

R Programming from Beginning to Expert

Published by Randy Fadler

July 2025

Contents

[Executive Summary: Mastering R: From Fundamentals to Real-World Solutions 5](#_Toc203393917)

[Chapter 1: Welcome to R 7](#_Toc203393918)

[1.1 What is R and why it's essential today? 7](#_Toc203393919)

[1.2 Installing R and RStudio 8](#_Toc203393920)

[1.3 Navigating the RStudio interface 8](#_Toc203393921)

[1.4 Overview of CRAN, packages, and repositories 9](#_Toc203393922)

[1.5 Writing your first line of R code 10](#_Toc203393923)

[Chapter 2: Language Fundamentals 11](#_Toc203393924)

[2.1 Variables, assignment operators, and comments 11](#_Toc203393925)

[2.2 Data types: numeric, character, logical 12](#_Toc203393926)

[2.3 Type coercion and inspection 14](#_Toc203393927)

[2.4 Arithmetic and logical operations 15](#_Toc203393928)

[2.5 Running Example Introduction: “Customer Feedback Analyzer” 19](#_Toc203393929)

[2.6 Wrap-up 20](#_Toc203393930)

[Chapter 3: Control Flow and Functions 21](#_Toc203393931)

[3.1 Conditional statements: if, else, switch 21](#_Toc203393932)

[3.2 Looping constructs: for, while, repeat 24](#_Toc203393933)

[3.3 Creating and calling functions 26](#_Toc203393934)

[3.4 Apply family (apply, lapply, sapply) for data iteration 28](#_Toc203393935)

[3.5 Wrap-up 31](#_Toc203393936)

[Chapter 4: Data Structures & Manipulation 32](#_Toc203393937)

[4.1 Vectors, lists, matrices, arrays 32](#_Toc203393938)

[4.2 Data frames and tibbles 35](#_Toc203393939)

[4.3 Indexing and subsetting techniques 36](#_Toc203393940)

[4.4 Combining and reshaping data (cbind, rbind, pivot) 39](#_Toc203393941)

[4.4.2 Reshaping Data with pivot\_wider() 40](#_Toc203393942)

[Wrap-up 46](#_Toc203393943)

[Chapter 5: Data Wrangling with Tidyverse 47](#_Toc203393944)

[5.1 Intro to dplyr, tidyr, stringr, readr 47](#_Toc203393945)

[5.2 Filtering, sorting, grouping, and summarizing with dplyr 48](#_Toc203393946)

[5.3 Cleaning text data and missing values with stringr and tidyr 52](#_Toc203393947)

[5.4 Evolving our running example with real-world CSVs 55](#_Toc203393948)

[Chapter 6: Data Visualization 58](#_Toc203393949)

[6.1 Intro to ggplot2: grammar of graphics 58](#_Toc203393950)

[6.2 Bar charts, histograms, boxplots, scatter plots 59](#_Toc203393951)

[6.3 Customizing labels, themes, and aesthetics 62](#_Toc203393952)

[6.4 Visual storytelling with customer feedback 64](#_Toc203393953)

[Chapter 7: Statistical Analysis in R 67](#_Toc203393954)

[7.1 Descriptive statistics 67](#_Toc203393955)

[7.2 T-tests, ANOVA, chi-squared tests 69](#_Toc203393956)

[7.3 Correlation and regression analysis 71](#_Toc203393957)

[7.4 Visualizing distributions and confidence intervals 72](#_Toc203393958)

[Chapter 8: Working with Dates & Strings 75](#_Toc203393959)

[8.1 Date/time objects with lubridate 75](#_Toc203393960)

[8.2 String manipulation with stringr 78](#_Toc203393961)

[8.3 Real-world applications: timestamped reviews and feedback parsing 80](#_Toc203393962)

[Chapter 9: Intermediate Functions & Environments 83](#_Toc203393963)

[9.1 Anonymous and nested functions 83](#_Toc203393964)

[9.2 Scope and environments 85](#_Toc203393965)

[Chapter 10: Advanced Topics and Real-World Integration 89](#_Toc203393966)

[10.1 Object-oriented programming in R (S3, S4, R6 systems) 89](#_Toc203393967)

[10.2 Performance optimization and Rcpp 93](#_Toc203393968)

[10.3 Connecting to external data sources 95](#_Toc203393969)

[Chapter 11: Reproducibility, Reporting, and Collaboration 98](#_Toc203393970)

[11.1 Reproducible research with R Markdown 98](#_Toc203393971)

[Chapter 12: Building Interactive Applications and Scalable Solutions 104](#_Toc203393972)

[12.1 Building interactive web applications with Shiny 104](#_Toc203393973)

[Appendix A: R and RStudio Quick Reference 109](#_Toc203393974)

[A.1 RStudio Interface Panes 109](#_Toc203393975)

[A.2 Basic R Commands 109](#_Toc203393976)

[A.3 dplyr Verbs Reference 110](#_Toc203393977)

[Appendix B: Common R Errors and Troubleshooting 111](#_Toc203393978)

[B.1 "could not find function "X"" 111](#_Toc203393979)

[B.2 "Error in filter(...): object 'X' not found" or "object 'X' not found" 111](#_Toc203393980)

[B.3 "Error: data must be a data frame, or other object coercible by fortify(), not a numeric 111](#_Toc203393981)

[B.4 "Error: unexpected 'token' in 'code'" 112](#_Toc203393982)

[B.5 "Error in [.data.frame(data, , variable) : undefined columns selected" 112](#_Toc203393983)

[B.6 "Error in file(file, "rt") : cannot open the connection" 112](#_Toc203393984)

[B.7 "Error: removed X rows containing non-finite values (stat\_bin)." (or similar warnings) 113](#_Toc203393985)

[Appendix C: Recommended R Packages for Further Exploration 113](#_Toc203393986)

[Appendix D: Glossary of Key Terms 114](#_Toc203393987)

# Executive Summary: Mastering R: From Fundamentals to Real-World Solutions

"Mastering R: From Fundamentals to Real-World Solutions" is a comprehensive and practical guide designed for developers and data scientists seeking to harness the full power of the R programming language. In today's data-driven world, R stands out as an indispensable tool for statistical computing, data analysis, visualization, and machine learning, lauded for its open-source accessibility, vast ecosystem of packages, and robust capabilities. This book serves as a roadmap for readers to navigate the R landscape, from initial setup to advanced application development.

The journey begins with a foundational introduction to R and RStudio, covering installation, interface navigation, and the crucial concept of CRAN packages. Readers then dive into language fundamentals, exploring variables, data types, basic operations, and R's powerful vectorization, all while building a practical "Customer Feedback Analyzer" example. This hands-on approach continues through chapters dedicated to control flow structures (conditionals, loops) and function creation, including advanced techniques like anonymous functions and the apply family for efficient data iteration.

A significant portion of the book is dedicated to data structures and manipulation, meticulously detailing vectors, lists, matrices, arrays, and data frames (including modern tibbles). It emphasizes robust indexing, subsetting, and reshaping techniques using cbind, rbind, pivot\_longer(), and pivot\_wider(). The book then transitions to the Tidyverse ecosystem, focusing on essential packages like dplyr for filtering, sorting, grouping, and summarizing data, tidyr for handling data structure, stringr for text cleaning, and readr for efficient data import.

Data visualization is covered in-depth with ggplot2, guiding readers through the creation of various plot types (bar charts, histograms, boxplots, scatter plots) and mastering customization for effective visual storytelling. The analytical capabilities of R are then explored through statistical analysis, including descriptive statistics, common inferential tests (t-tests, ANOVA, chi-squared), correlation, and regression, complemented by techniques for visualizing distributions and confidence intervals. Practical skills for handling dates, times, and strings are developed using lubridate and stringr, crucial for processing real-world, timestamped feedback data.

Finally, the book elevates readers' skills with intermediate functions and environments, explaining scope, anonymous, and nested functions. It progresses to advanced topics, including R's Object-Oriented Programming systems (S3, S4, R6), strategies for performance optimization (vectorization, Rcpp integration), and connecting to external data sources like databases, web APIs, and web scraping. The concluding chapter champions reproducibility and collaboration through R Markdown for dynamic reporting and Git/GitHub for version control, ensuring analyses are transparent, shareable, and scalable.

By blending foundational theory with practical, real-world applications centered around the "Customer Feedback Analyzer," "Mastering R" equips readers with the expertise to confidently import, clean, analyze, visualize, and report on data using R. It prepares individuals not just to understand R, but to leverage it as a powerful tool for solving complex data challenges in both development and data science contexts.

# Chapter 1: Welcome to R

Welcome to the world of R! Whether you are a seasoned developer or a data enthusiast, this chapter lays the groundwork for success with R. This chapter covers the basics: what R is, why it is crucial in today's data-driven world, and how to set everything up to begin coding.

## 1.1 What is R and why it's essential today?

R is a programming language, and a free software environment designed for statistical computing and graphics. Ross Ihaka and Robert Gentleman developed R in the 1990s, and it was first released in 2000. Its popularity has grown, especially in the last decade. [According to intro2r.com, it is maintained by the R Development Core Team](https://www.google.com/url?sa=i&source=web&rct=j&url=https://intro2r.com/chap1.html&ved=2ahUKEwjHjNrjubyOAxXEVTABHRa5L8MQy_kOegYIAwgAEAc&opi=89978449&cd&psig=AOvVaw1Kj9hvA8HCYSQ4zur-C9F2&ust=1752586005745000).

Why R?

* Open Source and Free: R is available under the GNU General Public License. This means it is free to use, distribute, and modify. This makes it accessible to individuals and organizations of all sizes.
* Built for Statistics: R is designed for statistical analysis, providing tools for basic descriptive statistics to advanced modeling and machine learning algorithms. Functions for linear regression, time series analysis, and statistical inference are readily available.
* Powerful Graphics: R excels in data visualization, allowing the creation of high-quality, customizable graphics for exploration and presentation. R provides the tools to bring data to life visually.
* Extensive Package Ecosystem: R's strength lies in its collection of user-contributed packages, which provide specialized functions and datasets. The Comprehensive R Archive Network (CRAN) hosts thousands of these packages, covering fields like bioinformatics, finance, and social sciences.
* Reproducibility: R promotes reproducible research practices. Writing code in scripts creates a clear and traceable record of analysis. Code can be shared with others, allowing them to replicate results and build upon the work.
* Active Community Support: R has a large community of users and developers. Ample support is available through online forums like Stack Overflow and dedicated websites when challenges arise.

## 1.2 Installing R and RStudio

Before programming in R, install R and RStudio, an Integrated Development Environment (IDE) that makes working with R more user-friendly and efficient.

1.2.1 Installing R

1. Visit CRAN: Open a web browser and go to [The Comprehensive R Archive Network (CRAN)](https://www.google.com/url?sa=i&source=web&rct=j&url=https://cran.r-project.org/&ved=2ahUKEwjHjNrjubyOAxXEVTABHRa5L8MQy_kOegYIAwgAEB8&opi=89978449&cd&psig=AOvVaw1Kj9hvA8HCYSQ4zur-C9F2&ust=1752586005745000).
2. Download R: Under the "Download and Install R" section, click on the link for the operating system (Windows, macOS, or Linux).
3. Choose the Latest Release: Select the latest release of R for the system.
4. Run the Installer: Open the installer file and follow the on-screen instructions. For most users, accepting the default settings will work.

1.2.2 Installing RStudio

1. Visit RStudio (Posit) Website: Go to the [official RStudio download page](https://www.google.com/url?sa=i&source=web&rct=j&url=https://posit.co/download/rstudio-desktop/&ved=2ahUKEwjHjNrjubyOAxXEVTABHRa5L8MQy_kOegYIAwgAECQ&opi=89978449&cd&psig=AOvVaw1Kj9hvA8HCYSQ4zur-C9F2&ust=1752586005745000).
2. Download RStudio Desktop: Choose the free "RStudio Desktop Open Source License" version.
3. Download the Installer: Click the "DOWNLOAD RSTUDIO DESKTOP" button for the operating system.
4. Run the Installer: Open the downloaded file and follow the installation instructions, typically accepting the default settings.
5. Launch RStudio: After installation, launch RStudio from the applications folder or desktop shortcut.

## 1.3 Navigating the RStudio interface

When opening RStudio, you'll see a multi-pane interface designed to streamline the R development workflow. While the layout is customizable, the default typically presents four main panes.

* Source Pane (Top-Left): This is the code editor where R scripts are written, edited, and saved (files with the .R extension). It features syntax highlighting, code completion, and tools to run selected lines or entire scripts.
* Console Pane (Bottom-Left): This is where R commands are executed, and the output of code is displayed. Commands can be typed directly into the console, but for longer or more complex code, it's better to write them in the Source Pane and send them to the console.
* Environment/History Pane (Top-Right):
  + Environment Tab: This tab displays the objects (variables, datasets, functions) currently loaded in the R session.
  + History Tab: This tab keeps a record of all the commands executed in the console.
* Files/Plots/Packages/Help Pane (Bottom-Right):
  + Files Tab: Allows navigation of files and folders in the working directory.
  + Plots Tab: Displays graphs or visualizations generated by the R code.
  + Packages Tab: Lists all installed R packages and allows you to load them into the current session.
  + Help Tab: Provides access to R's documentation system. Information about functions, datasets, and packages can be found here.

## 1.4 Overview of CRAN, packages, and repositories

R's functionality expands through its community, primarily through packages. A package is a collection of functions, data, and documentation that extends R's capabilities for specific tasks.

* CRAN (The Comprehensive R Archive Network): This is the primary repository for R packages. When a package is installed from CRAN, R retrieves it from one of the many CRAN mirror servers.
* Other Repositories: Packages can also be found on platforms like [Bioconductor](https://www.google.com/url?sa=i&source=web&rct=j&url=https://www.bioconductor.org/&ved=2ahUKEwjHjNrjubyOAxXEVTABHRa5L8MQy_kOegYIAwgAEDc&opi=89978449&cd&psig=AOvVaw1Kj9hvA8HCYSQ4zur-C9F2&ust=1752586005745000) (for biology-related packages) or [GitHub](https://www.google.com/url?sa=i&source=web&rct=j&url=https://github.com/&ved=2ahUKEwjHjNrjubyOAxXEVTABHRa5L8MQy_kOegYIAwgAEDg&opi=89978449&cd&psig=AOvVaw1Kj9hvA8HCYSQ4zur-C9F2&ust=1752586005745000) (often for development versions of packages or those that don't meet CRAN's requirements).
* Installing and Loading Packages:
  + To install a package (i.e., download and save it to your computer), use the install.packages() function. For example, to install the tidyverse package (a popular collection of data science packages), you would run: install.packages("tidyverse").
  + After installation, load the package into the current R session before using its functions. Use the library() function for this: library(tidyverse).

## 1.5 Writing your first line of R code

Let's write the first line of R code. Traditionally, the first program in any language is a "Hello, World!" program, and R is no exception.

1. Open an R Script: In RStudio, go to File > New File > R Script. A new blank script will open in the Source Pane.
2. Type Your Code: In the new script, type the following:

R

print("Hello, World!")

1. Run the Code: Place the cursor on the line and press Ctrl + Enter (or Cmd + Enter on a Mac).
2. Observe the Output: The output "Hello, World!" will appear in the Console Pane.

You have just executed the first R command. Values can also be assigned to variables (objects) in R using the assignment operator <-:

R

greeting <- "Hello, World!"

print(greeting)

This assigns the string "Hello, World!" to a variable named greeting, which then gets printed to the console.

1.6 Wrap-up

This chapter has provided an introduction to R and RStudio. You have learned what R is, why it's a valuable tool for data analysis, how to set up your environment, and how to execute your first lines of code. The next chapter will explore language fundamentals, covering variables, data types, and basic operations that form the building blocks of R programming. Practice is key, and experimenting with the code examples will solidify

# Chapter 2: Language Fundamentals

In R, storing and categorizing information, along with performing operations, is necessary, as in any programming language. This chapter introduces variables, basic data types, and fundamental operations. These serve as the foundation for all R programming tasks. The "Customer Feedback Analyzer" example will also be introduced.

## 2.1 Variables, assignment operators, and comments

Variables

A variable (or object) in R is a name given to a storage location that holds a value. This value can be a single number, text, a collection of values, or a more complex data structure. When creating a variable, a value is assigned to it. This allows you to refer to that value by its name later in your code.

Assignment Operators

In R, the most common way to assign a value to a variable uses the assignment operator <-. This can be read as "gets" or "is assigned." The equals sign = can also be used for assignment, but <- is generally preferred for clarity and consistency, especially when assigning values within function calls [4].

Here's how it works:

R

# Assigning a numeric value to a variable

customer\_id <- 101

# Assigning a character string to a variable

product\_name <- "Wireless Earbuds"

# Assigning a logical value

is\_premium\_customer <- TRUE

# You can also assign the result of an operation

total\_cost <- 25.50 \* 3

After these assignments, the variables customer\_id, product\_name, is\_premium\_customer, and total\_cost will appear in your RStudio Environment pane, showing their names and assigned values.

Comments

Comments are lines of code that R ignores. They explain the code, making it easier for yourself and others to understand what the code does. In R, comments begin with a hash symbol (#). Any text following the # on that line is considered a comment.

R

# This is a single-line comment.

# It explains the purpose of the next line of code.

# Assign a customer rating (out of 5)

customer\_rating <- 4.5 # This comment explains the '4.5'

# R will execute the code, but ignore the comments.

Using comments effectively is crucial for writing readable and maintainable R scripts.

## 2.2 Data types: numeric, character, logical

R stores different kinds of data using various data types. Understanding these is fundamental because they dictate what operations you can perform on your data. R features several fundamental (atomic) data types, but for most data analysis tasks, you'll primarily work with three: numeric, character, and logical.

2.2.1 Numeric

Numeric is the default data type for numbers in R. It covers real numbers (decimals) and integers. R stores all numbers as double-precision floating-point numbers, which are capable of storing both integers and decimals.

R

# An integer number

age <- 30

print(age)

class(age) # Check the data type

# A decimal number

price <- 49.99

print(price)

class(price) # Check the data type

While R treats all numbers as numeric (double), you can explicitly specify an integer type by appending L to the number. This can be useful for memory optimization in very large datasets or when interacting with other programming languages or databases that differentiate between integers and floating-point numbers.

R

# Explicitly storing an integer

product\_quantity <- 10L

print(product\_quantity)

class(product\_quantity) # Will show "integer"

2.2.2 Character

The character data type stores text or strings. Character values are enclosed in either single quotes (') or double quotes ("). It's generally best practice to use double quotes consistently.

R

# Storing a single word

feedback\_type <- "Positive"

print(feedback\_type)

class(feedback\_type)

# Storing a sentence

customer\_comment <- "The product arrived quickly and exceeded expectations."

print(customer\_comment)

class(customer\_comment)

2.2.3 Logical

The logical data type stores boolean values: TRUE or FALSE. They are often the result of logical comparisons or used for conditional execution. R also recognizes T and F as shorthand for TRUE and FALSE, respectively, but it's safer and clearer to use the full words.

R

# Result of a comparison

is\_high\_rating <- customer\_rating > 4

print(is\_high\_rating)

class(is\_high\_rating)

# Directly assigning a logical value

has\_warranty <- TRUE

print(has\_warranty)

class(has\_warranty)

## 2.3 Type coercion and inspection

Understanding data types is crucial because R sometimes automatically converts data from one type to another (coercion), which can lead to unexpected results if not handled consciously.

Type Coercion

When you try to combine different data types into a single vector (which is a fundamental concept we'll explore more in Chapter 4), R will coerce all elements to the most flexible type to prevent data loss. The hierarchy is typically: logical -> numeric -> character [3].

R

# Combining different types in a vector

mixed\_data <- c(1, "apple", TRUE)

print(mixed\_data)

class(mixed\_data) # What do you expect here?

# Combining numbers and logicals

numeric\_logical\_data <- c(10, FALSE, 20, TRUE)

print(numeric\_logical\_data)

class(numeric\_logical\_data) # What do you expect here?

In the first example, mixed\_data will be coerced to character because character is the most flexible type. The number 1 becomes "1", and TRUE becomes "TRUE". In the second example, FALSE becomes 0, and TRUE becomes 1, resulting in a numeric vector.

You can also explicitly coerce data types using functions like as.numeric(), as.character(), and as.logical().

R

numeric\_string <- "123"

converted\_number <- as.numeric(numeric\_string)

print(converted\_number)

class(converted\_number)

logical\_number <- 0

converted\_logical <- as.logical(logical\_number)

print(converted\_logical)

class(converted\_logical)

# Be careful when coercing:

bad\_number\_string <- "hello"

coerced\_bad\_number <- as.numeric(bad\_number\_string)

print(coerced\_bad\_number) # This will result in NA (Not Available) with a warning.

Type Inspection

Several functions help you inspect the data type of a variable:

* class(): Returns the class of an object (e.g., "numeric", "character", "logical").
* typeof(): Returns the internal R type of an object (e.g., "double", "character", "logical", "integer").
* is.numeric(), is.character(), is.logical(): These are logical tests that return TRUE if the object is of the specified type, and FALSE otherwise.

R

test\_value <- 15.7

class(test\_value)

typeof(test\_value)

is.numeric(test\_value)

is.character(test\_value)

text\_value <- "R is fun"

class(text\_value)

is.character(text\_value)

## 2.4 Arithmetic and logical operations

R is a powerful calculator, and it handles operations on entire vectors (collections of values) very efficiently. This concept is called vectorization, and it's a cornerstone of R programming.

2.4.1 Arithmetic Operations

Basic arithmetic operators work as expected:

|  |  |  |
| --- | --- | --- |
| Operator | Description | Example |
| + | Addition | 5 + 3 |
| - | Subtraction | 10 - 4 |
| \* | Multiplication | 6 \* 7 |
| / | Division | 20 / 5 |
| ^ or \*\* | Exponentiation | 2^3 or 2\*\*3 |
| %% | Modulo (remainder) | 10 %% 3 |
| %/% | Integer Division | 10 %/% 3 |

Vectorization in Action:  
Operations are applied element-wise when performing arithmetic on vectors.

R

# Create two numeric vectors

vector1 <- c(10, 20, 30)

vector2 <- c(2, 5, 1)

# Element-wise addition

sum\_vector <- vector1 + vector2

print(sum\_vector) # Output: [1] 12 25 31

# Element-wise multiplication

product\_vector <- vector1 \* vector2

print(product\_vector) # Output: [1] 20 100 30

If vectors are of different lengths, R will recycle the shorter vector to match the length of the longer one. This can sometimes be useful but often indicates a mistake if the lengths are not multiples of each other, resulting in a warning [5].

R

# Recycling example (no warning, 10 is recycled three times)

longer\_vector <- c(1, 2, 3, 4, 5, 6)

shorter\_vector <- c(10, 20)

result\_recycled <- longer\_vector + shorter\_vector

print(result\_recycled)

# Recycling example (with warning)

vector\_a <- c(1, 2, 3)

vector\_b <- c(10, 20)

result\_warning <- vector\_a + vector\_b # R will warn that the longer object length is not a multiple of the shorter object length.

print(result\_warning)

2.4.2 Logical Operations

Logical operators compare values and return TRUE or FALSE. They are crucial for filtering data and controlling program flow.

|  |  |  |
| --- | --- | --- |
| Operator | Description | Example |
| == | Equal to | x == y |
| != | Not equal to | x != y |
| < | Less than | x < y |
| <= | Less than or equal to | x <= y |
| > | Greater than | x > y |
| >= | Greater than or equal to | x >= y |
| & | Logical AND (element-wise) | x & y |
| && | Logical AND (scalar, short-circuiting) | x && y |
| ` | ` | Logical OR (element-wise) |
| ` |  | ` |
| ! | Logical NOT | !x |

Vectorization with Logical Operators:

R

# Create a vector of customer ratings

ratings <- c(5, 4, 2, 5, 3, 4)

# Which ratings are excellent (equal to 5)?

is\_excellent <- ratings == 5

print(is\_excellent) # Output: [1] TRUE FALSE FALSE TRUE FALSE FALSE

# Which ratings are good or excellent (greater than or equal to 4)?

is\_good\_or\_excellent <- ratings >= 4

print(is\_good\_or\_excellent) # Output: [1] TRUE TRUE FALSE TRUE FALSE TRUE

# Combining conditions: ratings that are good AND excellent

# (this is tautological for single values, but illustrates combining conditions)

good\_and\_excellent <- (ratings >= 4) & (ratings == 5)

print(good\_and\_excellent) # Output: [1] TRUE FALSE FALSE TRUE FALSE FALSE

# Applying logical NOT

not\_excellent <- !(ratings == 5)

print(not\_excellent) # Output: [1] FALSE TRUE TRUE FALSE TRUE TRUE

The && and || operators are typically used inside if statements and evaluate only the first element of each vector, stopping early if the outcome is already determined (short-circuiting) [2, 1]. For vector operations, & and | are the common choices.

## 2.5 Running Example Introduction: “Customer Feedback Analyzer”

To make the R learning journey more concrete, a "Customer Feedback Analyzer" will be built throughout this booklet. The goal is to process raw customer feedback data to extract insights, quantify sentiment, and visualize trends.

For now, let's start with a very simple representation of some initial feedback data using the variables and vectors just learned.

Imagine feedback was received for three product orders:

* Order 1001: "Excellent product, fast delivery!" (Rating: 5, Positive sentiment)
* Order 1002: "Average experience. The item was okay." (Rating: 3, Neutral sentiment)
* Order 1003: "Disappointed with the quality." (Rating: 2, Negative sentiment)

These pieces of information can be represented using vectors:

R

# Store order IDs as a numeric vector (or integer if preferred)

order\_ids <- c(1001, 1002, 1003)

print(order\_ids)

# Store customer feedback text as a character vector

feedback\_text <- c(

"Excellent product, fast delivery!",

"Average experience. The item was okay.",

"Disappointed with the quality."

)

print(feedback\_text)

# Store numerical ratings as a numeric vector

product\_ratings <- c(5, 3, 2)

print(product\_ratings)

# Store sentiment categories as a character vector

sentiment <- c("Positive", "Neutral", "Negative")

print(sentiment)

# Let's create a logical vector: were any ratings below 3?

is\_low\_rating <- product\_ratings < 3

print(is\_low\_rating)

This initial setup demonstrates how to use the fundamental data types and vectors to hold pieces of customer feedback data. In later chapters, combining these related vectors into more structured data structures (like data frames) and performing more complex analyses will be explained.

## 2.6 Wrap-up

This chapter provides a foundational understanding of R's language fundamentals:

* How to create variables and assign values using <-.
* The importance of comments (#) for code readability.
* The primary data types: numeric, character, and logical.
* The concept of type coercion and how to inspect data types using functions like class() and is.numeric().
* Performing arithmetic and logical operations, understanding R's powerful vectorization capabilities.
* Started the "Customer Feedback Analyzer" running example by representing initial feedback data using vectors.

These building blocks are essential. In Chapter 3, how to control the flow of R programs and create reusable functions will be explored, further empowering your ability to write sophisticated R code.

# Chapter 3: Control Flow and Functions

Code is rarely executed linearly from start to finish. Control flow mechanisms enable programs to make decisions and repeat tasks, while functions allow the creation of reusable blocks of code. Mastering these concepts is crucial for writing efficient, dynamic, and maintainable R scripts.

## 3.1 Conditional statements: if, else, switch

Conditional statements allow different code blocks to execute based on whether a condition is true or false.

3.1.1 if and else

The if statement evaluates a logical condition. If it is TRUE, the code block following if executes. Optionally, the else statement specifies a code block to run if the condition is FALSE.

Syntax:

R

if (condition) {

# Code to execute if condition is TRUE

} else {

# Code to execute if condition is FALSE (optional)

}

Example 1: Basic if statement

R

customer\_rating <- 4

if (customer\_rating >= 4) {

print("Customer is satisfied.")

}

Explanation: If customer\_rating is 4 or greater, the message "Customer is satisfied" is printed.

Example 2: if-else statement

R

customer\_rating <- 2

if (customer\_rating >= 4) {

print("Customer is satisfied.")

} else {

print("Customer needs attention.")

}

Explanation: Since customer\_rating is not greater than or equal to 4, the code in the else block executes, printing "Customer needs attention."

Example 3: Chaining else if  
Multiple conditions can be checked using else if. R evaluates conditions sequentially, and the first TRUE condition executes its block.

R

customer\_rating <- 3

if (customer\_rating == 5) {

print("Excellent feedback!")

} else if (customer\_rating >= 4) {

print("Positive feedback.")

} else if (customer\_rating >= 3) {

print("Neutral feedback, needs review.")

} else {

print("Negative feedback, urgent action required!")

}

Explanation: The code checks for a rating of 5, then a rating of 4 or higher. Since customer\_rating is 3, neither of the first two conditions is met. The third condition, customer\_rating >= 3, is TRUE, so "Neutral feedback, needs review" is printed.

ifelse(test, yes, no)

* test: A logical vector.
* yes: The value to return if test is TRUE.
* no: The value to return if test is FALSE.

For element-wise conditional logic on vectors, the ifelse() function is more efficient and concise than if-else statements within loops.

Example 4: Using ifelse() with customer ratings

R

product\_ratings <- c(5, 3, 2, 4, 1)

feedback\_category <- ifelse(product\_ratings >= 4, "Positive/Good", "Negative/Needs Improvement")

print(feedback\_category)

Explanation: This code categorizes each rating as "Positive/Good" if it's 4 or higher, or "Negative/Needs Improvement" otherwise. The result is a character vector with the corresponding categories for each rating.

3.1.2 switch

The switch() statement is ideal for handling multiple conditions based on a single expression or value, offering a more compact and readable alternative to long if-else if chains.

Syntax (using a string expression):  
switch(expression, case1 = result1, case2 = result2, ..., default = default\_result)

Example 5: Using switch() for sentiment analysis

R

sentiment\_type <- "Neutral"

action\_plan <- switch(sentiment\_type,

"Positive" = "Promote product, encourage reviews.",

"Neutral" = "Investigate further, seek specific feedback.",

"Negative" = "Contact customer, resolve issue immediately.",

"Unknown" = "Needs manual classification." # Default case

)

print(action\_plan)

Explanation: Based on the sentiment\_type variable, switch() matches the string to one of the cases and returns the corresponding action plan. If no match is found, the value of the Unknown case is returned. According to Fiveable Library, the switch statement in R can sometimes be less efficient than well-structured if-else statements due to R's implementation.

## 3.2 Looping constructs: for, while, repeat

Loops automate repetitive tasks, executing a block of code multiple times. However, in R, it's often more efficient to use vectorized operations or apply family functions than explicit loops, especially for large datasets.

3.2.1 for loop

The for loop iterates over the elements of a sequence (like a vector or list), executing the code block once for each element.

Syntax:

R

for (variable in sequence) {

# Code to execute for each element

}

Example 6: Iterating through customer feedback

R

feedback\_ids <- c("FB001", "FB002", "FB003")

for (id in feedback\_ids) {

print(paste("Processing feedback ID:", id))

}

Explanation: The loop iterates through each id in the feedback\_ids vector, printing a message for each one.

Example 7: Calculating average rating using a for loop (less efficient approach)

While possible, this approach is less efficient than vectorized operations for calculating statistics.

R

ratings <- c(5, 4, 2, 5, 3)

total\_sum <- 0

for (rating in ratings) {

total\_sum <- total\_sum + rating

}

average\_rating <- total\_sum / length(ratings)

print(paste("Average Rating:", average\_rating))

# More efficient vectorized approach:

# print(paste("Average Rating (vectorized):", mean(ratings)))

Explanation: The loop iterates through ratings, adding each value to total\_sum. After the loop, the average is calculated. This demonstrates a loop's functionality but highlights the benefits of R's built-in vectorized functions like mean().

3.2.2 while loop

A while loop repeatedly executes a block of code as long as a specified condition remains TRUE.

Syntax:

R

while (condition) {

# Code to execute as long as condition is TRUE

}

Example 8: Processing feedback until a certain condition is met

R

feedback\_queue <- 5 # Number of feedback items to process

processed\_count <- 0

while (feedback\_queue > 0) {

processed\_count <- processed\_count + 1

feedback\_queue <- feedback\_queue - 1

print(paste("Processed", processed\_count, "feedback items. Remaining:", feedback\_queue))

}

Explanation: The loop continues as long as feedback\_queue is greater than 0. In each iteration, processed\_count increases, and feedback\_queue decreases until the condition becomes FALSE.

3.2.3 repeat loop and break

The repeat loop executes a block of code indefinitely until a break statement is encountered within the loop's body. The break statement stops the loop's execution.

Syntax:

R

repeat {

# Code to execute

if (condition\_to\_break) {

break # Exit the loop

}

}

Example 9: Simulating feedback processing with a repeat loop

R

total\_feedback\_items <- 10

items\_processed <- 0

repeat {

items\_processed <- items\_processed + 1

print(paste("Processing item", items\_processed))

if (items\_processed >= total\_feedback\_items) {

print("All feedback items processed!")

break # Exit the loop once all items are processed

}

}

Explanation: This loop runs continuously until items\_processed reaches total\_feedback\_items, at which point the break statement terminates the loop.

## 3.3 Creating and calling functions

Functions are reusable blocks of code that perform specific tasks. Defining functions helps organize code, improves readability, and avoids repetition.

3.3.1 Creating a function

A function is defined using the function() keyword, followed by arguments in parentheses and the function body in curly braces {}.

Syntax:

R

function\_name <- function(argument1, argument2 = default\_value, ...) {

# Function body: code to be executed

# ...

return(result) # Optional: returns a value

}

3.3.2 Calling a function

To use a function, simply type its name followed by the arguments (values passed to the function) in parentheses.

Example 10: Simple greeting function

R

greet <- function(name) {

message <- paste("Hello,", name, "!")

print(message)

}

# Call the function

greet("Alice")

Explanation: The greet function takes one argument, name. When called with "Alice", it constructs a greeting message and prints it.

3.3.3 Argument defaults and return values

* Argument Defaults: Functions can have arguments with default values. If a user doesn't provide a value for such an argument, the default is used. This makes functions more flexible.
* Return Values: Functions can return a value using the return() statement. If return() is omitted, the function implicitly returns the value of the last evaluated expression.

Example 11: Function with default argument and return value

R

analyze\_rating <- function(rating, threshold = 3) { # default threshold is 3

if (rating > threshold) {

return("Above average")

} else if (rating == threshold) {

return("Average")

} else {

return("Below average")

}

}

# Call with default threshold

result1 <- analyze\_rating(rating = 4)

print(result1)

# Call with custom threshold

result2 <- analyze\_rating(rating = 2, threshold = 2.5)

print(result2)

Explanation: The analyze\_rating function takes rating and threshold. threshold has a default of 3. The function returns a string indicating the rating's performance relative to the threshold. When rating is 4 and threshold is the default 3, it returns "Above average". When rating is 2 and threshold is explicitly set to 2.5, it returns "Below average".

## 3.4 Apply family (apply, lapply, sapply) for data iteration

The **apply** family functions are powerful and efficient alternatives to loops, especially when performing operations on elements of lists, vectors, or margins of arrays/matrices. They are often faster and result in cleaner code than explicit loops for many common tasks.

3.4.1 lapply()

Applies a function to each element of a list (or vector treated as a list), always returning a list.

Syntax:  
lapply(X, FUN, ...)

* X: A list or vector.
* FUN: The function to apply.
* ...: Additional arguments to pass to FUN.

Example 12: Converting customer feedback text to lowercase

R

feedback\_comments <- list(

"Excellent product!",

"Item was okay.",

"Disappointed."

)

lowercase\_comments <- lapply(feedback\_comments, tolower)

print(lowercase\_comments)

Explanation: lapply() applies the tolower() function to each string in the feedback\_comments list, converting them to lowercase and returning the result as a list.

3.4.2 sapply()

A "simplifying" version of lapply(). It applies a function to each element of a list or vector and attempts to simplify the result to the most basic data structure possible, often a vector or matrix.

Syntax:  
sapply(X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)

* X, FUN, ...: Same as lapply().
* simplify: If TRUE (default), sapply() attempts to simplify the result.

Example 13: Checking character count in feedback comments

R

feedback\_comments <- c(

"Excellent product!",

"Item was okay.",

"Disappointed."

)

char\_counts <- sapply(feedback\_comments, nchar)

print(char\_counts)

Explanation: sapply() applies the nchar() function (which counts characters in a string) to each comment. Since the result for each comment is a single number, sapply() simplifies the output into a numeric vector.

3.4.3 apply()

Applies a function over the margins (rows or columns) of an array or matrix. It is commonly used for summarizing data across rows or columns of matrices or data frames.

Syntax:  
apply(X, MARGIN, FUN, ...)

* X: An array or matrix.
* MARGIN: A vector indicating which margins to apply FUN over (1 for rows, 2 for columns, c(1,2) for both).
* FUN: The function to apply.

Example 14: Calculating summary statistics for numerical feedback data

R

# Imagine we have a matrix of feedback scores for different product aspects

# Rows: Different feedback items

# Columns: Scores for Aspect A, Aspect B, Aspect C

feedback\_scores\_matrix <- matrix(c(

5, 4, 3,

4, 5, 4,

3, 2, 3,

5, 5, 5

), nrow = 4, byrow = TRUE, dimnames = list(

paste("Item", 1:4),

c("Aspect\_A", "Aspect\_B", "Aspect\_C")

))

print(feedback\_scores\_matrix)

# Calculate the mean score for each aspect (column)

mean\_by\_aspect <- apply(feedback\_scores\_matrix, 2, mean)

print(mean\_by\_aspect)

# Calculate the sum of scores for each feedback item (row)

sum\_by\_item <- apply(feedback\_scores\_matrix, 1, sum)

print(sum\_by\_item)

Explanation:

* The first apply() call calculates the mean for each column (MARGIN = 2), representing the average score for each aspect across all feedback items.
* The second apply() call calculates the sum for each row (MARGIN = 1), representing the total score for each feedback item across all aspects.

## 3.5 Wrap-up

In this chapter, the foundational concepts of control flow and functions in R have been explored:

* Conditional statements (if, else, switch) enable decision-making in code based on conditions.
* Looping constructs (for, while, repeat) automate repetitive tasks, though vectorized operations are often more efficient in R.
* Functions streamline code by defining reusable blocks, with features like default arguments and explicit return values.
* The apply family (apply, lapply, sapply) provides powerful alternatives to loops for efficiently iterating over data structures like lists, vectors, and matrices [1.6.

# Chapter 4: Data Structures & Manipulation

Chapter 2 introduced the atomic data types (numeric, character, logical) and briefly touched upon vectors, which store multiple elements of the same type. While vectors are the foundation, R offers a richer set of data structures to handle more complex and real-world data effectively. This chapter delves into these essential data structures and how to manipulate them.

## 4.1 Vectors, lists, matrices, arrays

4.1.1 Vectors

As discussed in Chapter 2>>, vectors are the most basic data structure in R, used to store a collection of elements of the same data type. Datamentor explains how to create vectors using functions like c(), seq(), and the colon operator (:). Operations on vectors are vectorized, meaning they apply element-wise, making R code concise and efficient. The index of vectors in R starts from 1.

R

# Creating a numeric vector

product\_prices <- c(29.99, 15.50, 89.00, 5.25)

print(product\_prices)

# Creating a character vector of product categories

product\_categories <- c("Electronics", "Apparel", "Home Goods", "Accessories")

print(product\_categories)

# Vectorized operation

discounted\_prices <- product\_prices \* 0.8 # 20% discount

print(discounted\_prices)

4.1.2 Lists

Unlike vectors, lists are heterogeneous data structures, meaning they can store elements of different types and even different structures like vectors, matrices, or other lists within them. Think of a list as a container where each item can be something entirely different – a shopping list where one item is a quantity, another is a specific type of fruit, and a third is a note. A list in R is an object consisting of an ordered collection of objects known as its components.

R

# Creating a list for a customer order

customer\_order <- list(

customer\_name = "Alice",

order\_id = 1004,

items = c("Laptop", "Mouse", "Keyboard"),

quantities = c(1, 1, 1),

total\_amount = 1250.00,

is\_premium = TRUE

)

print(customer\_order)

Accessing List Elements:  
Elements can be accessed using double square brackets [[]] or the dollar sign $ notation if the elements are named.

R

# Accessing by name

customer\_name <- customer\_order$customer\_name

print(customer\_name)

# Accessing by position

first\_item <- customer\_order[[3]]

print(first\_item)

4.1.3 Matrices

Matrices are two-dimensional, homogeneous data structures where all elements must be of the same type, arranged in rows and columns. They are essentially vectors with a dim attribute (dimensions: number of rows, number of columns). W3Schools explains how to create a matrix using the matrix() function by specifying the number of rows and columns.

R

# Creating a matrix of sales figures for products over quarters

sales\_data <- matrix(

c(120, 150, 130, 180, 200, 190, 140, 170), # Data elements

nrow = 4, # Number of rows

ncol = 2, # Number of columns

byrow = TRUE, # Fill by row (default is by column)

dimnames = list(

c("Q1", "Q2", "Q3", "Q4"), # Row names

c("Product A", "Product B") # Column names

)

)

print(sales\_data)

Accessing Matrix Elements:  
Elements are accessed using square brackets [] with [row, column] indexing.

R

# Accessing a single element (Q2 sales for Product A)

q2\_prod\_a\_sales <- sales\_data[2, 1]

print(q2\_prod\_a\_sales)

# Accessing an entire row (Q3 sales)

q3\_sales <- sales\_data[3, ]

print(q3\_sales)

# Accessing an entire column (Product B sales)

product\_b\_sales <- sales\_data[, 2]

print(product\_b\_sales)

4.1.4 Arrays

Arrays are generalizations of matrices to more than two dimensions. For example, a 3-dimensional array could store sales data across products, quarters, and regions. Like matrices, all elements within an array must be of the same data type.

R

# Creating a 3D array: sales data for products (2), quarters (3), and regions (2)

region\_sales <- array(

c(10, 12, 15, 8, 11, 14, # Product A

20, 25, 22, 18, 28, 23), # Product B

dim = c(2, 3, 2), # Dimensions: 2 products, 3 quarters, 2 regions

dimnames = list(

c("Prod A", "Prod B"), # Dimension 1: Products

c("Q1", "Q2", "Q3"), # Dimension 2: Quarters

c("East", "West") # Dimension 3: Regions

)

)

print(region\_sales)

Accessing Array Elements:  
Indexing uses square brackets [] with [dimension1, dimension2, dimension3, ...].

R

# Sales for Product A in Q1, East region

prod\_a\_q1\_east <- region\_sales[1, 1, 1]

print(prod\_a\_q1\_east)

# All sales for Q2 in the West region

q2\_west\_sales <- region\_sales[, 2, 2]

print(q2\_west\_sales)

## 4.2 Data frames and tibbles

4.2.1 Data Frames

Data frames are the most common and versatile data structure for storing tabular data in R. Imagine a spreadsheet: each column can hold a different data type (e.g., numeric, character, logical), but all elements within a column must be of the same type. Each row represents an observation, and each column represents a variable. A data frame in R is made up of three components: data, rows, and columns. You can create a data frame using the data.frame() function.

R

# Let's revisit our customer feedback data and structure it into a data frame

customer\_feedback\_df <- data.frame(

OrderID = c(1001, 1002, 1003, 1004, 1005),

FeedbackText = c(

"Excellent product, fast delivery!",

"Average experience. The item was okay.",

"Disappointed with the quality.",

"Great value for money, highly recommend.",

"Broken upon arrival, very unhappy."

),

Rating = c(5, 3, 2, 5, 1),

Sentiment = c("Positive", "Neutral", "Negative", "Positive", "Negative"),

IsVerified = c(TRUE, TRUE, FALSE, TRUE, TRUE)

)

print(customer\_feedback\_df)

4.2.2 Tibbles

Tibbles (from the tibble package, part of the [Tidyverse](https://www.google.com/url?sa=i&source=web&rct=j&url=https://tibble.tidyverse.org/&ved=2ahUKEwix8cLBw7yOAxWSlYkEHSxpBEAQy_kOegYIAwgAEC0&opi=89978449&cd&psig=AOvVaw3acZYRdA9ya-3WqkXaVpz2&ust=1752588618417000)) are an enhanced version of data frames, designed to be more modern and user-friendly, especially for data science workflows. They behave similarly to data frames but come with key improvements: Educative explains that tibbles have a more advanced print function that only displays the first ten rows and columns that fit on the screen, along with the data types of each column. Tibbles do not convert character strings to factors by default and are generally stricter, providing clearer error messages and preventing accidental data type changes or partial matching of names. You can create tibbles using tibble() or tribble().

First, install the tidyverse package if not already installed, then load the tibble library:

R

# install.packages("tidyverse") # Uncomment and run if you haven't installed tidyverse

library(tibble)

# Creating a tibble from vectors

customer\_feedback\_tbl <- tibble(

OrderID = c(1001, 1002, 1003, 1004, 1005),

FeedbackText = c(

"Excellent product, fast delivery!",

"Average experience. The item was okay.",

"Disappointed with the quality.",

"Great value for money, highly recommend.",

"Broken upon arrival, very unhappy."

),

Rating = c(5, 3, 2, 5, 1),

Sentiment = c("Positive", "Neutral", "Negative", "Positive", "Negative"),

IsVerified = c(TRUE, TRUE, FALSE, TRUE, TRUE)

)

print(customer\_feedback\_tbl)

Notice the cleaner printing style and the inclusion of data types under the column names.

## 4.3 Indexing and subsetting techniques

Accessing specific parts of your data structures is fundamental for analysis. R provides powerful indexing and subsetting methods using square brackets [] and, for lists and data frames, [[ ]] and $.

4.3.1 Positional Indexing

Using numbers to specify elements or ranges:

R

# For vectors

my\_vector <- c("A", "B", "C", "D", "E")

first\_element <- my\_vector[1]

print(first\_element) # Output: "A"

range\_elements <- my\_vector[2:4]

print(range\_elements) # Output: "B" "C" "D"

# For matrices (row, column)

print(sales\_data[1, 1]) # Q1, Product A sales

# For data frames (row, column)

first\_feedback <- customer\_feedback\_df[1, ] # First row, all columns

print(first\_feedback)

first\_three\_feedbacks <- customer\_feedback\_df[1:3, ] # First three rows

print(first\_three\_feedbacks)

# Select specific columns by position

selected\_columns <- customer\_feedback\_df[, c(1, 3)] # OrderID and Rating

print(selected\_columns)

4.3.2 Named Indexing

Using names to specify elements or columns:

R

# For lists

customer\_items <- customer\_order$items

print(customer\_items)

# For data frames

ratings\_column <- customer\_feedback\_df$Rating

print(ratings\_column)

# Alternatively for data frames (using double square brackets)

ratings\_column\_alt <- customer\_feedback\_df[["Rating"]]

print(ratings\_column\_alt)

# Selecting multiple columns by name in a data frame

subset\_df\_by\_name <- customer\_feedback\_df[, c("OrderID", "Sentiment")]

print(subset\_df\_by\_name)

4.3.3 Logical Indexing

Using logical vectors (TRUE/FALSE) to select elements based on a condition:

R

# For vectors: selecting high ratings

high\_ratings\_only <- product\_ratings[product\_ratings >= 4] # from vector in 4.1.1

print(high\_ratings\_only)

# For data frames: selecting positive feedback

positive\_feedback <- customer\_feedback\_df[customer\_feedback\_df$Sentiment == "Positive", ]

print(positive\_feedback)

# Selecting reviews with low ratings AND verified customers

low\_rating\_verified <- customer\_feedback\_df[

(customer\_feedback\_df$Rating < 3) & (customer\_feedback\_df$IsVerified == TRUE),

]

print(low\_rating\_verified)

4.3.4 Subsetting with subset() function

The subset() function provides a more readable way to subset data frames, especially for complex conditions.

R

# Select feedback where rating is less than 3

low\_rating\_subset <- subset(customer\_feedback\_df, Rating < 3)

print(low\_rating\_subset)

# Select feedback with a Rating of 5, only showing the OrderID and FeedbackText columns

excellent\_feedback\_details <- subset(

customer\_feedback\_df,

Rating == 5,

select = c(OrderID, FeedbackText)

)

print(excellent\_feedback\_details)

## 4.4 Combining and reshaping data (cbind, rbind, pivot)

Real-world data often comes from various sources or needs to be structured in different ways for analysis. R provides functions to combine and reshape data structures.

4.4.1 Combining Data

* cbind() (Column Bind): Combines objects (vectors, matrices, or data frames) side-by-side as columns. All objects must have the same number of rows. Statology.org explains that cbind stands for column-bind and is used to combine vectors, matrices and data frames by column
* rbind() (Row Bind): Combines objects (vectors, matrices, or data frames) stacked on top of each other as rows. All objects must have the same number of columns and matching column names. IONOS states that the rbind() function is useful for combining data line by line, often used to add new information to an existing data frame

R

# Example data for combining

additional\_feedback <- data.frame(

OrderID = c(1006, 1007),

FeedbackText = c("Fast delivery, great product.", "Customer service was unhelpful."),

Rating = c(4, 2),

Sentiment = c("Positive", "Negative"),

IsVerified = c(TRUE, FALSE)

)

# Using rbind to add new rows to the existing customer feedback data frame

updated\_feedback\_df <- rbind(customer\_feedback\_df, additional\_feedback)

print(updated\_feedback\_df)

# Creating a separate data frame with additional details (e.g., product ID)

product\_details <- data.frame(

OrderID = c(1001, 1002, 1003, 1004, 1005, 1006, 1007),

ProductID = c("P001", "P002", "P001", "P003", "P002", "P001", "P004")

)

# Using cbind to add product details (assuming OrderIDs are aligned)

# For more robust combining with non-aligned data, use merge() or join functions (Chapter 5)

combined\_data\_cbind <- cbind(updated\_feedback\_df, ProductID = product\_details$ProductID)

print(combined\_data\_cbind)

## 4.4.2 Reshaping Data with pivot\_wider()

While pivot\_longer() converts data from a wide format to a long format, pivot\_wider() performs the opposite operation. It takes data in a long format (where key-value pairs are stored across rows) and spreads it out into a wider format, increasing the number of columns and decreasing the number of rows. This is particularly useful when analyzing or visualizing data that needs to be summarized by categories or when comparing values across different groups.

To illustrate pivot\_wider(), let's create a "long" version of our customer feedback data that records multiple feedback types for each customer over time.

First, create an example dataset that includes different feedback metrics (like 'Rating' and 'Sentiment\_Score') for various customer interactions, along with the customer\_id and interaction\_date.

R

library(tidyverse)

# Create a sample long-format dataset for customer feedback

long\_feedback\_data <- tibble(

customer\_id = c(1, 1, 2, 2, 3, 3),

interaction\_date = as.Date(c("2023-01-15", "2023-01-20", "2023-02-10", "2023-02-12", "2023-03-01", "2023-03-05")),

metric = c("Rating", "Sentiment\_Score", "Rating", "Sentiment\_Score", "Rating", "Sentiment\_Score"),

value = c(4, 0.8, 3, 0.5, 5, 0.9)

)

print(long\_feedback\_data)

Explanation: In this long\_feedback\_data tibble, each row represents a specific metric (Rating or Sentiment\_Score) measured for a particular customer on a given interaction date. This format is useful for some analyses, but it's not ideal if you want to see the Rating and Sentiment\_Score side-by-side for each customer interaction.

To achieve this, pivot\_wider() can be used. It requires two main arguments:

* names\_from: The column whose unique values will become the new column names in the wider format.
* values\_from: The column that contains the values to populate the new wide-format columns.

Let's use pivot\_wider() to transform long\_feedback\_data:

R

# Pivot the long data to a wider format

wide\_feedback\_data <- long\_feedback\_data %>%

pivot\_wider(

names\_from = metric, # Take the values from the 'metric' column to create new column names

values\_from = value # Fill the new columns with values from the 'value' column

)

print(wide\_feedback\_data)

Explanation: The pivot\_wider() function reshaped the data frame. Now, Rating and Sentiment\_Score are separate columns, and each row represents a unique customer interaction with both metrics present. This wide format is often more intuitive for direct comparison or for some statistical models that expect each variable in its own column.

Handling Multiple Value Columns

Sometimes, you might have multiple value columns that need to be spread across. pivot\_wider() can handle this by specifying multiple columns in values\_from.

Consider a scenario where the customer feedback data also includes 'Review\_Length' in addition to 'Rating':

R

# A long-format dataset with multiple value columns

long\_feedback\_metrics <- tibble(

customer\_id = c(1, 1, 2, 2, 3, 3),

interaction\_date = as.Date(c("2023-01-15", "2023-01-15", "2023-02-10", "2023-02-10", "2023-03-01", "2023-03-01")),

metric = c("Rating", "Review\_Length", "Rating", "Review\_Length", "Rating", "Review\_Length"),

value = c(4, 50, 3, 30, 5, 120) # 'value' holds both rating and review length

)

print(long\_feedback\_metrics)

# This data is not truly tidy because 'value' represents two different things.

# However, for demonstration of pivot\_wider, let's assume it's like this for a moment.

In a more realistic scenario, there would be separate value columns for Rating and Review\_Length if they were collected separately. If you had data that looked like this:

R

# Better representation for multiple measures

another\_long\_feedback <- tibble(

customer\_id = c(1, 1, 2, 2),

interaction\_date = as.Date(c("2023-01-15", "2023-01-15", "2023-02-10", "2023-02-10")),

variable = c("Rating", "Review\_Length", "Rating", "Review\_Length"),

measurement = c(4, 50, 3, 30)

) %>%

pivot\_wider(names\_from = variable, values\_from = measurement)

print(another\_long\_feedback)

# Now, imagine we added a 'Source' column to this wide data

another\_long\_feedback\_with\_source <- tibble(

customer\_id = c(1, 1, 2, 2),

interaction\_date = as.Date(c("2023-01-15", "2023-01-15", "2023-02-10", "2023-02-10")),

feedback\_type = c("Rating", "Review\_Length", "Rating", "Review\_Length"),

value\_1 = c(4, 50, 3, 30),

value\_2 = c("App", "Web", "App", "Web") # Imagine value\_2 is the source

)

print(another\_long\_feedback\_with\_source)

# To pivot this, you might need to use `names\_glue` or combine columns

# For simplicity, let's stick to the simpler pivot\_wider() for now,

# but it's important to be aware of such complexities.

Handling Missing Values with values\_fill

When pivoting, if a combination of names\_from and id\_cols doesn't exist for a particular value\_from, pivot\_wider() will by default fill the corresponding cell with NA (Not Available). You can specify a different fill value using the values\_fill argument. [You can learn more about handling missing values at tidyr's pivot vignette](https://www.google.com/url?sa=i&source=web&rct=j&url=https://cran.r-project.org/web/packages/tidyr/vignettes/pivot.html&ved=2ahUKEwjx5pW617yOAxWQl4kEHcYkMqoQy_kOegYIAwgAECk&opi=89978449&cd&psig=AOvVaw2kI6jD22NVhWhNGEB0KeuT&ust=1752593971709000).

R

# Create a dataset with some implicit missing values

sparse\_feedback <- tibble(

customer\_id = c(1, 1, 2),

metric = c("Rating", "Sentiment\_Score", "Rating"),

value = c(4, 0.8, 3)

)

print(sparse\_feedback)

# Pivot with default NA fill

wide\_sparse\_feedback\_na <- sparse\_feedback %>%

pivot\_wider(names\_from = metric, values\_from = value)

print(wide\_sparse\_feedback\_na)

# Pivot with a specific fill value (e.g., 0)

wide\_sparse\_feedback\_filled <- sparse\_feedback %>%

pivot\_wider(names\_from = metric, values\_from = value, values\_fill = 0)

print(wide\_sparse\_feedback\_filled)

Explanation: In the first pivot, customer 2 has no Sentiment\_Score recorded, so it's filled with NA. In the second pivot, values\_fill = 0 ensures that missing Sentiment\_Score for customer 2 is replaced with 0, potentially indicating no recorded sentiment or a neutral sentiment depending on context.

Combining pivot\_longer() and pivot\_wider() for Complex Reshaping

The true power of these functions often becomes apparent when they are used together to perform more complex data transformations. You might find yourself pivoting data to a long format for easier manipulation, then pivoting it back to a wide format for a specific analysis or presentation.

For example, if you had multiple sets of measurements encoded in a wide format, you could use pivot\_longer() to tidy the data, perform calculations, and then use pivot\_wider() to get it into a format required by another tool or for reporting purposes.

4.4.3 Evolving our Running Example with pivot\_longer() and pivot\_wider()

Returning to the "Customer Feedback Analyzer," imagine that instead of having just a product\_rating and sentiment column, you have yearly ratings from repeat customers in separate columns for each year:

R

# Create a wider version of the customer feedback data for demonstration

customer\_yearly\_feedback <- tibble(

customer\_id = c(101, 102, 103),

name = c("Alice", "Bob", "Charlie"),

rating\_2022 = c(4, 3, 5),

rating\_2023 = c(5, 4, NA), # Charlie didn't provide feedback in 2023

sentiment\_2022 = c("Positive", "Neutral", "Positive"),

sentiment\_2023 = c("Positive", "Neutral", NA)

)

print(customer\_yearly\_feedback)

This format is useful for seeing annual ratings side-by-side but not ideal for analyzing rating trends over time using visualization tools like ggplot2 (which we'll explore in Chapter 6). For that, you'd want a "longer" format.

First, let's use pivot\_longer() to gather the rating\_ and sentiment\_ columns:

R

# Load the tidyverse if not already loaded

library(tidyverse)

# Pivot to a longer format

long\_customer\_feedback <- customer\_yearly\_feedback %>%

pivot\_longer(

cols = starts\_with("rating\_") | starts\_with("sentiment\_"), # Select columns to pivot

names\_to = c(".value", "year"), # Split column names into 'metric' and 'year'

names\_sep = "\_" # Use '\_' as the separator for splitting names

)

print(long\_customer\_feedback)

Explanation:

* cols = starts\_with("rating\_") | starts\_with("sentiment\_"): This selects all columns starting with "rating\_" or "sentiment\_".
* names\_to = c(".value", "year"): This is a powerful feature where .value indicates that part of the original column name (rating or sentiment) should become the new column name (i.e., rating and sentiment will become new columns) and year becomes a new column holding the year values.
* names\_sep = "\_": This specifies that the underscore character separates the metric and year components in the original column names. More on using names\_sep to separate values into multiple columns is available here.

The long\_customer\_feedback data frame is now much tidier. Each row represents a single observation (a customer's feedback for a specific year), and each variable (customer ID, name, year, rating, sentiment) has its own column. This structure is excellent for plotting trends over time, like how a customer's average rating changed from 2022 to 2023.

Now, imagine a scenario where after analyzing the long\_customer\_feedback, a presentation wants to compare Alice's 2022 rating against her 2023 rating in a single row for a report. This requires pivoting back to a wider format using pivot\_wider():

R

# Pivot the long data back to a wider format (e.g., for reporting)

comparison\_wide\_feedback <- long\_customer\_feedback %>%

pivot\_wider(

names\_from = year, # Years become new column names

values\_from = c(rating, sentiment), # Both 'rating' and 'sentiment' values are spread

names\_glue = "{.value}\_{year}" # Defines how new column names are constructed

)

print(comparison\_wide\_feedback)

Explanation:

* names\_from = year: The unique values from the year column (2022, 2023) will become new column names.
* values\_from = c(rating, sentiment): Both the rating and sentiment columns will have their values spread into the new wider columns.
* names\_glue = "{.value}\_{year}": This argument specifies how the new column names are created. .value refers to the names from values\_from (i.e., rating and sentiment), and year refers to the names from names\_from. This results in column names like rating\_2022, sentiment\_2022, rating\_2023, sentiment\_2023.

This transformed data frame is again in a wide format, but now it's focused on comparing annual metrics side-by-side for each customer, which could be beneficial for certain reporting tasks.

## Wrap-up

Reshaping data with pivot\_longer() and pivot\_wider() from the tidyr package is an essential skill for any R user. As noted by Steven P. Sanderson II, MPH, these functions allow the transformation of data from wide to long and vice-versa, making it suitable for analysis and visualization. pivot\_longer() is used to gather columns into rows, typically creating key-value pairs, which is ideal for tidying data and preparing it for functions that expect long formats (like ggplot2). pivot\_wider() is used to spread rows into columns, which is useful for creating summary tables or preparing data for specific analytical tasks. Mastering these tools empowers flexible data manipulation, ensuring the data is in the optimal format for any task.

In the next chapter, Chapter 5, we will delve deeper into the tidyverse ecosystem, exploring more powerful functions for filtering, sorting, grouping, and summarizing data, further enhancing the ability to analyze and extract insights from the "Customer Feedback Analyzer" dataset.

# Chapter 5: Data Wrangling with Tidyverse

You've learned the fundamentals of R programming and basic data manipulation. Now, it's time to supercharge your data wrangling skills with the tidyverse, a powerful collection of R packages designed for data science. The tidyverse packages work together seamlessly, following consistent design principles that make data transformation intuitive and efficient. This chapter focuses on the core tidyverse packages that will become your daily tools: dplyr, tidyr, stringr, and readr.

## 5.1 Intro to dplyr, tidyr, stringr, readr

The tidyverse isn't a single package but a family of packages. When you install and load tidyverse using install.packages("tidyverse") and library(tidyverse), you're loading a core set of packages that includes the ones covered in this chapter, among others like ggplot2 (for visualization) and tibble (a modern version of data frames). According to Study.com, the tidyverse is a collection of packages that help to import, organize, manipulate, and visualize data.

* dplyr (pronounced "dee-ply-er"): This package provides a grammar of data manipulation, offering a consistent set of "verbs" (functions) to perform common data transformations like filtering rows, selecting columns, creating new variables, and summarizing data.
* tidyr (pronounced "tie-dee-er"): This package focuses on tidying data, which means making it "long" or "wide" as needed. You encountered pivot\_longer() and pivot\_wider() in Chapter 4, which are key tidyr functions.
* stringr: This package simplifies working with strings (character data), making it easier to clean, extract, and manipulate text. Study.com notes that stringr is used for data preparation and to simplify working with strings.
* readr: This package provides fast and user-friendly functions for reading rectangular data (like CSV and TSV files) into R. It's often preferred over base R's read.csv() due to its speed and consistent output (tibbles). According to Read Rectangular Text Data, readr is designed to parse many types of data found in the wild while providing informative problem reports when parsing leads to unexpected results.

Throughout this chapter, the pipe operator (%>%) from the magrittr package (loaded with tidyverse) will be extensively used. This operator makes code more readable by passing the result of one function directly to the next, chaining operations together logically. OARC Stats reports that the tidyverse package helps in data import, management, and visualization, with dplyr, tidyr, magrittr, lubridate, and stringr being its key components.

## 5.2 Filtering, sorting, grouping, and summarizing with dplyr

dplyr provides a powerful and intuitive set of functions to manipulate data frames (or more specifically, tibbles). These are often referred to as "verbs" because they describe actions you perform on your data.

5.2.1 Filtering Rows with filter()

The filter() function allows you to select rows (observations) based on one or more conditions. It works similar to subsetting with [] but offers a cleaner syntax, especially with the pipe operator.

Syntax: dataframe %>% filter(condition1, condition2, ...)

R

library(tidyverse) # Ensure tidyverse is loaded

# Let's use our long\_customer\_feedback from Chapter 4

long\_customer\_feedback <- tibble(

customer\_id = c(101, 102, 103, 101, 102, 103),

name = c("Alice", "Bob", "Charlie", "Alice", "Bob", "Charlie"),

year = c(2022, 2022, 2022, 2023, 2023, 2023),

rating = c(4, 3, 5, 5, 4, NA),

sentiment = c("Positive", "Neutral", "Positive", "Positive", "Neutral", NA)

)

# Filter for feedback from a specific customer (e.g., Alice, customer\_id = 101)

alice\_feedback <- long\_customer\_feedback %>%

filter(customer\_id == 101)

print(alice\_feedback)

# Filter for feedback with a rating of 5 AND from the year 2023

excellent\_2023\_feedback <- long\_customer\_feedback %>%

filter(rating == 5, year == 2023) # Separate conditions with commas for AND

print(excellent\_2023\_feedback)

# Filter for feedback that is either "Positive" OR has a rating of 5

positive\_or\_excellent <- long\_customer\_feedback %>%

filter(sentiment == "Positive" | rating == 5) # Use | for OR

print(positive\_or\_excellent)

# Filter out rows where the rating is missing (NA)

feedback\_with\_ratings <- long\_customer\_feedback %>%

filter(!is.na(rating))

print(feedback\_with\_ratings)

5.2.2 Sorting Rows with arrange()

The arrange() function allows you to reorder the rows of your data frame based on the values in one or more columns.

Syntax: dataframe %>% arrange(column1, column2, ...)

R

# Arrange feedback by year in ascending order

feedback\_by\_year <- long\_customer\_feedback %>%

arrange(year)

print(feedback\_by\_year)

# Arrange feedback by customer ID (ascending) and then by rating (descending)

arranged\_feedback <- long\_customer\_feedback %>%

arrange(customer\_id, desc(rating)) # Use desc() for descending order

print(arranged\_feedback)

5.2.3 Grouping Data with group\_by() and Summarizing with summarize()

The "split-apply-combine" strategy is fundamental in data analysis. The data are split into groups, a function is applied to each group, and the results are then combined. Karl Broman mentions that many data analysis tasks can be approached using the “split-apply-combine” paradigm: split the data into groups, apply some analysis to each group, and then combine the results. dplyr makes this easier with group\_by() and summarize().

* group\_by(): This function groups data by one or more categorical variables. Subsequent operations (like summarize()) will be applied to each group independently.
* summarize() (or summarise()): This function collapses each group into a single row summary, calculating aggregate statistics (like mean, sum, count, min, max) for each group.

Syntax: dataframe %>% group\_by(grouping\_variable) %>% summarize(new\_column = aggregate\_function(variable))

R

# Calculate the average rating for each year

average\_rating\_by\_year <- long\_customer\_feedback %>%

group\_by(year) %>%

summarize(mean\_rating = mean(rating, na.rm = TRUE)) # na.rm = TRUE handles NA values

print(average\_rating\_by\_year)

# Calculate the number of feedback entries and average rating for each customer

customer\_summary <- long\_customer\_feedback %>%

group\_by(customer\_id, name) %>% # Group by multiple columns

summarize(

total\_feedback = n(), # n() counts the number of rows in each group

avg\_rating = mean(rating, na.rm = TRUE),

min\_rating = min(rating, na.rm = TRUE),

max\_rating = max(rating, na.rm = TRUE)

)

print(customer\_summary)

# Let's count how many positive, neutral, negative sentiments there are per year

sentiment\_counts\_by\_year <- long\_customer\_feedback %>%

filter(!is.na(sentiment)) %>% # Exclude NA sentiments

group\_by(year, sentiment) %>%

summarize(count = n()) %>%

arrange(year, desc(count))

print(sentiment\_counts\_by\_year)

5.2.4 Creating/Modifying Columns with mutate()

The mutate() function allows adding new columns to a data frame or modifying existing ones based on calculations using other columns.

Syntax: dataframe %>% mutate(new\_column = expression, another\_new\_column = another\_expression, ...)

R

# Add a new column indicating if the rating is above average

feedback\_with\_flag <- long\_customer\_feedback %>%

mutate(is\_above\_average = rating > mean(rating, na.rm = TRUE))

print(feedback\_with\_flag)

# Create a new column combining customer ID and year

feedback\_with\_id\_year <- long\_customer\_feedback %>%

mutate(customer\_year\_id = paste0("Cust", customer\_id, "\_", year))

print(feedback\_with\_id\_year)

# Let's create a categorized rating based on the numerical rating

categorized\_feedback <- long\_customer\_feedback %>%

mutate(

rating\_category = case\_when(

rating == 5 ~ "Excellent",

rating == 4 ~ "Good",

rating == 3 ~ "Average",

rating < 3 ~ "Poor",

TRUE ~ NA\_character\_ # Handle NA ratings and other cases

)

) %>%

arrange(customer\_id, year)

print(categorized\_feedback)

5.2.5 Selecting Columns with select()

The select() function allows you to choose specific columns or exclude them from your data frame.

Syntax: dataframe %>% select(column1, column2, ...) or dataframe %>% select(-column\_to\_exclude)

R

# Select only the customer ID, year, and rating columns

selected\_cols <- long\_customer\_feedback %>%

select(customer\_id, year, rating)

print(selected\_cols)

# Select all columns EXCEPT the 'name' column

all\_except\_name <- long\_customer\_feedback %>%

select(-name)

print(all\_except\_name)

# Select a range of columns

range\_selection <- long\_customer\_feedback %>%

select(customer\_id:year) # Select columns from customer\_id to year, inclusively

print(range\_selection)

# Select columns based on patterns (using helper functions)

pattern\_selection <- long\_customer\_feedback %>%

select(starts\_with("cust"), ends\_with("t")) # Select columns starting with 'cust' or ending with 't'

print(pattern\_selection)

## 5.3 Cleaning text data and missing values with stringr and tidyr

Real-world customer feedback often comes messy: inconsistent capitalization, extra spaces, special characters, and missing values. The stringr and tidyr packages are invaluable for cleaning these issues.

5.3.1 Text Cleaning with stringr

The stringr package provides a consistent and simple interface for common string manipulation tasks.

Let's imagine the raw feedback text contains inconsistencies:

R

raw\_feedback\_text <- c(

" Excellent product, fast delivery! ",

"Average experience. The item was okay.",

"DISAPPOINTED with the quality.",

"Great value for money (Highly Recommended)",

"Buggy software. Needs improvements."

)

raw\_feedback\_df <- tibble(feedback = raw\_feedback\_text)

print(raw\_feedback\_df)

# Remove leading/trailing whitespace

cleaned\_whitespace <- raw\_feedback\_df %>%

mutate(feedback\_cleaned = str\_trim(feedback))

print(cleaned\_whitespace)

# Convert to consistent casing (e.g., lowercase)

cleaned\_case <- cleaned\_whitespace %>%

mutate(feedback\_cleaned = str\_to\_lower(feedback\_cleaned))

print(cleaned\_case)

# Remove specific patterns (e.g., text in parentheses)

cleaned\_patterns <- cleaned\_case %>%

mutate(feedback\_cleaned = str\_remove\_all(feedback\_cleaned, "\\(.\*?\\)")) # Removes text within parentheses

print(cleaned\_patterns)

# You can chain these operations together

fully\_cleaned\_feedback <- raw\_feedback\_df %>%

mutate(

feedback\_cleaned = feedback %>%

str\_trim() %>%

str\_to\_lower() %>%

str\_remove\_all("\\(.\*?\\)")

)

print(fully\_cleaned\_feedback)

Explanation:

* str\_trim(): Removes whitespace from the beginning and end of strings.
* str\_to\_lower(): Converts strings to all lowercase. str\_to\_upper() is also available.
* str\_remove\_all(): Removes all occurrences of a specified pattern. The pattern "\\(.\*?\\)" uses regular expressions to match any text enclosed in parentheses. Regular expressions are a powerful tool for pattern matching in text, but they have their own syntax. We'll touch on them more in Chapter 8.

5.3.2 Handling Missing Values with tidyr

Missing data (represented as NA) is a common challenge. tidyr provides functions to deal with them systematically.

R

data\_with\_na <- tibble(

customer\_id = c(101, 102, 103, 104),

rating = c(5, 3, NA, 4),

comment = c("Good", "Okay", NA, "Excellent")

)

print(data\_with\_na)

# Drop rows with any NA values

complete\_cases <- data\_with\_na %>%

drop\_na()

print(complete\_cases)

# Drop rows with NA only in specific columns (e.g., 'rating')

complete\_ratings <- data\_with\_na %>%

drop\_na(rating)

print(complete\_ratings)

# Replace NA values with a specified value (e.g., 0 for ratings, "No comment" for text)

filled\_na <- data\_with\_na %>%

replace\_na(list(rating = 0, comment = "No comment"))

print(filled\_na)

# Fill missing values from the previous or next observation (useful for sequential data)

# Example: If a customer's rating is missing, fill it with their previous rating

sequential\_data <- tibble(

customer\_id = c(101, 101, 102, 102, 102),

year = c(2022, 2023, 2021, 2022, 2023),

rating = c(4, NA, 3, 5, NA)

) %>%

arrange(customer\_id, year) # Important to sort first for fill()

print(sequential\_data)

filled\_sequential <- sequential\_data %>%

group\_by(customer\_id) %>% # Fill separately for each customer

fill(rating, .direction = "down") # Fill NA ratings downwards

print(filled\_sequential)

filled\_sequential\_up <- sequential\_data %>%

group\_by(customer\_id) %>%

fill(rating, .direction = "up") # Fill NA ratings upwards

print(filled\_sequential\_up)

Explanation:

* drop\_na(): Removes rows containing NA values. You can specify which columns to check for NAs.
* replace\_na(): Replaces NA values with a given value. It's often used with a list when replacing NAs in multiple columns.
* fill(): Fills NA values in selected columns using the next or previous entry. This is useful for data where values are implicitly carried forward or backward. Remember to group\_by() relevant categorical variables (like customer\_id) before using fill() to ensure values are filled within groups and not across them.

## 5.4 Evolving our running example with real-world CSVs

The "Customer Feedback Analyzer" example can now be expanded by importing realistic customer feedback data. Instead of manually creating tibbles, the readr package allows reading data directly from files, such as CSVs.

Creating a Sample CSV File

To follow along, first create a CSV file named customer\_feedback.csv and save it in your RStudio project's working directory.

1. Open a text editor (like Notepad, Sublime Text, or RStudio's text editor).
2. Paste the following data:

csv

customer\_id,rating,comment,feedback\_source,timestamp

1001,5,"Excellent product! Fast delivery.","Web Form","2023-01-15 10:30:00"

1002,3,"Average experience. The item was okay.","Email","2023-01-16 14:00:00"

1003,2,"Disappointed with the quality.","App Review","2023-01-17 09:15:00"

1004,5,"Great value for money. Highly recommend.","Web Form","2023-01-18 11:45:00"

1005,4,"Satisfied with the purchase.","Email","2023-01-19 16:20:00"

1006,1,"Buggy software. Needs improvements.","App Review","2023-01-20 08:00:00"

1007,5,"Fantastic support team!","Phone Call","2023-01-21 13:00:00"

1008,3,"Product was okay, but delivery was slow.","Web Form","2023-01-22 17:05:00"

1009,NA,"Customer did not leave a rating.","Email","2023-01-23 09:30:00"

1010,4,"Good, but missing a feature.","Web Form","2023-01-24 10:00:00"

1. Save the file as customer\_feedback.csv in the same directory where your R script is located (or in a data subfolder, if you prefer, and adjust the path accordingly).

Importing Data with readr::read\_csv()

The read\_csv() function from the readr package is the go-to for importing comma-separated values. It automatically detects column types and provides helpful diagnostics.

R

library(tidyverse) # Loads readr as well

# Read the CSV file into an R data frame (tibble)

feedback\_data\_raw <- read\_csv("customer\_feedback.csv")

print(feedback\_data\_raw)

Explanation:  
When you run read\_csv(), it will display a message in the console indicating how it parsed the data, showing the detected column types (e.g., col\_double(), col\_character(), col\_datetime()). According to readr's documentation, it is designed to parse many types of data found in the wild while providing informative problem reports when parsing leads to unexpected results. This is a helpful feature of readr, allowing you to verify that the data types were imported as expected.

readr typically detects column types like col\_double, col\_character, and col\_datetime(). While it is possible to override these, the defaults are usually sufficient.

Basic Exploration and Initial Cleaning of the Imported Data

After loading the data, it is necessary to begin with initial cleaning and exploration using the dplyr and tidyr functions.

R

# View the structure of the imported data

glimpse(feedback\_data\_raw)

# Perform initial cleaning and transformation steps

cleaned\_feedback <- feedback\_data\_raw %>%

# 1. Standardize text data in the 'comment' column

mutate(

comment = comment %>%

str\_trim() %>% # Remove leading/trailing whitespace

str\_to\_lower() # Convert to lowercase for consistency

) %>%

# 2. Fill missing ratings (NA) with a default value, e.g., 0 or the mean

# Here, we'll replace NA ratings with 3 (representing "Average" or imputed neutral)

replace\_na(list(rating = 3)) %>%

# 3. Create a new column indicating whether the feedback is 'Positive', 'Negative', or 'Neutral'

mutate(

sentiment\_category = case\_when(

rating >= 4 ~ "Positive",

rating == 3 ~ "Neutral",

rating < 3 ~ "Negative",

TRUE ~ "Unknown" # Should not happen after NA replacement, but good practice

)

) %>%

# 4. Select and reorder columns for clarity

select(customer\_id, timestamp, feedback\_source, rating, sentiment\_category, comment) %>%

# 5. Arrange data by timestamp

arrange(timestamp)

print(cleaned\_feedback)

Explanation:  
This code demonstrates a data cleaning process:

1. Whitespace is removed and text in the comment column is converted to lowercase.
2. Missing rating values are replaced with 3.
3. A sentiment\_category is created based on the rating.
4. Columns are selected and reordered.
5. Data is sorted by timestamp.

Wrap-up

In this chapter, you have learned to use tidyverse packages like dplyr, tidyr, stringr, and readr for data manipulation, cleaning, and importing real-world data like the customer feedback dataset. This provides a solid base for data analysis. Chapter 6 will focus on visualizing this data using ggplot2.

# Chapter 6: Data Visualization

Data visualization is a crucial step in any data analysis workflow. It transforms raw data into easily digestible visual representations, revealing patterns, trends, and outliers that might be hidden in tables or summaries. In R, the ggplot2 package, part of the tidyverse, is the most popular and powerful tool for creating stunning and informative graphics. This chapter dives into the world of ggplot2, exploring its core principles and demonstrating how to create various plot types essential for analyzing and presenting customer feedback data. R for Data Science emphasizes that data visualization is a crucial component of data analysis.

## 6.1 Intro to ggplot2: grammar of graphics

ggplot2 is based on the Grammar of Graphics, a powerful framework developed by Leland Wilkinson that allows you to build any plot by combining independent components. Rather than being limited to predefined plots, you construct graphics layer by layer, mapping data variables to visual aesthetics.

The basic template for a ggplot2 plot involves these core components:

* ggplot(): Initializes a plot object and specifies the data frame to be used.
* aes() (Aesthetics): Defines how variables from your data are mapped to visual properties of the plot, such as the x-axis position, y-axis position, color, size, shape, or transparency.
* geom\_ (Geometrics): Specifies the geometric objects (layers) used to represent the data, such as points (geom\_point() for scatter plots), bars (geom\_bar() for bar charts), lines (geom\_line() for line graphs), or boxes (geom\_boxplot() for box plots).

The power of ggplot2 comes from adding these layers using the + operator, creating a plot step-by-step.

Let's use the cleaned\_feedback data frame created in Chapter 5 to illustrate these concepts.

R

library(tidyverse) # Ensure tidyverse (including ggplot2) is loaded

# Recreate cleaned\_feedback for continuity if not in current session

# You should have saved customer\_feedback.csv in your working directory.

feedback\_data\_raw <- read\_csv("customer\_feedback.csv")

cleaned\_feedback <- feedback\_data\_raw %>%

mutate(

comment = comment %>%

str\_trim() %>%

str\_to\_lower()

) %>%

replace\_na(list(rating = 3)) %>%

mutate(

sentiment\_category = case\_when(

rating >= 4 ~ "Positive",

rating == 3 ~ "Neutral",

rating < 3 ~ "Negative",

TRUE ~ "Unknown"

)

) %>%

select(customer\_id, timestamp, feedback\_source, rating, sentiment\_category, comment) %>%

arrange(timestamp)

print(cleaned\_feedback)

# Basic ggplot2 structure:

# Start by initializing the ggplot object with the data frame

# Then map variables to aesthetics within aes()

# Finally, add a geometric layer to specify how the data is plotted

# Example: An empty plot with data and aesthetics defined (no geom yet)

# This will show a blank canvas

ggplot(data = cleaned\_feedback, aes(x = rating, y = customer\_id))

# To make it visible, we need to add a geom

## 6.2 Bar charts, histograms, boxplots, scatter plots

ggplot2 provides specific geom\_ functions for common chart types. This section demonstrates some of the most frequently used ones, applying them to the customer feedback data. STHDA outlines how ggplot2 creates bar charts, which display the relationship between a numeric and a categorical variable.

6.2.1 Bar Charts

Bar charts are useful for displaying the distribution of categorical variables or comparing numerical values across different categories.

Use Case: Visualizing the count of feedback entries by feedback\_source or sentiment\_category.

R

# Bar chart of feedback sources

bar\_source <- cleaned\_feedback %>%

ggplot(aes(x = feedback\_source)) +

geom\_bar(fill = "steelblue") # geom\_bar automatically counts observations by default

print(bar\_source)

# Bar chart of sentiment categories

bar\_sentiment <- cleaned\_feedback %>%

ggplot(aes(x = sentiment\_category, fill = sentiment\_category)) + # Map 'sentiment\_category' to fill color

geom\_bar() +

labs(title = "Distribution of Customer Sentiments") # Add a title

print(bar\_sentiment)

# Bar chart showing average rating by feedback source (requires geom\_col with stat = "identity")

# First, calculate the average rating per source

avg\_rating\_by\_source <- cleaned\_feedback %>%

group\_by(feedback\_source) %>%

summarize(mean\_rating = mean(rating, na.rm = TRUE))

# Now plot the average rating

bar\_avg\_rating <- avg\_rating\_by\_source %>%

ggplot(aes(x = feedback\_source, y = mean\_rating, fill = feedback\_source)) +

geom\_col() + # geom\_col expects pre-calculated y-values

labs(

title = "Average Rating by Feedback Source",

y = "Average Rating",

x = "Feedback Source"

)

print(bar\_avg\_rating)

6.2.2 Histograms

Histograms are used to visualize the distribution of a single numerical variable by dividing the data into bins and counting the number of observations in each bin. [Appsilon provides examples of how to make stunning histograms in R](https://www.google.com/url?sa=i&source=web&rct=j&url=https://www.appsilon.com/post/ggplot2-histograms&ved=2ahUKEwjg2dCd3LyOAxWQSTABHS1-IgwQy_kOegYIAwgAECQ&opi=89978449&cd&psig=AOvVaw1kINU7eUbDDRsyFTkJOZSL&ust=1752595254091000).

Use Case: Understanding the distribution of rating values.

R

# Histogram of product ratings

hist\_rating <- cleaned\_feedback %>%

ggplot(aes(x = rating)) +

geom\_histogram(binwidth = 1, fill = "darkgreen", color = "white") + # Specify binwidth

labs(

title = "Distribution of Customer Ratings",

x = "Rating (1-5)",

y = "Count"

)

print(hist\_rating)

# Histograms are sensitive to binwidth. Experiment to find what best tells the story.

# For example, wider bins might mask details, while narrower ones might show noise.

6.2.3 Boxplots

Boxplots display the distribution of a numeric variable across different categories, showing the median, quartiles, and potential outliers. You can learn how to create and customize a box plot using the ggplot2 package in R.

Use Case: Comparing the distribution of rating across different feedback\_source categories.

R

# Boxplot of ratings by feedback source

box\_rating\_source <- cleaned\_feedback %>%

ggplot(aes(x = feedback\_source, y = rating, fill = feedback\_source)) +

geom\_boxplot() +

labs(

title = "Rating Distribution by Feedback Source",

x = "Feedback Source",

y = "Rating"

)

print(box\_rating\_source)

6.2.4 Scatter Plots

Scatter plots are used to visualize the relationship between two numerical variables.

Use Case: While the current dataset doesn't have two continuous numerical variables for a perfect scatter plot example, we can illustrate by converting one variable or exploring potential relationships if we had more detailed data (e.g., time\_spent\_on\_website vs rating). For demonstration, we can create a proxy using timestamp and rating if treating timestamp as a continuous measure of time progression.

R

# Scatter plot of rating over time

scatter\_time\_rating <- cleaned\_feedback %>%

ggplot(aes(x = timestamp, y = rating, color = sentiment\_category)) + # Map sentiment to color

geom\_point(alpha = 0.7) + # Add transparency to points

labs(

title = "Customer Ratings Over Time",

x = "Date and Time of Feedback",

y = "Rating"

)

print(scatter\_time\_rating)

# Adding a trend line

scatter\_time\_rating\_trend <- cleaned\_feedback %>%

ggplot(aes(x = timestamp, y = rating)) +

geom\_point(alpha = 0.7, aes(color = sentiment\_category)) +

geom\_smooth(method = "lm", se = FALSE, color = "black") + # Add a linear model trend line

labs(

title = "Customer Ratings Over Time with Trend",

x = "Date and Time of Feedback",

y = "Rating"

)

print(scatter\_time\_rating\_trend)

## 6.3 Customizing labels, themes, and aesthetics

ggplot2 allows extensive customization to make plots informative, visually appealing, and tailored to specific needs.

6.3.1 Adding Labels and Titles with labs()

The labs() function is used to set plot titles, subtitles, captions, and axis labels.

R

custom\_labels\_plot <- cleaned\_feedback %>%

ggplot(aes(x = sentiment\_category, fill = sentiment\_category)) +

geom\_bar() +

labs(

title = "Customer Sentiment Breakdown by Category",

subtitle = "Based on Ratings from Customer Feedback Data",

caption = "Data collected January 2023",

x = "Sentiment Category",

y = "Number of Customers"

)

print(custom\_labels\_plot)

6.3.2 Customizing Aesthetics

Aesthetics can be set globally for a geom\_ or mapped to variables within aes().

R

# Map color to feedback\_source for the scatter plot

scatter\_colored\_by\_source <- cleaned\_feedback %>%

ggplot(aes(x = timestamp, y = rating, color = feedback\_source)) + # 'color' mapped to 'feedback\_source'

geom\_point(size = 3, alpha = 0.8) + # Set point size and transparency globally

labs(title = "Ratings Over Time, Colored by Source")

print(scatter\_colored\_by\_source)

# Change point shape and size based on a variable

custom\_shape\_size <- cleaned\_feedback %>%

ggplot(aes(x = timestamp, y = rating, color = sentiment\_category, shape = feedback\_source)) +

geom\_point(size = 3) + # You can also map size to a numeric variable

labs(title = "Ratings Over Time, Colored by Sentiment, Shaped by Source")

print(custom\_shape\_size)

6.3.3 Using and Customizing Themes

Themes control the non-data components of a plot, like background color, fonts, gridlines, and legend appearance. R-Statistics.co provides a guide on customizing ggplot2 themes.

ggplot2 comes with several built-in themes (e.g., theme\_minimal(), theme\_bw(), theme\_classic()) that provide a quick way to change the plot's overall appearance.

R

# Apply a built-in theme (e.g., theme\_minimal)

minimal\_theme\_plot <- custom\_labels\_plot +

theme\_minimal()

print(minimal\_theme\_plot)

# Customize theme elements using theme()

custom\_theme\_plot <- custom\_labels\_plot +

theme\_bw() + # Start with a base theme

theme(

plot.title = element\_text(size = 16, face = "bold", hjust = 0.5), # Center and bold title

axis.title = element\_text(size = 12, face = "italic"), # Italicize axis titles

legend.position = "bottom", # Move legend to bottom

panel.grid.major = element\_line(linetype = "dashed", color = "lightgray"), # Customize gridlines

plot.background = element\_rect(fill = "honeydew") # Change background color

)

print(custom\_theme\_plot)

## 6.4 Visual storytelling with customer feedback

The goal of data visualization isn't just to make plots, but to tell a story about the data. Using the customer feedback data, let's create a visualization that highlights key insights. Our Coding Club suggests that data visualization can be used for storytelling with data.

Story: Customers using "App Review" are significantly more likely to leave negative feedback compared to those using "Web Form."

To investigate this, a bar chart can be used that shows the sentiment breakdown by feedback\_source.

R

# Calculate the percentage of each sentiment category per feedback source

sentiment\_proportions <- cleaned\_feedback %>%

group\_by(feedback\_source, sentiment\_category) %>%

summarize(count = n()) %>%

group\_by(feedback\_source) %>% # Group again to calculate percentages within each source

mutate(proportion = count / sum(count)) %>%

ungroup() # Ungroup for subsequent operations

print(sentiment\_proportions)

# Create a stacked bar chart showing sentiment distribution per source

sentiment\_story\_plot <- sentiment\_proportions %>%

ggplot(aes(x = feedback\_source, y = proportion, fill = sentiment\_category)) +

geom\_col(position = "stack") + # Stack the bars

labs(

title = "Sentiment Distribution by Feedback Source",

subtitle = "App Reviews show a higher proportion of Negative feedback",

x = "Feedback Source",

y = "Proportion of Feedback",

fill = "Sentiment"

) +

scale\_fill\_manual(values = c("Negative" = "firebrick", "Neutral" = "goldenrod", "Positive" = "darkgreen")) + # Custom colors

theme\_minimal() +

theme(

plot.title = element\_text(hjust = 0.5, face = "bold"),

plot.subtitle = element\_text(hjust = 0.5),

legend.position = "bottom"

)

print(sentiment\_story\_plot)

Explanation: This plot clearly shows the proportion of positive, neutral, and negative feedback for each source. The taller red section for "App Review" immediately highlights the point about negative feedback, helping to visually convey the data story.

Wrap-up

This chapter introduced ggplot2, the essential R package for data visualization. You learned the fundamental principles of the Grammar of Graphics and how to create common plot types like bar charts, histograms, boxplots, and scatter plots. You also explored how to customize plot aesthetics, labels, titles, and themes to create visually compelling and informative graphics. Remember that visualization is not just about creating pretty pictures; it's about effectively communicating insights from your data, as demonstrated with the customer feedback example.

Chapter 7 will move beyond visualization into the realm of statistical analysis, showing how to perform common tests and build models in R.

# Chapter 7: Statistical Analysis in R

After importing, cleaning, and visualizing the data, the next logical step is often to perform statistical analysis. R, being a language specifically designed for statistics, provides a vast array of tools to conduct various statistical tests and build models. This chapter introduces fundamental statistical concepts and shows how to implement them in R using the "Customer Feedback Analyzer" data as an example.

## 7.1 Descriptive statistics

Descriptive statistics summarize and describe the main features of a dataset. They provide simple summaries about the sample and the measures, helping to understand the distribution of variables numerically.

Basic descriptive statistics include:

* Measures of Central Tendency: These describe the center of the distribution (mean, median, mode).
* Measures of Dispersion: These describe the spread or variability of the data (range, variance, standard deviation, interquartile range). Bookdown explains different measures of dispersion, including the range, interquartile range, variance, and standard deviation.

Let's calculate some descriptive statistics for the rating column in the cleaned\_feedback data. GitHub Pages provides a quick way to get a summary of a dataset in R using the summary() function, which includes the min, max, mean, median, and first and third quartiles.

R

library(tidyverse)

# Ensure 'cleaned\_feedback' is available from previous chapters

feedback\_data\_raw <- read\_csv("customer\_feedback.csv") # Assuming customer\_feedback.csv is in your working directory

cleaned\_feedback <- feedback\_data\_raw %>%

mutate(

comment = comment %>% str\_trim() %>% str\_to\_lower()

) %>%

replace\_na(list(rating = 3)) %>%

mutate(

sentiment\_category = case\_when(

rating >= 4 ~ "Positive",

rating == 3 ~ "Neutral",

rating < 3 ~ "Negative",

TRUE ~ "Unknown"

)

) %>%

select(customer\_id, timestamp, feedback\_source, rating, sentiment\_category, comment) %>%

arrange(timestamp)

# Quick summary of the 'rating' column

summary(cleaned\_feedback$rating)

# Individual measures

mean\_rating <- mean(cleaned\_feedback$rating)

median\_rating <- median(cleaned\_feedback$rating)

sd\_rating <- sd(cleaned\_feedback$rating) # Standard deviation

var\_rating <- var(cleaned\_feedback$rating) # Variance

range\_rating <- range(cleaned\_feedback$rating) # Min and Max

iqr\_rating <- IQR(cleaned\_feedback$rating) # Interquartile Range

print(paste("Mean Rating:", round(mean\_rating, 2)))

print(paste("Median Rating:", median\_rating))

print(paste("Standard Deviation:", round(sd\_rating, 2)))

print(paste("Variance:", round(var\_rating, 2)))

print(paste("Range:", range\_rating[1], "-", range\_rating[2]))

print(paste("Interquartile Range:", round(iqr\_rating, 2)))

# You can also group and summarize to get descriptive stats by category

desc\_stats\_by\_source <- cleaned\_feedback %>%

group\_by(feedback\_source) %>%

summarize(

mean\_rating = mean(rating, na.rm = TRUE),

median\_rating = median(rating, na.rm = TRUE),

sd\_rating = sd(rating, na.rm = TRUE),

count = n()

)

print(desc\_stats\_by\_source)

## 7.2 T-tests, ANOVA, chi-squared tests

Inferential statistics uses sample data to make inferences about a larger population. This section explores several common inferential tests. The City University of New York explains that inferential statistics uses sample data to learn about the whole population.

7.2.1 T-tests

T-tests are used to compare the means of two groups to determine if they are significantly different from each other. There are several types:

* One-sample t-test: Compares the mean of a single sample to a known population mean (or a hypothesized value).
* Independent Samples t-test: Compares the means of two independent groups. Andrew Farina provides code for running an independent-samples t-test in R, including specifying the formula, data, and options for paired or Welch's t-test.
* Paired Samples t-test: Compares the means of two related groups (e.g., before-and-after measurements on the same individuals).

Example: Independent Samples T-test

Let's investigate if there is a significant difference in rating between "Web Form" and "App Review" feedback sources.

R

# Prepare data for t-test: Filter for the two groups of interest

t\_test\_data <- cleaned\_feedback %>%

filter(feedback\_source %in% c("Web Form", "App Review")) %>%

select(feedback\_source, rating)

# Perform independent samples t-test

# Formula: dependent\_variable ~ independent\_variable

t\_test\_result <- t.test(rating ~ feedback\_source, data = t\_test\_data)

print(t\_test\_result)

The output of the t.test() includes the t-statistic, degrees of freedom (df), and the p-value. A p-value less than the chosen significance level (e.g., 0.05) suggests a significant difference between the means of the two groups. The conf.int shows the confidence interval for the difference in means. By default, R's t.test() performs Welch's t-test, which doesn't assume equal variances.

7.2.2 ANOVA (Analysis of Variance)

ANOVA compares the means of three or more groups to see if at least one group mean is significantly different.

Example: One-way ANOVA

Is there a significant difference in rating across the different feedback\_source categories?

R

# Ensure feedback\_source is treated as a factor

cleaned\_feedback$feedback\_source <- as.factor(cleaned\_feedback$feedback\_source)

# Perform one-way ANOVA

# Formula: dependent\_variable ~ independent\_variable (factor)

anova\_result <- aov(rating ~ feedback\_source, data = cleaned\_feedback)

summary(anova\_result)

The ANOVA summary provides an F-statistic and a p-value. A p-value below the significance level indicates a significant difference in means among the feedback sources. If the ANOVA is significant, post-hoc tests (like Tukey HSD) can identify which specific groups differ.

7.2.3 Chi-squared Tests

Chi-squared tests examine relationships between categorical variables.

* Chi-squared Goodness-of-Fit Test: Compares observed frequencies to expected frequencies in a single categorical variable to test a hypothesized distribution.
* Chi-squared Test of Independence: Determines if two categorical variables are associated.

Example: Chi-squared Test of Independence

Is there an association between feedback\_source and sentiment\_category?

R

# Create a contingency table (cross-tabulation) of the two categorical variables

contingency\_table <- table(cleaned\_feedback$feedback\_source, cleaned\_feedback$sentiment\_category)

print(contingency\_table)

# Perform Chi-squared Test of Independence

chi\_sq\_result <- chisq.test(contingency\_table)

print(chi\_sq\_result)

The output includes the chi-squared statistic, degrees of freedom, and the p-value. A p-value less than the significance level suggests a significant association between the two categorical variables.

## 7.3 Correlation and regression analysis

7.3.1 Correlation

Correlation measures the strength and direction of a linear relationship between two quantitative variables. The cor() function calculates the correlation coefficient, while cor.test() tests its significance.

R

# For demonstration, let's create a hypothetical 'review\_length' numeric variable

# In a real scenario, this would come from your data (e.g., str\_length(comment))

cleaned\_feedback <- cleaned\_feedback %>%

mutate(review\_length = str\_length(comment))

# Calculate Pearson correlation between rating and review\_length

# By default, cor() computes Pearson correlation.

cor\_pearson <- cor(cleaned\_feedback$rating, cleaned\_feedback$review\_length, use = "complete.obs") # 'use' handles NAs

print(paste("Pearson Correlation:", round(cor\_pearson, 2)))

# Calculate Spearman correlation (for non-parametric or ordinal data)

# Use method="spearman".

cor\_spearman <- cor(cleaned\_feedback$rating, cleaned\_feedback$review\_length, method = "spearman", use = "complete.obs")

print(paste("Spearman Correlation:", round(cor\_spearman, 2)))

# Test for significance of Pearson correlation

cor\_test\_result <- cor.test(cleaned\_feedback$rating, cleaned\_feedback$review\_length, method = "pearson")

print(cor\_test\_result)

Interpretation: The correlation coefficient ranges from -1 (strong negative linear relationship) to 1 (strong positive linear relationship), with values near 0 indicating a weak or no linear relationship. The cor.test() output provides the p-value to determine statistical significance.

7.3.2 Regression Analysis

Regression analysis models the relationship between a dependent variable and one or more independent variables. lm() is used to fit linear models.

* Simple Linear Regression: One independent variable.
* Multiple Linear Regression: Two or more independent variables.

Example: Simple Linear Regression

Can we predict rating based on review\_length?

R

# Perform simple linear regression

# Formula: dependent\_variable ~ independent\_variable

lm\_model <- lm(rating ~ review\_length, data = cleaned\_feedback)

summary(lm\_model)

The summary shows coefficients for the intercept and predictor, with standard errors, t-values, and p-values to indicate significance. R-squared (Multiple R-squared) shows the proportion of the dependent variable's variance explained by the independent variable(s).

## 7.4 Visualizing distributions and confidence intervals

Visualizing statistical analysis results is crucial. Histograms and density plots assess variable distributions, helping check assumptions like normality. Boxplots visualize group differences.

R

# Density plot of ratings, separated by sentiment category

density\_rating\_by\_sentiment <- cleaned\_feedback %>%

ggplot(aes(x = rating, fill = sentiment\_category)) +

geom\_density(alpha = 0.6) +

labs(

title = "Rating Distribution by Sentiment Category",

x = "Rating",

y = "Density"

) +

scale\_fill\_manual(values = c("Negative" = "firebrick", "Neutral" = "goldenrod", "Positive" = "darkgreen"))

print(density\_rating\_by\_sentiment)

# Boxplot again, highlighting group differences

boxplot\_rating\_by\_source <- cleaned\_feedback %>%

ggplot(aes(x = feedback\_source, y = rating, fill = feedback\_source)) +

geom\_boxplot() +

labs(title = "Rating Distribution Across Feedback Sources")

print(boxplot\_rating\_by\_source)

7.4.2 Visualizing Confidence Intervals

Confidence intervals show a range for the true population parameter and can be visualized with error bars or around regression lines.

R

# Example 1: Visualize the mean rating with 95% confidence intervals per feedback source

# First, calculate the mean and standard error per group

mean\_ci\_data <- cleaned\_feedback %>%

group\_by(feedback\_source) %>%

summarize(

mean\_rating = mean(rating, na.rm = TRUE),

se\_rating = sd(rating, na.rm = TRUE) / sqrt(n()), # Standard Error

upper\_ci = mean\_rating + 1.96 \* se\_rating, # Approx. 95% CI upper bound

lower\_ci = mean\_rating - 1.96 \* se\_rating # Approx. 95% CI lower bound

)

print(mean\_ci\_data)

# Plot with error bars

mean\_ci\_plot <- mean\_ci\_data %>%

ggplot(aes(x = feedback\_source, y = mean\_rating, fill = feedback\_source)) +

geom\_col() +

geom\_errorbar(aes(ymin = lower\_ci, ymax = upper\_ci), width = 0.2, color = "black") +

labs(

title = "Mean Rating with 95% Confidence Intervals by Source",

y = "Mean Rating",

x = "Feedback Source"

)

print(mean\_ci\_plot)

# Example 2: Confidence Interval around a regression line (using geom\_smooth)

# (Re-using the timestamp vs rating example, assuming some linearity for demonstration)

regression\_ci\_plot <- cleaned\_feedback %>%

ggplot(aes(x = timestamp, y = rating)) +

geom\_point(alpha = 0.7) +

geom\_smooth(method = "lm", color = "blue", fill = "lightblue") + # Displays 95% CI by default

labs(

title = "Customer Ratings Over Time with Regression Line and CI",

x = "Date and Time of Feedback",

y = "Rating"

)

print(regression\_ci\_plot)

Wrap-up

This chapter covered essential statistical analysis techniques in R, including descriptive statistics, t-tests, ANOVA, chi-squared tests, and simple linear regression. You also learned to visualize distributions and confidence intervals. These skills are crucial for drawing robust conclusions from data.

Chapter 8 will cover working with dates, times, and strings for more advanced analysis.

# Chapter 8: Working with Dates & Strings

In real-world data analysis, dates, times, and strings (textual data) are frequently encountered and often require specific handling. R provides robust tools for these tasks, with the lubridate package simplifying date-time operations and the stringr package offering consistent functions for string manipulation. This chapter delves into these powerful packages and applies them to the "Customer Feedback Analyzer" to extract deeper insights from timestamps and textual feedback. DataCamp explains how `parse\_date\_time` takes an input character or Date vector and returns an output of class POSIXct. Stack Overflow discusses using the anytime package in R for parsing dates with different formats.

## 8.1 Date/time objects with lubridate

While base R has some capabilities for working with dates and times (e.g., as.Date(), POSIXct, POSIXlt), the lubridate package makes these operations much more intuitive and user-friendly, handling various formats and time zone complexities with ease. UC Berkeley Statistics Department mentions that the base R `as.Date` function handles dates but not times, while POSIXct and POSIXlt handle dates and times with time zone control.

8.1.1 Creating Date and Datetime Objects

lubridate provides functions for parsing dates and times directly from character strings. These functions are named based on the order of year (y), month (m), day (d), hour (h), minute (m), and second (s) components in the string. The University of Virginia highlights that `lubridate`'s functions are named based on the order of month, day, and year components, e.g., `mdy()` for "May 11, 1996".

* ymd(): Parses dates like "2023-01-15" or "2023/01/15". The Comprehensive R Archive Network notes that `ymd()` parses dates with year first, followed by month and then day.
* mdy(): Parses dates like "01/15/2023".
* dmy(): Parses dates like "15-01-2023".
* For dates with time, add \_h, \_hm, or \_hms (e.g., ymd\_hms() for "2023-01-15 10:30:00"). The Comprehensive R Archive Network explains that for dates with time information, users can add `h`, `m`, and/or `s` to the function name, such as `ymd\_hms()` for the most common datetime format.

R

library(tidyverse) # Ensure lubridate is loaded

# Use the cleaned\_feedback data frame from Chapter 5

feedback\_data\_raw <- read\_csv("customer\_feedback.csv")

cleaned\_feedback <- feedback\_data\_raw %>%

mutate(

comment = comment %>% str\_trim() %>% str\_to\_lower()

) %>%

replace\_na(list(rating = 3)) %>%

mutate(

sentiment\_category = case\_when(

rating >= 4 ~ "Positive",

rating == 3 ~ "Neutral",

rating < 3 ~ "Negative",

TRUE ~ "Unknown"

)

) %>%

select(customer\_id, timestamp, feedback\_source, rating, sentiment\_category, comment) %>%

arrange(timestamp)

# The 'timestamp' column is likely already parsed correctly by read\_csv,

# but if it were a character, this is how you'd explicitly convert it:

char\_timestamp <- "2023-01-15 10:30:00"

datetime\_object <- ymd\_hms(char\_timestamp)

print(datetime\_object)

class(datetime\_object)

char\_date <- "January 15, 2023"

date\_object <- mdy(char\_date)

print(date\_object)

class(date\_object)

8.1.2 Extracting Date and Time Components

lubridate makes it easy to extract specific components from date-time objects using simple functions. The University of Virginia highlights that `lubridate` provides functions for every permutation of "m", "d", "y" to format dates, simplifying the extraction of date components.

R

# Extract components from the 'timestamp' column

feedback\_with\_date\_parts <- cleaned\_feedback %>%

mutate(

year = year(timestamp),

month = month(timestamp, label = TRUE), # 'label = TRUE' for abbreviated month names

day = day(timestamp),

wday = wday(timestamp, label = TRUE, abbr = TRUE), # Day of the week

hour = hour(timestamp),

minute = minute(timestamp),

second = second(timestamp)

)

print(head(feedback\_with\_date\_parts))

8.1.3 Date and Time Arithmetic

Performing calculations with dates and times, such as finding durations or adding/subtracting time units, is straightforward with lubridate.

* Durations: Exact time spans (e.g., number of seconds). Use dweeks(), dhours(), dminutes(), etc.
* Periods: Human-readable time spans that respect calendar boundaries (e.g., months have different numbers of days). Use weeks(), hours(), minutes(), etc. R for Data Science (2e) provides more details on working with periods and durations.

R

# Calculate the duration since the earliest feedback entry

first\_timestamp <- min(cleaned\_feedback$timestamp)

feedback\_with\_duration <- cleaned\_feedback %>%

mutate(

days\_since\_first\_feedback = as.period(timestamp - first\_timestamp, unit = "day"),

hours\_since\_first\_feedback = as.duration(timestamp - first\_timestamp) / dhours(1)

)

print(head(feedback\_with\_duration))

# Add or subtract time

future\_date <- date\_object + days(7)

print(paste("Date seven days from", date\_object, "is", future\_date))

# Example: Feedback received within the first hour of a day (hypothetical)

feedback\_first\_hour <- cleaned\_feedback %>%

filter(hour(timestamp) < 1) # Using the hour() function extracted earlier

# The cleaned\_feedback tibble currently has no feedback within the first hour

# Let's see some feedback during peak morning hours (e.g., 9 am to 11 am)

morning\_feedback <- cleaned\_feedback %>%

filter(hour(timestamp) >= 9 & hour(timestamp) < 12)

print(morning\_feedback)

## 8.2 String manipulation with stringr

Cleaning and analyzing textual data (strings) is crucial for understanding open-ended customer comments. The stringr package provides a consistent and user-friendly set of functions, all prefixed with str\_, to simplify string manipulation. GitHub mentions that the `stringr` package focuses on the most important and commonly used string manipulation functions, while `stringi` provides a comprehensive set.

8.2.1 Detecting Patterns: str\_detect()

str\_detect() checks if a pattern exists within a string, returning TRUE or FALSE.

Use Case: Identify comments containing specific keywords related to product quality or service.

R

# Check for comments mentioning "quality" or "buggy"

feedback\_quality\_issues <- cleaned\_feedback %>%

mutate(

has\_quality\_issue = str\_detect(comment, "quality|buggy"), # Use '|' for OR condition in regex

has\_delivery\_issue = str\_detect(comment, "delivery|shipping")

) %>%

filter(has\_quality\_issue | has\_delivery\_issue) # Filter for either issue

print(feedback\_quality\_issues)

8.2.2 Extracting Information: str\_extract() and str\_extract\_all()

str\_extract() extracts the first match of a pattern, while str\_extract\_all() extracts all matches. These functions are often used with regular expressions (regex) to define the pattern. Bookdown explains that `str\_extract()` and `str\_extract\_all()` functions extract substrings corresponding to a pattern.

Use Case: Extracting specific product mentions or error codes if they follow a pattern in comments.

R

# Let's imagine comments sometimes mention product codes like "PROD-XYZ"

sample\_comments <- c(

"Issue with PROD-123. Buggy software.",

"Great product PROD-456!",

"No specific product mentioned.",

"Defect in PROD-789 and also PROD-001."

)

product\_code\_data <- tibble(comment = sample\_comments) %>%

mutate(

first\_product\_code = str\_extract(comment, "PROD-[0-9]{3}"), # Extracts the first match

all\_product\_codes = str\_extract\_all(comment, "PROD-[0-9]{3}") # Extracts all matches as a list

)

print(product\_code\_data)

8.2.3 Replacing and Modifying Strings: str\_replace() and str\_replace\_all()

These functions replace the first or all occurrences of a pattern with new text.

Use Case: Masking sensitive information or standardizing terminology.

R

# Replace "excellent" with "superb" in comments

feedback\_standardized <- cleaned\_feedback %>%

mutate(

comment\_replaced = str\_replace(comment, "excellent", "superb"),

comment\_replaced\_all = str\_replace\_all(comment, "product", "item")

)

print(select(feedback\_standardized, comment, comment\_replaced, comment\_replaced\_all))

8.2.4 Splitting Strings: str\_split() and str\_split\_fixed()

These functions split a string into multiple parts based on a delimiter or pattern. str\_split() returns a list, while str\_split\_fixed() returns a matrix.

Use Case: Separating a customer's name from their email address or tags from a string.

R

# Hypothetical: Split feedback source into main type and sub-type (if applicable)

feedback\_with\_source\_parts <- cleaned\_feedback %>%

mutate(

source\_parts = str\_split(feedback\_source, " "), # Split by space

first\_source\_word = str\_split\_fixed(feedback\_source, " ", n = 2)[, 1] # Extract first word

)

print(select(feedback\_with\_source\_parts, feedback\_source, source\_parts, first\_source\_word))

## 8.3 Real-world applications: timestamped reviews and feedback parsing

Combining lubridate and stringr allows for powerful analysis of timestamped text data like customer reviews.

Case Study: Analyzing Peak Feedback Times and Text Content

Objective: Determine if feedback sentiment varies at different times of the day or if specific keywords are more prevalent during certain hours.

1. Extract Time-of-Day: Use lubridate to get the hour or part of the day.
2. Analyze Sentiment by Time: Use dplyr to group and summarize.
3. Identify Keywords by Time: Use stringr to detect patterns and count occurrences by time.

R

# 1. Extract hour of the day and categorize into shifts

feedback\_by\_time <- cleaned\_feedback %>%

mutate(

feedback\_hour = hour(timestamp),

time\_shift = case\_when(

feedback\_hour >= 6 & feedback\_hour < 12 ~ "Morning",

feedback\_hour >= 12 & feedback\_hour < 18 ~ "Afternoon",

feedback\_hour >= 18 & feedback\_hour < 24 ~ "Evening",

TRUE ~ "Night"

) %>% factor(levels = c("Morning", "Afternoon", "Evening", "Night")) # Order factors

)

# 2. Analyze sentiment distribution by time shift

sentiment\_by\_shift <- feedback\_by\_time %>%

group\_by(time\_shift, sentiment\_category) %>%

summarize(count = n(), .groups = 'drop') %>%

group\_by(time\_shift) %>%

mutate(proportion = count / sum(count)) %>%

ungroup()

print(sentiment\_by\_shift)

# Visualize sentiment distribution by time shift

sentiment\_by\_shift\_plot <- sentiment\_by\_shift %>%

ggplot(aes(x = time\_shift, y = proportion, fill = sentiment\_category)) +

geom\_col(position = "stack") +

labs(

title = "Customer Sentiment by Time of Day",

x = "Time Shift",

y = "Proportion of Feedback",

fill = "Sentiment"

) +

scale\_fill\_manual(values = c("Negative" = "firebrick", "Neutral" = "goldenrod", "Positive" = "darkgreen")) +

theme\_minimal()

print(sentiment\_by\_shift\_plot)

# 3. Identify keywords by time shift (e.g., are "buggy" comments more frequent at night?)

buggy\_comments\_by\_shift <- feedback\_by\_time %>%

mutate(is\_buggy = str\_detect(comment, "buggy|improvements")) %>%

group\_by(time\_shift) %>%

summarize(

total\_comments = n(),

buggy\_count = sum(is\_buggy, na.rm = TRUE),

proportion\_buggy = buggy\_count / total\_comments

)

print(buggy\_comments\_by\_shift)

This analysis helps identify operational patterns related to customer experience. For instance, if Negative sentiment spikes during Night shifts, it might indicate insufficient support coverage or product performance issues during those hours.

Wrap-up

This chapter equipped you with essential tools for handling two pervasive data types: dates and times using lubridate, and strings using stringr. You learned how to parse, extract components, perform arithmetic, detect patterns, extract specific information, replace text, and split strings. The "Customer Feedback Analyzer" example demonstrated the practical application of these skills in processing timestamped reviews and extracting valuable insights from textual data.

These capabilities are fundamental for robust data cleaning, preparation, and analysis, especially when working with real-world, often messy, datasets. Building on this, Chapter 9 will delve into intermediate programming concepts, including functions and environments, to further enhance coding skills in R.

# Chapter 9: Intermediate Functions & Environments

So far, you've mastered the basics of R, including data structures, manipulation with dplyr and tidyr, visualization with ggplot2, and handling dates and strings. This chapter elevates your R programming skills by diving into more advanced concepts surrounding functions and the crucial topic of environments and scoping. Understanding these concepts is essential for writing efficient, reusable, and debuggable R code, enabling more complex data analysis workflows.

## 9.1 Anonymous and nested functions

Functions are fundamental building blocks in R, allowing you to encapsulate code for reusability. Beyond the standard named functions defined with function(), R also offers anonymous and nested functions, providing flexibility for specific programming patterns.

9.1.1 Anonymous Functions

An anonymous function is a function defined and used without assigning it a name. They are often used for short, single-purpose operations within another function call, especially with functions that apply a function to elements or groups (like lapply, sapply, or functions within the purrr package, which is also part of the tidyverse).

Scenario: Suppose you have a list of numerical vectors and want to calculate the square root of each number, but only if the number is positive.

R

# A list of numeric vectors

data\_list <- list(

set1 = c(4, 9, -1, 16),

set2 = c(25, 0, 36),

set3 = c(-5, 100)

)

# Using an anonymous function with lapply to apply a custom operation to each element

# The anonymous function takes 'x' as input, applies the sqrt if x >= 0, else returns NA

result\_sqrt\_positive <- lapply(data\_list, function(x) {

ifelse(x >= 0, sqrt(x), NA)

})

print(result\_sqrt\_positive)

Explanation:  
The function(x) { ifelse(x >= 0, sqrt(x), NA) } is an anonymous function. It's defined directly as an argument to lapply without being assigned to a variable. It processes each element x from the data\_list, calculating the square root only for positive numbers and returning NA otherwise.

Anonymous functions streamline code by avoiding the need to create and manage many small, named functions that are only used once.

9.1.2 Nested Functions

A nested function (or inner function) is a function defined inside another function (the outer function). It has access to the variables of its enclosing function's environment (lexical scoping), even after the outer function has finished executing. This concept is powerful for creating functions that are factories for other functions, or for encapsulating helper logic.

Scenario: Create a function that generates another function to calculate a "weighted rating" based on different importance factors.

R

# Outer function: creates a weighting function

create\_weighted\_rating\_calculator <- function(weight\_comment = 0.5, weight\_source = 0.3, weight\_base\_rating = 0.2) {

# Inner (nested) function: calculates the weighted rating

weighted\_rating\_calculator <- function(base\_rating, comment\_length, is\_premium\_source) {

# Accesses weights from the outer function's environment

weighted\_value <- (base\_rating \* weight\_base\_rating) +

(comment\_length \* weight\_comment) +

(is\_premium\_source \* weight\_source)

return(weighted\_value)

}

return(weighted\_rating\_calculator) # Return the inner function

}

# Use the outer function to create a specific weighting function

# This function assigns higher weight to comment length

my\_custom\_calculator <- create\_weighted\_rating\_calculator(weight\_comment = 0.6, weight\_source = 0.2, weight\_base\_rating = 0.2)

# Use the generated function

# For demonstration, assume comment\_length and is\_premium\_source are derived from feedback

rating\_val <- 4

comment\_len\_val <- 50 # Hypothetical derived value

is\_premium\_val <- 1 # 1 for premium source, 0 otherwise

calculated\_weighted\_rating <- my\_custom\_calculator(rating\_val, comment\_len\_val, is\_premium\_val)

print(paste("Calculated weighted rating:", calculated\_weighted\_rating))

# Create another weighting function with different weights

another\_calculator <- create\_weighted\_rating\_calculator(weight\_comment = 0.3, weight\_source = 0.5, weight\_base\_rating = 0.2)

calculated\_another\_rating <- another\_calculator(rating\_val, comment\_len\_val, is\_premium\_val)

print(paste("Calculated another weighted rating:", calculated\_another\_rating))

Explanation:  
create\_weighted\_rating\_calculator is the outer function. It takes weighting parameters and returns the weighted\_rating\_calculator function. The key is that weighted\_rating\_calculator "remembers" the weight\_comment, weight\_source, and weight\_base\_rating values from the environment in which it was created, even after create\_weighted\_rating\_calculator has finished. This concept is closely tied to closures and R's lexical scoping rules.

## 9.2 Scope and environments

Understanding scope and environments is fundamental to comprehending how R finds the values associated with variables and functions.

9.2.1 Environments

In R, an environment is a collection of named objects (variables, functions, etc.). Every R session starts with a global environment, which contains all the objects created at the console or loaded from scripts. When a function is called, a new environment is created for that function's execution.

Environments form a hierarchy:

* Global Environment: The top-level environment where objects defined interactively or in scripts are stored.
* Package Environments: When a package is loaded (e.g., library(dplyr)), its functions and data are placed in their own environments, typically attached to the search path.
* Function Environments: Each time a function is called, it creates a new environment to store its local variables and arguments. This environment's parent is typically the environment where the function was *created* (not necessarily where it was called).

R

# You can view the current environment and its parent

current\_env <- environment()

print(current\_env)

print(parent.env(current\_env))

# Objects in the global environment

global\_var <- "I'm in the global environment"

my\_function <- function() {

local\_var <- "I'm local to my\_function"

print(global\_var) # Can access global\_var due to search path

print(local\_var)

# environment() here refers to the environment \*inside\* my\_function

print(environment())

print(parent.env(environment()))

}

my\_function()

# Trying to access local\_var outside the function will result in an error

# print(local\_var) # Error: object 'local\_var' not found

Explanation:  
When my\_function() is called, a new environment is created for it. Inside this environment, local\_var is defined. The function can still find global\_var because R searches up the chain of parent environments (the search path) until it finds a matching object name. Once my\_function() finishes, its local environment and local\_var are typically removed.

9.2.2 Scoping Rules

R uses lexical scoping (also known as static scoping). This means that a function's search for the value of a variable depends on where the function was *defined* (its parent environment), not where it was *called*.

R

x <- 10 # Global x

my\_outer\_function <- function() {

x <- 20 # x local to my\_outer\_function

my\_inner\_function <- function() {

print(x) # Which x will this see?

}

my\_inner\_function()

}

my\_outer\_function() # Output: 20

# Now, define another function at the global level

another\_inner\_function <- function() {

print(x) # Which x will this see?

}

another\_inner\_function() # Output: 10

Explanation:

* When my\_inner\_function is called inside my\_outer\_function, it finds x within my\_outer\_function's environment (where my\_inner\_function was *defined*), not the global x.
* another\_inner\_function was *defined* in the global environment, so when it searches for x, it finds the global x first.

This distinction is crucial: a function carries its definition environment with it, determining how it resolves variable names. This ensures predictable behavior, as a function will always find variables in the same way, regardless of where or when it's executed.

Understanding environments and scoping is particularly important when working with closures (like the nested function example earlier), S3 and S4 classes (which aren't covered here but rely heavily on these concepts), and when debugging complex R code where variables might be unexpectedly shadowed or unavailable.

Wrap-up

Chapter 9 moved beyond the basic operations to introduce intermediate programming concepts vital for becoming an effective R programmer. You explored:

* Anonymous Functions: Creating and using single-purpose functions on the fly within other function calls.
* Nested Functions: Defining functions within functions to create specialized tools and leverage lexical scoping.
* Environments and Scope: Understanding how R stores objects and resolves variable names based on where functions are defined (lexical scoping).

These concepts lay the foundation for writing more sophisticated, modular, and maintainable R code. They are particularly relevant when building custom functions, working with advanced R packages, or developing larger R projects. The next steps in your R journey might involve exploring object-oriented programming in R (S3, S4, R6 systems), package development, or diving into specific advanced topics based on your field of interest.

# Chapter 10: Advanced Topics and Real-World Integration

Having built a solid foundation in R programming, data manipulation, and visualization, this chapter expands into more specialized and advanced topics. It covers how R can be used with object-oriented programming paradigms, how to optimize code for performance, and how to connect to external data sources beyond simple CSV files. These skills are crucial for developing robust, efficient, and scalable R solutions for complex real-world problems.

## 10.1 Object-oriented programming in R (S3, S4, R6 systems)

While R's primary strength lies in functional programming and vectorized operations, it also supports object-oriented programming (OOP). OOP allows you to organize code around objects that combine data and functions (methods) that operate on that data. R offers several OOP systems, each with different strengths and use cases.

10.1.1 S3 Classes

S3 is R's oldest and most informal OOP system. It relies on a concept called "generic functions" and "method dispatch." A generic function (like print(), summary(), or plot()) behaves differently depending on the class of its first argument.

How it works:

1. Generic Function: A function like print() is a generic. When you call print(my\_object), R looks at the class of my\_object.
2. Method Dispatch: R then searches for a method named print.class\_of\_my\_object() (e.g., print.data.frame, print.factor).
3. Method Execution: If it finds a matching method, it executes that specific version of the function.

Defining an S3 Class and Method:

R

# 1. Create a regular list (or other data structure)

customer\_feedback\_summary <- list(

customer\_id = 105,

avg\_rating = 4.2,

num\_feedback = 10,

most\_common\_source = "Web Form"

)

# 2. Assign a class attribute to the object

class(customer\_feedback\_summary) <- "feedback\_summary"

# 3. Define a custom print method for this class

print.feedback\_summary <- function(x, ...) {

cat("--- Customer Feedback Summary ---\n")

cat("Customer ID:", x$customer\_id, "\n")

cat("Average Rating:", round(x$avg\_rating, 1), "\n")

cat("Total Feedback Entries:", x$num\_feedback, "\n")

cat("Most Common Source:", x$most\_common\_source, "\n")

cat("---------------------------------\n")

}

# Now, when you print the object, the custom method is used

print(customer\_feedback\_summary)

# You can still access its components directly

print(customer\_feedback\_summary$avg\_rating)

Explanation:  
S3 is flexible and easy to implement, especially for simple scenarios where you want to customize how objects are printed, summarized, or plotted. It doesn't enforce strict rules, making it quick to get started but potentially less robust for very complex systems.

10.1.2 S4 Classes

S4 is a more formal and rigorous OOP system introduced later. It provides stricter definition of classes (including slots for data and explicit inheritance) and methods (allowing multi-argument dispatch). S4 requires more upfront work but offers greater guarantees and encapsulation. It's often used in advanced statistical modeling packages.

Defining an S4 Class (Simplified):

R

# Define the S4 class

setClass("CustomerReview",

slots = c(

customer\_id = "numeric",

rating = "numeric",

comment = "character",

timestamp = "POSIXct"

)

)

# Create an instance of the S4 class

review\_s4 <- new("CustomerReview",

customer\_id = 1011,

rating = 4,

comment = "Quick delivery!",

timestamp = ymd\_hms("2023-05-10 14:00:00")

)

# Access slots using the @ operator

print(review\_s4@comment)

# Define a method for the S4 class (simplified example)

setMethod("show", "CustomerReview", function(object) {

cat("--- Customer Review ---\n")

cat("ID:", object@customer\_id, "\n")

cat("Rating:", object@rating, "\n")

cat("Comment:", object@comment, "\n")

cat("Timestamp:", as.character(object@timestamp), "\n")

cat("-----------------------\n")

})

# Printing the S4 object will now use the custom show method

show(review\_s4)

Explanation: S4 classes define specific slots (variables) and their types, providing more structure and type safety than S3. Methods are defined for specific combinations of generic functions and class signatures.

10.1.3 R6 Classes

R6 is a newer OOP system that closely resembles OOP paradigms found in languages like Java or Python. It's built on encapsulated objects with reference semantics, meaning objects can be modified "in place" rather than creating copies, which can be important for performance with large objects.

R

library(R6)

# Define an R6 class

CustomerFeedback <- R6Class("CustomerFeedback",

public = list(

customer\_id = NULL,

rating = NULL,

comment = NULL,

timestamp = NULL,

initialize = function(id, rate, comm, ts) {

self$customer\_id <- id

self$rating <- rate

self$comment <- comm

self$timestamp <- ts

self # Return self for chaining

},

# Method to update rating

update\_rating = function(new\_rating) {

self$rating <- new\_rating

message(paste("Rating for ID", self$customer\_id, "updated to", self$rating))

self # Return self for chaining

},

# Method to print feedback details

print = function(...) {

cat("--- Customer Feedback (R6) ---\n")

cat("ID:", self$customer\_id, "\n")

cat("Rating:", self$rating, "\n")

cat("Comment:", self$comment, "\n")

cat("Timestamp:", as.character(self$timestamp), "\n")

cat("------------------------------\n")

}

)

)

# Create an instance of the R6 class

feedback\_r6 <- CustomerFeedback$new(

id = 1012,

rate = 3,

comm = "Initially okay.",

ts = ymd\_hms("2023-06-01 09:00:00")

)

feedback\_r6$print()

# Modify the object in place using a method

feedback\_r6$update\_rating(4)$print() # Chain methods

Explanation:  
R6 classes define public and private members, methods, and an initialize method (constructor). They are invoked using the $ operator for methods and fields. The key difference is that R6 objects are modified by reference, making them behave more like objects in other traditional OOP languages.

The choice between S3, S4, or R6 depends on the specific needs of your project. S3 is great for simple, flexible customizations, S4 for more formal structure and type checking, and R6 when you need encapsulated objects with reference semantics.

## 10.2 Performance optimization and Rcpp

For computationally intensive tasks, R's interpreted nature can sometimes be a bottleneck. This section covers techniques to optimize R code, including using efficient programming practices and integrating C++ code via the Rcpp package for significant speedups.

10.2.1 Optimizing R Code

1. Vectorization: Always prefer vectorized operations over explicit for loops in R. R's built-in functions and tidyverse functions are often highly optimized for vectorization.
2. Apply Family & purrr: Use functions like lapply(), sapply(), vapply(), or the purrr equivalents (map(), map\_df(), etc.) instead of for loops when iterating over lists or vectors.
3. Data Structures: Choose appropriate data structures. For example, data.table or tibble can sometimes offer performance advantages over base R data.frame for specific operations.
4. Avoid Unnecessary Copies: Be mindful of operations that create unnecessary copies of large objects, especially within loops. Modifying objects in place where possible or pre-allocating memory can help.
5. Profiling: Use R's profiling tools (Rprof() or the profvis package) to identify bottlenecks in your code before attempting optimization. Don't optimize until you know *what* to optimize.

10.2.2 Integrating C++ with Rcpp

Rcpp is a powerful package that allows seamless integration of C++ code into R, dramatically speeding up computationally intensive parts of your analysis. You write functions in C++ and expose them to R, leveraging C++'s speed while retaining R's convenience.

Example: Calculating Euclidian Distance (C++ vs. R)

Let's imagine a scenario where frequent calculation of the Euclidean distance between two points is needed, such as with customer geographical data or feature similarity.

R Version:

R

# R function for Euclidean distance

euclidean\_distance\_R <- function(p1, p2) {

sqrt(sum((p1 - p2)^2))

}

point\_a <- c(1, 2, 3)

point\_b <- c(4, 5, 6)

dist\_R <- euclidean\_distance\_R(point\_a, point\_b)

print(paste("R Euclidean Distance:", round(dist\_R, 2)))

Rcpp Version:

1. Create a C++ file: Create a new file (e.g., src/euclidean\_distance.cpp) in your project's src folder. (If you don't have a src folder, create one).
2. Paste the C++ code:

cpp

#include <Rcpp.h>

using namespace Rcpp;

*// [[Rcpp::export]]*

double euclidean\_distance\_rcpp(NumericVector p1, NumericVector p2) {

int n = p1.size();

double sum\_sq\_diff = 0;

for (int i = 0; i < n; ++i) {

sum\_sq\_diff += pow(p1[i] - p2[i], 2);

}

return sqrt(sum\_sq\_diff);

}

1. Source the C++ code in R:

R

library(Rcpp)

# Source the C++ file (adjust path if needed)

sourceCpp("src/euclidean\_distance.cpp")

# Now you can use the Rcpp function

dist\_rcpp <- euclidean\_distance\_rcpp(point\_a, point\_b)

print(paste("Rcpp Euclidean Distance:", round(dist\_rcpp, 2)))

Performance Comparison:

R

library(microbenchmark)

# Generate larger vectors for testing

long\_point\_a <- rnorm(1000)

long\_point\_b <- rnorm(1000)

# Compare performance

benchmark\_result <- microbenchmark(

R\_version = euclidean\_distance\_R(long\_point\_a, long\_point\_b),

Rcpp\_version = euclidean\_distance\_rcpp(long\_point\_a, long\_point\_b),

times = 1000 # Run each function 1000 times

)

print(benchmark\_result)

Explanation:  
The [[Rcpp::export]] attribute makes the C++ function callable from R. NumericVector is an Rcpp type that handles R numeric vectors. Rcpp versions often run significantly faster, especially for tasks involving loops or element-wise calculations.

## 10.3 Connecting to external data sources

Beyond CSV files, R can connect to a wide array of external data sources like databases, web APIs, and even perform web scraping.

10.3.1 Databases

Connecting R to databases (e.g., SQL Server, PostgreSQL, MySQL, SQLite) is typically done using the DBI package along with specific database drivers (e.g., RSQLite, RMySQL, RPostgres).

Steps:

1. Install the driver package: install.packages("RSQLite")
2. Load DBI and the driver: library(DBI), library(RSQLite)
3. Establish connection: con <- dbConnect(RSQLite::SQLite(), dbname = "my\_database.sqlite")
4. Query data: dbReadTable(con, "customers") or dbGetQuery(con, "SELECT \* FROM customers WHERE rating > 4")
5. Write/Manipulate data: dbWriteTable(), dbExecute() (for SQL statements)
6. Disconnect: dbDisconnect(con)

R

# Example using an in-memory SQLite database

library(DBI)

library(RSQLite)

library(tidyverse)

# 1. Connect to an in-memory SQLite database

con <- dbConnect(RSQLite::SQLite(), ":memory:")

# 2. Write our cleaned\_feedback data frame to the database as a table

dbWriteTable(con, "customer\_feedback", cleaned\_feedback, overwrite = TRUE)

# 3. Query data from the database

# Get all feedback from "Web Form" source

web\_form\_feedback <- dbGetQuery(con, "SELECT \* FROM customer\_feedback WHERE feedback\_source = 'Web Form'")

print(web\_form\_feedback)

# Get average rating per sentiment category

avg\_rating\_db <- dbGetQuery(con, "SELECT sentiment\_category, AVG(rating) as mean\_rating FROM customer\_feedback GROUP BY sentiment\_category")

print(avg\_rating\_db)

# 4. Disconnect from the database

dbDisconnect(con)

10.3.2 Web APIs (Application Programming Interfaces)

APIs provide structured ways to interact with web services and retrieve data (often in JSON or XML format). The httr package is excellent for making HTTP requests (GET, POST, etc.), and packages like jsonlite handle parsing the responses.

Steps:

1. Make an HTTP request: GET("https://api.example.com/data")
2. Check status code: status\_code(response) (200 is success)
3. Extract content: content(response, "text")
4. Parse content: fromJSON(raw\_content) (for JSON)

*(Note: A live API example requires a specific API endpoint and credentials, which are beyond the scope of a general example. However, the httr and jsonlite packages are the primary tools.)*

10.3.3 Web Scraping

Web scraping involves programmatically extracting data from web pages. The rvest package (part of the tidyverse) simplifies this process.

Steps:

1. Read HTML: read\_html("https://example.com")
2. Select elements: html\_nodes(webpage, "CSS selector")
3. Extract data: html\_text(selected\_elements) or html\_attr(selected\_elements, "attribute\_name")

*(Note: Ethical considerations and website terms of service are crucial when web scraping. Some sites may have APIs as a preferred way to access data.) Web scraping may also involve dynamic websites, which present their own challenges.*

Wrap-up

This chapter propelled you into advanced R topics, equipping you with skills to tackle more complex programming challenges. You gained an understanding of R's Object-Oriented Programming systems (S3, S4, R6), allowing you to choose the right paradigm for your needs. You learned performance optimization techniques, including the use of Rcpp for integrating high-speed C++ code. Finally, you explored how R connects with the outside world, enabling interaction with databases via DBI and driver packages, web APIs using httr and jsonlite, and even performing web scraping with rvest. These advanced capabilities are essential for building scalable

# Chapter 11: Reproducibility, Reporting, and Collaboration

In the world of data science, simply performing an analysis isn't enough. It's crucial that the work is reproducible (others can get the same results), reportable (insights are communicated clearly), and collaborative (can be easily shared and worked on with others). This chapter brings these essential aspects together using R Markdown for dynamic reporting and Git/GitHub for version control and collaboration.

## 11.1 Reproducible research with R Markdown

R Markdown is a powerful tool for creating dynamic documents that seamlessly blend narrative text, R code, and the output generated by the code (figures, tables, model summaries). When knitted, the document is processed by knitr, executing the embedded R code and embedding the results into the final output format. This ensures your analysis is fully reproducible – if the data or code changes, re-knitting the document updates everything automatically .

11.1.1 Introduction to R Markdown

An R Markdown file (.Rmd) is a plain text file that contains three main components:

1. YAML metadata: Defines document settings (title, author, output format).
2. Markdown text: Narrative text formatted with Markdown syntax (headings, bold, lists).
3. R code chunks: Blocks of R code that are executed when the document is knitted.

yaml

---

title: "Customer Feedback Analysis Report"

author: "Your Name"

date: "July 14, 2025"

output: html\_document

---

R

# A basic R Markdown example

# (This code would be placed within an .Rmd file)

## Introduction

This report presents an analysis of customer feedback data from our recent product launch.

Key metrics like average rating and sentiment distribution will be examined.

```{r setup, include=FALSE}

# This chunk runs code but doesn't show it in the output.

knitr::opts\_chunk$set(echo = FALSE, message = FALSE, warning = FALSE)

library(tidyverse)

library(lubridate)

library(stringr)

# Load the cleaned\_feedback data (assuming customer\_feedback.csv is in your directory)

feedback\_data\_raw <- read\_csv("customer\_feedback.csv")

cleaned\_feedback <- feedback\_data\_raw %>%

mutate(

comment = comment %>% str\_trim() %>% str\_to\_lower()

) %>%

replace\_na(list(rating = 3)) %>%

mutate(

sentiment\_category = case\_when(

rating >= 4 ~ "Positive",

rating == 3 ~ "Neutral",

rating < 3 ~ "Negative",

TRUE ~ "Unknown"

)

) %>%

select(customer\_id, timestamp, feedback\_source, rating, sentiment\_category, comment) %>%

arrange(timestamp)

Overall Customer Sentiment

The average customer rating is r round(mean(cleaned\_feedback$rating), 2).

This section visualizes the distribution of sentiment categories.

{r sentiment\_bar\_chart, fig.width=8, fig.height=5}

sentiment\_plot <- cleaned\_feedback %>%

ggplot(aes(x = sentiment\_category, fill = sentiment\_category)) +

geom\_bar() +

labs(

title = "Distribution of Customer Sentiments",

x = "Sentiment Category",

y = "Number of Customers"

) +

scale\_fill\_manual(values = c("Negative" = "firebrick", "Neutral" = "goldenrod", "Positive" = "darkgreen")) +

theme\_minimal()

print(sentiment\_plot)

11.1.2 Code Chunks and Options

R code chunks are enclosed in backticks ( `` ). Options within the curly braces {}` control how chunks behave.

* echo=FALSE: Prevents R code from being shown in the output.
* eval=FALSE: Shows R code but prevents it from being executed.
* include=FALSE: Runs R code but prevents both the code and its output from appearing in the document.
* message=FALSE, warning=FALSE: Hides messages and warnings from the output.
* fig.width, fig.height: Controls the dimensions of generated plots.
* cache=TRUE: Caches the results of the chunk, speeding up subsequent knitting if the code hasn't changed.

11.1.3 Output Formats

R Markdown supports various output formats, including:

* HTML: Default, interactive documents.
* PDF: High-quality static documents (requires LaTeX installation).
* Word: Documents compatible with Microsoft Word.
* Presentations: ioslides, beamer, PowerPoint presentations.
* Dashboards: Using packages like flexdashboard.
* Websites and Books: Using packages like blogdown, bookdown.

Simply change the output field in the YAML metadata to render the document in a different format.

11.2 Version control with Git and GitHub

Version control is essential for tracking changes to code, collaborating with others, and ensuring the reproducibility of projects. Git is a popular version control system, and GitHub is a web-based hosting service for Git repositories that facilitates collaboration.

11.2.1 Setting up Git and GitHub with RStudio

1. Install Git: Download and install Git for your operating system .
2. Configure RStudio: Go to Tools > Global Options > Git/SVN in RStudio. Ensure Enable version control interface for RStudio projects is checked and that the path to the Git executable is correct.
3. Configure Git: Set your Git username and email address in your terminal or Git Bash.

bash

git config --global user.name "Your Name"

git config --global user.email "your.email@example.com"

1. Create a New RStudio Project with Git:
   * Go to File > New Project > New Directory > New Project.
   * Give your project a name and choose a directory.
   * Crucially, check the box that says Create a git repository.

11.2.2 Basic Git Workflow

1. Make Changes: Modify your .R scripts, .Rmd files, or any other project files.
2. Stage Changes: In the Git pane (usually top-right in RStudio), check the boxes next to the files you want to include in your next commit.
3. Commit Changes: Click the Commit button. A new window opens where you write a clear and concise Commit message describing the changes you made. Then click Commit. A commit creates a snapshot of your project at that point in time.
4. Push to GitHub: If your project is linked to a GitHub repository, click the Push button (green up-arrow icon) to upload your committed changes to the remote repository.

11.2.3 Collaborating with GitHub

1. Create a Repository on GitHub: Go to GitHub.com, create a new repository.
2. Clone the Repository (for collaborators): In RStudio, File > New Project > Version Control > Git. Paste the GitHub repository URL. This downloads a local copy of the repository.
3. Pull Changes: Before working, pull the latest changes from the remote repository to ensure your local copy is up-to-date. Click the Pull button (blue down-arrow icon).
4. Resolve Conflicts: If multiple people modify the same part of a file, Git will notify you of conflicts when you try to merge changes. You'll need to manually resolve these.
5. Issues and Pull Requests: Use GitHub's Issues feature to track bugs or tasks, and Pull Requests to propose and review changes before merging them into the main project branch.

11.2.4 Project Structure and Best Practices

* Organize Project Files: Keep data in a data/ folder, scripts in R/ or scripts/, figures in figures/, and reports in reports/.
* Use .Rproj files: RStudio projects bundle settings and the working directory, making it easy to share projects.
* .gitignore: Use a .gitignore file to prevent unnecessary or sensitive files (like raw data, large binaries, API keys) from being tracked by Git.
* R Markdown for Documentation: Use R Markdown for reports, analyses, and even your project's README.md file.
* Modularize Code: Break down complex tasks into smaller functions, stored in separate .R files.
* renv for Package Management: Consider using renv to create isolated, reproducible R environments for your projects, ensuring that everyone uses the exact same package versions.
* Knit Often: When working in R Markdown, knit frequently to catch errors early.
* Commit Often, Push Regularly: Small, frequent commits with descriptive messages are easier to manage than large, infrequent ones. Pushing regularly keeps the remote repository updated.

Wrap-up

This chapter solidified your journey towards becoming a proficient R user by focusing on three pillars: reproducibility, reporting, and collaboration. You learned how to leverage the power of R Markdown to create dynamic, reproducible reports and presentations, seamlessly integrating code, output, and narrative. Furthermore, you gained practical skills in using Git and GitHub for version control, enabling effective change tracking and collaborative development on your R projects. Mastering these tools ensures that your data analyses are not only insightful but also transparent, reliable, and easily shareable. This concludes the core chapters of "Mastering R: From Fundamentals to Real-World Solutions." The journey continues with applying these skills to your specific data challenges!

# Chapter 12: Building Interactive Applications and Scalable Solutions

You've built a strong foundation in R, from data cleaning and analysis to reporting and version control. This chapter takes your skills to the next level by focusing on how to build interactive applications for broader audiences and how to handle increasingly large datasets. These topics are crucial for transitioning from personal data analysis to creating production-ready tools and working effectively with big data.

## 12.1 Building interactive web applications with Shiny

Interactive web applications allow users to explore data and analyses without needing R installed or knowledge of coding. Shiny is an R package that makes building these interactive applications simple and powerful. Tilburg Science Hub explains that Shiny allows users to create interactive dashboards, data visualizations, and reports without needing knowledge of HTML, CSS, or Javascript.

12.1.1 Introduction to Shiny and Reactivity

Every Shiny app has two main components:

1. User Interface (UI): Defines the layout and appearance of the application, including input controls (sliders, dropdowns) and outputs (plots, tables).
2. Server Function: Contains the R code that performs calculations, generates plots, and produces outputs based on user inputs.

The core concept in Shiny is reactivity, where changes in user input trigger updates only in the necessary parts of the application, ensuring dynamism and responsiveness.

12.1.2 A Simple Shiny App Example

The following code demonstrates a simple Shiny app that visualizes the distribution of customer ratings, allowing users to filter by feedback source:

R

library(shiny)

library(tidyverse)

# Assume 'cleaned\_feedback' is loaded as in previous chapters

# For a standalone Shiny app, you'd typically load data inside the app or use a global.R file

feedback\_data\_raw <- read\_csv("customer\_feedback.csv") # Read in your CSV

cleaned\_feedback <- feedback\_data\_raw %>%

mutate(

comment = comment %>% str\_trim() %>% str\_to\_lower()

) %>%

replace\_na(list(rating = 3)) %>%

mutate(

sentiment\_category = case\_when(

rating >= 4 ~ "Positive",

rating == 3 ~ "Neutral",

rating < 3 ~ "Negative",

TRUE ~ "Unknown"

)

) %>%

select(customer\_id, timestamp, feedback\_source, rating, sentiment\_category, comment) %>%

arrange(timestamp)

# Define the User Interface (UI)

ui <- fluidPage(

titlePanel("Customer Rating Distribution by Source"), # Application title

sidebarLayout(

sidebarPanel(

selectInput(

inputId = "source\_filter", # Input ID to refer to in the server

label = "Select Feedback Source:",

choices = c("All", unique(cleaned\_feedback$feedback\_source)), # Options for dropdown

selected = "All"

)

),

mainPanel(

plotOutput("rating\_histogram") # Output ID for the plot

)

)

)

# Define the Server Logic

server <- function(input, output) {

# Reactive expression to filter data based on user input

filtered\_data <- reactive({

if (input$source\_filter == "All") {

cleaned\_feedback

} else {

cleaned\_feedback %>%

filter(feedback\_source == input$source\_filter)

}

})

# Render the histogram plot

output$rating\_histogram <- renderPlot({

filtered\_data() %>% # Call the reactive expression

ggplot(aes(x = rating, fill = sentiment\_category)) +

geom\_histogram(binwidth = 1, color = "white") +

labs(title = paste("Rating Distribution for", input$source\_filter, "Feedback"),

x = "Rating (1-5)",

y = "Count") +

scale\_fill\_manual(values = c("Negative" = "firebrick", "Neutral" = "goldenrod", "Positive" = "darkgreen")) +

theme\_minimal()

})

}

# Run the application

shinyApp(ui = ui, server = server)

Explanation: The ui defines the layout with input and output elements, while the server function uses reactive expressions (reactive({}) and renderPlot({})) to dynamically filter data and update the plot based on user input. shinyApp(ui, server) launches the application.

12.1.3 Customizing and Deploying Shiny Apps

Shiny apps can be customized with CSS, HTML (using the htmltools package), and JavaScript. Deployment options include shinyapps.io, a private Shiny Server, or Docker containers.

12.2 Handling large datasets and distributed computing

Working with "big data" in R often requires strategies beyond loading entire datasets into memory.

12.2.1 Strategies for Large Datasets

Several strategies can be employed:

* Efficient File Formats: Use formats like Parquet or Feather with the arrow package.
* Chunking/Streaming Data: Process data in smaller pieces using packages like data.table and readr.
* Database Backends: Store data in databases (e.g., PostgreSQL, DuckDB) and use dplyr with dbplyr to perform computations within the database.
* Specialized Packages: Utilize data.table for fast in-memory manipulation, disk.frame for out-of-memory data, and arrow for interoperable large dataset handling.

12.2.2 Distributed Computing with R

For processing data across multiple machines, R can integrate with distributed computing frameworks:

1. SparkR and sparklyr: These packages provide R interfaces to Apache Spark. sparklyr offers a dplyr-like syntax for working with Spark.

R

# Example using sparklyr (requires a Spark installation/connection)

# library(sparklyr)

# library(dplyr)

# # Connect to a local Spark instance (or a remote cluster)

# sc <- spark\_connect(master = "local")

# # Copy cleaned\_feedback data to Spark

# feedback\_spark <- copy\_to(sc, cleaned\_feedback, "feedback\_spark", overwrite = TRUE)

# # Perform dplyr-like operations on Spark data (executed on Spark)

# avg\_rating\_spark <- feedback\_spark %>%

# group\_by(sentiment\_category) %>%

# summarize(mean\_rating = mean(rating, na.rm = TRUE)) %>%

# collect() # Bring results back to R

# print(avg\_rating\_spark)

# # Disconnect from Spark

# spark\_disconnect(sc)

Explanation: sparklyr translates dplyr code into Spark operations, allowing computation on large datasets without loading them into R's memory. collect() retrieves summarized results.

1. Parallel Processing: The parallel package or foreach with doParallel can be used to speed up independent tasks on multi-core machines.

R

# Example: Using parallel processing to simulate multiple analyses

# (This is illustrative; actual speedup depends on task complexity and number of cores)

library(parallel)

num\_cores <- detectCores() - 1 # Use all but one core

cl <- makeCluster(num\_cores) # Create a cluster

# Perform a simulated analysis in parallel

results <- parLapply(cl, 1:5, function(i) {

# Simulate some heavy computation

Sys.sleep(1) # Pause for 1 second

mean(rnorm(1000 \* i))

})

print(results)

# Stop the cluster

stopCluster(cl)

Wrap-up

This final chapter introduced building interactive web applications with Shiny and handling large datasets through strategies like efficient file formats, database backends, and distributed computing with sparklyr. These skills are essential for creating production-ready tools and working with big data.

Congratulations on completing "Mastering R: From Fundamentals to Real-World Solutions." Continue practicing, exploring new packages, and engaging with the community to further your expertise!

# Appendix A: R and RStudio Quick Reference

This appendix provides a quick reference for common R commands and RStudio features discussed throughout the book.

## A.1 RStudio Interface Panes

* Source Pane (Top-Left): Code editor for writing and saving scripts (.R, .Rmd).
* Console Pane (Bottom-Left): Where R commands are executed, and output is displayed.
* Environment/History Pane (Top-Right): Lists active objects (variables, data frames) and command history.
* Files/Plots/Packages/Help Pane (Bottom-Right): Navigation, plot display, package management, and help documentation.

## A.2 Basic R Commands

|  |  |  |
| --- | --- | --- |
| Command | Description | Example |
| <- or = | Assignment operator | x <- 5 |
| # | Comment | # This is a comment |
| print() | Displays output to the console | print("Hello") |
| c() | Combines values into a vector | my\_vector <- c(1, 2, 3) |
| library() | Loads an installed package into the session | library(tidyverse) |
| install.packages() | Installs a package from CRAN | install.packages("ggplot2") |
| ?function\_name or help() | Access help documentation for a function | ?mean or help(mean) |
| rm() | Removes objects from the environment | rm(my\_variable) |
| ls() | Lists objects in the current environment | ls() |
| class() | Determines the class (type) of an object | class(my\_vector) |
| str() | Displays the structure of an object (useful for data frames) | str(my\_data\_frame) |
| head() / tail() | View the first/last few rows of a data frame | head(my\_data\_frame) |

## A.3 dplyr Verbs Reference

|  |  |  |
| --- | --- | --- |
| dplyr Verb | Description | Example |
| filter() | Selects rows based on conditions | df %>% filter(rating > 3) |
| select() | Selects or deselects columns | df %>% select(customer\_id, rating) |
| mutate() | Creates new columns or modifies existing ones | df %>% mutate(is\_high = rating >= 4) |
| arrange() | Reorders rows by columns | df %>% arrange(year, desc(rating)) |
| group\_by() | Groups data by categorical variables | df %>% group\_by(feedback\_source) |
| summarize() | Summarizes grouped data into single rows | df %>% summarize(avg\_rating = mean(rating)) |
| %>% (pipe) | Passes output of one function as input to the next | df %>% filter(...) %>% select(...) |

# Appendix B: Common R Errors and Troubleshooting

Encountering error messages is a normal part of programming. This section outlines some of the most common R errors and strategies for resolving them. blog.revolutionanalytics.com shares the most common R error messages, including "could not find function", "Error in if", "Error in eval", and "subscript out of bounds". The Epidemiologist R Handbook discusses common R errors and potential solutions, such as typo errors, package errors, and issues with using the wrong data file.

## B.1 "could not find function "X""

* Cause: You are trying to use a function (e.g., ggplot, filter) from a package that has not been loaded into the current R session. Or, there's a typo in the function name.
* Solution: Use library(package\_name) to load the necessary package. Double-check the function's spelling.

## B.2 "Error in filter(...): object 'X' not found" or "object 'X' not found"

* Cause: You're referencing a variable or object that either doesn't exist in your current environment, or you're trying to use it in a context where it's not visible (e.g., dplyr verbs acting on a column name that doesn't exist in the data frame).
* Solution: Check the spelling of the variable. Ensure the data frame has the column you're trying to access (names(my\_data\_frame)). If inside a function, review scoping rules.

B.3 "Error: data must be a data frame, or other object coercible by fortify(), not a numeric vector"

* Cause: You're passing the wrong type of data to a function that expects a data frame, vector, or other specific type. For example, trying to use ggplot() on a simple vector instead of a data frame.
* Solution: Check the function's documentation to see the expected input type. Ensure your data is in the correct format (e.g., use as.data.frame() or as\_tibble() if needed).

## B.4 "Error: unexpected 'token' in 'code'"

* Cause: Syntax error. Missing parenthesis, bracket, quote, or a stray character.
* Solution: Carefully check the line of code indicated (and often the line just before it) for unmatched symbols. RStudio's syntax highlighting and auto-completion can help.

## B.5 "Error in [.data.frame(data, , variable) : undefined columns selected"

* Cause: Attempting to select a column by name that does not exist in the data frame using base R subsetting ([]).
* Solution: Verify the column name spelling. Use names(my\_data\_frame) to see available columns.

## B.6 "Error in file(file, "rt") : cannot open the connection"

* Cause: R cannot find the file you're trying to read. Common causes include typos in the filename/path or the file not being in the current working directory.
* Solution:
  + Verify the filename and extension.
  + Ensure the file is in your RStudio project's working directory (getwd()).
  + Use an absolute path or navigate to the correct working directory (setwd("path/to/directory") - though relying on RStudio projects is generally better).
  + Ensure the file isn't open and locked by another program (like Excel).

## B.7 "Error: removed X rows containing non-finite values (stat\_bin)." (or similar warnings)

* Cause: This is often a warning, not an error. Functions, especially plotting functions, may remove rows containing NA (missing) values or NaN (Not a Number) values by default when performing calculations or plotting.
* Solution:
  + Understand if these missing values are expected.
  + Use na.rm = TRUE in functions like mean() or sum() to exclude NAs.
  + Use filter(!is.na(column\_name)) to remove rows with NAs in specific columns.
  + Use replace\_na() to fill NAs with a specific value (see Chapter 5).
  + Use drop\_na() to remove rows with NAs (see Chapter 5).

# Appendix C: Recommended R Packages for Further Exploration

This book covered the foundational R packages essential for data science. However, the R ecosystem is vast, with thousands of specialized packages available on CRAN and GitHub. Here are a few recommendations to expand your toolkit. [GitHub Pages recommends `adv-r.hadley.nz` for sharpening programming skills in R](https://www.google.com/url?sa=i&source=web&rct=j&url=https://bluefoxr.github.io/COINrDoc/appendix-r-resources.html&ved=2ahUKEwiSk42m37yOAxVESTABHa2XMp8Qy_kOegYIAwgAECg&opi=89978449&cd&psig=AOvVaw1jS_81IZseOndFDP9Gkeee&ust=1752596077227000).

* Data Manipulation & Transformation:
  + data.table: A high-performance package for data manipulation, particularly efficient for large datasets.
  + janitor: Provides functions for cleaning dirty data (e.g., standardizing column names, removing empty rows).
  + duckdb: An in-process SQL OLAP database that integrates seamlessly with R, allowing efficient SQL queries on large datasets without needing a separate database server.
* Data Visualization:
  + plotly: Creates interactive web-based graphs, allowing users to hover over points, zoom, and pan.
  + ggiraph: Makes ggplot2 graphics interactive, supporting tooltips, zooming, and click actions.
  + leaflet: For creating interactive maps.
* Statistical Modeling:
  + tidymodels (collection of packages like parsnip, recipes, tune, yardstick): A meta-package for tidying and streamlining machine learning workflows.
  + caret: Provides a unified interface for many machine learning models and data preprocessing steps.
  + lme4: For fitting linear and generalized linear mixed-effects models.
* Text Analysis:
  + quanteda: A powerful and efficient framework for quantitative text analysis.
  + tidytext: Integrates text mining with tidyverse principles for easy manipulation and analysis of text.
* Reproducibility & Reporting:
  + bookdown: Creates books and long-form documents from R Markdown files.
  + blogdown: Builds websites and blogs using R Markdown.
  + flexdashboard: Easily creates interactive dashboards from R Markdown.
  + renv: Manages project-specific R environments, ensuring reproducibility across different systems.
* Performance:
  + profvis: An interactive tool for profiling R code to identify performance bottlenecks.

# Appendix D: Glossary of Key Terms

* Aesthetics (aes()): In ggplot2, the visual properties of a plot (e.g., x-position, y-position, color, size) mapped to data variables.
* Anonymous Function: An unnamed function defined and used inline, often for simple, single-use tasks.
* ANOVA (Analysis of Variance): A statistical test used to compare the means of three or more groups.
* Atomic Vector: The simplest R data structure, holding multiple elements of the *same* data type.
* Class: An attribute of an R object that determines how generic functions (like print()) will behave when applied to it.
* Coercion: The automatic or explicit conversion of data from one type to another (e.g., numeric to character).
* Console Pane: The area in RStudio where R commands are executed and output is displayed.
* CRAN (Comprehensive R Archive Network): The primary repository for R packages.
* Data Frame: A tabular R data structure where columns can hold different data types, but all elements within a column must be of the same type. Rows represent observations, columns represent variables.
* dplyr: A tidyverse package providing a grammar for data manipulation.
* Environment: A collection of named objects (variables, functions). Every R session and function call has an associated environment.
* Factor: An R data type used to store categorical data with predefined levels.
* geom\_ (Geometrics): In ggplot2, the visual markers used to represent data (e.g., geom\_point for scatter plots, geom\_bar for bar charts).
* Git: A distributed version control system for tracking changes in source code during software development.
* GitHub: A web-based platform for hosting Git repositories, facilitating collaboration.
* Global Environment: The default environment in an R session where user-defined objects reside.
* ggplot2: A tidyverse package for creating data visualizations based on the Grammar of Graphics.
* knitr: An R package that processes R Markdown documents, executing code chunks and embedding results.
* Lexical Scoping: R's rule for resolving variable names, based on where a function was *defined*, not where it was called.
* List: An R data structure that can hold elements of different data types (including other lists or data frames).
* lubridate: An R package for simplifying date and time manipulation.
* magrittr (%>% pipe operator): Facilitates chaining operations by passing the output of one function as the input to the next.
* NA (Not Available): R's representation for missing values.
* Package: A collection of R functions, data, and documentation that extends R's capabilities.
* readr: A tidyverse package for fast and user-friendly import of rectangular data.
* Reactivity: The core concept in Shiny where changes in inputs automatically trigger updates in outputs.
* Reproducible Research: An approach to research that ensures others can replicate results using the same code, data, and environment.
* Rcpp: An R package for seamlessly integrating C++ code into R for performance optimization.
* R Markdown: A file format for creating dynamic documents that combine narrative text, R code, and generated output.
* Shiny: An R package for building interactive web applications.
* Source Pane: The RStudio editor where R scripts are written and saved.
* S3 / S4 / R6 Classes: R's different object-oriented programming systems.
* stringr: A tidyverse package for consistent and intuitive string manipulation.
* T-test: A statistical test used to compare the means of two groups.
* tibble: A modern, tidyverse-friendly alternative to data.frame, with improved printing and subsetting behaviors.
* tidyr: A tidyverse package for tidying data, primarily with pivot\_longer() and pivot\_wider().
* tidyverse: A collection of R packages designed for data science, sharing a common design philosophy and making data analysis more intuitive.
* Vectorization: Performing operations on entire vectors at once in R, often leading to much faster code execution than loops.
* Version Control: A system (like Git) for tracking and managing changes to files over time.
* Working Directory: The default location where R looks for files to read and saves files it creates.