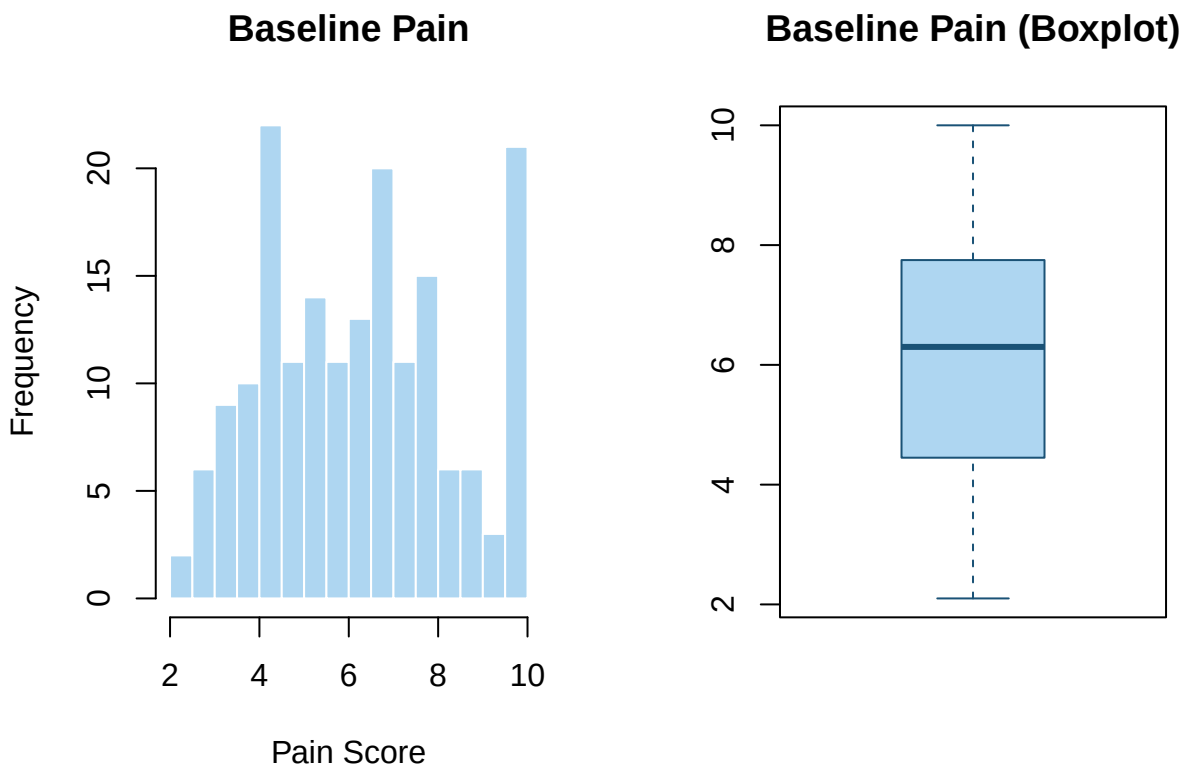
2.8.2 Step 2: Univariate Summary — Numerical Variables

```
# Quick numeric summary
library(psych)
psych::describe(course_df[, sapply(course_df, is.numeric)])
```

```
#>               vars   n  mean    sd median trimmed   mad  min max range  skew
#> age              1 180 52.45 12.62  54.00   52.74 13.34 24.0  85  61.0 -0.17
#> baseline_pain    2 180  6.27  2.10   6.30    6.21  2.45  2.1  10   7.9  0.21
#> week4_pain       3 180  4.34  2.44   4.35    4.27  2.45  0.0  10  10.0  0.26
#> week8_pain       4 180  3.40  2.46   3.10    3.22  2.74  0.0  10  10.0  0.53
#>               kurtosis   se
#> age              -0.46 0.94
#> baseline_pain    -0.89 0.16
#> week4_pain       -0.47 0.18
#> week8_pain       -0.35 0.18
```

2.8.3 Step 3: Visualise Key Variables

```
par(mfrow = c(1, 2))
hist(course_df$baseline_pain, main = "Baseline Pain", xlab = "Pain Score",
     col = "#AED6F1", border = "white", breaks = 15)
boxplot(course_df$baseline_pain, main = "Baseline Pain (Boxplot)",
        col = "#AED6F1", border = "#1A5276")
```



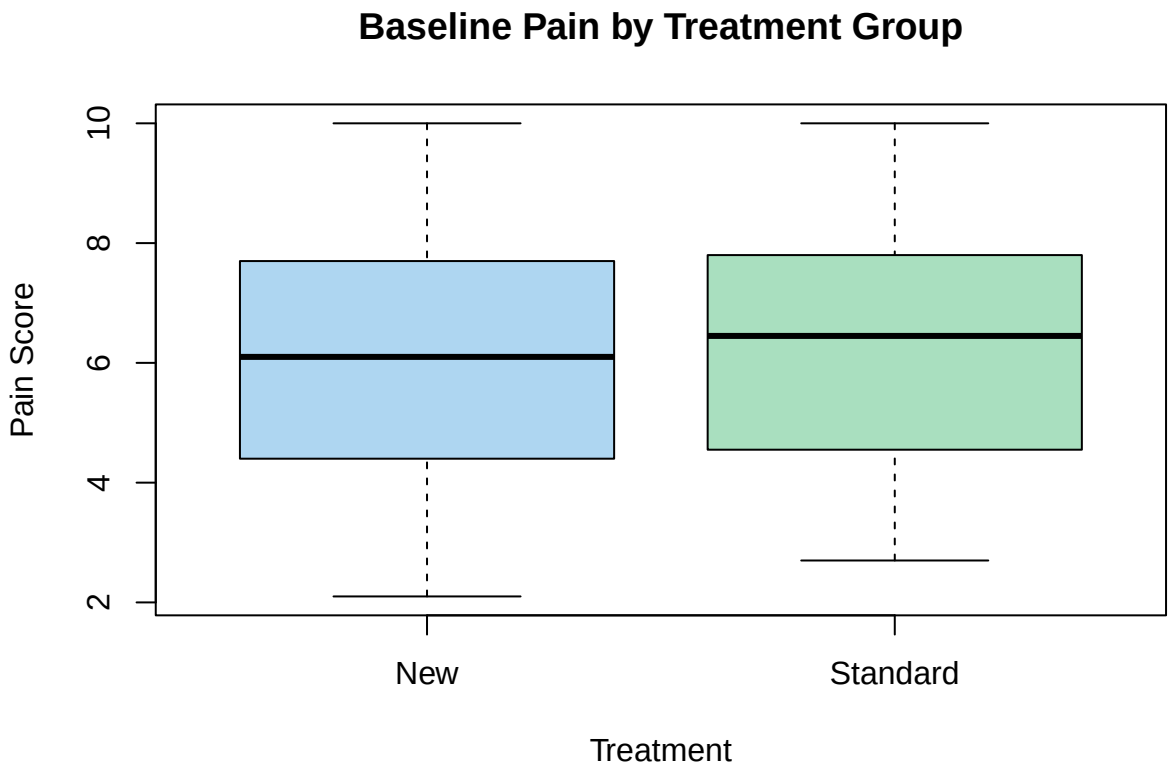

```
par(mfrow = c(1, 1))
```

2.8.4 Step 4: Group Comparison

```
# Mean baseline pain by treatment group
tapply(course_df$baseline_pain, course_df$treatment, mean, na.rm = TRUE)
```

```
#>      New Standard
#> 6.189583 6.371429
```

```
# Side-by-side boxplots
boxplot(baseline_pain ~ treatment, data = course_df,
        col = c("#AED6F1", "#A9DFBF"),
        main = "Baseline Pain by Treatment Group",
        xlab = "Treatment", ylab = "Pain Score")
```



2.8.5 Step 5: Publication Summary Table

```
library(gtsummary)
tbl_summary(course_df,
            by = treatment,
            statistic = list(all_continuous() ~ "{mean} ({sd})"),
            missing = "no") |>
  add_p() |>
  bold_labels()
```

2.8.6 Lab Exercises

1. Compute the CV (coefficient of variation) for `baseline_pain` and `week8_pain`. Which variable has greater relative variability?
2. Write a custom R function `my_summary(x)` that returns a named vector with min, Q1, median, mean, Q3, max, and number of NAs.
3. Apply `my_summary()` to every numeric column of `course_df` using `sapply()`.

Characteristic	**New** N = 96[1]	**Standard** N = 84[1]	**p-value**
id			
P001	1 (1.0%)	0 (0%)	
P002	0 (0%)	1 (1.2%)	
P003	1 (1.0%)	0 (0%)	
P004	0 (0%)	1 (1.2%)	
P005	0 (0%)	1 (1.2%)	
P006	0 (0%)	1 (1.2%)	
P007	1 (1.0%)	0 (0%)	
P008	1 (1.0%)	0 (0%)	
P009	1 (1.0%)	0 (0%)	
P010	1 (1.0%)	0 (0%)	
P011	1 (1.0%)	0 (0%)	
P012	1 (1.0%)	0 (0%)	
P013	0 (0%)	1 (1.2%)	
P014	0 (0%)	1 (1.2%)	
P015	0 (0%)	1 (1.2%)	
P016	1 (1.0%)	0 (0%)	
P017	1 (1.0%)	0 (0%)	
P018	0 (0%)	1 (1.2%)	
P019	0 (0%)	1 (1.2%)	
P020	0 (0%)	1 (1.2%)	
P021	0 (0%)	1 (1.2%)	
P022	0 (0%)	1 (1.2%)	
P023	0 (0%)	1 (1.2%)	
P024	1 (1.0%)	0 (0%)	
P025	1 (1.0%)	0 (0%)	
P026	1 (1.0%)	0 (0%)	
P027	0 (0%)	1 (1.2%)	
P028	0 (0%)	1 (1.2%)	
P029	1 (1.0%)	0 (0%)	
P030	1 (1.0%)	0 (0%)	
P031	0 (0%)	1 (1.2%)	
P032	0 (0%)	1 (1.2%)	
P033	1 (1.0%)	0 (0%)	
P034	1 (1.0%)	0 (0%)	
P035	1 (1.0%)	0 (0%)	
P036	0 (0%)	1 (1.2%)	
P037	0 (0%)	1 (1.2%)	
P038	1 (1.0%)	0 (0%)	
P039	0 (0%)	1 (1.2%)	
P040	0 (0%)	1 (1.2%)	
P041	0 (0%)	1 (1.2%)	
P042	0 (0%)	1 (1.2%)	
P043	1 (1.0%)	0 (0%)	
P044	1 (1.0%)	0 (0%)	
P045	1 (1.0%)	0 (0%)	
P046	1 (1.0%)	0 (0%)	
P047	1 (1.0%)	0 (0%)	
P048	0 (0%)	1 (1.2%)	
P049	0 (0%)	1 (1.2%)	
P050	1 (1.0%)	0 (0%)	
P051	1 (1.0%)	0 (0%)	
P052	0 (0%)	1 (1.2%)	
P053	1 (1.0%)	0 (0%)	

Chapter 3

Confidence Interval Estimation and Resampling Methods

This chapter answers a question you've probably wondered about: what do you do when the textbook formula for a confidence interval doesn't quite fit your data? The answer is **resampling** — and the bootstrap is its most important technique.

3.1 Parametric Confidence Intervals (Review)

This section uses `mtcars` (built into R) for a clean, roughly symmetric variable, so the parametric approach looks trustworthy — setting up a contrast with the skewed data later in this chapter.

```
t_result <- t.test(mtcars$mpg)
t_result$conf.int
```

```
#> [1] 17.91768 22.26357
#> attr(,"conf.level")
#> [1] 0.95
```

```
t_result$estimate
```

```
#> mean of x
#>  20.09062
```

This interval relies on the sampling distribution of the mean being approximately normal — which the Central Limit Theorem guarantees *reasonably well* here, because `mpg` isn't too skewed and the formula for the mean's standard error is well established.

Check Before You Run This (One-Sample t-test Diagnostics) Before running a parametric one-sample t-test, verify these three assumptions:

1. **Continuous Variable**: The dependent variable must be measured on a continuous scale (interval or ratio).
2. **Independence**: The observations must be independent of one another (no paired or repeated measures).
3. **Normality**: The data should be approximately normally distributed. For small samples ($n < 30$), verify this with a Shapiro-Wilk test (`shapiro.test(x)`). For large samples ($n \geq 30$), the Central Limit Theorem makes the test robust to minor normality violations.

The Math Behind It: Manual t-test Calculation A one-sample t-test compares the sample mean ($\bar{x}$) to a hypothesized population mean (μ_0, which defaults to 0). The test statistic is calculated as:

$$t = \frac{\bar{x} - \mu_0}{s/\sqrt{n}}$$

where s is the sample standard deviation and n is the sample size. The denominator $s/\sqrt{n}$ is the standard error of the mean (SE).

Let's calculate this manually in R to verify it matches `t.test(mtcars$mpg)` exactly:

```
x <- mtcars$mpg
n <- length(x)
xbar <- mean(x)
s <- sd(x)
se <- s / sqrt(n)

# 1. Calculate the t-statistic (testing H0: mu = 0)
t_stat <- (xbar - 0) / se
t_stat
```

```
#> [1] 18.85693
```

```
# 2. Look up the two-tailed p-value using the pt() cumulative density function
p_val <- 2 * pt(-abs(t_stat), df = n - 1)
p_val
```

```
#> [1] 1.526151e-18
```

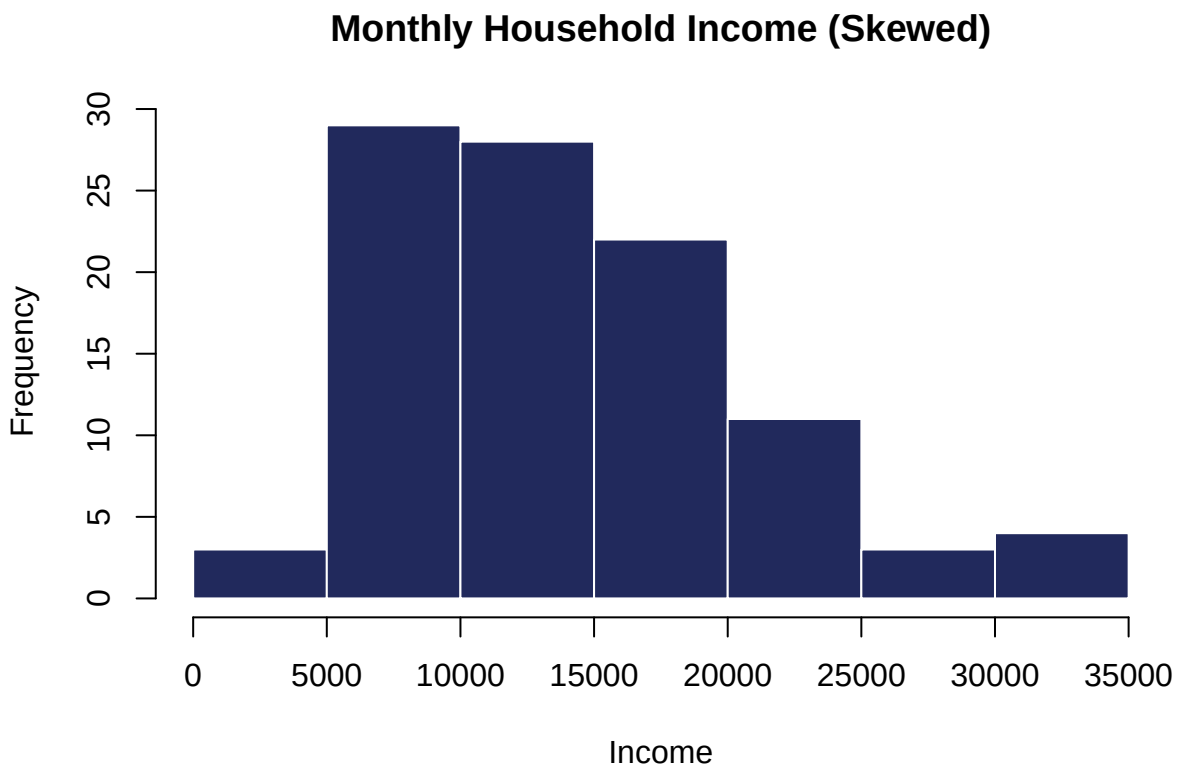
These match the test statistic ($t = 18.857$) and p-value ($p = 1.53 \times 10^{-18}$) from `t.test()` exactly!

How to Report This in a Paper: > "A standard one-sample t-test indicated that the mean fuel efficiency of the sampled cars ($\bar{x} = 20.09$ mpg, $SD = 6.03$) was statistically significantly higher than zero, $t(31) = 18.86$, $p < .001$."

A 95% confidence interval does **not** mean "there's a 95% chance the true mean is in this range." It means: if you repeated this entire sampling-and-estimation process many times, about 95% of the intervals constructed this way would contain the true population mean. The randomness is in the *procedure*, not in the fixed (but unknown) true value.

3.1.1 Where the parametric approach breaks down

```
income <- read.csv("data/survey_income.csv")
hist(income$monthly_income, main = "Monthly Household Income (Skewed)",
     xlab = "Income", col = "#21295C", border = "white")
```




```
mean(income$monthly_income)
```

```
#> [1] 14418.37
```

```
median(income$monthly_income)
```

```
#> [1] 13381.5
```

Notice the gap between mean and median — a strong sign of skew. A t-based CI for the mean assumes approximate normality of the *sampling distribution* of the mean; with skewed data and this sample size, that assumption is shakier, and there's no simple textbook formula at all for some statistics we might want (like the *median*).

Run `t.test(income$monthly_income)$conf.int`. Now imagine you wanted a CI for the **median** income instead of the mean. Try `median.test()` — it doesn't exist in base R! This is exactly the gap resampling fills.

3.2 The Logic of Resampling

The core idea: if we can't derive a sampling distribution mathematically, we can **approximate it empirically** by treating our one sample as a stand-in for the population, and repeatedly drawing new samples *from it, with replacement.*

3.2.1 Step 1: Understanding "Sampling with Replacement"

Before resampling our big dataset, let's look at how R's `sample()` function works on a tiny vector of 5 numbers: `x <- c(10, 20, 30, 40, 50)`.

- **Sampling WITHOUT Replacement (Shuffling)**: Every number can only be chosen once.

```r
  x <- c(10, 20, 30, 40, 50)
  sample(x, size = 5, replace = FALSE)
```

```
#> [1] 50 10 40 20 30
```

- **Sampling WITH Replacement (Resampling)**: Each time R picks a number, it "puts it back" into the bag before picking the next one. This means some numbers might appear multiple times, and others might not appear at all!

```r
  sample(x, size = 5, replace = TRUE)
```

```
#> [1] 30 40 40 50 40
```

```r
  sample(x, size = 5, replace = TRUE)
```

```
#> [1] 20 20 30 50 10
```

Notice that in the output, some numbers repeat (e.g., 30 twice) and some are missing. This matches how bootstrapping operates!

3.2.2 Step 2: Drawing ONE Resample of Our Dataset

Now let's apply this to our `income$monthly_income` dataset (which contains 100 observations). Let's draw *one* bootstrap sample (of the same size, 100, with replacement) and compute its mean:

```r
n <- length(income$monthly_income)
# Draw a resample of size n
one_resample <- sample(income$monthly_income, size = n, replace = TRUE)

# Compare original mean vs resampled mean
mean(income$monthly_income)
```

```
#> [1] 14418.37
```

```r
mean(one_resample)
```

```
#> [1] 15040.49
```

3.2.3 Step 3: Drawing 5 Resamples to Watch the Mean Fluctuate

If we repeat this process 5 times, we get 5 slightly different means:

```r
sapply(1:5, function(i) {
  resample <- sample(income$monthly_income, size = n, replace = TRUE)
  mean(resample)
})
```

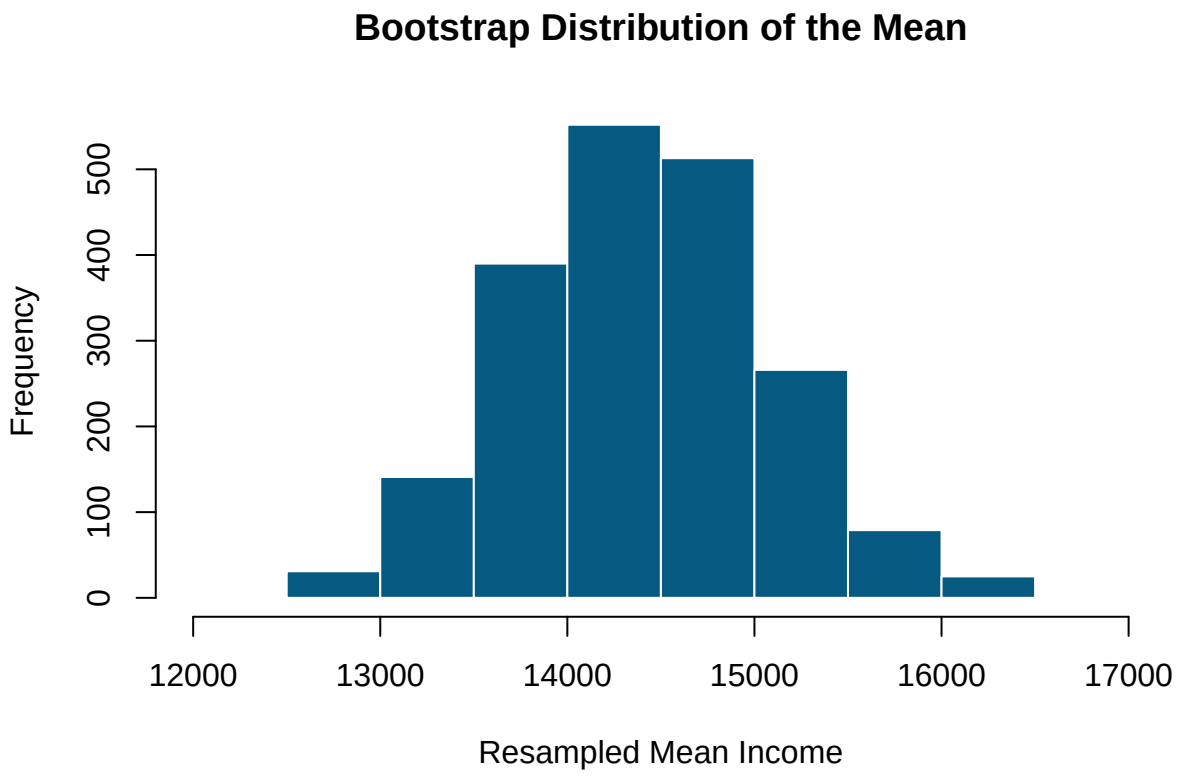
```
#> [1] 13840.46 13902.35 14971.23 12668.58 14394.38
```

Each resample gives a slightly different mean. The *spread* of these means across thousands of resamples represents the uncertainty of our original sample mean estimate.

3.3 The Manual Bootstrap Algorithm (Using a Loop)

To build a full sampling distribution, we repeat this process $B = 2000$ times using a `for` loop, saving the resampled mean in an empty vector at each step:

```
# 1. Choose replication size (B = 2000 is standard)
B <- 2000

# 2. Set up an empty vector to store results
boot_means <- numeric(B)

# 3. Run the loop
for (i in 1:B) {
  resample <- sample(income$monthly_income, replace = TRUE)
  boot_means[i] <- mean(resample)
}

# 4. Plot the results!
hist(boot_means,
     main = "Bootstrap Distribution of the Mean",
     xlab = "Resampled Mean Income",
     col = "#065A82",
     border = "white")
```



We can now compute the standard error (the standard deviation of our resampled means) and compare it to the original mean:

```
mean(boot_means)          # Center of bootstrap distribution (matches sample mean)
```

```
#> [1] 14410.82
```

```
sd(boot_means)             # Bootstrap Standard Error!
```

```
#> [1] 667.6659
```

3.4 The Professional Workflow: Using the `boot` Package

While writing a loop is excellent for learning, modern R programmers use the **`boot`** package. It is faster and automatically calculates multiple types of confidence intervals.

To use the `boot` package, we must write a custom function that accepts two arguments:

1. **`data`**: The dataset.
2. **`indices`**: A vector of row numbers that `boot` generates during resampling.

3.4.1 How do "Indices" work? (The Index Mechanics)

Students often find the `indices` argument confusing. Let's see what it does in detail:

Imagine your dataset is `d <- c("A", "B", "C")`. During bootstrap resampling, the `boot` package generates a random set of row indices, for example: `i <- c(1, 1, 3)`. Inside our function, R subsets our data using these indices: `d[i]`, which outputs `c("A", "A", "C")`.

Let's test this manually:

```
# Let's write a simple mean function that uses index subsetting
mean_fn <- function(data, indices) {
  resampled_subset <- data[indices]
  return(mean(resampled_subset))
}

# Run it manually on the full data using regular indices (1 to 100):
mean_fn(income$monthly_income, 1:100)
```

```
#> [1] 14418.37
```

```
# Run it manually using a set of random resampled indices:
random_indices <- sample(1:100, replace = TRUE)
mean_fn(income$monthly_income, random_indices)
```

```
#> [1] 14735.64
```

Now that our function works, we pass it to the **`boot()`** function, which handles the loop and index generation automatically:

```
library(boot)

# Run the bootstrap simulation
boot_result <- boot(data = income$monthly_income, statistic = mean_fn, R = 2000)

# Print the boot summary
boot_result
```

```
#>
#> ORDINARY NONPARAMETRIC BOOTSTRAP
#>
#>
```

```
#> Call:
#> boot(data = income$monthly_income, statistic = mean_fn, R = 2000)
#>
#>
#> Bootstrap Statistics :
#>     original    bias    std. error
#> t1* 14418.37 -10.36739    671.0528
```

Notice: `original` *shows the original sample mean, and* `std.error` *shows the bootstrap standard error (matching our manual loop standard deviation).*

3.5 Bootstrap Confidence Intervals: Percentile and BCa

Once we run `boot()`, we can extract the standard error and compute confidence intervals using `boot.ci()`:

```
# Extract the bootstrap SE (t holds the resampled statistic values)
sd(boot_result$t)
```

```
#> [1] 671.0528
```

```
# Calculate both 95% Percentile and BCa Confidence Intervals
boot.ci(boot_result, type = c("perc", "bca"))
```

```
#> BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
#> Based on 2000 bootstrap replicates
#>
#> CALL :
#> boot.ci(boot.out = boot_result, type = c("perc", "bca"))
#>
#> Intervals :
#> Level     Percentile            BCa
#> 95%   (13111, 15758 )   (13199, 15831 )
#> Calculations and Intervals on Original Scale
```

Bias-Corrected and Accelerated (BCa) Confidence Intervals

While the **Percentile Bootstrap Interval** (obtained via `type = "perc"`) is simple and intuitive, it assumes that the bootstrap distribution is unbiased and has constant variance. If the bootstrap distribution is skewed or has non-constant variance (heteroscedasticity), percentile intervals can fail to provide the nominal coverage (e.g., a "95% interval" might only contain the parameter 91% of the time).

To correct for these limitations, the **Bias-Corrected and Accelerated (BC_a)** interval (obtained via `type = "bca"`) introduces two correction factors:

1. **Bias Correction (z_0)**: Corrects for asymmetry in the bootstrap distribution (the extent to which the median of the bootstrap replicates differs from the original sample estimate).
2. **Acceleration (a)**: Corrects for changes in the standard error of the statistic as the parameter value changes (skewness in the sampling distribution).

The BC_a interval is more computationally intensive and requires larger bootstrap replications ($R \geq 1000$ or 2000), but it is mathematically superior and preferred for skewed or non-normal data.

Let's compare these to the classical t-based parametric interval:

```
t.test(income$monthly_income)$conf.int
```

```
#> [1] 13097.04 15739.70
#> attr(,"conf.level")
#> [1] 0.95
```

Check Before You Run This (Bootstrap Confidence Interval Diagnostics) Before relying on bootstrap confidence intervals, verify these conditions:

1. **Representative Sample**: The sample must be representative of the underlying population. The bootstrap cannot correct for selection bias.
2. **Independence**: The observations must be independent of one another (no clustering or serial correlation).
3. **Replication Size (R)**: Ensure $R \geq 1000$ (preferably 2000) for stable percentile and BC_a intervals. Small values of R lead to highly unstable endpoints.

How to Report This in a Paper: > "The mean monthly household income was estimated to be \$14,418.37 (Bootstrap SE = \$671.05). A 95% bias-corrected and accelerated (BC_a) bootstrap confidence interval for the mean was [\$13,199, \$15,831], indicating that the true mean is likely within this range."

3.5.1 Bootstrapping a statistic with no textbook formula: the median

```
median_fn <- function(data, indices) median(data[indices])
boot_median <- boot(income$monthly_income, statistic = median_fn, R = 2000)
boot.ci(boot_median, type = c("perc", "bca"))
```

```
#> BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
#> Based on 2000 bootstrap replicates
#>
#> CALL :
#> boot.ci(boot.out = boot_median, type = c("perc", "bca"))
#>
#> Intervals :
#> Level     Percentile            BCa
#> 95%   (11687, 15155 )   (11561, 14885 )
#> Calculations and Intervals on Original Scale
```

Bootstrapping Variance

In many clinical and scientific studies, we are interested in comparing the *variability* (variance) of a measurement. Unlike the mean, sample variance does not have a clean, robust parametric confidence interval formula under non-normality. Let's bootstrap the variance of our skewed income dataset:

```
var_fn <- function(data, indices) var(data[indices])
boot_var <- boot(income$monthly_income, statistic = var_fn, R = 2000)
boot_var
```

```
#>
#> ORDINARY NONPARAMETRIC BOOTSTRAP
#>
#>
#> Call:
#> boot(data = income$monthly_income, statistic = var_fn, R = 2000)
```

```
#>
#>
#> Bootstrap Statistics :
#>     original    bias    std. error
#> t1* 44344993 -372290.9     5985341
```

```
boot.ci(boot_var, type = c("perc", "bca"))
```

```
#> BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
#> Based on 2000 bootstrap replicates
#>
#> CALL :
#> boot.ci(boot.out = boot_var, type = c("perc", "bca"))
#>
#> Intervals :
#> Level     Percentile            BCa
#> 95%   (32943582, 56292079 )   (34421297, 58907968 )
#> Calculations and Intervals on Original Scale
```

Second example — bootstrapping a correlation coefficient, using `mtcars`:

```
cor_fn <- function(data, indices) {
  d <- data[indices, ]
  cor(d$hp, d$mpg)
}
boot_cor <- boot(mtcars, statistic = cor_fn, R = 2000)
boot_cor$t0                      # the original correlation
```

```
#> [1] -0.7761684
```

```
boot.ci(boot_cor, type = "perc")
```

```
#> BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
#> Based on 2000 bootstrap replicates
#>
#> CALL :
#> boot.ci(boot.out = boot_cor, type = "perc")
#>
#> Intervals :
#> Level     Percentile
#> 95%   (-0.8733, -0.6904 )
#> Calculations and Intervals on Original Scale
```

Using the `course_dataset.csv` file (load it with `read.csv("data/course_dataset.csv")`), bootstrap a 95% CI for the **median** of `baseline_pain`. Compare it to a t-based CI for the mean of the same variable — are they telling a similar or different story?

3.6 Bootstrap Confidence Intervals for Proportions

```
course_df <- read.csv("data/course_dataset.csv")
p_hat <- mean(course_df$response == "Responder")
p_hat
```

```
#> [1] 0.7277778
```

Wald CI (parametric, via `prop.test`):

```
n_responders <- sum(course_df$response == "Responder")
n_total <- nrow(course_df)
prop.test(n_responders, n_total)$conf.int
```

```
#> [1] 0.6555826 0.7900526
#> attr(,"conf.level")
#> [1] 0.95
```

Bootstrap CI:

```
prop_fn <- function(data, indices) {
  mean(data[indices] == "Responder")
}
boot_prop <- boot(course_df$response, statistic = prop_fn, R = 2000)
boot.ci(boot_prop, type = "perc")
```

```
#> BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
#> Based on 2000 bootstrap replicates
#>
#> CALL :
#> boot.ci(boot.out = boot_prop, type = "perc")
#>
#> Intervals :
#> Level     Percentile
#> 95%   ( 0.6611,  0.7944 )
#> Calculations and Intervals on Original Scale
```

With a moderate proportion (not extremely close to 0 or 1) and a reasonably large sample (as here, where $\hat{p} \approx 0.73$), the Wald and bootstrap intervals usually agree closely. The bootstrap approach earns its keep most clearly when the proportion is close to 0 or 1, or when the sample size is small, situations where the Wald interval can behave badly (even extending below 0% or above 100%, which is nonsensical for a proportion).

Second example — a small, extreme proportion where the two methods diverge more:

```
set.seed(1)
rare_event <- rbinom(30, size = 1, prob = 0.05)   # only ~5% "successes", small n
mean(rare_event)
```

```
#> [1] 0.03333333
```

```
prop.test(sum(rare_event), length(rare_event))$conf.int
```

```
#> [1] 0.001742467 0.190530216
#> attr(,"conf.level")
#> [1] 0.95
```

```
boot.ci(boot(rare_event, function(d, i) mean(d[i]), R = 2000), type = "perc")
```

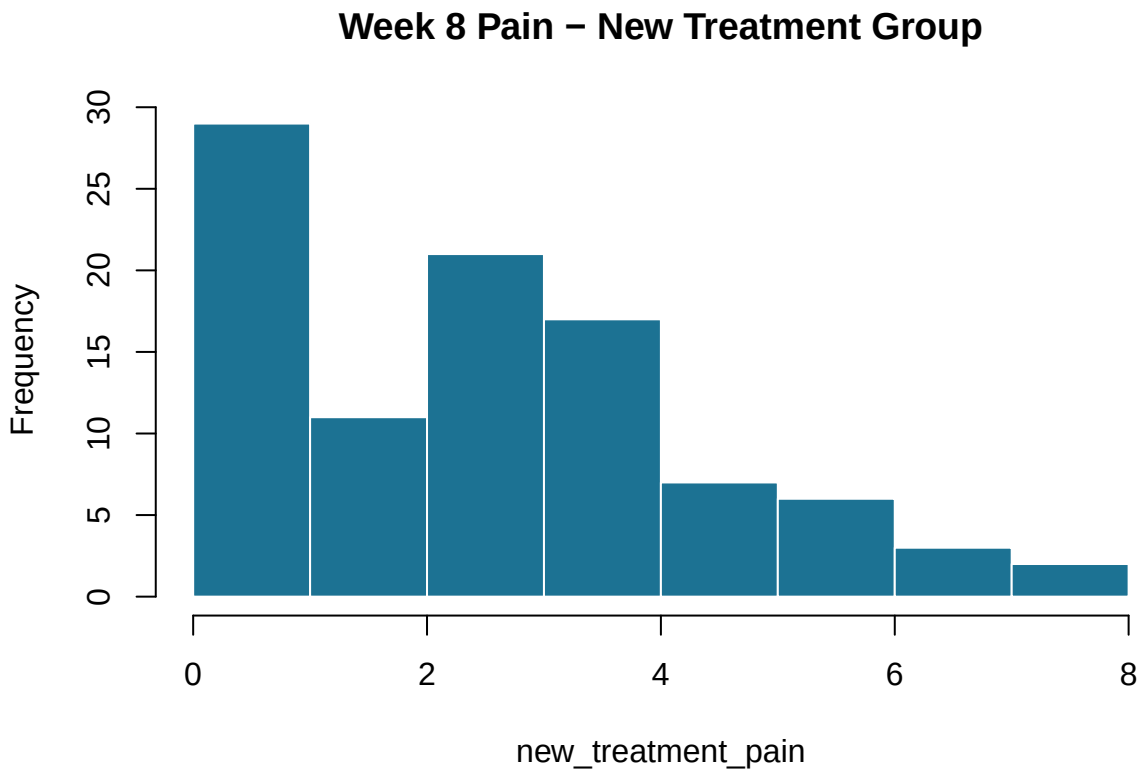
```
#> BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
#> Based on 2000 bootstrap replicates
#>
#> CALL :
#> boot.ci(boot.out = boot(rare_event, function(d, i) mean(d[i]),
#>     R = 2000), type = "perc")
#>
#> Intervals :
#> Level     Percentile
#> 95%   ( 0.0,  0.1 )
#> Calculations and Intervals on Original Scale
```

Notice how much wider and more asymmetric the bootstrap interval looks here — that asymmetry is real information the symmetric Wald interval simply cannot represent.

3.7 An End-to-End Bootstrap Workflow

Here's a repeatable process you can apply to any new dataset:

```r
# STEP 1: State the question
# "What is the mean week-8 pain score for patients on the New treatment?"

# STEP 2: Check assumptions
new_treatment_pain <- course_df$week8_pain[course_df$treatment == "New"]
hist(
  new_treatment_pain,
  main = "Week 8 Pain - New Treatment Group",
  col = "#1C7293", border = "white"
)
```




```r
shapiro.test(new_treatment_pain)
```

```
#>
#>  Shapiro-Wilk normality test
#>
#> data:  new_treatment_pain
#> W = 0.94252, p-value = 0.0003735
```

```r
# STEP 3: Run the bootstrap
mean_fn2 <- function(data, indices) mean(data[indices])
boot_final <- boot(new_treatment_pain, statistic = mean_fn2, R = 2000)

# STEP 4: Construct the CI
ci_final <- boot.ci(boot_final, type = "perc")
ci_final
```

```
#> BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
#> Based on 2000 bootstrap replicates
#>
#> CALL :
#> boot.ci(boot.out = boot_final, type = "perc")
#>
#> Intervals :
#> Level     Percentile
#> 95%   ( 2.083,  2.828 )
#> Calculations and Intervals on Original Scale
```

```r
# STEP 5: Report
cat(sprintf(
  paste0(
    "Mean week-8 pain in the New treatment group: %.2f ",
    "(95%% bootstrap CI: %.2f to %.2f)\n"
  ),
  mean(new_treatment_pain),
  ci_final$percent[4],
  ci_final$percent[5]
))
```

```
#> Mean week-8 pain in the New treatment group: 2.45 (95% bootstrap CI: 2.08 to 2.83)
```

Repeat this exact five-step workflow, but for the **Standard** treatment group instead of New. Do the two confidence intervals overlap? What would that suggest about the two treatments?

- Parametric CIs rely on assumptions (usually approximate normality) that skewed or small data can violate.
- The bootstrap approximates a sampling distribution empirically, by resampling your data *with replacement* many times.
- `boot()` + `boot.ci()` is the standard R workflow — and it generalizes to *any* statistic, even ones with no textbook CI formula (medians, correlations, custom metrics).
- Bootstrap CIs for proportions are especially valuable when the sample is small or the proportion is near 0 or 1.
- A repeatable five-step workflow (question → check assumptions → bootstrap → CI → report) will serve you for the rest of this course and beyond.

Next, in Chapter 4, we move from continuous variables to categorical data — contingency tables, agreement measures, and the art of reporting them compactly.

Chapter 3 covered:

- **Correct interpretation**: A 95% CI does *not* mean "95% probability the parameter is in this interval" — it means 95% of intervals constructed this way, over repeated samples, would contain the true parameter
- **Parametric CIs**: Based on the t-distribution (mean), normal approximation (proportion), and their differences — require distributional assumptions
- **Bootstrap principle**: Treat the sample as a proxy for the population; resample with replacement B times; the variability of the bootstrap statistic estimates the sampling variability
- **Bootstrap CI types**: Percentile (simple), Basic (bias-corrected), BCa (bias-corrected and accelerated — most accurate)
- **When to bootstrap**: Small samples, unknown distributions, complex estimators (correlation, ratio, regression coefficients)

Method	Formula	When to Use
One-sample t	$\bar{x} \pm t_{n-1,\alpha/2} \cdot s/\sqrt{n}$	Normal population or $n \geq 30$
Proportion (Wald)	$\hat{p} \pm z_{\alpha/2}\sqrt{\hat{p}(1-\hat{p})/n}$	$n\hat{p} \geq 10$, $n(1-\hat{p}) \geq 10$
Bootstrap (percentile)	$[\hat{\theta}^*_{(B\alpha/2)}, \hat{\theta}^*_{(B(1-\alpha/2))}]$	Any estimator, any distribution

Coming up: Chapter 4 moves from continuous outcomes to categorical data — contingency tables, association tests, and agreement measures.

3.8 Lab 3: Confidence Intervals for the Course Dataset

Objective: Construct and compare parametric and bootstrap CIs for multiple estimands from the `course_dataset`.

Dataset: `course_dataset.csv`

3.8.1 Step 1: Load Data

```
library(boot)
course_df <- read.csv("data/course_dataset.csv")
```

3.8.2 Step 2: Parametric CI for Mean Baseline Pain

```
# Using t.test (returns CI automatically)
t_result <- t.test(course_df$baseline_pain, conf.level = 0.95)
cat("95% CI for mean baseline pain:
")
```

```
#> 95% CI for mean baseline pain:
```

```
cat(round(t_result$conf.int, 2), "
")
```

```
#> 5.97 6.58
```

3.8.3 Step 3: CI for Proportion of Responders

```r
n_resp <- sum(course_df$response == "Responder")
n_total <- nrow(course_df)
# Wilson CI (more accurate than Wald for proportions)
prop.test(n_resp, n_total, conf.level = 0.95)
```

```
#>
#>  1-sample proportions test with continuity correction
#>
#> data:  n_resp out of n_total, null probability 0.5
#> X-squared = 36.45, df = 1, p-value = 1.566e-09
#> alternative hypothesis: true p is not equal to 0.5
#> 95 percent confidence interval:
#>  0.6555826 0.7900526
#> sample estimates:
#>         p
#> 0.7277778
```

3.8.4 Step 4: Bootstrap CI for the Median

```r
set.seed(42)
median_fn <- function(data, indices) median(data[indices])
boot_out <- boot(course_df$baseline_pain, statistic = median_fn, R = 2000)
boot.ci(boot_out, type = c("perc", "basic"), conf = 0.95)
```

```
#> BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
#> Based on 2000 bootstrap replicates
#>
#> CALL :
#> boot.ci(boot.out = boot_out, conf = 0.95, type = c("perc", "basic"))
#>
#> Intervals :
#> Level      Basic              Percentile
#> 95%   ( 6.0,  6.9 )   ( 5.7,  6.6 )
#> Calculations and Intervals on Original Scale
```

3.8.5 Step 5: Bootstrap CI for the Difference in Means (New vs Standard Treatment)

```r
set.seed(42)
diff_fn <- function(data, indices) {
  d <- data[indices, ]
  mean(d$baseline_pain[d$treatment == "New"], na.rm = TRUE) -
  mean(d$baseline_pain[d$treatment == "Standard"], na.rm = TRUE)
}
boot_diff <- boot(course_df, statistic = diff_fn, R = 2000, strata =
 ↪  as.factor(course_df$treatment))
boot.ci(boot_diff, type = c("perc", "norm", "basic"), conf = 0.95)
```

```
#> BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
#> Based on 2000 bootstrap replicates
#>
#> CALL :
```

```
#> boot.ci(boot.out = boot_diff, conf = 0.95, type = c("perc", "norm",
#>     "basic"))
#>
#> Intervals :
#> Level      Normal              Basic              Percentile
#> 95%   (-0.7911,  0.4238 )   (-0.7949,  0.4164 )   (-0.7801,  0.4312 )
#> Calculations and Intervals on Original Scale
```

3.8.6 Lab Exercises

1. Compute a 90% and 99% t-interval for mean `week8_pain`. How do they differ in width?
2. Compute bootstrap CIs for the correlation between `baseline_pain` and `week8_pain` using `R = 1000` resamples.
3. Compare the parametric and bootstrap 95% CI for the mean of `baseline_pain`. Are they similar? If they differ, what might explain the difference?

Chapter 4

Analysis of Contingency Tables and Compact Reporting

This chapter is about categorical data — variables measured in categories rather than numbers. You'll learn to build contingency tables, test for association and trend, measure agreement between raters, handle paired categorical data, control for confounding variables, and produce clean summary tables for reports.

Most of this chapter uses **course_dataset** (180 simulated patients across three hospital sites) because it was specifically built to contain every kind of categorical relationship this chapter covers: a treatment/response 2×2 table, a site-stratified design, an ordinal exposure variable, two raters' severity gradings, and paired pre/post symptom data. We'll bring in `InsectSprays` (built into R) for one contrasting example.

```
course_df <- read.csv("data/course_dataset.csv")
str(course_df)
```

```
#> 'data.frame':    180 obs. of  14 variables:
#>  $ id             : chr  "P001" "P002" "P003" "P004" ...
#>  $ site           : chr  "Site C" "Site B" "Site A" "Site C" ...
#>  $ treatment      : chr  "New" "Standard" "New" "Standard" ...
#>  $ age            : int  59 68 33 53 58 49 42 40 67 49 ...
#>  $ sex            : chr  "Male" "Male" "Male" "Male" ...
#>  $ activity_level : chr  "Moderate" "High" "Moderate" "Low" ...
#>  $ baseline_pain  : num  8.7 6.7 7.6 5.5 6 4.1 6.8 6 5.8 6.6 ...
#>  $ week4_pain     : num  5.8 6.5 3.9 2.7 5.9 4.6 6.6 2.5 4.4 2.9 ...
#>  $ week8_pain     : num  5.1 5.6 0.1 2.4 5.1 4.6 4.2 2.4 3.5 1.3 ...
#>  $ response       : chr  "Responder" "Non-Responder" "Responder" "Responder" ...
#>  $ symptom_pre    : chr  "Present" "Absent" "Present" "Present" ...
#>  $ symptom_post   : chr  "Absent" "Present" "Present" "Absent" ...
#>  $ rater1_severity: chr  "Severe" "Mild" "Moderate" "Severe" ...
#>  $ rater2_severity: chr  "Severe" "Mild" "Moderate" "Severe" ...
```

4.1 Contingency Table Structure

```
tab_2x2 <- table(course_df$treatment, course_df$response)
tab_2x2
```

```
#>
#>              Non-Responder Responder
#>   New                    4        92
#>   Standard              45        39
```

```
tab_3x3 <- table(course_df$rater1_severity, course_df$rater2_severity)
tab_3x3
```

```
#>
#>            Mild Moderate Severe
#>   Mild       53       10      2
#>   Moderate    8       62      6
#>   Severe      3        4     32
```

Adding margins (row/column totals):

```
addmargins(tab_2x2)
```

```
#>
#>            Non-Responder Responder Sum
#>   New                  4        92  96
#>   Standard            45        39  84
#>   Sum                 49       131 180
```

Using `xtabs()` with formula notation (handy for more than two variables):

```
xtabs(~ treatment + response, data = course_df)
```

```
#>           response
#> treatment  Non-Responder Responder
#>   New                  4        92
#>   Standard            45        39
```

Manual Table Generation Using Loops

Before using R's built-in `table()` or `xtabs()` functions, it is highly instructive to see how R builds a contingency table algorithmically. We can initialize a blank matrix and loop through the dataset row by row, incrementing the counts:

```
# 1. Initialize an empty 2x2 matrix with zeros
row_labels <- c("New", "Standard")
col_labels <- c("Non-Responder", "Responder")
manual_tab <- matrix(0, nrow = 2, ncol = 2,
                     dimnames = list(row_labels, col_labels))

# 2. Loop through each row of the dataset and increment counts
for (i in 1:nrow(course_df)) {
  trt <- course_df$treatment[i]
  resp <- course_df$response[i]

  if (trt %in% row_labels && resp %in% col_labels) {
    manual_tab[trt, resp] <- manual_tab[trt, resp] + 1
  }
}
manual_tab
```

```
#>          Non-Responder Responder
#> New                  4        92
#> Standard            45        39
```

This manual loop matches the output of `table(course_df$treatment, course_df$response)` exactly, showing the underlying logic of cross-tabulation.

4.2 Chi-Square and Fisher's Exact Test

The **Chi-Square (χ^2) Test of Independence** is used to determine whether there is a statistically significant association between two categorical variables.

- **Null Hypothesis (H_0)**: The two variables are independent (no association).
- **Alternative Hypothesis (H_1)**: The two variables are dependent (associated).

4.2.1 Step 1: Running the Chi-Square Test

Let's run the test on our `tab_2x2` (Treatment vs. Pain Relief):

```
chisq_result <- chisq.test(tab_2x2)
# Print the results
print(chisq_result)
```

```
#>
#>  Pearson's Chi-squared test with Yates' continuity correction
#>
#> data:  tab_2x2
#> X-squared = 52.729, df = 1, p-value = 3.83e-13
```

Check Before You Run This (Chi-Square & Fisher's Diagnostics) Before running a Chi-Square test of independence:

1. **Categorical Variables**: Both variables must be nominal or ordinal.
2. **Independence of Observations**: Each subject must contribute to exactly one cell in the table (no repeated measures, which require McNemar's test).
3. **Adequate Expected Counts**: The expected count in each cell must be ≥ 5 (or at least 80% of cells ≥ 5 and no cells < 1). If this is violated, use **Fisher's Exact Test** instead.

4.2.2 Step 2: Understanding the Output

The output lists three key values:

1. `X-squared`: The test statistic ($\chi^2 = 52.73$). A larger value indicates a greater difference between what we observed and what we would expect if the variables were independent.
2. `df`: Degrees of freedom. For a table of r rows and c columns, $\text{df} = (r-1) \times (c-1)$. In our 2×2 table, $\text{df} = (2-1) \times (2-1) = 1$.
3. `p-value`: The probability of observing such an extreme association by chance. Here, $p = 3.83 \times 10^{-13}$, which is extremely small (far below the standard 0.05 threshold), indicating a very strong, highly significant association.

The Math Behind It: Manual Chi-Square Calculation & Yates' Correction The mathematical formula for the standard Pearson Chi-Square statistic is:

$$\chi^2 = \sum \frac{(O-E)^2}{E}$$

where O represents the observed cell counts and E represents the expected cell counts.

However, for 2×2 tables, R's `chisq.test()` defaults to applying **Yates' continuity correction** to prevent overestimating significance in small tables. The corrected formula is:

$$\chi^2_{\text{Yates}} = \sum \frac{(|O-E| - 0.5)^2}{E}$$

Let's compute both manually in R to see how they match R's output:

```r
# 1. Extract observed and expected values as vectors
obs <- as.vector(tab_2x2)
exp <- as.vector(chisq_result$expected)

# 2. Standard Pearson Chi-Square (without correction)
chi_standard <- sum((obs - exp)^2 / exp)
chi_standard
```

```
#> [1] 55.19418
```

```r
pchisq(chi_standard, df = 1, lower.tail = FALSE)
```

```
#> [1] 1.091917e-13
```

```r
# 3. Yates' Corrected Chi-Square (matching R's default chisq.test output)
chi_yates <- sum((abs(obs - exp) - 0.5)^2 / exp)
chi_yates
```

```
#> [1] 52.72863
```

```r
pchisq(chi_yates, df = 1, lower.tail = FALSE)
```

```
#> [1] 3.829673e-13
```

Notice that `chi_yates` matches R's default output (52.729) exactly. If you disable the correction in R using `chisq.test(tab_2x2, correct = FALSE)`, it will output `chi_standard` (55.194).

How to Report This in a Paper: > "A Pearson's chi-square test with Yates' continuity correction revealed a statistically significant association between treatment and pain response, $\chi^2(1) = 52.73$, $p < .001$. The association was strong, with a Phi coefficient (effect size) of $\phi = \sqrt{52.73/180} = 0.54$."

4.2.3 Step 3: Checking Assumptions (Expected Cell Counts)

The Chi-Square test is an **approximation** that assumes a large sample size. It becomes unreliable if the **expected cell counts** are too small.

- **The Rule of Thumb**: At least 80% of expected cell counts must be ≥ 5, and no cell should have an expected count < 1.

Let's check the expected counts calculated by R:

```r
# Extract the expected counts matrix
chisq_result$expected
```

```
#>
#>            Non-Responder Responder
#>   New            26.13333  69.86667
#>   Standard       22.86667  61.13333
```

Pedagogical note: R calculates expected values using the formula:

$$\text{Expected Count} = \frac{\text{Row Total} \times \text{Column Total}}{\text{Grand Total}}$$

For example, the expected count for (New, Non-Responder) is $(96 \times 49)/180 = 26.13$. *Since all our expected values are* ≥ 22.87 *(which is well above the threshold of* 5*), the Chi-Square test assumptions are perfectly satisfied.*

4.2.4 Step 4: Small Samples — Fisher's Exact Test

If any expected cell count is < 5, the Chi-Square test is invalid. In such cases, we use **Fisher's Exact Test**, which calculates the *exact* hypergeometric probability of the table.

Let's run Fisher's test:

```
fisher_result <- fisher.test(tab_2x2)
print(fisher_result)
```

```
#>
#>  Fisher's Exact Test for Count Data
#>
#> data:  tab_2x2
#> p-value = 1.341e-14
#> alternative hypothesis: true odds ratio is not equal to 1
#> 95 percent confidence interval:
#>  0.009392081 0.115710445
#> sample estimates:
#> odds ratio
#> 0.03841056
```

Notice that Fisher's test also gives you:

- **Odds Ratio (OR)**: Under independence, OR $= 1$. Here, the odds ratio is 0.038, which represents the odds of being a *Non-Responder* in the *New* treatment group compared to the *Standard* treatment group. Since the odds ratio is much less than 1, patients on the *New* treatment have a massive 96.2% reduction in the odds of being a Non-Responder (meaning they are dramatically more likely to be Responders!).
- **95% Confidence Interval for the OR**: Since this interval (0.009 to 0.116) does not contain 1 and is far below it, the association is highly statistically significant, matching the extremely small p-value ($p = 1.34 \times 10^{-14}$).

4.2.5 Step 5: Testing Larger Tables ($r \times c$)

Let's see what happens when we test a larger 3×3 table (`tab_3x3` representing Rater 1 vs Rater 2 Severity classifications):

```
# Run Chi-Square test
chisq_3x3 <- chisq.test(tab_3x3)
print(chisq_3x3)
```

```
#>
#>  Pearson's Chi-squared test
#>
#> data:  tab_3x3
#> X-squared = 189.53, df = 4, p-value < 2.2e-16
```

```
# Check Expected Cell Counts (Crucial!)
chisq_3x3$expected
```

```
#>
#>                Mild Moderate    Severe
#>   Mild     23.11111 27.44444 14.444444
#>   Moderate 27.02222 32.08889 16.888889
#>   Severe   13.86667 16.46667  8.666667
```

Note: In this case, all expected cell counts are above 5 (ranging from 8.67 to 32.09), so the Chi-Square approximation is technically valid. We show Fisher's exact test here as a demonstration of how it can be applied to larger tables, or as a robustness check.

Because we want to demonstrate the method, we can run Fisher's Exact Test:

```
fisher.test(tab_3x3)
```

```
#>
#>  Fisher's Exact Test for Count Data
#>
#> data:  tab_3x3
#> p-value < 2.2e-16
#> alternative hypothesis: two.sided
```

Exercise: Test on InsectSprays Using R's built-in `InsectSprays` dataset:

1. Re-categorize the insect counts into a binary column: `InsectSprays$high_count <- ifelse(InsectSprays$count > 10, "High", "Low")`.
2. Generate a table of `spray` (type of spray) × `high_count`.
3. Check the expected counts for this table. Can you use a Chi-Square test, or should you run Fisher's exact test? Execute the appropriate test.

4.3 Trend Tests for Ordinal Data

When your exposure variable is ordinal (has a natural order), a trend test uses that order and is more powerful than a generic chi-square test.

```
trend_tab <- table(course_df$activity_level, course_df$response)
trend_tab
```

```
#>
#>            Non-Responder Responder
#>   High                 6        31
#>   Low                 19        48
#>   Moderate            24        52
```

Using base R's `prop.trend.test()` (a Cochran-Armitage-style trend test built into R):

```
# prop.trend.test needs: number of "successes",
# total n, and a numeric score per group
successes <- trend_tab[, "Responder"]
totals <- rowSums(trend_tab)
# Low=1, Moderate=2, High=3
prop.trend.test(successes, totals, score = c(1, 2, 3))
```

```
#>
#>  Chi-squared Test for Trend in Proportions
#>
#> data:  successes out of totals ,
#>  using scores: 1 2 3
#> X-squared = 2.6318, df = 1, p-value = 0.1047
```

Using the `coin` package for a linear-by-linear association test:

```
library(coin)
course_df$activity_level <- factor(
  course_df$activity_level,
  levels = c("Low", "Moderate", "High"),
  ordered = TRUE
)
lbl_test(table(course_df$activity_level, course_df$response))
```

```
#>
#>  Asymptotic Linear-by-Linear Association Test
#>
#> data:  Var2 by Var1 (Low < Moderate < High)
#> Z = -1.0883, p-value = 0.2765
#> alternative hypothesis: two.sided
```

Both trend tests above assume the categories are genuinely ordered and roughly evenly spaced in the underlying construct they represent. If your "ordinal" categories are really just unordered labels with no natural progression, a trend test doesn't make sense — use a regular chi-square test instead.

Compare the p-value from `prop.trend.test()` above to a plain `chisq.test(trend_tab)`. Which is smaller? Given that `activity_level` genuinely has an order, which result should you trust more?

4.4 Agreement Measures: Kappa

When two raters independently classify the same subjects, **percent agreement** overstates how good the agreement really is, because some agreement happens by chance alone. Cohen's kappa corrects for this.

Check Before You Run This (Cohen's Kappa Diagnostics) Before running a Kappa test:

1. **Identical Categories**: Both raters must use the exact same set of mutually exclusive, categorical rating levels (e.g., both use Mild/Moderate/Severe).
2. **Paired Observations**: The subjects being rated must be identical, and each subject must be rated by both raters.
3. **Independent Raters**: The two raters must perform their ratings independently of each other.

```
library(psych)
kappa_result <- cohen.kappa(cbind(
  course_df$rater1_severity,
  course_df$rater2_severity
))
kappa_result
```

```
#> Call: cohen.kappa1(x = x, w = w, n.obs = n.obs, alpha = alpha, levels = levels,
#>     w.exp = w.exp)
#>
#> Cohen Kappa and Weighted Kappa correlation coefficients and confidence boundaries
#>                  lower estimate upper
#> unweighted kappa  0.63     0.72  0.80
#> weighted kappa    0.67     0.76  0.86
#>
#>  Number of subjects = 180
```

Since severity is ordinal, the **weighted kappa** (which penalizes a "Mild vs. Severe" disagreement more than a "Mild vs. Moderate" disagreement) is more appropriate.

Under the hood:

- **Unweighted Kappa**: Treats all off-diagonal disagreements equally.
- **Weighted Kappa**: Assigns weights to off-diagonal cells depending on their distance from the diagonal (representing agreement).
  - **Linear weights** (the default or `weight = "linear"`): Penalty increases linearly with the number of categories separating the ratings.
  - **Quadratic weights** (often preferred in medical literature): Penalty increases quadratically with the distance, placing much heavier penalties on extreme disagreements.

Let's inspect the weighted kappa output:

```
kappa_result$weighted.kappa
```

```
#> [1] 0.7612732
```

The Math Behind It: Manual Kappa Calculation Cohen's Kappa (κ) is calculated as:

$$\kappa = \frac{p_o - p_e}{1 - p_e}$$

where p_o represents the observed proportion of agreement, and p_e represents the expected proportion of agreement by chance alone.

Let's calculate this manually in R to verify it matches `cohen.kappa()`:

```
# 1. Create a table of the two raters
rater_a <- factor(c("Mild","Mild","Moderate","Severe","Severe","Moderate"))
rater_b_bad  <- factor(c("Severe","Mild","Mild","Moderate","Mild","Severe"))
tab <- table(rater_a, rater_b_bad)

# 2. Observed agreement (sum of the diagonal cells / grand total)
po <- sum(diag(tab)) / sum(tab)
po
```

```
#> [1] 0.1666667
```

```
# 3. Expected agreement (sum of row probability * col probability)
row_prob <- rowSums(tab) / sum(tab)
col_prob <- colSums(tab) / sum(tab)
pe <- sum(row_prob * col_prob)
pe
```

```
#> [1] 0.3333333
```

```
# 4. Compute manual kappa
manual_k <- (po - pe) / (1 - pe)
manual_k
```

```
#> [1] -0.25
```

This matches R's `cohen.kappa(cbind(rater_a, rater_b_bad))$kappa` output (-0.25) exactly!

Landis & Koch benchmarks for interpreting kappa: < 0 Poor, 0.00–0.20 Slight, 0.21–0.40 Fair, 0.41–0.60 Moderate, 0.61–0.80 Substantial, 0.81–1.00 Almost Perfect. These are widely used rules of thumb, not universal laws — always report the actual value, not just the label.

How to Report This in a Paper: > "Inter-rater reliability between the two clinical raters on patient symptom severity was substantial (unweighted Cohen's $\kappa = 0.72$, quadratic weighted $\kappa = 0.76$)."

Second example — perfect agreement vs. poor agreement:

```
# identical ratings:
rater_b_good <- factor(c(
  "Mild","Mild","Moderate","Severe","Severe","Moderate"
))

cohen.kappa(cbind(rater_a, rater_b_good))$kappa
```

```
#> [1] 1
```

```
cohen.kappa(cbind(rater_a, rater_b_bad))$kappa
```

```
#> [1] -0.25
```

Simulate your own pair of raters with `sample()`, deliberately making them agree about 90% of the time (hint: use `ifelse(runif(n) < 0.9, rater_a, sample(...))` as in the dataset generator). Compute kappa and confirm it lands in the "Substantial" or "Almost Perfect" range.

4.5 Paired Categorical Tests: McNemar and Cochran's Q

4.5.1 McNemar's test — two paired binary measurements

```
mcnemar_tab <- table(course_df$symptom_pre, course_df$symptom_post)
mcnemar_tab
```

```
#>
#>           Absent Present
#>   Absent      24      20
#>   Present     79      57
```

```
mcnemar.test(mcnemar_tab)
```

```
#>
#>  McNemar's Chi-squared test with continuity correction
#>
#> data:  mcnemar_tab
#> McNemar's chi-squared = 33.98, df = 1, p-value = 5.569e-09
```

McNemar's test ignores the patients whose status didn't change (Present→Present or Absent→Absent) and focuses entirely on the *discordant* pairs — patients who changed status in one direction or the other. It tests whether those changes are symmetric (equally likely in both directions) or not.

4.5.2 Cochran's Q test — generalizing McNemar to 3+ repeated measurements

Cochran's Q isn't available in a commonly pre-packaged apt-installed library, but the formula is simple enough to implement directly — which is also a great way to *really* understand what it's doing:

```
cochran_q <- function(x) {
  # x: matrix, rows = subjects,
  # columns = repeated binary measurements (0/1)
  x <- as.matrix(x)
  k <- ncol(x)
```

```
  Ri <- rowSums(x)
  Cj <- colSums(x)
  Q <- (k - 1) * (k * sum(Cj^2) - sum(Cj)^2) /
       (k * sum(Ri) - sum(Ri^2))
  list(
    statistic = Q,
    df = k - 1,
    p.value = pchisq(Q, k - 1, lower.tail = FALSE)
  )
}

# Simulate three repeated binary measurements per patient
# (e.g., symptom present at 3 visits)
set.seed(42)
n_pat <- 40
visit1 <- rbinom(n_pat, 1, 0.6)
visit2 <- rbinom(n_pat, 1, 0.45)
visit3 <- rbinom(n_pat, 1, 0.30)
repeated_symptoms <- cbind(visit1, visit2, visit3)

cochran_q(repeated_symptoms)
```

```
#> $statistic
#> [1] 2.066667
#>
#> $df
#> [1] 2
#>
#> $p.value
#> [1] 0.3558189
```

If you'd rather use a maintained package for Cochran's Q, `RVAideMemoire::cochran.qtest()` provides one — install it with `install.packages("RVAideMemoire")`. We implemented it manually above so this book keeps working even without every optional package installed, and so you can see precisely how the test statistic is built.

4.6 Controlling for a Stratifying Variable: The CMH Test

A treatment-response association might look different — or vanish, or even reverse — once you account for which hospital site a patient came from. The Cochran-Mantel-Haenszel test checks the association *while controlling for* a stratifying variable.

```
# Unstratified: ignore site entirely
chisq.test(table(course_df$treatment, course_df$response))
```

```
#>
#>  Pearson's Chi-squared test with Yates' continuity correction
#>
#> data:  table(course_df$treatment, course_df$response)
#> X-squared = 52.729, df = 1, p-value = 3.83e-13
```

```
# Stratified by site
cmh_array <- table(course_df$treatment, course_df$response, course_df$site)
mantelhaen.test(cmh_array)
```

```
#>
#>  Mantel-Haenszel chi-squared test with continuity correction
#>
#> data:  cmh_array
#> Mantel-Haenszel X-squared = 51.393, df = 1, p-value = 7.559e-13
#> alternative hypothesis: true common odds ratio is not equal to 1
#> 95 percent confidence interval:
#>  0.01378706 0.12237804
#> sample estimates:
#> common odds ratio
#>        0.04107595
```

If the stratified (CMH) result tells a meaningfully different story than the unstratified chi-square result, `site` is likely a **confounder** — a variable associated with both the exposure and outcome that was distorting the raw, unstratified relationship. This is one of the most important ideas in observational data analysis.

Look at the individual 2×2 tables inside `cmh_array` (try `cmh_array[,, "Site A"]`, and so on for the other sites). Does the treatment-response relationship look consistent across all three sites, or does it vary a lot? What does that suggest about whether CMH pooling is appropriate here?

4.7 Compact Reporting: Tables That Publish

4.7.1 A fully-manual baseline table (always works, no extra packages needed)

```r
build_baseline_table <- function(data, vars, strata) {
  groups <- unique(data[[strata]])
  result <- data.frame(Variable = vars)
  for (g in groups) {
    subset_data <- data[data[[strata]] == g, ]
    result[[g]] <- sapply(vars, function(v) {
      x <- subset_data[[v]]
      if (is.numeric(x)) {
        sprintf("%.1f (%.1f)", mean(x, na.rm = TRUE), sd(x, na.rm = TRUE))
      } else {
        paste0(names(sort(table(x), decreasing = TRUE))[1], " most common")
      }
    })
  }
  result
}

build_baseline_table(course_df, vars = c("age", "sex"), strata = "treatment")
```

```
#>   Variable                New           Standard
#> 1      age        53.7 (13.3)        51.0 (11.7)
#> 2      sex Female most common Female most common
```

This is exactly what packages like `tableone` and `gtsummary` automate for you — computing mean (SD) for numeric variables and counts/percentages for categorical ones, split by group. Understanding the manual version means you'll never be stuck if a package isn't available, and you'll better understand what those packages' output actually means.

4.7.2 Using `tableone` and `gtsummary` for Professional Reports

In practice, medical and statistical researchers do not compile these tables manually. They use packages like **`tableone`** or **`gtsummary`** which automate this process and generate beautifully formatted tables with a single line of code.

Characteristic	**New** N = 96[1]	**Standard** N = 84[1]	**p-value**
age	56 (44, 65)	52 (43, 58)	
sex			
Female	50 (52%)	45 (54%)	
Male	46 (48%)	39 (46%)	
baseline_pain	6.10 (4.40, 7.70)	6.45 (4.55, 7.80)	

[1]Median (Q1, Q3); n (%)

Here is how you generate a baseline table using the `tableone` package:

```
library(tableone)
# Generate a Table 1 object
t1 <- CreateTableOne(vars = c("age", "sex", "baseline_pain"),
                     strata = "treatment", data = course_df)
# Print it (renders in standard markdown/text format)
print(t1)
```

```
#>                              Stratified by treatment
#>                               New           Standard      p      test
#>   n                              96             84
#>   age (mean (SD))             53.74 (13.29) 50.98 (11.72)  0.143
#>   sex = Male (%)                 46 (47.9)      39 (46.4)   0.960
#>   baseline_pain (mean (SD))  6.19 (2.09)    6.37 (2.13)    0.564
```

Here is how you generate a highly polished HTML baseline table using the `gtsummary` package:

```
library(gtsummary)
course_df |>
  tbl_summary(by = treatment, include = c(age, sex, baseline_pain)) |>
  add_p()
```

Why do the p-values differ between tableone and gtsummary? A student comparing the two tables above side-by-side will notice that for `age`, the `tableone` package reports a p-value of 0.143, whereas `gtsummary` reports 0.083. This is not an error!

- `tableone` defaults to continuous variables being summarized by **mean (SD)** and compared using a parametric **t-test** (or ANOVA).
- `gtsummary` defaults to continuous variables being summarized by **median (IQR)** and compared using a non-parametric **Wilcoxon rank-sum test** (or Kruskal-Wallis). Always check the defaults of the reporting packages you use!

Extend `build_baseline_table()` above to also report a percentage breakdown for categorical variables (currently it just reports the most common category, which loses information). This is a genuinely useful exercise — you'll end up with something close to what `tableone` does internally.

- `table()` and `xtabs()` build contingency tables; `addmargins()` adds totals.
- Fisher's exact test is safer than chi-square when expected counts are small; check `chisq.test(...)$expected` to decide.
- Trend tests (`prop.trend.test()`, `coin::lbl_test()`) use the natural ordering of an ordinal exposure for more statistical power.
- Cohen's kappa corrects raw percent agreement for chance — always prefer weighted kappa for ordinal categories.
- McNemar's test handles paired binary data; Cochran's Q generalizes it to 3+ repeated binary measurements.

- The CMH test lets you check an association while controlling for a stratifying variable, revealing potential confounding.
- Building a manual summary table once is worth doing — it demystifies what packages like `tableone` and `gtsummary` automate.

Next, in Chapter 5, we turn to hypothesis testing for continuous or ordinal outcomes when parametric assumptions (like normality) don't hold.

Chapter 4 covered:

- **Cross-tabulations**: `table()`, `prop.table()`, and `gtsummary::tbl_cross()` for publication-ready display
- **Chi-square test**: Tests independence between two categorical variables; requires expected frequencies ≥ 5
- **Fisher's exact test**: Preferred when expected frequencies are small (< 5) or the sample is small
- **2×2 measures of association**: Odds ratio (OR), relative risk (RR), and their confidence intervals
- **Cohen's kappa** (κ): Measures agreement between two raters beyond chance; $\kappa = (p_o - p_e)/(1 - p_e)$
- **McNemar test**: For paired 2×2 tables — tests marginal homogeneity; uses only the discordant cells

Test	Use Case	Key Assumption
Chi-square	Independence, 2+ groups	Expected freq. ≥ 5 in all cells
Fisher's exact	Independence, small samples	No minimum expected frequency
Cohen's kappa	Inter-rater agreement	Two raters, same categories
McNemar	Paired proportions	Matched pairs / repeated measures

Coming up: Chapter 5 addresses what to do when the normality assumption fails — non-parametric rank-based tests that work on any distribution.

4.8 Lab 4: Analysing Response Patterns in the Course Dataset

Objective: Apply contingency table methods to examine associations and agreement in the course dataset.

Dataset: `course_dataset.csv`

4.8.1 Step 1: Load and Prepare

```r
library(gtsummary)
library(psych)
course_df <- read.csv("data/course_dataset.csv")
```

4.8.2 Step 2: Cross-tabulation

```r
# Frequency table: response vs treatment
freq_tbl <- table(course_df$response, course_df$treatment)
print(freq_tbl)
```

```
#>
#>                 New Standard
#>   Non-Responder   4       45
#>   Responder      92       39
```

```
# Row percentages
prop.table(freq_tbl, margin = 1) |> round(3)
```

```
#>
#>                   New Standard
#>   Non-Responder 0.082    0.918
#>   Responder     0.702    0.298
```

4.8.3 Step 3: Chi-Square Test

```
chi_result <- chisq.test(freq_tbl)
cat("Chi-square statistic:", round(chi_result$statistic, 3), "
")
```

```
#> Chi-square statistic: 52.729
```

```
cat("Degrees of freedom:", chi_result$parameter, "
")
```

```
#> Degrees of freedom: 1
```

```
cat("p-value:", round(chi_result$p.value, 4), "
")
```

```
#> p-value: 0
```

```
# Check expected frequencies
chi_result$expected
```

```
#>
#>                       New Standard
#>   Non-Responder 26.13333 22.86667
#>   Responder     69.86667 61.13333
```

4.8.4 Step 4: Simulate Inter-Rater Agreement (Kappa)

```
# Simulate two raters classifying the same 80 patients
set.seed(42)
rater1 <- sample(c("Normal", "Abnormal"), 80, replace = TRUE, prob = c(0.6, 0.4))
rater2 <- ifelse(runif(80) < 0.85, rater1,
                 ifelse(rater1 == "Normal", "Abnormal", "Normal"))
agreement_tbl <- table(Rater1 = rater1, Rater2 = rater2)
psych::cohen.kappa(agreement_tbl)
```

	treatment		
	New	Standard	**Total**
response			
Non-Responder	4 (8.2%)	45 (92%)	49 (100%)
Responder	92 (70%)	39 (30%)	131 (100%)
Total	96 (53%)	84 (47%)	180 (100%)

```
#> Call: cohen.kappa1(x = x, w = w, n.obs = n.obs, alpha = alpha, levels = levels,
#>     w.exp = w.exp)
#>
#> Cohen Kappa and Weighted Kappa correlation coefficients and confidence boundaries
#>                  lower estimate upper
#> unweighted kappa  0.61     0.75  0.89
#> weighted kappa    0.61     0.75  0.89
#>
#>  Number of subjects = 80
```

4.8.5 Step 5: Publication Table

```
tbl_cross(course_df, row = response, col = treatment,
          percent = "row") |>
  bold_labels()
```

4.8.6 Lab Exercises

1. Is the association between `response` and `treatment` statistically significant? Write a 2-sentence conclusion.
2. The chi-square test assumes all expected frequencies ≥ 5. Check this for the `course_df` table. If violated, what alternative should you use?
3. Run Fisher's exact test on the same table. Does the conclusion change?

Chapter 5

Non-Parametric Tests and Applications Using R

By the end of this chapter, you will be able to:

- Select the correct non-parametric test given the data structure, number of groups, and pairing status
- Implement the Sign test and Wilcoxon Signed-Rank test for one-sample and paired data
- Apply the Mann-Whitney U test to compare two independent groups
- Conduct Kruskal-Wallis ANOVA and appropriate post-hoc tests for 3+ independent groups
- Run the Friedman test for repeated-measures or blocked designs
- Report results in proper scientific writing format (APA / journal style)

> **Why this chapter?** A clinical researcher studies pain reduction in 15 patients. Pain scores are on a 0–10 scale and are heavily skewed — the t-test's normality assumption is clearly violated. Non-parametric tests solve this: they work on *ranks* instead of raw values, require no distributional assumptions, and remain valid for small samples. This chapter gives you the complete toolkit for distribution-free inference.

- **Chapters 2 & 3**: Descriptive statistics and confidence intervals
- **Hypothesis testing**: p-values, null/alternative hypotheses, Type I and II errors
- **Chapter 3**: Shapiro-Wilk test for normality (introduced there)

This final chapter covers the classic non-parametric hypothesis tests — the rank-and-sign-based alternatives to the t-test and ANOVA family, used when normality assumptions don't hold or sample sizes are small.

This chapter uses `course_dataset` for most examples, `exam_pairs` (a small pre/post exam-score dataset) for the paired-test section, and the built-in datasets `PlantGrowth`, `ToothGrowth`, and `sleep` to show the same tests applied to completely different research questions.

```
course_df <- read.csv("data/course_dataset.csv")
exam_df <- read.csv("data/exam_pairs.csv")
```

5.1 Motivation, Advantages, and Limitations

Advantages: no distributional assumption required; robust to outliers and skew; valid for small samples; work directly on ranks or signs rather than raw values.

Limitations: generally less statistically powerful than the parametric equivalent *when the parametric assumptions genuinely hold*; hypotheses are framed around medians/distributions/ranks rather than means, which changes how you word your conclusions; fewer options exist for complex, multi-factor designs.

Non-Parametric Test Selection Matrix

Use this matrix to identify the correct test and R implementation based on your data structure:

Scenario / Design	Dependent Var Type	Independent Var / Groups	Parametric Equivalent	Non-Parametric Test	R Implementation
One Sample (Proportion)	Binary / Categorical	None (compared to constant)	One-sample z-test	**Binomial Test**	`binom.test(x, n, p)`
One Sample (Median)	Continuous / Ordinal	None (compared to constant)	One-sample t-test	**Sign Test**	*Custom function / binomial*
Two Independent Groups	Continuous / Ordinal	Binary (2 independent groups)	Independent t-test	**Mann-Whitney U / Wilcoxon Rank-Sum**	`wilcox.test(y ~ group)`
Two Paired Measurements	Continuous / Ordinal	Paired / Repeated (2 related measurements)	Paired t-test	**Wilcoxon Signed-Rank**	`wilcox.test(x, y, paired = TRUE)`
Three+ Independent Groups	Continuous / Ordinal	Multiclass (3+ independent groups)	One-way ANOVA	**Kruskal-Wallis**	`kruskal.test(y ~ group)`
Three+ Repeated Measures	Continuous / Ordinal	Multiclass (3+ related measurements)	Repeated measures ANOVA	**Friedman Test**	`friedman.test(y ~ time | id)`

Checking normality with the Shapiro-Wilk test is a common first step:

```
shapiro.test(course_df$baseline_pain)
```

```
#>
#>  Shapiro-Wilk normality test
#>
#> data:  course_df$baseline_pain
#> W = 0.96216, p-value = 8.788e-05
```

```
shapiro.test(course_df$week8_pain)
```

```
#>
#>  Shapiro-Wilk normality test
#>
#> data:  course_df$week8_pain
#> W = 0.95497, p-value = 1.665e-05
```

Both p-values are well below 0.05, meaning we reject the assumption of normality — non-parametric methods are the safer choice for this variable.

5.2 One-Sample Tests

5.2.1 Binomial test — for a proportion against a hypothesized value

```r
n_responders <- sum(course_df$response == "Responder")
n_total <- nrow(course_df)
binom.test(n_responders, n_total, p = 0.5)
```

```
#>
#>  Exact binomial test
#>
#> data:  n_responders and n_total
#> number of successes = 131, number of trials = 180, p-value = 8.018e-10
#> alternative hypothesis: true probability of success is not equal to 0.5
#> 95 percent confidence interval:
#>  0.6565617 0.7913269
#> sample estimates:
#> probability of success
#>              0.7277778
```

This tests whether the true response rate differs from a hypothesized 50%.

5.2.2 Sign test — for a median against a hypothesized value

The `BSDA` package provides `SIGN.test()` directly, but the sign test is simple enough to build from a single insight: **it's just a binomial test on the count of values above vs. below the hypothesized median.**

Check Before You Run This (One-Sample Sign Test Diagnostics) Before running a Sign Test:

1. **Ordinal or Continuous Variable**: The dependent variable should be measured on an ordinal or continuous scale.
2. **Independence**: Observations must be independent of one another.
3. **No Distributional Assumptions**: Unlike the Wilcoxon signed-rank test, the Sign Test does *not* assume that the distribution is symmetric around the median, making it the most robust test of all.

```r
manual_sign_test <- function(x, mu = 0) {
  diffs <- x - mu
  diffs <- diffs[diffs != 0]  # ties are dropped
  n_pos <- sum(diffs > 0)
  binom.test(n_pos, length(diffs), p = 0.5)
}

manual_sign_test(course_df$week8_pain, mu = 4)
```

```
#>
#>  Exact binomial test
#>
#> data:  n_pos and length(diffs)
#> number of successes = 63, number of trials = 179, p-value = 9.116e-05
#> alternative hypothesis: true probability of success is not equal to 0.5
#> 95 percent confidence interval:
#>  0.2821879 0.4267169
#> sample estimates:
#> probability of success
#>              0.3519553
```

The sign test only uses the *direction* (sign) of each deviation from the hypothesized value, throwing away information about magnitude. This makes it extremely robust — but also less powerful than a test that uses ranks, like the Wilcoxon signed-rank test coming up next.

How to Report This in a Paper: > "A one-sample sign test was conducted to determine if the median week-8 pain score differed from a clinical threshold of 4.0. The test indicated that the median pain score (Mdn = 3.1) was statistically significantly lower than 4.0, $p < .001$ ($n_{\text{above}} = 63, n_{\text{below}} = 116$)."

Second example — built-in `sleep` dataset:

```
data(sleep)
group1 <- sleep$extra[sleep$group == 1]
manual_sign_test(group1, mu = 0)
```

```
#>
#>  Exact binomial test
#>
#> data:  n_pos and length(diffs)
#> number of successes = 5, number of trials = 9, p-value = 1
#> alternative hypothesis: true probability of success is not equal to 0.5
#> 95 percent confidence interval:
#>  0.2120085 0.8630043
#> sample estimates:
#> probability of success
#>              0.5555556
```

Using `PlantGrowth$weight` (built into R), test whether the median plant weight differs from a hypothesized value of 5.0, using both `manual_sign_test()` and `binom.test()` directly — confirm you get identical results.

5.3 Two Independent Groups: Mann-Whitney U (Wilcoxon Rank-Sum)

The **Mann-Whitney U Test** (also known as the **Wilcoxon Rank-Sum Test**) compares the distributions of two independent groups. It is the non-parametric alternative to the independent two-sample t-test.

Check Before You Run This (Mann-Whitney U Diagnostics) Before running a Mann-Whitney U test, verify these assumptions:

1. **Ordinal or Continuous Dependent Variable**: The outcome variable must be ordinal or continuous.
2. **Independent Groups**: The independent variable must consist of two categorical, independent groups.
3. **Independence of Observations**: Observations within and between groups must be independent.

- **Null Hypothesis** (H_0): The distributions of the two groups are equal.
- **Alternative Hypothesis** (H_1): The distributions of the two groups are not equal.

5.3.1 How Rank-Sum Testing Works (The Logic of Ranks)

Instead of comparing means, this test ranks **all** observations from both groups combined from smallest to largest. Let's see how this works on a tiny dummy dataset:

Dummy Example: * **Group A**: `c(5, 12)` * **Group B**: `c(3, 8)`

1. **Combine and Sort**: The combined dataset sorted is `c(3, 5, 8, 12)`.
2. **Assign Ranks**:
   - Value `3` (Group B) → Rank **1**
   - Value `5` (Group A) → Rank **2**
   - Value `8` (Group B) → Rank **3**
   - Value `12` (Group A) → Rank **4**
3. **Sum Ranks for each group**:

- Sum for Group A (R_1): $2 + 4 = 6$
- Sum for Group B (R_2): $1 + 3 = 4$

If the rank sums are very different, it suggests one group systematically has larger values than the other.

5.3.2 Step-by-Step R Execution

Let's run this test on our `course_df` comparing week 8 pain levels between the Treatment and Control groups:

```r
# Run the test using formula notation: numeric_variable ~ grouping_variable
wilcox_result <- wilcox.test(week8_pain ~ treatment, data = course_df)
print(wilcox_result)
```

```
#>
#>  Wilcoxon rank sum test with continuity correction
#>
#> data:  week8_pain by treatment
#> W = 2201, p-value = 1.499e-07
#> alternative hypothesis: true location shift is not equal to 0
```

5.3.3 Understanding the Output

1. **`W` (Test Statistic)**: This is calculated from the sum of ranks of the first group.
2. **`p-value`**: Here, $p = 1.50 \times 10^{-7}$, which is extremely small and well below the 0.05 significance threshold. We reject the null hypothesis and conclude that week 8 pain levels differ significantly between the Treatment and Control groups.
3. **Warning ("cannot compute exact p-value with ties")**: R typically displays a warning message in the console when ties are present because it cannot compute the mathematically *exact* distribution (since multiple patients reported the *same* pain score, like two patients having a score of 3). R automatically splits ranks for tied values (giving them the average rank) and uses a normal approximation. This warning is expected and normal for datasets with integer or rounded scores.

The Math Behind It: Manual Mann-Whitney U Calculation The U statistic for the first group is calculated as:

$$U_1 = R_1 - \frac{n_1(n_1 + 1)}{2}$$

where R_1 is the sum of ranks for Group 1, and n_1 is its sample size.

For large samples, the distribution of U closely approximates a normal distribution. Under the null hypothesis, the mean (μ_U) and standard error (σ_U) of U are:

$$\mu_U = \frac{n_1 n_2}{2}, \quad \sigma_U = \sqrt{\frac{n_1 n_2 (n_1 + n_2 + 1)}{12}}$$

The test statistic is converted to a Z-score:

$$Z = \frac{U_1 - \mu_U}{\sigma_U}$$

Let's compute this manually in R to verify it matches R's output:

```r
# 1. Extract groups
g1 <- course_df$week8_pain[course_df$treatment == "New"]
g2 <- course_df$week8_pain[course_df$treatment == "Standard"]
n1 <- length(g1)
n2 <- length(g2)

# 2. Combine and rank
all_ranks <- rank(c(g1, g2))
r1 <- sum(all_ranks[1:n1])

# 3. Calculate U statistic (matches W exactly)
u1 <- r1 - n1 * (n1 + 1) / 2
u1
```

```
#> [1] 2201
```

```r
# 4. Z normal approximation
mu_u <- (n1 * n2) / 2
sigma_u <- sqrt((n1 * n2 * (n1 + n2 + 1)) / 12)
z_score <- (u1 - mu_u) / sigma_u
z_score
```

```
#> [1] -5.250062
```

```r
# 5. Two-tailed p-value
p_val <- 2 * pnorm(-abs(z_score))
p_val
```

```
#> [1] 1.520481e-07
```

Our manual U_1 matches R's W (2201) exactly! The manual p-value (1.52×10^{-7}) is extremely close to R's output (1.50×10^{-7}). The minor difference is because R's `wilcox.test()` adjusts the standard error (σ_U) to account for ties in the data and applies a continuity correction of 0.5 by default.

How to Report This in a Paper: > "A Mann-Whitney U test indicated that week 8 pain scores were statistically significantly lower in the New Treatment group ($Mdn = 2.4$, $n = 96$) compared to the Standard Treatment group ($Mdn = 4.3$, $n = 84$), $W = 2201$, $p < .001$. The effect size, calculated as rank-biserial correlation, was $r = 1 - \frac{2W}{n_1 n_2} = 1 - \frac{2(2201)}{96 \times 84} = 0.45$, representing a moderate-to-strong effect."

5.3.4 Second Example: Using `ToothGrowth` (Built-in)

Let's compare tooth length (`len`) by supplement type (`supp`: orange juice `OJ` vs ascorbic acid `VC`):

```r
data(ToothGrowth)
wilcox.test(len ~ supp, data = ToothGrowth)
```

```
#>
#>  Wilcoxon rank sum exact test
#>
#> data:  len by supp
#> W = 575.5, p-value = 0.06366
#> alternative hypothesis: true location shift is not equal to 0
```

5.3.5 Third Example: Subsetting Groups

If a dataset has 3 groups (e.g., control, treatment 1, treatment 2) and we only want to compare two, we must subset it first:

```r
# Subset control and treatment 2 from PlantGrowth
pg_subset <- subset(PlantGrowth, group %in% c("ctrl", "trt2"))

# Run the test
wilcox.test(weight ~ group, data = pg_subset)
```

```
#>
#>  Wilcoxon rank sum exact test
#>
#> data:  weight by group
#> W = 25, p-value = 0.06301
#> alternative hypothesis: true location shift is not equal to 0
```

Exercise: Mann-Whitney U

1. Load `course_dataset.csv`.
2. Subset the data to compare `week8_pain` between **Site A** and **Site B** only.
3. Run the Wilcoxon Rank-Sum test and state your statistical conclusion.

5.4 Paired Measurements: Wilcoxon Signed-Rank Test

The **Wilcoxon Signed-Rank Test** is used to compare two related samples, matched samples, or repeated measurements on a single sample to assess whether their population mean ranks differ. It is the non-parametric alternative to the paired-samples t-test.

Check Before You Run This (Wilcoxon Signed-Rank Diagnostics) Before running a Wilcoxon Signed-Rank test, verify these assumptions:

1. **Ordinal or Continuous Dependent Variable**: The paired variables must be ordinal or continuous.
2. **Paired/Matched Observations**: Each subject must have exactly two paired measurements (e.g., pre-test vs. post-test).
3. **Symmetric Differences**: The distribution of differences between the two paired groups should be approximately symmetric around the median of differences.

The Math Behind It: Wilcoxon Signed-Rank Test Let $D_i = X_{1,i} - X_{2,i}$ be the differences between the paired observations.

1. Exclude differences that are zero ($D_i = 0$).
2. Rank the absolute differences $|D_i|$ from smallest to largest, assigning average ranks to ties.
3. Sum the ranks of the positive differences (W^+) and the negative differences (W^-).
4. The test statistic W (in R, printed as V) is:

$$V = \sum_{D_i > 0} \text{Rank}(|D_i|)$$

   For large samples, the test statistic V is converted to a Z-score using the mean and standard error of the signed-rank distribution.

This section switches to **`exam_pairs`** — 25 students with a pre-test and post-test score — a cleaner example of genuinely paired data than repeating the pain-score columns again.

```r
head(exam_df, 4)
```

```
#>   student_id pre_test post_test
#> 1          1       54        68
#> 2          2       67        71
#> 3          3       62        78
#> 4          4       64        89
```

```r
wilcox_paired <- wilcox.test(exam_df$pre_test, exam_df$post_test, paired = TRUE)
print(wilcox_paired)
```

```
#>
#>  Wilcoxon signed rank exact test
#>
#> data:  exam_df$pre_test and exam_df$post_test
#> V = 8.5, p-value = 1.55e-06
#> alternative hypothesis: true location shift is not equal to 0
```

How to Report This in a Paper: > "A Wilcoxon signed-rank test was conducted to compare exam performance before and after the tutorial. The test indicated that student scores were statistically significantly higher after the tutorial ($Mdn = 71$) compared to before the tutorial ($Mdn = 61$), $V = 8.5$, $p < .001$, indicating a strong positive effect of the intervention."

A very common mistake: running the **unpaired** version of this test (`wilcox.test(pre_test, post_test)` without `paired = TRUE`) on data that's actually paired. This throws away the pairing information — the fact that the same student appears in both columns — and can give a misleading result. Always ask: are these the same subjects measured twice, or two different independent groups?

Second example — with the course dataset's baseline vs. week 8 pain:

```r
wilcox.test(course_df$baseline_pain, course_df$week8_pain, paired = TRUE)
```

```
#>
#>  Wilcoxon signed rank test with continuity correction
#>
#> data:  course_df$baseline_pain and course_df$week8_pain
#> V = 15820, p-value < 2.2e-16
#> alternative hypothesis: true location shift is not equal to 0
```

Using `exam_df`, also run the *unpaired* version of the same test by mistake (`wilcox.test(exam_df$pre_test, exam_df$post_test)`, no `paired = TRUE`). Compare the p-value to the correct paired version above — how different is the conclusion?

5.5 Three or More Independent Groups: Kruskal-Wallis

The **Kruskal-Wallis Test** is a rank-based non-parametric test that can be used to determine if there are statistically significant differences between three or more groups on an ordinal or continuous dependent variable. It is the non-parametric alternative to the one-way ANOVA.

Check Before You Run This (Kruskal-Wallis Diagnostics) Before running a Kruskal-Wallis test, verify these assumptions:

1. **Ordinal or Continuous Dependent Variable**: The outcome variable must be ordinal or continuous.
2. **Independent Groups**: The independent grouping variable must contain 3 or more categorical, independent groups.
3. **Independence of Observations**: Observations within and between groups must be independent.

The Math Behind It: Kruskal-Wallis H Test Like other rank tests, observations from all groups are combined and ranked. The test statistic H is computed as:

$$H = \frac{12}{N(N+1)} \sum_{j=1}^{k} \frac{R_j^2}{n_j} - 3(N+1)$$

where:

- k is the number of groups.
- n_j is the sample size of group j.
- N is the total sample size across all groups ($N = \sum n_j$).
- R_j is the sum of ranks for group j.

Under the null hypothesis, H follows a Chi-Square (χ^2) distribution with $k-1$ degrees of freedom. The p-value is computed in R using `pchisq(H, df = k - 1, lower.tail = FALSE)`.

```
krusq_result <- kruskal.test(week8_pain ~ site, data = course_df)
print(krusq_result)
```

```
#>
#>  Kruskal-Wallis rank sum test
#>
#> data:  week8_pain by site
#> Kruskal-Wallis chi-squared = 1.5744, df = 2, p-value = 0.4551
```

A significant result tells you *at least one* site differs from the others — but not which one(s). Follow up with pairwise comparisons and a multiple-comparison correction:

```
pairwise.wilcox.test(
  course_df$week8_pain,
  course_df$site,
  p.adjust.method = "bonferroni"
)
```

```
#>
#>  Pairwise comparisons using Wilcoxon rank sum test with continuity correction
#>
#> data:  course_df$week8_pain and course_df$site
#>
#>        Site A Site B
#> Site B 0.78   -
#> Site C 0.94   1.00
#>
#> P value adjustment method: bonferroni
```

Why the correction matters: if you ran many pairwise tests at the usual 0.05 threshold without any adjustment, you'd accumulate a much higher chance of at least one false positive purely by chance. The Bonferroni correction (dividing your significance threshold by the number of comparisons) is a simple, conservative way to control that risk.

How to Report This in a Paper: > "A Kruskal-Wallis test showed no statistically significant difference in week 8 pain levels across the three hospital sites, $H(2) = 1.57$, $p = .455$. The median pain scores were 3.40 for Site A, 2.85 for Site B, and 3.00 for Site C. Consistent with the overall test, post-hoc pairwise Wilcoxon rank-sum tests with Bonferroni correction showed no significant differences between any pair of sites (all adjusted $p \geq .78$)."

Second example — `PlantGrowth`, all three groups:

```r
kruskal.test(weight ~ group, data = PlantGrowth)
```

```
#>
#>  Kruskal-Wallis rank sum test
#>
#> data:  weight by group
#> Kruskal-Wallis chi-squared = 7.9882, df = 2, p-value = 0.01842
```

```r
pairwise.wilcox.test(
  PlantGrowth$weight,
  PlantGrowth$group,
  p.adjust.method = "bonferroni"
)
```

```
#>
#>  Pairwise comparisons using Wilcoxon rank sum exact test
#>
#> data:  PlantGrowth$weight and PlantGrowth$group
#>
#>      ctrl  trt1
#> trt1 0.590 -
#> trt2 0.189 0.027
#>
#> P value adjustment method: bonferroni
```

Run `kruskal.test()` on `course_df$baseline_pain` grouped by `activity_level` (Low/Moderate/High). Is there a significant difference? If so, which pairs differ?

5.6 Repeated Measures on the Same Subjects: The Friedman Test

The **Friedman Test** is the non-parametric version of the one-way repeated measures ANOVA. It is used when the same subjects are measured three or more times under different conditions (or at different time points).

Check Before You Run This (Friedman Test Diagnostics) Before running a Friedman test, verify these assumptions:

1. **Ordinal or Continuous Dependent Variable**: The outcome variable must be ordinal or continuous.
2. **Repeated Measures / Blocks**: One group of subjects is measured on 3 or more occasions (or matched blocks).
3. **No Missing Data**: The test requires a complete block design. If a subject is missing a score at any time point, R will throw an error or exclude the subject.

The Math Behind It: Friedman Q Test For each subject (or block), the k measurements are ranked from 1 to k. The test statistic Q is:

$$Q = \frac{12}{nk(k+1)} \sum_{j=1}^{k} R_j^2 - 3n(k+1)$$

where:

- n is the number of subjects (blocks).
- k is the number of repeated measurements (groups).
- R_j is the sum of ranks for treatment/time point j.

Under the null hypothesis, Q follows a Chi-Square (χ^2) distribution with $k-1$ degrees of freedom.

- **Null Hypothesis (H_0)**: The distributions (medians) across all time points are equal.
- **Alternative Hypothesis (H_1)**: At least one time point distribution differs from the others.

5.6.1 Step 1: Reshaping Wide Data to Long Format

In our `course_df` dataset, the pain scores for each patient are stored in three separate columns: `baseline_pain`, `week4_pain`, and `week8_pain` (**wide format**). To run the Friedman test, R requires the data to be in **long format** (where each observation is its own row, with a subject ID, a time point label, and the score).

Let's use **`tidyr::pivot_longer()`** (which you learned in Chapter 2!) to reshape the data:

```
library(tidyr)
library(dplyr)

# 1. Select the relevant columns to see the wide structure:
course_df %>%
  select(id, baseline_pain, week4_pain, week8_pain) %>%
  head(3)
```

```
#>     id baseline_pain week4_pain week8_pain
#> 1 P001           8.7        5.8        5.1
#> 2 P002           6.7        6.5        5.6
#> 3 P003           7.6        3.9        0.1
```

```
# 2. Pivot the pain columns into long format:
long_pain <- course_df %>%
  select(id, baseline_pain, week4_pain, week8_pain) %>%
  pivot_longer(
    cols = c(baseline_pain, week4_pain, week8_pain),
    names_to = "time",
    values_to = "pain"
  )

# 3. View the reshaped long structure:
head(long_pain, 6)
```

```
#> # A tibble: 6 x 3
#>   id    time           pain
#>   <chr> <chr>         <dbl>
#> 1 P001  baseline_pain   8.7
#> 2 P001  week4_pain      5.8
#> 3 P001  week8_pain      5.1
#> 4 P002  baseline_pain   6.7
#> 5 P002  week4_pain      6.5
#> 6 P002  week8_pain      5.6
```

5.6.2 Step 2: Running the Friedman Test

Now that the data is in long format, we can run `friedman.test()`:

```r
# Formula: value_variable ~ grouping_variable | subject_id
friedman_result <- friedman.test(pain ~ time | id, data = long_pain)
print(friedman_result)
```

```
#>
#>  Friedman rank sum test
#>
#> data:  pain and time and id
#> Friedman chi-squared = 264.98, df = 2, p-value < 2.2e-16
```

5.6.3 Understanding the Output

- **`Friedman chi-squared`**: The test statistic.
- **`p-value`**: Here, $p < 2.2 \times 10^{-16}$ (extremely small). We reject the null hypothesis and conclude that pain scores differ significantly across the three time points.

How to Report This in a Paper: > "A Friedman test indicated that pain levels differed statistically significantly across the three time points (baseline, week 4, week 8), $\chi^2(2) = 264.98$, $p < .001$ (Kendall's W effect size $= Q/(n(k-1)) = 0.74$, representing a moderate-to-strong effect). Post-hoc pairwise Wilcoxon signed-rank tests with Bonferroni correction showed a statistically significant reduction in pain scores at every step (all adjusted $p < .001$)."

5.6.4 Step 3: Post-Hoc Pairwise Comparisons

Just like the Kruskal-Wallis test, the Friedman test only tells us that *at least one* time point differs. To find out which specific time points are different, we perform pairwise Wilcoxon signed-rank tests with a Bonferroni correction:

```r
# Run pairwise paired Wilcoxon tests
pairwise_results <- pairwise.wilcox.test(
  x = long_pain$pain,
  g = long_pain$time,
  paired = TRUE,
  p.adjust.method = "bonferroni"
)
print(pairwise_results)
```

```
#>
#>  Pairwise comparisons using Wilcoxon signed rank test with continuity correction
#>
#> data:  long_pain$pain and long_pain$time
#>
#>            baseline_pain week4_pain
#> week4_pain <2e-16        -
#> week8_pain <2e-16        <2e-16
#>
#> P value adjustment method: bonferroni
```

5.6.5 Understanding Post-Hoc Output

The table lists adjusted p-values for each comparison:

- `week4_pain` vs `baseline_pain`: $p < 0.001$ (Significant)

- `week8_pain` vs `baseline_pain`: $p < 0.001$ (Significant)
- `week8_pain` vs `week4_pain`: $p < 0.001$ (Significant)

We conclude that pain scores decrease significantly at each subsequent follow-up interval.

The dedicated post-hoc test for Friedman is the **Nemenyi test**, available via `PMCMRplus::frdAllPairsNemenyiTest()`. The pairwise Wilcoxon approach shown above is a widely-used approximation when that package is not installed.

Exercise: Friedman Post-Hoc Interpretation Verify the pairwise output on your own console. Explain in your own words why we need a "Bonferroni adjustment" when running multiple pairwise tests.

5.7 Capstone: Selecting, Justifying, Running, and Reporting a Test

Let's work through a complete example end to end, exactly as you'll be asked to do in the lab.

Research question: "Is there a difference in `week4_pain` across the three `sites`?"

```
# STEP 1: Identify the design
# Three independent groups (Site A, B, C) ->
# candidates are one-way ANOVA or Kruskal-Wallis

# STEP 2: Check assumptions
tapply(course_df$week4_pain, course_df$site, shapiro.test)
```

```
#> $`Site A`
#>
#>  Shapiro-Wilk normality test
#>
#> data:  X[[i]]
#> W = 0.974, p-value = 0.2025
#>
#>
#> $`Site B`
#>
#>  Shapiro-Wilk normality test
#>
#> data:  X[[i]]
#> W = 0.97897, p-value = 0.254
#>
#>
#> $`Site C`
#>
#>  Shapiro-Wilk normality test
#>
#> data:  X[[i]]
#> W = 0.97421, p-value = 0.4369
```

```
# STEP 3: Select the test
# At least one group shows meaningful non-normality ->
# Kruskal-Wallis is the safer choice

# STEP 4: Run and extract results
kw_result <- kruskal.test(week4_pain ~ site, data = course_df)
kw_result
```

```
#>
#>  Kruskal-Wallis rank sum test
#>
#> data:  week4_pain by site
#> Kruskal-Wallis chi-squared = 1.3809, df = 2, p-value = 0.5014
```

```
# STEP 5: Post-hoc if needed
if (kw_result$p.value < 0.05) {
  print(pairwise.wilcox.test(
    course_df$week4_pain,
    course_df$site,
    p.adjust.method = "bonferroni"
  ))
}
```

```
# STEP 6: Report in plain language
cat(sprintf(
  paste0(
    "A Kruskal-Wallis test found %s evidence of alpha difference in ",
    "week-4 pain scores across the three sites (H = %.2f, p = %.3f).\n"
  ),
  ifelse(
    kw_result$p.value < 0.05,
    "statistically significant",
    "no statistically significant"
  ),
  kw_result$statistic,
  kw_result$p.value
))
```

```
#> A Kruskal-Wallis test found no statistically significant evidence of alpha difference in week-4 pa
```

Pick your own research question from `course_dataset` that this chapter hasn't already answered (for example: "Does `sex` relate to `response`?" — careful, that one might actually need Chapter 4's tools, not this chapter's! — or "Is there a difference in `age` between responders and non-responders?"). Work through all six capstone steps yourself, ending with a one-paragraph plain-language write-up.

- Non-parametric tests trade some statistical power for robustness to non-normality and outliers.
- One-sample: binomial test (proportions) or sign test (medians) — the sign test is just a binomial test on the count of values above/below the hypothesized value.
- Two independent groups: Mann-Whitney U / Wilcoxon rank-sum, via `wilcox.test(y ~ group)`.
- Two paired measurements: Wilcoxon signed-rank, via `wilcox.test(x, y, paired = TRUE)` — always double check pairing is correct.
- 3+ independent groups: Kruskal-Wallis, followed by pairwise Wilcoxon with a correction if significant.
- 3+ repeated measurements: Friedman test, using long-format data and a correctly specified `blocks` term.
- A consistent six-step workflow (design → assumptions → test selection → run → post-hoc → plain-language report) will serve you well beyond this course.

Non-Parametric Scenario Bank (Practice Exercises)

Practice identifying the correct statistical test for each of the following research scenarios using the decision matrix from Section 5.1.1. (Note: These exercises focus on test selection and do not require code).

Scenario 1

> A hospital administrator wants to compare the patient satisfaction ratings (measured on a 1-to-5 Likert scale) between Ward A ($n = 45$) and Ward B ($n = 50$). Satisfaction scores are highly

skewed. * **Answer:** Mann-Whitney U / Wilcoxon Rank-Sum test (comparing two independent groups on ordinal data).

Scenario 2

A clinical researcher measures the pain levels (on a 0-to-10 visual analog scale) of 30 chronic pain patients at three distinct time points: baseline, month 1, and month 3. The pain scores violate normality assumptions at all time points. * **Answer:** Friedman Test (comparing three related measurements on the same subjects over time).

Scenario 3

A psychologist wants to test if the median score on a cognitive test in a sample of elderly adults ($n = 18$) is significantly different from the national average median of 70. The sample data shows severe outliers. * **Answer:** Sign Test (one-sample median test with no symmetry assumptions).

Scenario 4

A geneticist wants to compare the expression levels of a specific gene across five different mouse strains (8 mice per strain). The expression levels are highly non-normal. * **Answer:** Kruskal-Wallis test (comparing 3+ independent groups on continuous, non-normal data).

Scenario 5

A diagnostic study compares the classifications ("Mild", "Moderate", "Severe") of 100 radiographs made by a resident doctor and an expert radiologist. The researcher wants to assess the level of agreement between the two. * **Answer:** Weighted Cohen's Kappa (measuring agreement between two raters on ordinal classifications).

5.8 Where to Go From Here

You've now covered the full arc of this course: from R fundamentals, through descriptive statistics and programming, to resampling-based confidence intervals, categorical data analysis, and non-parametric hypothesis testing. The Appendix that follows documents every dataset used throughout this book in one place, for quick reference.

Chapter 5 covered:

- **When to use non-parametric tests**: Non-normal data, small samples ($n < 20$), ordinal outcomes, or when robustness to outliers is needed
- **One-sample / paired tests**:
  - *Sign test*: Tests whether median equals a value; uses only the *sign* of differences → least powerful
  - *Wilcoxon Signed-Rank test*: Uses *ranks* of absolute differences → more powerful than Sign test
- **Two-sample tests**:
  - *Mann-Whitney U* (= Wilcoxon Rank-Sum): Compares medians / distributions of two independent groups
- **Multi-group tests**:
  - *Kruskal-Wallis*: Non-parametric one-way ANOVA; significant result → follow with Dunn's post-hoc test

– *Friedman test*: Non-parametric repeated-measures ANOVA; for blocked / matched designs

Design	Parametric	Non-Parametric	R Function
1 sample vs. constant	One-sample t	Sign / Wilcoxon SR	`wilcox.test(x, mu=)`
2 independent groups	Independent t	Mann-Whitney U	`wilcox.test(y ~ g)`
2 paired measurements	Paired t	Wilcoxon SR	`wilcox.test(x,y,paired=TRUE)`
3+ independent groups	One-way ANOVA	Kruskal-Wallis	`kruskal.test(y ~ g)`
3+ repeated measures	Repeated ANOVA	Friedman	`friedman.test(y ~ t | id)`

Congratulations — you have completed the Statistical Computing and Non-Parametric Inference Using R course! You now have a complete toolkit: R programming (Ch.1), descriptive analysis (Ch.2), confidence intervals and the bootstrap (Ch.3), categorical data analysis (Ch.4), and distribution-free hypothesis testing (Ch.5).

5.9 Lab 5: Full Non-Parametric Workflow on the Course Dataset

Objective: Apply the complete non-parametric testing workflow — from normality checking through to post-hoc analysis and reporting — on a real dataset.

Datasets: `course_dataset.csv` and `exam_pairs.csv`

5.9.1 Step 1: Load Data and Check Normality

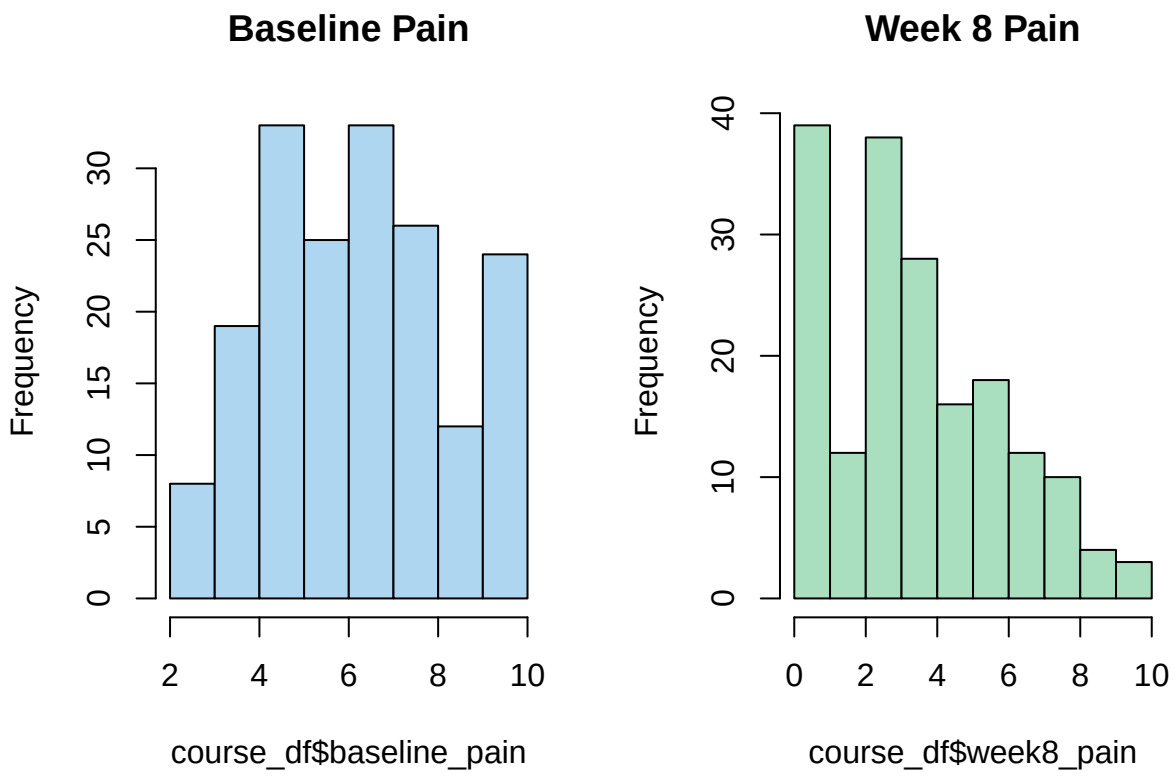
```
course_df <- read.csv("data/course_dataset.csv")
exam_df   <- read.csv("data/exam_pairs.csv")

# Shapiro-Wilk test for baseline_pain
shapiro.test(course_df$baseline_pain)
```

```
#>
#>  Shapiro-Wilk normality test
#>
#> data:  course_df$baseline_pain
#> W = 0.96216, p-value = 8.788e-05
```

```
shapiro.test(course_df$week8_pain)
```

```
#>
#>  Shapiro-Wilk normality test
#>
#> data:  course_df$week8_pain
#> W = 0.95497, p-value = 1.665e-05
```

```
# Visual check
par(mfrow = c(1, 2))
hist(course_df$baseline_pain, main = "Baseline Pain", col = "#AED6F1")
hist(course_df$week8_pain,    main = "Week 8 Pain",   col = "#A9DFBF")
```




```
par(mfrow = c(1, 1))
```

5.9.2 Step 2: Paired Analysis (Wilcoxon Signed-Rank)

```
# Did pain decrease from baseline to week 8?
wilcox.test(course_df$baseline_pain, course_df$week8_pain,
            paired = TRUE, alternative = "greater")
```

```
#>
#>  Wilcoxon signed rank test with continuity correction
#>
#> data:  course_df$baseline_pain and course_df$week8_pain
#> V = 15820, p-value < 2.2e-16
#> alternative hypothesis: true location shift is greater than 0
```

Interpret: Report W statistic, p-value, and whether you reject H_0.

5.9.3 Step 3: Two-Group Comparison (Mann-Whitney)

```
# Does baseline pain differ between treatment groups?
wilcox.test(baseline_pain ~ treatment, data = course_df)
```

```
#>
#>  Wilcoxon rank sum test with continuity correction
#>
#> data:  baseline_pain by treatment
#> W = 3863, p-value = 0.6288
#> alternative hypothesis: true location shift is not equal to 0
```

5.9.4 Step 4: Kruskal-Wallis for 3+ Groups

```r
# Compare baseline pain across response categories
kruskal.test(baseline_pain ~ response, data = course_df)
```

```
#>
#>  Kruskal-Wallis rank sum test
#>
#> data:  baseline_pain by response
#> Kruskal-Wallis chi-squared = 15.065, df = 1, p-value = 0.0001039
```

```r
# Post-hoc (if significant)
if (kruskal.test(baseline_pain ~ response, data = course_df)$p.value < 0.05) {
  library(PMCMRplus)
  kwAllPairsDunnTest(baseline_pain ~ as.factor(response), data = course_df,
                     p.adjust.method = "bonferroni")
}
```

```
#>           Non-Responder
#> Responder 1e-04
```

5.9.5 Step 5: Write a Results Paragraph

Using the results above, write a 3–4 sentence results section in APA format, as you would in a journal paper. Include: the test used, the test statistic, degrees of freedom (if applicable), p-value, and a plain-English conclusion.

5.9.6 Lab Exercises

1. Run the Sign test (manual binomial approach) on the paired pain data. Does the conclusion match the Wilcoxon Signed-Rank result? Why might they differ?
2. Use the `exam_pairs` dataset: run a Wilcoxon Signed-Rank test to check if exam scores improved from pre-test to post-test.
3. Compute the effect size $r = Z/\sqrt{N}$ for the Mann-Whitney test in Step 3. Is the effect small, medium, or large (by Cohen's conventions)?

Dataset Reference

This appendix documents every dataset used across all five chapters, so you can look one up whenever you forget its structure. It also includes the script that regenerates every custom dataset from scratch, in case your copies get lost or modified.

.1 Custom Course Datasets

`student_scores`

Used in: Chapters 1–2. A small gradebook-style dataset.

```
scores <- read.csv("data/student_scores.csv")
str(scores)
```

```
#> 'data.frame':    40 obs. of  5 variables:
#>  $ id           : int  1 2 3 4 5 6 7 8 9 10 ...
#>  $ name         : chr  "Student_01" "Student_02" "Student_03" "Student_04" ...
#>  $ group        : chr  "A" "A" "A" "B" ...
#>  $ score        : int  63 66 49 89 69 95 92 69 77 92 ...
#>  $ hours_studied: num  4.7 5.3 4.6 4.4 1.5 4.5 1.4 7.3 3 5.3 ...
```

```
head(scores, 3)
```

```
#>   id       name group score hours_studied
#> 1  1 Student_01     A    63           4.7
#> 2  2 Student_02     A    66           5.3
#> 3  3 Student_03     A    49           4.6
```

Column	Meaning
`id`	Student identifier
`name`	Student name
`group`	Class section (A or B)
`score`	Exam score, 0–100
`hours_studied`	Self-reported study hours

Also provided as `data/student_scores.xlsx` and `data/student_scores.txt` for import practice.

`course_dataset`

Used in: Chapters 3–5. The "running dataset" for the whole second half of the course — a simulated clinical trial.

```r
course_df <- read.csv("data/course_dataset.csv")
str(course_df)
```

```
#> 'data.frame':    180 obs. of  14 variables:
#>  $ id             : chr  "P001" "P002" "P003" "P004" ...
#>  $ site           : chr  "Site C" "Site B" "Site A" "Site C" ...
#>  $ treatment      : chr  "New" "Standard" "New" "Standard" ...
#>  $ age            : int  59 68 33 53 58 49 42 40 67 49 ...
#>  $ sex            : chr  "Male" "Male" "Male" "Male" ...
#>  $ activity_level : chr  "Moderate" "High" "Moderate" "Low" ...
#>  $ baseline_pain  : num  8.7 6.7 7.6 5.5 6 4.1 6.8 6 5.8 6.6 ...
#>  $ week4_pain     : num  5.8 6.5 3.9 2.7 5.9 4.6 6.6 2.5 4.4 2.9 ...
#>  $ week8_pain     : num  5.1 5.6 0.1 2.4 5.1 4.6 4.2 2.4 3.5 1.3 ...
#>  $ response       : chr  "Responder" "Non-Responder" "Responder" "Responder" ...
#>  $ symptom_pre    : chr  "Present" "Absent" "Present" "Present" ...
#>  $ symptom_post   : chr  "Absent" "Present" "Present" "Absent" ...
#>  $ rater1_severity: chr  "Severe" "Mild" "Moderate" "Severe" ...
#>  $ rater2_severity: chr  "Severe" "Mild" "Moderate" "Severe" ...
```

```r
head(course_df, 3)
```

```
#>     id   site treatment age  sex activity_level baseline_pain week4_pain
#> 1 P001 Site C       New  59 Male       Moderate           8.7        5.8
#> 2 P002 Site B  Standard  68 Male           High           6.7        6.5
#> 3 P003 Site A       New  33 Male       Moderate           7.6        3.9
#>   week8_pain      response symptom_pre symptom_post rater1_severity
#> 1        5.1     Responder     Present       Absent          Severe
#> 2        5.6 Non-Responder      Absent      Present            Mild
#> 3        0.1     Responder     Present      Present        Moderate
#>   rater2_severity
#> 1          Severe
#> 2            Mild
#> 3        Moderate
```

Column	Meaning
`id`	Patient identifier
`site`	Hospital site (A/B/C) — the stratifying variable for CMH
`treatment`	Standard or New
`age`, `sex`	Demographics
`activity_level`	Ordinal: Low/Moderate/High — used for trend tests
`baseline_pain`, `week4_pain`, `week8_pain`	Pain scores (0–10) at three time points — used for bootstrap and Friedman examples
`response`	Responder/Non-Responder — the binary outcome
`symptom_pre`, `symptom_post`	Paired binary symptom status — used for McNemar
`rater1_severity`, `rater2_severity`	Two raters' ordinal severity grades — used for kappa

`survey_income`

Used in: Chapter 3, to illustrate a genuinely skewed variable.

```
income <- read.csv("data/survey_income.csv")
str(income)
```

```
#> 'data.frame':    100 obs. of  3 variables:
#>  $ household_id  : int  1 2 3 4 5 6 7 8 9 10 ...
#>  $ region        : chr  "Urban" "Urban" "Rural" "Rural" ...
#>  $ monthly_income: int  17581 5260 24417 22408 20646 19483 11935 24541 6938 30536 ...
```

```
head(income, 3)
```

```
#>   household_id region monthly_income
#> 1            1  Urban          17581
#> 2            2  Urban           5260
#> 3            3  Rural          24417
```

exam_pairs

Used in: Chapter 5, for the paired Wilcoxon signed-rank example.

```
exam_df <- read.csv("data/exam_pairs.csv")
str(exam_df)
```

```
#> 'data.frame':    25 obs. of  3 variables:
#>  $ student_id: int  1 2 3 4 5 6 7 8 9 10 ...
#>  $ pre_test  : int  54 67 62 64 46 62 61 63 72 59 ...
#>  $ post_test : int  68 71 78 89 62 56 72 67 86 73 ...
```

```
head(exam_df, 3)
```

```
#>   student_id pre_test post_test
#> 1          1       54        68
#> 2          2       67        71
#> 3          3       62        78
```

.2 Built-In R Datasets Used in This Book

These come with base R (or the `datasets` package, which is always loaded) — you never need to download or import them.

Dataset	Used in	What it contains
`mtcars`	Ch. 1, 3	32 cars: mpg, horsepower, weight, and other specs
`iris`	Ch. 1	150 flowers: petal/sepal measurements across 3 species
`PlantGrowth`	Ch. 1, 5	Plant weights under control vs. two treatments
`airquality`	Ch. 2	Daily air quality measurements, New York, 1973
`chickwts`	Ch. 2	Chick weights under six different feed types
`InsectSprays`	Ch. 4	Insect counts under six different spray types

Dataset	Used in	What it contains
`ToothGrowth`	Ch. 5	Guinea pig tooth growth under two supplement types
`sleep`	Ch. 5	Extra sleep hours under two drugs

Load any of these with `data(name)` (though most are available immediately without even calling `data()`).

.3 Regenerating All Custom Datasets

If your copies of the custom datasets are ever lost, corrupted, or you want to verify reproducibility, this script recreates all of them exactly (it uses a fixed random seed):

```r
set.seed(2026)

## student_scores.csv
n1 <- 40
student_scores <- data.frame(
  id = 1:n1,
  name = paste0("Student_", sprintf("%02d", 1:n1)),
  group = sample(c("A", "B"), n1, replace = TRUE),
  score = round(pmin(100, pmax(0, rnorm(n1, mean = 68, sd = 14)))),
  hours_studied = round(pmax(0, rnorm(n1, mean = 5, sd = 2)), 1)
)
write.csv(student_scores, "data/student_scores.csv", row.names = FALSE)

## course_dataset.csv
n2 <- 180
site <- sample(
  c("Site A", "Site B", "Site C"),
  n2, replace = TRUE, prob = c(.4, .35, .25)
)
treatment <- sample(c("Standard", "New"), n2, replace = TRUE)
age <- pmin(pmax(round(rnorm(n2, 52, 12)), 21), 85)
sex <- sample(c("Male", "Female"), n2, replace = TRUE, prob = c(.45, .55))
baseline_pain <- round(
  pmin(10, pmax(0, rgamma(n2, shape = 6, rate = .9))), 1
)
effect <- ifelse(treatment == "New", 2.6, 1.4)
week4_pain <- round(
  pmin(10, pmax(0, baseline_pain - rnorm(n2, effect, 1.1))), 1
)
week8_pain <- round(
  pmin(10, pmax(0, week4_pain - rnorm(n2, effect * .5, 1.0))), 1
)
response <- ifelse(
  (baseline_pain - week8_pain) / pmax(baseline_pain, .5) >= .3,
  "Responder", "Non-Responder"
)
symptom_pre <- sample(
  c("Present", "Absent"), n2, replace = TRUE, prob = c(.75, .25)
)
symptom_post <- ifelse(
  response == "Responder",
  sample(c("Present", "Absent"), n2, replace = TRUE, prob = c(.25, .75)),
  sample(c("Present", "Absent"), n2, replace = TRUE, prob = c(.65, .35))
)
```

```
sev <- c("Mild", "Moderate", "Severe")
rater1 <- sample(sev, n2, replace = TRUE, prob = c(.3, .45, .25))
rater2 <- ifelse(
  runif(n2) < .75, rater1, sample(sev, n2, replace = TRUE)
)
activity_level <- sample(
  c("Low", "Moderate", "High"), n2, replace = TRUE, prob = c(.35, .4, .25)
)
course_dataset <- data.frame(
  id = sprintf("P%03d", 1:n2), site, treatment, age, sex,
  activity_level = factor(
    activity_level, levels = c("Low", "Moderate", "High"), ordered = TRUE
  ),
  baseline_pain, week4_pain, week8_pain, response, symptom_pre, symptom_post,
  rater1_severity = factor(rater1, levels = sev, ordered = TRUE),
  rater2_severity = factor(rater2, levels = sev, ordered = TRUE)
)
write.csv(course_dataset, "data/course_dataset.csv", row.names = FALSE)

## survey_income.csv
n3 <- 100
survey_income <- data.frame(
  household_id = 1:n3,
  region = sample(c("Urban","Rural"), n3, replace = TRUE),
  monthly_income = round(rlnorm(n3, meanlog = 9.5, sdlog = .55))
)
write.csv(survey_income, "data/survey_income.csv", row.names = FALSE)

## exam_pairs.csv
n4 <- 25
pre <- round(rnorm(n4, 60, 10))
post <- round(pre + rnorm(n4, 8, 6))
exam_pairs <- data.frame(student_id = 1:n4,
                         pre_test = pmin(100, pmax(0, pre)),
                         post_test = pmin(100, pmax(0, post)))
write.csv(exam_pairs, "data/exam_pairs.csv", row.names = FALSE)
```

.4 Package Installation Checklist

```
install.packages(c(
  "boot", "psych", "coin", "readxl", "tidyr", "dplyr", "ggplot2",
  "tableone", "gtsummary", "BSDA", "PMCMRplus", "writexl"
))
```

Every package marked `eval=FALSE` in code chunks throughout this book (mainly `tableone`, `gtsummary`, `BSDA`, and `PMCMRplus`) was not available in the environment this book was built in, but the code shown is exactly what you should run once you've installed them on your own machine.

Appendix A

Bonus Chapter: Accessing R Source Code, Package Internals, and Repositories

One of R's greatest strengths is that it is completely open source. This means you do not have to treat R functions, packages, or interactive widgets as "black boxes." Every line of R code written by other developers—including R's core team—is available for you to inspect, study, and modify.

This bonus chapter explains how to access, extract, and inspect R source code from within R itself, from CRAN/GitHub repositories, and from web portals like `rdrr.io`.

A.1 Accessing Source Code Directly within R

A.1.1

1. Inspecting Standard R Functions {-}

The easiest way to view the source code of any R function is to type the function's name in the Console **without parentheses** and press Enter:

```
# Define the function in this session
skewness_manual <- function(x) {
  n <- length(x)
  m <- mean(x)
  (sum((x - m)^3) / n) / (sum((x - m)^2) / n)^(3/2)
}

# View the source code by printing its name without parentheses
skewness_manual
```

```
## function (x)
## {
##     n <- length(x)
##     m <- mean(x)
##     (sum((x - m)^3)/n)/(sum((x - m)^2)/n)^(3/2)
## }
```

For package functions, simply load the package first or use the namespace operator `:::`

```r
library(psych)
# View the source code of the skew function from the psych package
psych::skew
```

```
## function (x, na.rm = TRUE, type = 3)
## {
##     if (length(dim(x)) == 0) {
##         if (na.rm) {
##             x <- x[!is.na(x)]
##         }
##         sdx <- sd(x, na.rm = na.rm)
##         mx <- mean(x)
##         n <- length(x[!is.na(x)])
##         switch(type, {
##             skewer <- sqrt(n) * (sum((x - mx)^3, na.rm = na.rm)/(sum((x -
##                 mx)^2, na.rm = na.rm)^(3/2)))
##         }, {
##             skewer <- n * sqrt(n - 1) * (sum((x - mx)^3, na.rm = na.rm)/((n -
##                 2) * sum((x - mx)^2, na.rm = na.rm)^(3/2)))
##         }, {
##             skewer <- sum((x - mx)^3)/(n * sd(x)^3)
##         })
##     }
##     else {
##         skewer <- rep(NA, dim(x)[2])
##         if (is.matrix(x)) {
##             mx <- colMeans(x, na.rm = na.rm)
##         }
##         else {
##             mx <- apply(x, 2, mean, na.rm = na.rm)
##         }
##         sdx <- apply(x, 2, sd, na.rm = na.rm)
##         for (i in 1:dim(x)[2]) {
##             n <- length(x[!is.na(x[, i]), i])
##             switch(type, {
##                 skewer[i] <- sqrt(n) * (sum((x[, i] - mx[i])^3,
##                   na.rm = na.rm)/(sum((x[, i] - mx[i])^2, na.rm = na.rm)^(3/2)))
##             }, {
##                 skewer[i] <- n * sqrt(n - 1) * (sum((x[, i] -
##                   mx[i])^3, na.rm = na.rm)/((n - 2) * sum((x[,
##                   i] - mx[i])^2, na.rm = na.rm)^(3/2)))
##             }, {
##                 skewer[i] <- sum((x[, i] - mx[i])^3, na.rm = na.rm)/(n *
##                   sdx[i]^3)
##             })
##         }
##     }
##     return(skewer)
## }
## <bytecode: 0x5606c81a9390>
## <environment: namespace:psych>
```

A.1.2

2. Accessing Unexported (Hidden) Functions {-}

Many packages contain private helper functions that are not exported to the user workspace. If you try

to view them using `package::function_name`, R will throw an error. To view these unexported functions, use the **triple colon (`:::`)** operator or the **`getAnywhere()`** function:

```
# Attempting this will fail as it is unexported
psych::skew.default

# This works! Accessing the unexported S3 method directly
psych:::skew.default

# Alternatively, search all loaded namespaces for the function
getAnywhere("skew.default")
```

A.1.3

3. Primitive and Internal C/Fortran Code {-}

Some core R functions operate at a very low level for performance. When you print their source code, you will see `.Primitive()` or `.Internal()`:

```
# Print the sum function
sum
```

```
## function (..., na.rm = FALSE)  .Primitive("sum")
```

These functions call compiled C or Fortran code directly. To inspect this code, you must view the official R source code repository (e.g., in the `src/main/` directory of the R source code).

A.2 Inspecting Code Online via rdrr.io

If you do not have R open or want a convenient, searchable web-based interface to read R code, the **`rdrr.io`** portal is an invaluable tool.

- **What it is**: `rdrr.io` is an online package index that hosts the documentation, source code, and dependency maps for all CRAN, Bioconductor, and popular GitHub R packages.
- **How to use it**:
  1. Go to https://rdrr.io/ in your browser.
  2. Search for any package (e.g., `psych`) or specific function (e.g., `cohen.kappa`).
  3. Under the package page, click on **"Source code"** to browse the package's internal file structure.
  4. Navigate to the `R/` directory. All R functions are defined here. For example, you can read the raw R source for the `skew()` function at rdrr.io/cran/psych/src/R/skew.R.

A.3 Accessing Code from Repositories (CRAN & GitHub)

A.3.1

1. Downloading Package Source Tarballs from CRAN {-}

CRAN stores every R package in two formats: compiled binary files (for users to install) and raw source archives (called **tarballs**, ending in `.tar.gz`). To inspect a package's full structure, including non-R assets like C++ code or raw datasets:

1. Go to the package's CRAN page (e.g., `https://cran.r-project.org/package=psych`).
2. Download the **Package source** file (e.g., `psych_2.4.4.tar.gz`).
3. Extract the archive using an archiving utility (like 7-Zip on Windows) or from R itself:

```r
# Download and unpack a package source archive manually
download.file("https://cran.r-project.org/src/contrib/psych_2.4.4.tar.gz",
 ↪  "psych.tar.gz")
untar("psych.tar.gz", exdir = "psych_source")
```

4. Open the extracted directory. You will see a standard R package structure:
   - **`R/`**: Contains all R code files.
   - **`src/`**: Contains compiled C, C++, or Fortran code files (if the package uses them).
   - **`man/`**: Contains the documentation files (written in `.Rd` format).
   - **`inst/`**: Contains extra installed assets, templates, and raw resource files.

A.3.2

2. Exploring Code on GitHub {-}

Most modern R packages are developed openly on GitHub before being published on CRAN. GitHub provides code versioning history, allowing you to see exactly *how* a function's code has changed over time.

- To find a package's development code, search GitHub or look for the "URL" field on the package's CRAN page.
- You can install the bleeding-edge developer version of any GitHub-hosted R package using the `remotes` package:

```r
# Install directly from a GitHub repository
remotes::install_github("personality-project/psych")
```

A.4 Accessing HTML Widgets and Assets

R Markdown and Shiny support interactive components called **HTML Widgets** (powered by the `htmlwidgets` package). These widgets (such as interactive maps in `leaflet` or datatables in `DT`) bridge R and JavaScript.

To inspect how an HTML widget works:

1. Locate where R stores installed packages on your local system using `find.package()`:

```r
# Locate the installation path of the DT package
find.package("DT")
```

2. Navigate to the `htmlwidgets/` subfolder inside that path: `system.file("htmlwidgets", package = "DT")`
3. Inside this directory, you will find:
   - The R binding script (e.g., `datatable.R` which defines the R parameters).
   - The JavaScript binding script (e.g., `datatables.js` which initializes the JS library).
   - The `.yaml` configuration file mapping R arguments to JavaScript assets.

Appendix B

Appendix C: Glossary of Terms

This appendix provides precise definitions of the key statistical computing and inferential concepts covered throughout this book.

Type I Error (Alpha)

Symbol: α The probability of rejecting the null hypothesis when it is actually true (a false positive). For example, concluding that a new treatment is effective when in reality it does not differ from the standard treatment. The threshold is typically set at 0.05.

Type II Error (Beta)

Symbol: β The probability of failing to reject the null hypothesis when it is actually false (a false negative). For example, concluding that a new treatment has no effect when it actually does. The probability of avoiding a Type II error $(1 - \beta)$ is known as the statistical power of a test.

Degrees of Freedom (df)

The number of independent pieces of information that go into estimating a parameter or calculating a test statistic. In categorical tables, it is computed as $(r - 1)(c - 1)$. In a one-sample t-test, it is $n - 1$.

Effect Size

A quantitative measure of the magnitude of a statistical phenomenon or relationship, independent of the sample size. Examples include the Phi coefficient (ϕ) and Cramér's V for chi-square tests, and the rank-biserial correlation r for Wilcoxon tests. Unlike p-values, which depend heavily on sample size, effect sizes indicate the practical significance of a result.

Discordant Pairs

In paired categorical data (e.g., analyzed using McNemar's test), discordant pairs are subjects who changed their status between two measurements (e.g., Yes $\rightarrow$ No, or No $\rightarrow$ Yes). In contrast, concordant pairs are those whose status remained unchanged.

Bias-Corrected and Accelerated (BCa) Bootstrap

An advanced resampling method for constructing confidence intervals that adjusts for both bias (asymmetry between the median of the bootstrap estimates and the original sample estimate) and acceleration (skewness in the sampling distribution). It is more computationally intensive than percentile bootstrap intervals but achieves superior coverage accuracy for skewed data.

Lexical Scoping

The static scoping rules in R that determine how R searches for variables. Lexical scoping means that R looks up free variables in the environment where the function was *defined*, rather than the environment from which the function is *called*.

Parametric vs. Non-Parametric

- **Parametric tests** (like t-tests and ANOVA) assume that the data follows a specific probability distribution (usually normal) and estimate parameters such as means and variances.
- **Non-parametric tests** (like Wilcoxon and Kruskal-Wallis) make no assumptions about the underlying distribution, operating instead on ranks, signs, or medians. They are robust to outliers and skew but generally have lower statistical power when parametric assumptions are met.

Odds Ratio (OR)

A measure of association between an exposure and an outcome. The odds ratio represents the odds that an outcome will occur given a particular exposure, compared to the odds of the outcome occurring in the absence of that exposure. An OR = 1 indicates no association.

Cohen's Kappa

Symbol: κ A statistic that measures inter-rater agreement for qualitative (categorical) items, correcting for the agreement that would be expected by chance alone. Unweighted kappa treats all disagreements equally, while weighted kappa (linear or quadratic) penalizes disagreements based on their ordinal distance.

Yates' Continuity Correction

An adjustment made to the chi-square formula in 2×2 tables by subtracting 0.5 from the absolute difference between observed and expected counts. It prevents overestimating significance (inflating Type I error) in small samples.

Quantile

Points partition the range of a probability distribution into contiguous intervals with equal probabilities. The p-quantile is the value x such that the probability of the variable being less than or equal to x is exactly p (e.g., the 95th percentile).

Cumulative Probability

The probability that a random variable X takes a value less than or equal to x, calculated as $P(X \leq x)$. In R, cumulative probabilities are computed using the `p...()` functions.

Vectorization

A programming paradigm in R where operations are applied to entire vectors or matrices at once rather than looping through individual elements. Vectorized code leverages low-level compiled implementations (C/Fortran) and is significantly faster than standard R loops.

Appendix C

Appendix D: R and RStudio Cheatsheets

To assist you with your class labs, examinations, and future research work, three high-quality reference cheatsheets have been bundled with this book. You can access and download them directly from the links below:

C.0.1

1. Base R Cheat Sheet {-}

- **Description**: A comprehensive two-page reference for core R commands, covering vector creation, subsetting, programming loops, math functions, matrix/list/data frame manipulation, basic statistical distributions, and plotting.
- **Access Link**: Download Base R Cheat Sheet (PDF)

C.0.2

2. Advanced R Cheat Sheet {-}

- **Description**: A detailed guide to advanced R concepts, including environment structures, lexical scoping rules, S3/S4 object-oriented programming systems, function environments, subsetting rules, and debugging/defensive programming methods.
- **Access Link**: Download Advanced R Cheat Sheet (PDF)

C.0.3

3. RStudio IDE Cheat Sheet {-}

- **Description**: A visual guide to the RStudio Integrated Development Environment, covering code editor shortcuts, package management, debugger tool execution, git version control integration, and shiny application development resources.
- **Access Link**: Download RStudio IDE Cheat Sheet (PDF)

Appendix D

Appendix E: Solutions to End-of-Chapter Labs

This appendix provides canonical R code solutions for all five End-of-Chapter Labs across the course. Use these solutions to verify your code logic after completing each chapter lab.

D.1 Lab 1 Solutions: R Basics and Data Structures

```r
# 1. Create vectors
student_ids <- 101:105
exam_scores <- c(78.5, 92.0, 84.5, 65.0, 90.0)
pass_status <- factor(c("Pass", "Pass", "Pass", "Fail", "Pass"),
                      levels = c("Fail", "Pass"), ordered = TRUE)

# 2. Build data frame
grades_df <- data.frame(
  ID = student_ids,
  Score = exam_scores,
  Status = pass_status
)

# 3. Calculate mean of passing students
mean_passing <- mean(grades_df$Score[grades_df$Status == "Pass"])
print(paste("Mean passing score:", round(mean_passing, 2)))
```

D.2 Lab 2 Solutions: Descriptive Statistics and Data Summaries

```r
# Load dataset
dataset_path <- system.file("extdata", "student_scores.csv", package = "scnpir")
if (dataset_path == "") dataset_path <- "data/student_scores.csv"
scores_df <- read.csv(dataset_path)

# Calculate summary statistics by group
```

```r
aggregate(score ~ group, data = scores_df, FUN = function(x) {
  c(Mean = mean(x), SD = sd(x), Median = median(x), IQR = IQR(x))
})
```

D.3 Lab 3 Solutions: Bootstrap Resampling and Confidence Intervals

```r
library(boot)

# Load dataset
income_path <- system.file("extdata", "survey_income.csv", package = "scnpir")
if (income_path == "") income_path <- "data/survey_income.csv"
income_df <- read.csv(income_path)

# Define statistic function for bootstrap median
stat_median <- function(data, indices) {
  median(data[indices, "monthly_income"])
}

# Run 1000 bootstrap iterations
set.seed(42)
boot_results <- boot(data = income_df, statistic = stat_median, R = 1000)

# Compute 95% Percentile Confidence Interval
boot.ci(boot_results, type = c("perc", "basic"))
```

D.4 Lab 4 Solutions: Analysis of Contingency Tables

```r
# Create 2x2 matrix
trial_data <- matrix(c(45, 15, 20, 40), nrow = 2, byrow = TRUE,
                     dimnames = list(Treatment = c("Drug_A", "Placebo"),
                                     Outcome = c("Recovered", "Not_Recovered")))

# Add margins
addmargins(trial_data)

# Chi-Square test with Yates' continuity correction
chi_res <- chisq.test(trial_data)
print(chi_res)

# Fisher's Exact Test & Odds Ratio
fisher_res <- fisher.test(trial_data)
print(fisher_res)
```

D.5 Lab 5 Solutions: Non-Parametric Hypothesis Testing

```r
# Load dataset
course_path <- system.file("extdata", "course_dataset.csv", package = "scnpir")
if (course_path == "") course_path <- "data/course_dataset.csv"
course_df <- read.csv(course_path)

# Mann-Whitney U test between New and Standard treatment formats
mw_res <- wilcox.test(baseline_pain ~ treatment, data = course_df, exact = FALSE)
print(mw_res)

# Kruskal-Wallis test across 3 study sites
kw_res <- kruskal.test(baseline_pain ~ site, data = course_df)
print(kw_res)
```