Statistics and Data Analysis with R: A Lab-Based Approach

Ezekiel Ogundepo

Table of contents

Preface

Welcome to **"Statistics and Data Analysis with R: A Lab-Based Approach"** by Ezekiel Ogundepo. This book is born out of a passion for teaching and a belief in learning by doing. Over the years, I've seen countless students transform their understanding and skills through hands-on experience, and it is this transformative journey that I hope to guide you through in these pages.

R has emerged as a powerful tool for data analysis, statistics, and visualisation. Whether you're a student stepping into the world of data science for the first time, a professional seeking to enhance your analytical capabilities, or simply a curious mind eager to explore new horizons, this book is designed to meet you where you are.

The approach we've taken is straightforward yet effective: each chapter presents lab-based experiments and exercises that encourage you to roll up your sleeves and dive into coding. Rather than overwhelming you with abstract theory, we focus on practical application, allowing you to see immediate results from the concepts you learn. This method not only reinforces your understanding but also builds confidence as you witness your own progress.

We begin with the basics—navigating the RStudio interface, performing simple calculations, and understanding fundamental data types. From there, we delve into more complex structures like vectors, matrices, and data frames, equipping you with the tools to manipulate and analyse data effectively. As you progress, you'll learn to write custom functions, manage packages, handle real-world data, and ensure the reproducibility of your analyses.

One of the unique aspects of this book is its emphasis on real-world applications. The labs are crafted to mirror challenges you might face outside the classroom or office, bridging the gap between learning and doing. By the end of this book, you'll not only understand the mechanics of R programming but also how to apply it to solve meaningful problems.

I have written this book in a conversational tone, much like how I would teach in a classroom or guide a colleague. My aim is to make the material accessible and engaging, stripping away unnecessary jargon without sacrificing depth or clarity. I've also included plenty of examples, exercises, and tips to support your learning journey.

Remember, programming is as much an art as it is a science. It requires patience, practice, and a willingness to experiment. Don't be discouraged by mistakes—they are stepping stones to mastery. I encourage you to take your time with each lab, explore variations of the examples provided, and most importantly, enjoy the process of learning.

Thank you for choosing this book as your guide into the world of R programming. I am excited to accompany you on this journey and look forward to the insights and discoveries that await you.

Figure 1: Author's Enthusiastic Invitation to Explore R Programming

Happy coding!

Part I

Your R Journey

1 Getting Started with R

1.1 Introduction

Welcome to **Lab 1**! In this first chapter, we'll embark on an exciting journey into the world of R programming and the powerful RStudio Integrated Development Environment (IDE). Whether you're new to programming or already familiar with other languages, this lab is designed to lay a solid foundation for future data analysis and statistical computing explorations.

1.2 Learning Objectives

By the end of the lab, you will be able to:

- **Explore the RStudio Interface**
  Get acquainted with the four main panes of RStudio and understand how each contributes to a smooth and efficient coding experience.
- **Perform Basic Calculations**
  Learn how to use R as a calculator, performing arithmetic operations while understanding the order of operations.
- **Understand Atomic Data Types**
  Delve into the fundamental data types in R—such as numeric, character, and logical types—which are essential building blocks for working with data.
- **Assigning Variables:**
  Practice creating variables, assigning values to them, and following proper naming conventions—an essential skill for organizing your code.
- **Using Conditional Statements**
  Explore how to control the flow of your programs using `if`, `else if`, and `else` statements, along with logical operators, allowing your code to make decisions based on conditions.

By completing this lab, you'll be comfortable with the RStudio environment and equipped to perform basic calculations, manipulate data types, assign variables, and write simple scripts that make decisions based on conditions. This is your first step toward mastering R and unlocking its potential for data analysis and statistical computing.

1.3 Prerequisites

Before starting this lab, you should have:

- Basic computer knowledge (navigating files, installing software).
- An interest in learning programming and data analysis.
- No prior programming experience is required.

1.4 Why Learn R Programming?

R is a powerful programming language and software environment extensively used for statistical computations, data cleaning, data analysis, and data visualisation. It is a vital tool for statisticians, data scientists, and anyone interested in data mining. Since its inception, R has become a cornerstone in data analysis, celebrated for its versatility and strong community support.

1.4.1 A Brief History of R's Development

The development of R programming commenced in 1993, spearheaded by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand. They released the initial version on StatLib, marking the beginning of R's evolution as an open-source tool designed to empower the statistical and data analysis community.

By 1997, R had solidified its status as a GNU project, reinforcing its commitment to free software principles and collaborative innovation. The release of version 1.0.0 in 2000 was pivotal, establishing a stable and reliable platform for statistical computing and data analysis.

Over the years, R has continued to evolve with the introduction of transformative packages such as:

- ggplot2 (2005): Revolutionized data visualization with a powerful and flexible grammar of graphics.
- dplyr (2014): Streamlined data manipulation tasks, making it easier to transform and summarize data.
- tidyverse suite (2016): Provided an integrated collection of packages for data science workflows, promoting a consistent and efficient approach to data analysis.

In 2023, R celebrated its 30th anniversary, a testament to its journey from a niche academic tool to a widely adopted resource across industries worldwide. This milestone underscores R's enduring robustness, adaptability, and the strength of its vibrant community[1].

1.4.2 Key Reasons to Learn R

- Free and Open Source: R is entirely free, making it accessible to everyone.
- Extensive Community Support: An active global community constantly develops and shares resources.
- Industry Application: From tech giants like Google and Facebook to finance leaders like JPMorgan and HSBC, R is a trusted tool across industries.

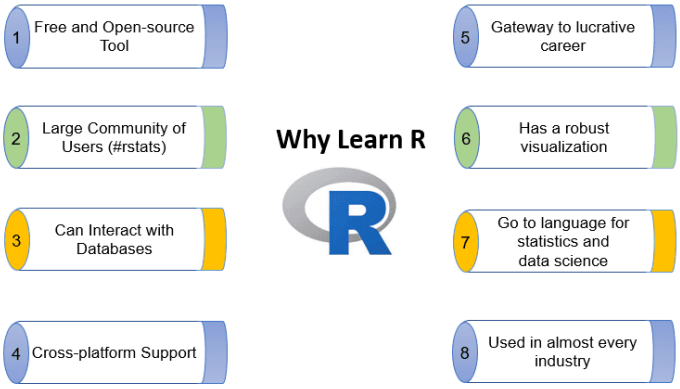


Figure 1.1: Compelling Reasons to Learn R

1.4.3 Companies Using R for Analytics

R's widespread adoption is evident in the diversity of industries leveraging its capabilities. From creating predictive models to visualising business trends, companies like Facebook, Google, Deloitte, and HSBC rely on R for analytics.

[1]For a detailed breakdown of R's development, refer to the comprehensive timeline created by Tim Brock, Colin Gillespie, and the Jumping Rivers team at https://www.jumpingrivers.com/blog/r-timeline.

Figure 1.2: Major Companies Using R Programming

For instance, data analysts at Netflix use R to understand viewing patterns and recommend shows to users. Healthcare professionals employ R to analyse patient data for better treatment outcomes. By learning R, you're gaining a skill that is in demand across various industries.

1.4.4 A Steep Yet Rewarding Learning Curve

R's learning curve can be steep initially. However, its design makes once-difficult tasks easier and intuitive. As you progress, you will automate complex workflows, create visually compelling graphics, and perform advanced analyses efficiently.

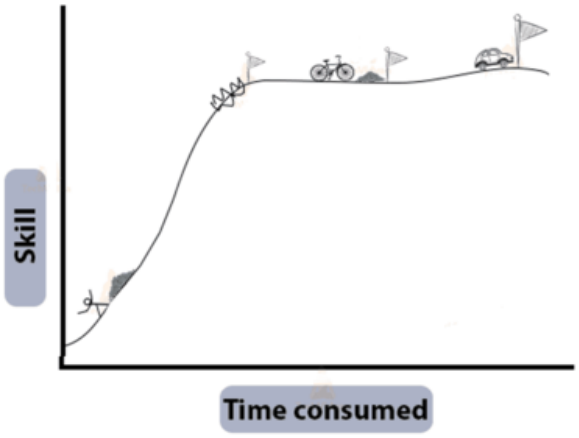
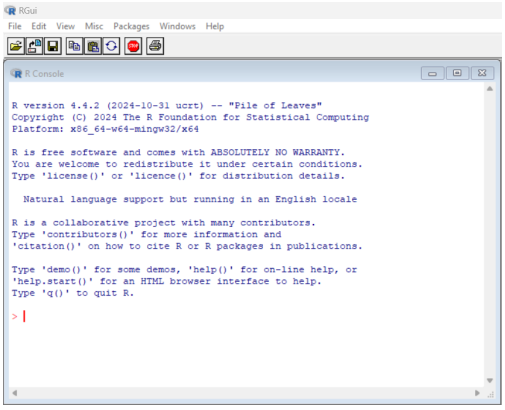


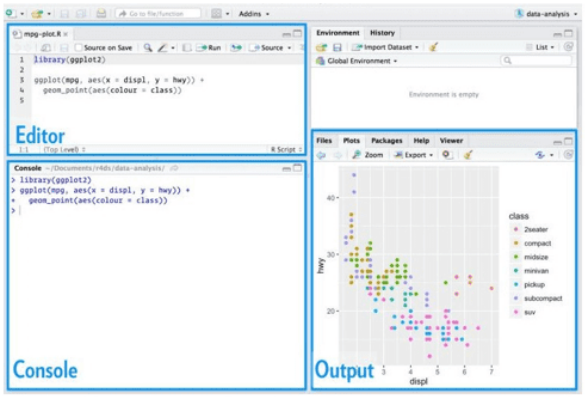
Figure 1.3: The Learning Curve of R Programming

1.5 Experiment 1.1: Installing R and RStudio

R is the core programming language (Figure 1.4a), while RStudio is an integrated development environment (IDE) that facilitates writing, executing, and debugging R code (Figure 1.4b).



(a) R Graphical User Interface (Rgui)



(b) RStudio Interface

Figure 1.4: Comparing the R GUI and RStudio IDE Interfaces

1.5.1 Installing R

The installation process for R varies slightly depending on your operating system:

- **For Windows Users:**

  Visit the CRAN (Comprehensive R Archive Network) website at this link. Download the latest version of R for Windows, then follow the installation prompts to complete the setup.

- **For Mac Users:**

  Visit the CRAN website for Mac at this link. Download the appropriate version for your macOS, and follow the on-screen instructions to install it.

1.5.2 Installing RStudio

Once R is installed, you should install RStudio, which provides an easier interface for interacting with R.

- Visit the RStudio download page. Select the free version of RStudio Desktop, and download the appropriate installer for your operating system (Windows, macOS, or Linux). Then, run the installer and follow the instructions.

Code Execution Guidance

After installing R and RStudio, open RStudio to ensure everything works correctly. The R console will appear in the lower-left pane, indicating that R is ready to use.

With R and RStudio installed you're ready to start your journey into data analysis, statistical computing, and programming with R!

1.5.3 Practice Quiz 1.1

Question 1:

What is the primary role of R in the R programming environment?

a) A user interface for writing code

b) A programming language for statistical computing

c) A package manager

d) A data visualization tool

Question 2:

Which of the following best describes RStudio?

a) A standalone programming language

b) A text editor for writing R scripts

c) An Integrated Development Environment (IDE) for R

d) A package repository for R

Question 3:

Which of the following is the correct sequence of steps to install R and RStudio on your computer?

a) Install RStudio first, then install R from the CRAN website.

b) Install R from the CRAN website first, then install RStudio.

c) Download both R and RStudio from the RStudio website and install them simultaneously.

d) Install R from the Microsoft Store, then install RStudio from the CRAN website.

Question 4:

Which keyboard shortcut runs the current line of code in RStudio on Windows?

a) `Ctrl + S`

b) `Ctrl + Enter`

c) `Alt + R`

d) `Shift + Enter`

Question 5:

After successful installation, which pane in RStudio indicates that R is ready to use?

a) Source Pane

b) Console Pane

c) Environment Pane

d) Files Pane

See the Solution to Quiz 1.1

1.6 Experiment 1.2: Exploring the RStudio Interface

Now that you have R and RStudio installed let's explore the RStudio interface. Understanding the layout and purpose of each pane will help you navigate and use RStudio effectively.

1.6.1 The Four Panes of RStudio

RStudio is divided into four main panes, each serving a specific purpose to enhance your coding workflow[2].

[2]For a detailed overview of all RStudio's features, see the RStudio User Guide at https://docs.posit.co/ide/user.

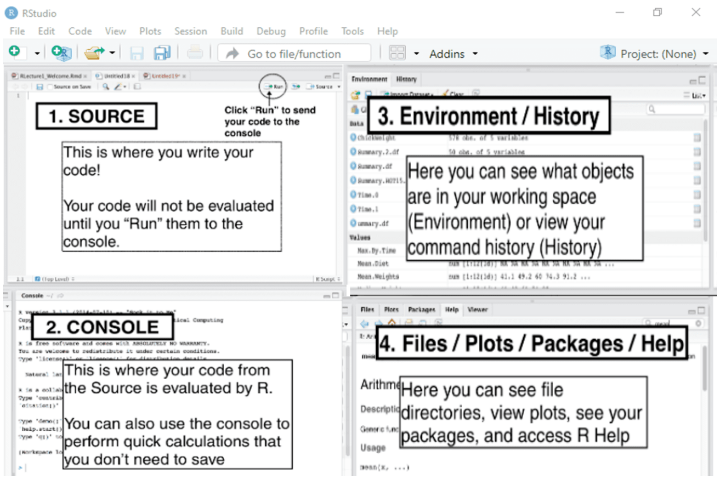


Figure 1.5: Annotated Overview of Key RStudio Panels

1.6.1.1 Source Pane

- This is where you write your R code. Think of it as your notepad or a place to draft your work.
- The code you write here won't run until you specifically tell it to. You do this by clicking the "Run" button or using the keyboard shortcut (`Ctrl + Enter` for Windows or `Cmd + Enter` for Mac).
- The Source Pane is excellent for writing scripts you can save and use later.

1.6.1.2 Console Pane

- This is the heart of R's interaction with you. It's where R evaluates your commands.
- When you "Run" your code from the Source, it appears here, and R processes it immediately.
- You can also directly type commands here for quick calculations or testing. However, anything you type in the console won't be saved if you close RStudio.

1.6.1.3 Environment/History Pane

- **Environment Tab**: This shows all the variables, data frames, and objects you've created in your current R session. It's like a snapshot of everything you're working with.

- **History Tab**: This records every command you've entered, allowing you to track your actions.

1.6.1.4 Files/Plots/Packages/Help Pane

- **Files Tab**: View and manage the files on your computer, similar to a file explorer.
- **Plots Tab**: Displays any graphs or charts you create with your R code.
- **Packages Tab**: Shows the packages (additional tools and functions) available in R and allows you to install, load, or update them as needed.
- **Help Tab**: This is your go-to place for understanding how functions work. If you're unsure about something, R's built-in documentation will be here to guide you.

> How to Run Code in RStudio
>
> To execute the code in the Source Pane:
>
> 1. Place your cursor on the line of code you want to run.
> 2. Press `Ctrl + Enter` (Windows) or `Cmd + Enter` (Mac) to run the current line.
> 3. To run multiple lines, highlight the code block and use the same shortcut.
> 4. Observe the output in the Console Pane.
>
> This practice will help you test code snippets as you progress through the lab.

1.6.2 Performing Basic Calculations in R

R is a powerful and versatile tool for performing all standard arithmetic operations. It supports a range of basic operators, including Addition (`+`), Subtraction (`-`), Multiplication (`*`), Division (`/`), Exponentiation (`^` or `**`), Modulo (`%%`), and Parentheses (`()`), which allow for grouping operations to enforce precedence.

Table 1.1 below summarizes the basic arithmetic operations in R, including their mathematical symbols, corresponding R operators, examples, and results.

Table 1.1: Arithmetic Operations in R: Symbols, Operators, and Examples

Arithmetic Operations	**Mathematical Symbol**	**R Operator**	**Examples**	**Result**
Addition	`+`	`+`	`3 + 2`	`5`

Arithmetic Operations	Mathematical Symbol	R Operator	Examples	Result
Subtraction	-	-	`3 - 1`	2
Multiplication	×	*	`3 * 2`	6
Division	÷	/	`4 / 2`	2
Exponentiation	a^b	^ or **	`2 ^ 2` or `2 ** 2`	4
Parentheses	( )	( )	`2 * (2 + 1)`	6
Modulus	`3 mod 2`	%%	`3 %% 2`	1

Examples:

Here are examples of basic arithmetic operations in R:

```
6 + 12 - 8 # Performs addition and subtraction
```

```
#> [1] 10
```

```
2 * 3 # Multiplies two numbers
```

```
#> [1] 6
```

```
100 / 50 # Divides 100 by 50
```

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

```
3 * 5 / 3 # Combines multiplication and division
```

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

```
3^2 # Raises 3 to the power of 2 (can also use 3**2)
```

```
#> [1] 9
```

Modulo Operation

The modulus (or "mod") operator returns the remainder after division. For example, `9 mod 2 = 1` because dividing `9/2 = 4` leaves a remainder of `1`. In R, this is written as:

```r
9 %% 2 # Returns 1
```

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

Parenthesis or brackets

Parentheses are used to group operations and override the default precedence of operators. In mathematics, you may know this as BODMAS (Brackets, Orders, Division, Multiplication, Addition, Subtraction). In programming, we use BEDMAS: Brackets, Exponentiation, Division, Multiplication, Addition, Subtraction.

For example:

```r
3 * (2 + 3) # Evaluates (2 + 3) first, then multiplies the result by 3
```

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

```r
(3 + 2) * (6 - 4) # Groups operations with parentheses
```

```
#> [1] 10
```

Square Root Calculations

Use the `sqrt()` function to calculate square roots,. For example:

```r
sqrt(125)
```

```
#> [1] 11.18034
```

You can also combine square roots with other operations:

```r
19 / sqrt(19)
```

```
#> [1] 4.358899
```

1.6.3 Comments in R

Comments in R begin with the `#` symbol and are ignored during execution. They are essential for:

1. Making your code easier to understand.
2. Helping others interpret your code.
3. Documenting your thought process.

Example:

```
# Multiplying 2 by 8

2 * 8
```

```
#> [1] 16
```

It is a good practice to add a space after the `#` for better readability:

```
3 + 6 # Adding 3 and 6
```

```
#> [1] 9
```

1.6.4 Comparison Operators

Comparison operators compare values and return `TRUE` or `FALSE`, known as logical. The following are the most common comparison operators in R:

- Equal to (`==`)
- Not equal to (`!=`)
- Greater than (`>`)
- Less than (`<`)
- Greater than or equal to (`>=`)
- Less than or equal to (`<=`)

```
5 == 3 # Returns FALSE
```

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

```
25 != 10 # Returns TRUE
```

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

```
100 > 30 # Returns TRUE
```

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

```
60 >= 45 # Returns TRUE
```

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

```
100 <= 1000 # Returns TRUE
```

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

Common Errors and Debugging Tips

- **Syntax Errors**: Missing commas, unmatched parentheses, or misspelt functions can cause errors.
- **Tip:** Read error messages carefully; they often indicate the line number and type of error.
- **Incorrect Operator Use**: Using = instead of == for comparison.
- **Tip:** Remember that = is for assignment, == is for comparison.

1.6.5 Practice Quiz 1.2

Question 1:

Which pane in RStudio is primarily used for writing and editing R scripts?

a) `Console Pane`

b) `Source Pane`

c) `Environment Pane`

d) `Files Pane`

Question 2:

What does the Environment Tab in RStudio display?

a) Available packages and their statuses

b) Active variables, data frames, and objects in the current session

c) The file directory of your project

d) Graphical plots and visualizations

Question 3:

How can you execute a selected block of code in the Source Pane?

a) Press `Ctrl + S`

b) Press `Ctrl + Enter`

c) Click the "Run" button

d) Both b) and c)

Question 4:

Which pane would you use to install and load R packages?

a) `Source Pane`

b) `Console Pane`

c) `Files Pane`

d) `Packages Tab` within Files/Plots/Packages/Help Pane

Question 5:

Where can you find R's built-in documentation and help files within RStudio?

a) `Source Pane`

b) `Console Pane`

c) `Environment Pane`

d) `Help Tab` within Files/Plots/Packages/Help Pane

See the Solution to Quiz 1.2

1.6.6 Exercise 1.2.1: Basic Calculations

- **Explore RStudio**

  Open RStudio and familiarise yourself with the four panes.

- **Perform Calculations**

  Perform the following calculations in the Source Pane, adding comments where appropriate:

  - $2 + 6 - 12$
  - $4 \times 3 - 8$
  - $81 \div 6$
  - $16 \bmod 3$
  - 2^3
  - $(3 + 2) \times (6 - 4) + 2$

See the Solution to Exercise 1.2.1

1.7 Experiment 1.3: Understanding Atomic Data Types and Variable Assignment

Now that you're comfortable using R for basic arithmetic and understand how to write comments and use comparison operators, let's delve into how R handles different data types. In this section, we'll explore atomic data types and learn how to assign values to variables, which are fundamental concepts in programming.

1.7.1 Atomic Data Types

R works with several atomic data types:

- Numeric: Integers (e.g., `4`, `-2`) or doubles (e.g., `4.7`, `-0.26`)
- Character: Text strings enclosed in quotes (e.g., `"Nigeria"`, `"Hello world"`)
- Logical: Boolean values (`TRUE`, `FALSE`)
- Complex: Represents numbers with real and imaginary parts (e.g., `2 + 3i`, `-1.5 - 4i`).

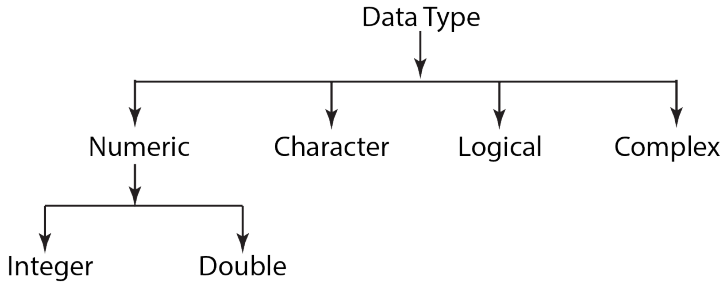


Figure 1.6: Data Types in R Programming

You can determine the data type of an object using the `class()` function.

```
class(2) # Returns "numeric"
```

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

```
class("Anthony Joshua") # Returns "character"
```

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

```
class(TRUE) # Returns "logical"
```

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

```
class(2 + 3i) # Returns "complex"
```

```
#> [1] "complex"
```

1.7.2 Variable Assignment

When working in R, you'll often find yourself storing values, results, or objects for later use. This is where *variables* come in. Variables allow you to hold onto data so that you can reference it easily whenever you need it. Assigning a value to a variable is straightforward in R, and you can do this using the assignment operator, which is `<-` or `=`. While both work, you'll notice that most R users prefer `<-` for assignments[3]. This preference is largely based on convention and readability, as it helps keep your code clean and consistent.

Let's look at a few examples of the variable assignments in action. Here, we'll assign different types of data to variables.

```
number <- 10 # 'number' now holds the value 10

class(number) # Returns "numeric"
```

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

```
state <- "Lagos"

class(state) # Returns "character"
```

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

After running these lines, each variable (`number`, `state`) stores a value you can reuse or modify later in your code. For instance, if you want to check the value of `number`, just type:

```
number
```

```
#> [1] 10
```

And R will display the stored value.

[3] You might wonder why R uses `<-` instead of the `=` symbol that you might see in other programming languages. While you can use `=` for assignment in R, it's generally preferred to use `<-` for clarity. This is partly because `=` is also used in function arguments, so sticking to `<-` makes your code easier to read and helps avoid confusion.

> **Tip**
>
> If you're using a Windows, a quick way to type the assignment operator `<-` is by pressing `ALT + _`; on a Mac, you can use `Option + _`. This shortcut can save time as you write and assign variables in R.

Once you've assigned a value to a variable, you can use that variable in expressions. For instance:

```
x <- 15

y <- 12
```

```
x + 1
```

```
#> [1] 16
```

```
x + y
```

```
#> [1] 27
```

It's also good to know that you can overwrite variables if needed. Say you assigned `x <- 15`, but later, you decide `x` should be `20`. You can just assign it again:

```
x <- 20
```

Now, every time you call `x`, R will know that its value is `20`, not `15` anymore.

1.7.3 Rules for Naming Variables

- Must start with a letter.
- Can contain letters, numbers, underscores `_`, or dots `.` after the first letter.
- No spaces or special characters.
- R is case-sensitive (`Age` and `age` are different variables).

Best Practices

- **Name Your Variables Clearly**: Choose names that describe their data, like `total_sales` or `average_height`, rather than generic names like `x` or `y`. Using clear, descriptive variable names is a best practice because it makes your code easier to understand and maintain. This way, anyone reading your code can quickly grasp the purpose of each variable without needing additional explanations.
- **Avoid Overwriting R's Built-in Functions**: Names like `mean`, `sum`, and `data` are already used by R, so avoid using these as variable names to prevent errors.

In short, variable assignment is like giving a shortcut name to a value or a piece of data. Once assigned, you can call on that name whenever needed, making your code easier to follow and maintain. And remember, R is pretty flexible, so don't worry too much if you make a mistake – you can permanently reassign or update your variables as you go!

1.7.4 Exercise 1.3.1: A Quick Hands-On

Try it yourself! Create a variable named `my_name` and assign your name to it. Then, print a greeting that says:
Hello, [Your Name]!

See the Solution to Exercise 1.3.1

1.7.5 Reflective Exercise 1.3.2: Best Practices and Pitfalls in Variable Naming

In this exercise, you will explore the differences between acceptable and unacceptable variable names in R. Understanding why some naming conventions work and others don't is essential for writing clean, error-free code.

Instructions:

1. Review the Table 1.2 below and identify why each name is either acceptable or unacceptable according to R's variable naming rules.
2. **Answer the following questions**:
   - Why are some variable names acceptable while others are not?
   - What makes the acceptable variable names follow R's rules and best practices?
3. **Reflect** on how these rules can help make your code more readable and easier to debug.

Table 1.2: Comparison of Valid and Invalid Variable Names

S/N	Acceptable Variable Names	Unacceptable Variable Names
1	health.status	health(status)
2	covid_19_cases	covid-19-cases
3	budget2024	2024budget
4	sales_price_2024	sales price 2024

1.7.6 Data Type Conversions

In R, data comes in various types, such as numeric, character, logical, and complex. Sometimes, you'll need to convert data from one type to another—a process known as **typecasting**. This is essential when performing operations requiring data to be in a specific format.

1.7.6.1 Using `as.` Functions for Typecasting

R provides a set of `as.` functions that make typecasting straightforward. These functions allow you to explicitly convert variables to a desired data type. Table 1.3 summarising these functions:

Table 1.3: Common Functions to Convert Between Data Types

Data Type Converting To	How to Do It
Numeric	as.numeric(variable_name)
Character	as.character(variable_name)
Logical	as.logical(variable_name)
Complex	as.complex(variable_name)

Example: Converting Character to Numeric

Suppose you have a variable `weight` that is currently a character string:

```
weight <- "64.45"

class(weight) # Returns "character"
```

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

To perform numerical operations on `weight`, you need to convert it to a numeric type:

```
weight_num <- as.numeric(weight)

class(weight_num) # Returns "numeric"
```

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

Now, `weight_num` is of numeric type, and you can use it in arithmetic calculations:

```
weight_num * 2
```

```
#> [1] 128.9
```

1.7.6.2 Handling `NA` Results

Sometimes, R cannot convert a value to the desired type. When this happens, it returns `NA` (Not Available) and a warning message. This often occurs in the following situations:

- **Converting Non-Numeric Characters to Numeric**: If a character string contains letters or symbols that cannot be interpreted as numbers.
- **Converting Non-Boolean Strings to Logical**: If the string does not represent `TRUE` or `FALSE`.

Example 1:

```
height <- "161.5 cm"

as.numeric(height) # Returns NA with a warning
```

```
#> Warning: NAs introduced by coercion
```

```
#> [1] NA
```

In this case, the string `"161.5 cm"` includes non-numeric characters (`" cm"`), so R cannot convert it to a numeric value.

Example 2:

```r
smiling_face <- "No"

as.logical(smiling_face)
```

```
#> [1] NA
```

Here, `"No"` does not correspond to `TRUE` or `FALSE`, so the conversion fails.

Common Errors and Debugging Tips

- **`NA` Values After Conversion**: Occurs when non-numeric characters are present in a string being converted to numeric.
- **Tip:** Clean your data to ensure it contains only the characters you expect.
- **Variable Not Found**: This occurs when you try to use a variable that hasn't been defined.
- **Tip:** Ensure you've assigned a value to the variable and that it's spelled correctly.

Best Practices

- **Inspect Your Data**: Before converting, check your data to ensure it's in the correct format.
- **Handle NAs Appropriately**: Use functions like `is.na()` to identify and manage `NA` values after conversion.
- **Clean Data When Necessary**: Remove or replace unwanted characters that may prevent successful conversion.

By understanding how to perform data type conversions and handle potential issues, you'll be better equipped to manipulate and analyze data effectively in R.

1.7.7 Practice Quiz 1.3

Question 1:

Which function is used to determine the class of an object in R?

a) `vector()`

b) `c()`

c) `class()`

d) `typeof()`

Question 2:

What will the class of the following object be in R?

```
my_var <- TRUE
```

a) `numeric`

b) `character`

c) `logical`

d) `complex`

Question 3:

Which of the following is an acceptable variable name in R?

a) `2nd_place`

b) `total-sales`

c) `average_height`

d) `user name`

Question 4:

How can you convert a character string `"123"` to a numeric type in R?

a) `to.numeric("123")`

b) `as.numeric("123")`

c) `convert("123", "numeric")`

d) `numeric("123")`

Question 5:

What will be the result of the following R code?

```
weight <- "60.4 kg"
weight_numeric <- as.numeric(weight)
```

a) `60.4`

b) `"60.4"`

c) `NA` with a warning

d) `NULL`

See the Solution to Quiz 1.3

1.7.8 Exercise 1.3.3: Variable Assignment and Data Types

Determine the classes of the following variables and convert them if necessary. Fill in the blanks (indicated by `---`) to complete the code.

```
age <- 15

---(age) # What is the class?
```

```
weight <- "60.4 kg"

class(---)  # What is the class?

# Can you convert weight to numeric?
weight_numeric <- ---(weight)
```

```
smile_face <- "FALSE"

---(smile_face)  # What is the class?

# What happens if you convert smile_face to logical?
smile_face_logical <- as.logical(---)
```

See the Solution to Exercise 1.3.3

Common Errors and Debugging Tips

- **Converting Strings with Units**: Direct conversion may fail due to non-numeric characters.
- **Tip:** Use `gsub()` to remove unwanted characters before conversion.

1.8 Experiment 1.4: Conditional Statements in R

Conditional statements are a vital tool for controlling the flow of your program based on logical conditions. They allow you to execute different blocks of code depending on whether certain conditions are true or false, making your code dynamic and adaptable. The primary constructs are `if`, `else`, `else if`.

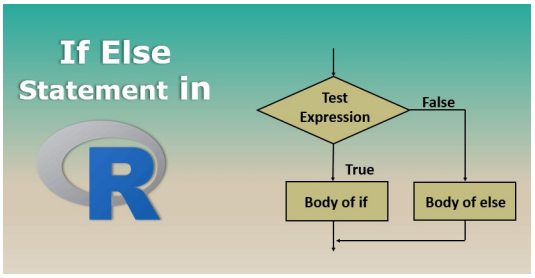


Figure 1.7: If-Else Statement in R Programming

1.8.1 The `if` Statement

This is the most basic conditional construct. It executes code only if a specified condition is `TRUE`.

```
x <- 5
if (x > 3) {
  print("x is greater than 3")
}
```

```
#> [1] "x is greater than 3"
```

1.8.2 The `else` Statement

Provides an alternative set of instructions if the `if` condition is `FALSE`.

```
x <- 2
if (x > 3) {
  print("x is greater than 3")
} else {
  print("x is not greater than 3")
}
```

```
#> [1] "x is not greater than 3"
```

1.8.3 The `else if` Statement

The `else if` can be used to check situations with multiple conditions sequentially. It provides an additional condition check after the initial `if` statement.

```
x <- 3
if (x > 5) {
  print("x is greater than 5")
} else if (x == 5) {
  print("x is equal to 5")
} else {
  print("x is less than 5")
}
```

```
#> [1] "x is less than 5"
```

Using Logical Operators

You can combine conditions using logical operators:

- AND (`&`): Both conditions must be `TRUE`.
- OR (`|`): At least one condition must be `TRUE`.
- NOT (`!`): Inverts the logical value.

Example using AND (`&`):

```r
x <- 8
y <- 12

if (x < 10 & y > 10) {
  print("Both conditions are true")
} else {
  print("At least one condition is false")
}
```

In this example, the `if` statement checks if both `x < 10` **and** `y > 10` are `TRUE`. Since both conditions are `TRUE`, the output will be:

```
"Both conditions are true"
```

Example using OR (`|`):

```r
a <- 3
b <- 20

if (a < 5 | b > 25) {
  print("At least one condition is true")
} else {
  print("Neither condition is true")
}
```

In this example, the `if` statement checks if either `a` is less than 5 **or** `b` is greater than 25. Since `a < 5` is `TRUE`, the output will be:

```
"At least one condition is true"
```

Example using NOT (`!`):

```r
c <- FALSE

if (!c) {
  print("The condition is false")
} else {
  print("The condition is true")
}
```

Here, the `if` statement uses the NOT operator to check if `c` is not `TRUE`. Since `c` is `FALSE`, `!c` becomes `TRUE`, and the output will be:

```
"The condition is false"
```

1.8.4 The `ifelse()` Function

The `ifelse()` function is a vectorised form of conditional statements. It applies a condition to each element of a vector and returns one value if the condition is `TRUE` and another value if the condition is `FALSE`. The syntax is as follows:

```
ifelse(condition, value_if_true, value_if_false)
```

Where:

- **`condition`**: A logical expression to evaluate.
- **`value_if_true`**: The value to return if the condition is `TRUE`.
- **`value_if_false`**: The value to return if the condition is `FALSE`.

Example:

```
number <- 21

ifelse(number %% 2 == 0, "Even", "Odd")
```

```
#> [1] "Odd"
```

In this example, `ifelse()` checks whether `number %% 2 == 0` (that is, whether `number` is even). If it is even, it returns `"Even"`; otherwise, it returns `"Odd"`. For more advanced uses, see Chapter 2.5.5.5.

1.8.5 The `switch` Function

The `switch()` function is a control flow statement that allows you to execute different pieces of code based on the value of an expression. It's particularly useful when you have multiple conditions to check and want a cleaner alternative to lengthy `if...else` statements.

There are two primary ways to use `switch()` in R:

1. **Numeric Switching**: Where the expression evaluates to a numeric index.
2. **Character Switching**: Where the expression evaluates to a character string matching one of the named alternatives.

The general structure of `switch()` function is as follows:

```
switch(EXPR,
  ...
)
```

Where:

- `EXPR`: An expression that evaluates to a numeric value or a character string.
- `...`: A sequence of alternatives (unnamed or named arguments).

The `switch()` function uses the same syntax for both numeric and character expressions. The behavior of the function depends on the type of the `EXPR` argument you provide.

When to Use `switch()`

- When you have a variable that can take on multiple known values and you want to execute different code based on each value.
- To improve code readability over multiple `if...else` statements.
- When performance is a consideration, as `switch()` can be more efficient than multiple `if...else` checks.

Example 1: Day of the Week Activities Using Character Switching

Suppose you want to plan activities based on the day of the week.

```r
day <- "Saturday"

activity <- switch(day,
  Monday = "Go to the gym",
  Tuesday = "Attend a cooking class",
  Wednesday = "Work from home",
  Thursday = "Meet friends for dinner",
  Friday = "Watch a movie",
  Saturday = "Go hiking",
  Sunday = "Rest and recharge",
  "Invalid day"
)

print(paste("Today's activity:", activity))
```

```
#> [1] "Today's activity: Go hiking"
```

Explanation

- **Variable `day`**: Contains the day of the week as a string.
- **Using `switch()`**:
  - Matches `day` against the provided day names.
  - If a match is found, returns the corresponding activity.
  - If no match is found, returns `"Invalid day"`.

Example 2: Mapping Codes to Descriptions Using Character Switching

Suppose you have status codes that need to be mapped to descriptive messages.

```r
status_code <- 404

message <- switch(as.character(status_code),
  "200" = "OK: The request has succeeded.",
  "301" = "Moved Permanently: The resource has moved.",
  "400" = "Bad Request: The request could not be understood.",
  "401" = "Unauthorized: Authentication is required.",
  "404" = "Not Found: The resource could not be found.",
  "500" = "Internal Server Error: The server encountered an error.",
  "Unknown Status Code"
)
```

```r
print(message)
```

```
#> [1] "Not Found: The resource could not be found."
```

Explanation:

- **Variable `status_code`**: Contains an HTTP status code.
- **Converting to Character**: `as.character(status_code)` because `switch()` with character matching requires a string.
- **Using `switch()`**:
  - Matches the status code against the provided cases.
  - Returns the corresponding message or `"Unknown Status Code"` if no match is found.

Example 3: Simple Calculator Using Numeric Switching

Let's create a simple calculator that performs operations based on a numeric choice.

```r
# User inputs
num1 <- 10
num2 <- 5
# Options: 1 for addition, 2 for subtraction, 3 for multiplication, 4 for division
choice <- 3

# Use switch() to perform the selected operation
result <- switch(choice,
  num1 + num2, # If choice == 1
  num1 - num2, # If choice == 2
  num1 * num2, # If choice == 3
  if (num2 != 0) num1 / num2 else "Division by zero error", # If choice == 4
  "Invalid operation"
) # Default if choice > number of cases

# Display the result
print(paste("The result is:", result))
```

```
#> [1] "The result is: 50"
```

Explanation

- **Variables**:
  - `num1`, `num2`: Numbers to operate on.
  - `choice`: Numeric choice of operation.
- **Using `switch()`**:
  - Since `choice` is numeric, `switch()` selects the expression based on position.
    - 1: `num1 + num2`
    - 2: `num1 - num2`
    - 3: `num1 * num2`
    - 4: Division with a check for division by zero.
  - If `choice` exceeds the number of provided alternatives (4), the default `"Invalid operation"` is returned.

Common Errors and Debugging Tips

- **Missing Braces `{}`**: Forgetting to include braces in `if` statements.
- **Tip:** Always include braces even if there's only one line of code inside.
- **Incorrect Logical Operators**: Using `&&` instead of `&` or `||` instead of `|` can lead to unexpected results.
- **Tip:** Use `&` and `|` for vectorized operations, which is common in R.

1.8.6 Practice Quiz 1.4

Question 1:

What will be the output of the following R code?

```r
number <- 10
if (number %% 2 == 0) {
  print("Even")
} else {
  print("Odd")
}
```

a) `Odd`

b) `Even`

c) `TRUE`

d) `FALSE`

Question 2:

Which logical operator in R returns `TRUE` only if both conditions are `TRUE`?

a) `|` (OR)

b) `&` (AND)

c) `!` (NOT)

d) `^` (XOR)

Question 3:

In the `switch()` function, what does the following code return when `choice` is 3?

```
num1 <- 10
num2 <- 5
choice <- 3

result <- switch(choice,
  num1 + num2,
  num1 - num2,
  num1 * num2,
  "Invalid operation"
)

print(result)
```

a) `15`

b) `5`

c) `50`

d) `"Invalid operation"`

Question 4:

What is the purpose of including a default case in a `switch()` statement?*

a) To handle cases where the expression matches multiple conditions

b) To execute a block of code if none of the specified cases match

c) To prioritize certain cases over others

d) To initialize variables within the switch

Question 5:

Which of the following uses the NOT (`!`) operator correctly in an `if` statement?

a)

```
if (!c) {
  print("The condition is false")
}
```

b)

```
if (c!) {
  print("The condition is false")
}
```

c)

```
if (c != TRUE) {
  print("The condition is false")
}
```

d) Both a) and c)

See the Solution to Quiz 1.4

1.8.7 Exercise 1.4.1: Conditional Statements

Task 1

What is the output of the following code?

```r
number <- 10

if (number %% 2 == 0) {
  print("Even")
} else {
  print("Odd")
}
```

Task 2

Given `m <- 5` and `n <- 7`, write code that prints:

- "m is greater than n" if `m > n`
- "m is less than n" if `m < n`
- "m and n are equal" if `m == n`

See the Solution to Exercise 1.4.1

1.8.8 Exercise 1.4.2: Menu Selection Using `switch()`

Simulate a simple text-based menu where a user selects an option. Use the `switch()` function to determine the action based on the user's selection.

Your Task:

1. **Simulate User Input**:
   - Assign a value to a variable `option` to represent the user's selection.
   - Possible options: `"balance"`, `"deposit"`, `"withdraw"`, `"exit"`.
2. **Use the `switch()` Function**:
   - Match the value of `option` to the appropriate case using `switch()`.
   - For each case, assign a message that describes the action.

Possible Options and Messages:

- **"balance"**: Display "Your current balance is $1,000."
- **"deposit"**: Display "Enter the amount you wish to deposit."
- **"withdraw"**: Display "Enter the amount you wish to withdraw."
- **"exit"**: Display "Thank you for using our banking services."
- **Default**: Display "Invalid selection. Please choose a valid option."

3. **Include a Default Case**:
    - If the user input does not match any of the specified options, provide a default message indicating an invalid selection.
4. **Display the Message**:
    - Use `print()` to display the message corresponding to the user's selection.

Here's a starting point for your code:

```
# Simulate user input
option <- "---"  # Options could be "balance", "deposit", "withdraw", "exit"

# Use switch() to determine the action
message <- switch(...,
                  balance = "You have $1,000 in your account.",
                  deposit = ...,
                  withdraw = "How much would you like to withdraw?",
                  "Invalid selection. Please choose a valid option.")

# Display the message
print(...)
```

Replace the `...` with the correct values and complete the exercise!

See the Solution to Exercise 1.4.2

1.8.9 Exercise 1.4.3: Mini-Project - Basic Calculator in R

Using what you've learned about arithmetic operations, variables, and conditional statements, create a simple calculator program in R that:

- Prompts the user to enter two numbers.
- Asks the user to choose an operation: addition, subtraction, multiplication, or division.
- Performs the operation and displays the result.

Hint:

- Use the `readline()` function to get user input.
- Convert the input to numeric using `as.numeric()`.
- Use a `switch()` statement to handle the operation selection.

See the Solution to Exercise 1.4.3

1.9 Further Reading

To further enhance your R programming skills, here are some excellent resources:

- **RStudio Cheat Sheets:**
  https://www.rstudio.com/resources/cheatsheets
- **Introduction to R by RStudio:**
  https://education.rstudio.com/learn/beginner
- **YaRrr! The Pirate's Guide to R by Nathaniel D. Phillips**
  https://bookdown.org/ndphillips/YaRrr
- **R for Data Science by Hadley Wickham, Mine Çetinkaya-Rundel, and Garrett Grolemund**
  https://r4ds.hadley.nz
- **R for Data Science: Exercise Solutions by Jeffrey B. Arnold**
  https://jrnold.github.io/r4ds-exercise-solutions
- **Big Book of R by Oscar Baruffa**
  https://www.bigbookofr.com

1.10 Reflective Summary

Congratulations on completing **Lab 1**! You've taken your first steps into R programming and have covered much ground:

- **Installed R and RStudio**
  Set up your programming environment.
- **Explored the RStudio Interface**
  Learned how to use RStudio's four main panes to write, execute, and manage your R code effectively.
- **Performed Basic Calculations in R**
  Practiced using R for arithmetic operations and understood operator precedence.
- **Understood Atomic Data Types and Variable Assignment**
  Explored numeric, character, and logical data types, and learned how to identify and convert between them.
- **Used Conditional Statements in R**
  Controlled the flow of your programs using `if`, `else if`, and `else` statements, and used logical operators.

Take a moment to reflect on how these foundational skills can be applied to solving real-world problems. As you progress through this book, each lab will build on these essentials, guiding you toward proficiency in data analysis with R.

Keep experimenting with the code and exploring built-in functions. The more you practice, the more confident and comfortable you'll become.

> **What's Next?**
>
> In the next lab, we'll delve into R's fundamental data structures, including vectors, matrices, and data frames—key data manipulation and analysis tools.

2 Understanding Data Structures

2.1 Introduction

Welcome to **Lab 2**! In this lab, we'll delve into the fundamental data structures in R that are essential for data analysis and manipulation. Understanding these data structures will empower you to handle data efficiently and perform various operations crucial for statistical analysis and data science tasks.

2.2 Learning Objectives

By completing this lab, you will be able to:

- **Identify Fundamental Data Structures**
  Recognize and describe the key characteristics of vectors, matrices, data frames, and lists in R.
- **Create Data Structures**
  Construct vectors, matrices, data frames, and lists using appropriate functions and syntax in R.
- **Manipulate Data Structures**
  Perform operations such as indexing, slicing, and modifying elements within vectors, matrices, data frames, and lists.
- **Apply Appropriate Operations and Functions**
  Utilize relevant R functions and operators to perform calculations and transformations specific to each data type.
- **Demonstrate Understanding Through Application**
  Solve problems and complete exercises that require the correct application of operations and functions to manipulate and analyze data within these structures.

By completing this lab, you'll have a solid foundation in working with vectors, matrices, data frames, and lists, setting you up for success in more advanced topics.

2.3 Prerequisites

Before starting this lab, you should have:

- Completed **Lab 1** or have basic knowledge of R programming fundamentals.
- Familiarity with basic arithmetic operations and variable assignment in R.
- An interest in learning how to manage and manipulate data effectively.

2.4 Exploring Data Structures in R

R offers several fundamental data structures to handle diverse data and analytical needs. These include vectors, matrices, data frames, and lists.

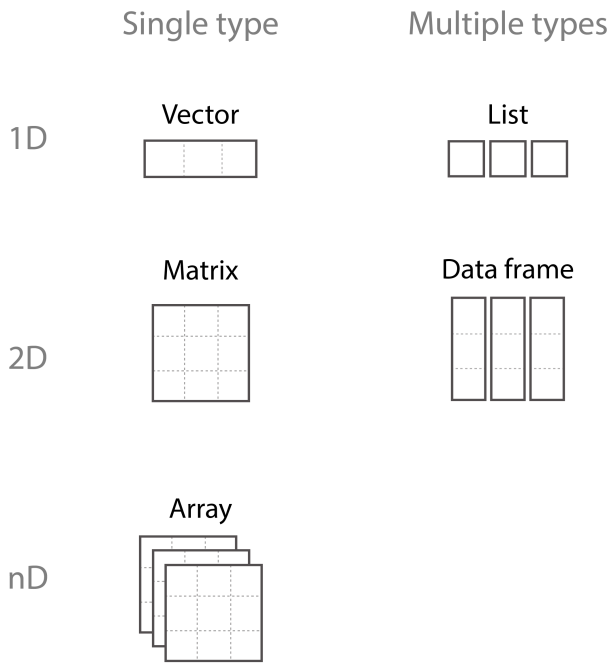


Figure 2.1: Data Structures in R Programming

2.5 Experiment 2.1: Vector

A vector is a one-dimensional array in R that stores elements of the same data type. Vectors are the most basic and frequently used data structures in R and are essential for data manipulation

and analysis.

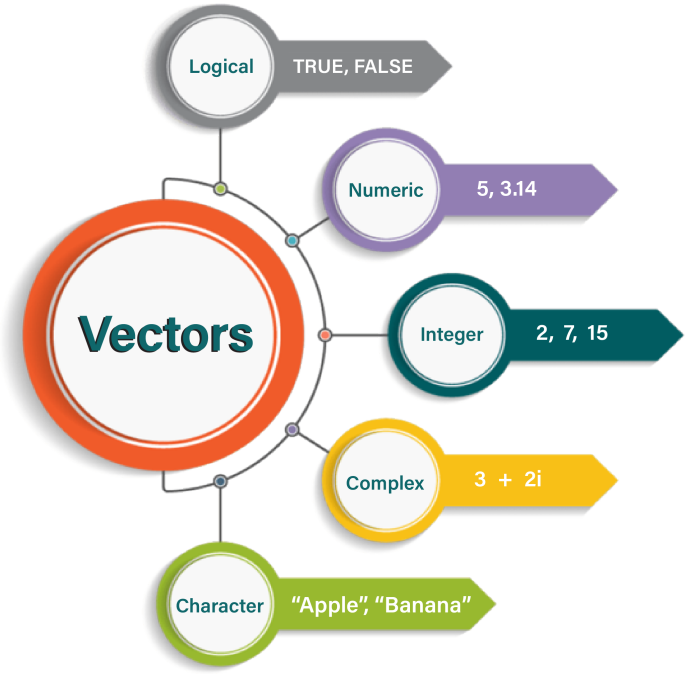


Figure 2.2: Types of Vectors in R Programming

2.5.1 Creating a Vector

You can create a vector in R using the `c()` function[1], which stands for "combine" or "concatenate."

Example 1: Eye Colour

```
eye_colour <- c("Green", "Blue", "Brown", "Green", "Green", "Brown", "Blue")

eye_colour
```

```
#> [1] "Green" "Blue"  "Brown" "Green" "Green" "Brown" "Blue"
```

Here, `eye_colour` is a character vector, as it contains text values and the `c()` function makes it easy to combine individual elements into a single vector.

[1]The `c()` function is used to combine or concatenate values into a vector. It serves as a fundamental tool for creating vectors of various data types, such as numeric, character, or logical values.

Best Practice:

When using the `c()` function, it's a good practice to add a space after each comma. This makes your code cleaner and easier to read.

Example 2: Product Prices

```
# Vector of product prices in USD

product_prices <- c(19.99, 5.49, 12.89, 99.99, 49.95)

product_prices
```

```
#> [1] 19.99  5.49 12.89 99.99 49.95
```

2.5.2 Checking the Type of a Vector

To determine the data type of a vector, use the `class()` function:

```
class(product_prices)
```

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

2.5.3 Length of a Vector

The length of a vector refers to the total number of elements it contains. You can determine this using the `length()` function, which returns an integer representing the total count of elements in the vector.

```
covid_confirmed <- c(31, 30, 37, 25, 33, 34, 26, 32, 23, 45)

length(covid_confirmed)
```

```
#> [1] 10
```

2.5.4 Advanced Vector Creation

While the simplest way to create a vector is by directly assigning values, R provides powerful tools for generating and manipulating vectors tailored to specific needs. This section explores advanced methods like `:`, `c()`, `seq()`, `rep()`, and `sample()` to create flexible, efficient, and custom vectors.

2.5.4.1 : Operator - Quick Sequence Generation

The : operator generates a sequence of integers between a starting and an ending value.

Syntax:

```
start:end
```

Where:

- `start`: The starting value of the sequence.
- `end`: The ending value of the sequence.

Examples:

- Create a sequence from 1 to 10:

```
sequence <- 1:10
sequence
```

```
#>  [1]  1  2  3  4  5  6  7  8  9 10
```

- Create a descending sequence:

```
reverse_sequence <- 10:1
reverse_sequence
```

```
#>  [1] 10  9  8  7  6  5  4  3  2  1
```

> **Use Case**
>
> The : operator is ideal for quickly generating ranges for indexing, loops, or other operations requiring consecutive integers.

2.5.4.2 `c()` Function - Vector Concatenation

The `c()` function combines individual elements or existing vectors into a single vector.

Syntax:

```
c(element1, element2, ..., elementN)
```

Where:

- `element1`, `element2`, ..., `elementN`: The individual values or vectors to be combined into a single vector.

Examples:

- Combine numeric elements:

```
numbers <- c(2, 4, 6, 8)
numbers
```

```
#> [1] 2 4 6 8
```

- Concatenate multiple vectors:

```
vector1 <- c(1, 2, 3)
vector2 <- c(4, 5, 6)

combined <- c(vector1, vector2)
combined
```

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

- Combine mixed data types:

```
mixed <- c(10, "Apple", TRUE)
mixed
```

```
#> [1] "10"    "Apple" "TRUE"
```

> **Use Case**
>
> All elements are coerced to the same data type (character in this case). Use `c()` to append data, merge multiple vectors, or build vectors from scratch.

2.5.4.3 `seq()` Function - Custom Sequence Generation

The `seq()` function allows for more flexible sequence generation compared to the `:` operator. You can specify the step size or desired length of the sequence.

Syntax:

```
seq(from, to, by, length)
```

Where:

- `from`: The starting value of the sequence.
- `to`: The ending value of the sequence.
- `by`: The increment (step size) between values. Defaults to `1` if not specified.
- `length`: The desired number of elements in the sequence. Overrides `by` if specified.

Examples:

- Generate a sequence with a positive step size:

```
custom_step <- seq(1, 10, by = 2)
custom_step
```

```
#> [1] 1 3 5 7 9
```

- Generate a sequence with a negative non-integer step size:

```
descending <- seq(5, 1, by = -0.5)
descending
```

```
#> [1] 5.0 4.5 4.0 3.5 3.0 2.5 2.0 1.5 1.0
```

- Generate a sequence of a specific length:

```
custom_length <- seq(1, 10, length = 5)
custom_length
```

```
#> [1]  1.00  3.25  5.50  7.75 10.00
```

> **Use Case**
>
> The `seq()` function is perfect for creating sequences with integer (non-integer) steps or evenly spaced data points.

2.5.4.4 `rep()` Function - Value Replication

The `rep()` function replicates values multiple times, creating patterns or duplications as needed.

Syntax:

```
rep(x, times, each)
```

Where:

- `x`: The vector or value to be replicated.
- `times`: The number of times to repeat the entire vector.
- `each`: The number of times to repeat each element of the vector.

Examples:

- Repeat a value multiple times:

```
rep("God is good!", 5)
```

```
#> [1] "God is good!" "God is good!" "God is good!" "God is good!"
#> [5] "God is good!"
```

- Repeat a vector multiple times:

```
rep(1:3, times = 2)
```

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

- Repeat each element of a vector:

```
rep(c("Female", "Male"), each = 2)
```

```
#> [1] "Female" "Female" "Male"   "Male"
```

> **Use Case**
>
> Use `rep()` to generate repetitive patterns for simulation, modeling, or experimental data.

2.5.4.5 `sample()` Function - Random Sampling

The `sample()` function generates random samples from a vector, with or without replacement.

Syntax:

```
sample(x, size, replace = FALSE, prob = NULL)
```

Where:

- `x`: The vector to sample from.
- `size`: The number of samples to draw.
- `replace`: Logical; whether sampling is with replacement (`TRUE`) or without replacement (`FALSE`).
- `prob`: A vector of probabilities for each element in `x`.

Examples:

- Random sample without replacement:

```
fruit <- c("Apple", "Banana", "Cherry", "Orange", "Pineapples", "Grape", "Pawpaw")

sample(x = fruit, size = 4)
```

```
#> [1] "Grape"  "Orange" "Banana" "Pawpaw"
```

Example output; results vary due to randomness

- Random sample with replacement:

```
sample(1:10, 5, replace = TRUE)
```

```
#> [1] 1 9 7 6 7
```

Example output; results vary due to randomness

Set a Seed for Reproducibility

When you perform random sampling in R, the results may vary each time you run the code. By using `set.seed()`, you ensure that the random number generator starts from the same point, producing consistent results every time the code is executed.

Examples

```r
# Set a seed for consistent results
set.seed(123)

# Generate a random sample of 5 numbers
sample(1:10, 5)
```

```
#> [1]  3 10  2  8  6
```

This output will always be the same with `set.seed(123)`

- Weighted Sampling:

  Weighted sampling is a method of random sampling where each element in the population is assigned a probability of being selected. These probabilities, or weights, determine the likelihood of an element being chosen, allowing certain elements to have a higher or lower chance of selection compared to others.

  Example:

  Let's simulate 10 flips of a biased coin, where the probability of getting a Head (H) is 0.7 and the probability of a Tail(T) is 0.3.

  ```r
  # Sample with probabilities

  set.seed(1923)

  coin_face <- c("H", "T") # Possible outcomes

  prob <- c(0.7, 0.3) # Higher probability for "Head"

  # Perform weighted sampling

  sampled_prob <- sample(coin_face, prob = prob, size = 10, replace = TRUE)

  sampled_prob
  ```

  ```
  #>  [1] "H" "H" "H" "H" "T" "T" "T" "T" "T" "H"
  ```

> **Use Case**
>
> Use `sample()` for bootstrapping, simulations, or creating random subsets from data.

2.5.5 Vector Operations

When working with vectors, most basic operations are applied element-by-element, making it both intuitive and efficient to transform and analyse collections of data.

2.5.5.1 Arithmetic Operations

Arithmetic operations on vectors are performed element-wise. For example, consider a vector of product prices:

```
# Vector of product prices in USD

product_prices <- c(19.99, 5.49, 12.89, 99.99, 49.95)
```

If we want to apply a 10% discount to all prices, we can multiply the entire vector by `0.9`:

```
# Apply a discount of 10% to all product prices

discounted_prices <- product_prices * 0.9

discounted_prices
```

```
#> [1] 17.991  4.941 11.601 89.991 44.955
```

In this example, each element in `product_prices` is multiplied by 0.9, producing the `discounted_prices` vector.

> Common Pitfall
>
> - **Data Type Coercion**: Mixing data types in a vector can lead to unexpected results.
> - **Tip**: Ensure all elements are of the same data type to avoid automatic coercion.

2.5.5.2 Mathematical and Statistical Functions

R provides many built-in functions that operate on vectors and return summary values or transformed vectors. These functions are vectorised, meaning they automatically apply to every element where relevant. Examples include:

- `sum(product_prices)` – returns the sum of all prices.
- `mean(product_prices)` – returns the average price.
- `max(product_prices)` / `min(product_prices)` – returns the maximum or minimum value.
- `sd(product_prices)` / `var(product_prices)` – returns the standard deviation or variance of the product prices.
- `round(product_prices, digits = 1)` – rounds all prices to one decimal place.

2.5.5.3 Element-wise Functions

Functions like `sqrt()`, `log()`, `exp()`, and `abs()` apply mathematically to each element:

```
sqrt_prices <- sqrt(product_prices)
sqrt_prices
```

```
#> [1] 4.471018 2.343075 3.590265 9.999500 7.067531
```

```
log_prices <- log(product_prices)
log_prices
```

```
#> [1] 2.995232 1.702928 2.556452 4.605070 3.911023
```

2.5.5.4 Sorting and Ordering

You can also sort vectors or determine the order of elements:

```
product_prices <- c(19.99, 5.49, 12.89, 99.99, 49.95)

sorted_prices <- sort(product_prices)
sorted_prices
```

```
#> [1]  5.49 12.89 19.99 49.95 99.99
```

2.5.5.5 Vectorised Conditional Operations with `ifelse()`

The `ifelse()` statement is a vectorised conditional function in R. It evaluates a condition for each element of a vector, returning one value if the condition is `TRUE` and another value if it is `FALSE`. This makes it an essential function for data transformation and analysis.

Example 1: Categorise Product Prices

Suppose we have a vector of product prices in USD, and we want to categorise them as "Affordable" or "Expensive" based on whether they are below or above $50.

```
# Vector of product prices in USD
product_prices <- c(30, 75, 50, 20, 100)

# Categorise prices as "Affordable" or "Expensive"
price_category <- ifelse(product_prices < 50, "Affordable", "Expensive")

# Print the result
price_category
```

```
#> [1] "Affordable" "Expensive"  "Expensive"  "Affordable" "Expensive"
```

Example 2: Apply a Discount for Expensive Products

If a product price is above $50, apply a 10% discount. Otherwise, keep the price unchanged.

```
# Apply discount for products above $50
discounted_prices <- ifelse(product_prices > 50, product_prices * 0.9, product_prices)

# Print the result
discounted_prices
```

```
#> [1] 30.0 67.5 50.0 20.0 90.0
```

2.5.6 Vector selection

To select elements of a vector, use square brackets `[]` with the index of the element(s). R indexing starts at 1 (i.e. R uses 1-based indexing).

For example:

Weekday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
index	1	2	3	4	5	6	7

Example 1

```
weekday <- c(
  "Monday", "Tuesday", "Wednessday", "Thursday", "Friday", "Saturday", "Sunday"
)
```

Access the first weekday:

```
weekday[1]
```

```
#> [1] "Monday"
```

Access the second weekday:

```
weekday[2]
```

```
#> [1] "Tuesday"
```

Access multiple elements

```
weekday[c(2, 4)]
```

```
#> [1] "Tuesday"  "Thursday"
```

2.5.7 Reflection Question 2.1.1

Why is it important to know that R uses 1-based indexing?

See the Solution to Reflection Question 2.1.1

In addition to selecting elements by their positions, you can select elements of a vector based on conditions using **comparison operators**. This method involves creating a logical vector of `TRUE` and `FALSE` values by applying a condition to the vector, and then using this logical vector to index the original vector.

Example 2

Let's start with a numeric vector representing daily temperatures:

```r
# Create a numeric vector of temperatures
temperatures <- c(72, 65, 70, 68, 75, 80, 78)
```

Select temperatures greater than 70 degrees:

```r
# Apply the condition and select elements
temperatures[temperatures > 70]
```

```
#> [1] 72 75 80 78
```

Explanation

- `temperatures > 70` creates a logical vector: `[TRUE, FALSE, FALSE, FALSE, TRUE, TRUE, TRUE]`.
- `temperatures[temperatures > 70]` selects elements where the condition is `TRUE`.

Select temperature that are even.

```r
temperatures[temperatures %% 2 == 0]
```

```
#> [1] 72 70 68 80 78
```

Explanation

- `temperatures %% 2` computes the remainder when each temperature is divided by 2.
- `temperatures %% 2 == 0` returns `TRUE` for even temperature.

Example 3

Using our `weekday` vector:

```r
weekday <- c(
  "Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
  "Saturday", "Sunday"
)
```

Select weekdays that are “Saturday” or “Sunday”:

```r
# Create a logical vector for the condition
is_weekend <- weekday == "Saturday" | weekday == "Sunday"

# Select elements based on the condition
weekend_days <- weekday[is_weekend]

# Display the result
weekend_days
```

```
#> [1] "Saturday" "Sunday"
```

Explanation

- `weekday == "Sunday"` returns `[FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, TRUE]`.
- Using `|` (logical OR) combines the two conditions.
- The final logical vector is `[FALSE, FALSE, FALSE, FALSE, FALSE, TRUE, TRUE]`.

Selecting Weekdays (Excluding Weekends)

```r
# Create a logical vector for weekdays
is_weekday <- !(weekday %in% c("Saturday", "Sunday"))

# Select elements based on the condition
weekday_days <- weekday[is_weekday]

# Display the result
weekday_days
```

```
#> [1] "Monday"    "Tuesday"   "Wednesday" "Thursday"  "Friday"
```

Explanation

- `weekday %in% c("Saturday", "Sunday")` checks if each element is in the vector `c("Saturday", "Sunday")`.
- The `%in%` operator returns `[FALSE, FALSE, FALSE, FALSE, FALSE, TRUE, TRUE]`.
- Using `!` negates the logical vector to `[TRUE, TRUE, TRUE, TRUE, TRUE, FALSE,`

`FALSE]`.

Code Style Guidelines

- **Naming Conventions:** Use descriptive variable names in `snake_case`.
- **Indentation:** Use consistent indentation for readability.
- **Comments:** Explain complex code with comments.

2.5.8 Exercise 2.1.1: Vector Selection

Given the monthly sales figures (in units) for a product over a year:

`120, 135, 150, 160, 155, 145, 170, 180, 165, 175, 190, 200`

Your Tasks:

1. Create a vector named `monthly_sales` containing the data.
2. Access the sales figures for March, June, and December.
3. Find the sales figures that are less than 60
4. Calculate the average sales for the first quarter (January to March).
5. Extract the sales figures for the last month of each quarter of the year

See the Solution to Exercise 2.1.1

2.5.9 Factor Vectors

Factors are a fundamental data structure in R used to represent categorical variables, where each value belongs to a specific category or "level." Unlike character vectors, factors store categorical data more efficiently and are essential in statistical modeling, especially when using functions like `summary()` or performing analyses that require categorical distinctions.

2.5.9.1 Why Use Factors?

- Statistical Modeling: Factors are crucial in statistical modeling as they explicitly designate categorical variables, allowing R's modeling functions to handle them appropriately.
- Efficient Storage: Factors store categorical data more efficiently than character vectors by using integer codes under the hood.

- Data Integrity: By restricting data to predefined categories (levels), factors help maintain data integrity and reduce errors due to invalid entries.
- Using Factors in Plots: Factors are particularly useful when creating plots, as they allow for meaningful ordering of categories.

2.5.9.2 Creating a Factor Vector

You can create a factor vector using the `factor()` function, converting a character or numeric vector into a factor.

Example 1: Product Categories

Suppose we have a vector of product categories:

```
# Vector of product categories
product_categories <- c(
  "Electronics", "Clothing", "Electronics",
  "Furniture", "Clothing", "Electronics"
)

product_categories
```

```
#> [1] "Electronics" "Clothing"    "Electronics" "Furniture"   "Clothing"
#> [6] "Electronics"
```

Converting to a factor:

```
# Convert to factor
product_categories_factor <- factor(product_categories)

# Display the factor vector
product_categories_factor
```

```
#> [1] Electronics Clothing    Electronics Furniture   Clothing    Electronics
#> Levels: Clothing Electronics Furniture
```

Inspecting the Factor Levels:

```
levels(product_categories_factor)
```

```
#> [1] "Clothing"    "Electronics" "Furniture"
```

Checking the Data Type:

```
class(product_categories_factor)
```

```
#> [1] "factor"
```

If you already have a character vector, you can convert it to a factor vector using the `as.factor()` function.

Example 2: Ethnicity Categories

```
ethnicity <- c(
  "African", "Asian", "Latin American", "African", "Latin American",
  "Asian", "African", "European"
)

ethnicity_factor <- as.factor(ethnicity)

ethnicity_factor
```

```
#> [1] African        Asian          Latin American African
#> [5] Latin American Asian          African        European
#> Levels: African Asian European Latin American
```

> **Note**
>
> The levels are automatically determined and sorted alphabetically unless specified otherwise.

2.5.9.3 Ordered Factors

Sometimes, categorical variables have a natural ordering (e.g., "Low" < "Medium" < "High"). In such cases, you can create an ordered factor by specifying the levels in the desired order and setting `ordered = TRUE`.

Example 3: Satisfaction Ratings

```r
# Vector of satisfaction ratings

satisfaction_ratings <- c(
  "High", "Medium", "Low", "High",
  "Medium", "Low", "High"
)

# Create an ordered factor
satisfaction_factors <- factor(satisfaction_ratings,
  levels = c("Low", "Medium", "High"),
  ordered = TRUE
)

# Display the ordered factor
satisfaction_factors
```

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

Checking the data type:

```r
class(satisfaction_factors)
```

```
#> [1] "ordered" "factor"
```

2.5.9.4 Using `summary()` with Factors:

The `summary()` function provides a count of each category in a factor vector:

```r
# Summarise the satisfaction factors

summary(satisfaction_factors)
```

```
#>    Low Medium   High
#>      2      2      3
```

2.5.9.5 Converting Factors to Numeric Values

Be cautious when converting factors to numeric values. Direct conversion using `as.numeric()` will return the internal integer codes representing the factor levels, which may not correspond to the actual values you expect.

Incorrect Way:

```
# Incorrect conversion
as.numeric(product_categories_factor)
```

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

> Tip
>
> The numbers represent the position of each level in the levels attribute, not the original data.

Correct Way:

First, convert the factor to a character vector, then to numeric (if applicable):

```
# Correct conversion (if levels are numeric strings)
as.numeric(as.character(product_categories_factor))
```

```
#> Warning: NAs introduced by coercion
```

```
#> [1] NA NA NA NA NA NA
```

> Warning
>
> This will produce `NA`s because the levels are not numeric strings. For factors with numeric levels, this method works correctly.

Example with Numeric Levels:

```
# Numeric factor example
numeric_factor <- factor(c("1", "3", "5", "2"))

# Correct conversion
numeric_values <- as.numeric(as.character(numeric_factor))

numeric_values
```

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

Common Pitfalls

- **Unintended Ordering**: By default, factors are unordered. If your categorical variable has a natural order, specify it using the `levels` argument and set `ordered = TRUE`.

```r
# Ordered factor example
factor_variable <- factor(c("Low", "Medium", "High"),
  levels = c("Low", "Medium", "High"),
  ordered = TRUE
)
```

- **Converting Factors**: Always be cautious when converting factors to numeric types to avoid unexpected results. Use `as.character()` before `as.numeric()` if necessary.
- **Missing Levels**: If you specify levels that are not present in your data, R will include them with a count of zero.

```r
feedback_ratings <- c(
  "Good", "Excellent", "Poor", "Fair",
  "Good", "Excellent", "Fair"
)

# Specifying extra levels
feedback_factors <- factor(feedback_ratings,
  levels = c("Poor", "Fair", "Good", "Very Good", "Excellent"),
  ordered = TRUE
)

# Summarise
summary(feedback_factors)
```

```
#>      Poor      Fair      Good Very Good Excellent
#>         1         2         2         0         2
```

Ensure that the levels you specify match the data or be aware that additional levels will appear with zero counts.

2.5.9.6 Removing Unused Factor Levels

When working with factors in R, you might encounter situations where, after subsetting your data, the factor levels still include categories that no longer exist in your dataset. These unused levels can lead to misleading analyses and cluttered visualizations. To ensure that your factor variables accurately reflect the data, it's important to remove these unused levels. This can be easily achieved using the `droplevels()` function.

Why Remove Unused Levels?

- Accurate Analysis: Statistical functions and models may misinterpret unused levels, affecting results.
- Clean Visualizations: Plots and charts can display empty categories, making them confusing.
- Data Integrity: Keeping only relevant levels ensures your data accurately represents the current state.

Example 1: Removing Unused Levels from a Factor

Suppose you have a factor vector representing customer feedback ratings:

```
# Original factor with all levels
feedback_ratings <- c(
  "Good", "Excellent", "Poor",
  "Fair", "Good", "Excellent", "Fair"
)

feedback_factors <- factor(feedback_ratings,
  levels = c("Poor", "Fair", "Good", "Excellent"),
  ordered = TRUE
)

# Display the original factor
feedback_factors
```

```
#> [1] Good      Excellent Poor      Fair      Good      Excellent Fair
#> Levels: Poor < Fair < Good < Excellent
```

Now, you decide to exclude the "Poor" ratings from your analysis:

```r
# Subset the data to exclude "Poor" ratings
subset_feedback <- feedback_factors[feedback_factors != "Poor"]

# Display the subsetted factor
subset_feedback
```

```
#> [1] Good      Excellent Fair      Good      Excellent Fair
#> Levels: Poor < Fair < Good < Excellent
```

> **Observation**
>
> Even after subsetting, the level "Poor" remains in the levels attribute, despite not being present in the data.

Using `droplevels()` to Remove Unused Levels

To clean up the factor and remove any levels that are no longer used, apply the `droplevels()` function:

```r
# Remove unused levels
subset_feedback <- droplevels(subset_feedback)

# Display the cleaned factor
subset_feedback
```

```
#> [1] Good      Excellent Fair      Good      Excellent Fair
#> Levels: Fair < Good < Excellent
```

Now, the factor levels accurately reflect only the categories present in the data.

```r
# Check the levels of the cleaned factor
levels(subset_feedback)
```

```
#> [1] "Fair"      "Good"      "Excellent"
```

> **Note**
>
> **Always Check Levels After Subsetting:** It's good practice to check the levels of your factor variables after any subsetting operation.

```r
# Check levels before and after dropping unused levels
levels(feedback_factors)
```

```
#> [1] "Poor"      "Fair"      "Good"      "Excellent"
```

```r
levels(subset_feedback)
```

```
#> [1] "Fair"      "Good"      "Excellent"
```

Apply `droplevels()` to Data Frames: If your data frame contains factor columns and you've performed row-wise subsetting, you can apply `droplevels()` to the entire data frame to clean all factor columns at once.

```r
# Assuming 'df' is your data frame
df_clean <- droplevels(df)
```

Use in Modeling and Visualization: Clean factor levels ensure that statistical models and plots accurately represent your data without misleading categories.

2.5.10 Reflection Question 2.1.2

- How does converting character vectors to factors benefit data analysis in R?
- When would you use a factor instead of a character vector in R?

See the Solution to Reflection Question 2.1.2

2.5.11 Practice Quiz 2.1

Question 1:

Which function is used to create a vector in R?

a) `vector()`

b) `c()`

c) `list()`

d) `data.frame()`

Question 2:

Given the vector:

`v <- c(2, 4, 6, 8, 10)`

What is the result of `v * 3`?

a) `c(6, 12, 18, 24, 30)`

b) `c(2, 4, 6, 8, 10, 3)`

c) `c(6, 12, 18, 24)`

d) An error occurs

Question 3:

In R, is the vector `c(TRUE, FALSE, TRUE)` considered a numeric vector?

a) True
b) False

Question 4:

What will be the output of the following code?

```
numbers <- c(1, 3, 5, 7, 9)
numbers[2:4]
```

a) `1, 3, 5`

b) `3, 5, 7`

c) `5, 7, 9`

d) `2, 4, 6`

Question 5:

Which of the following best describes a factor in R?

a) A numerical vector

b) A categorical variable with predefined levels

c) A two-dimensional data structure

d) A list of vectors

Question 6:

Which function is used to create sequences including those with either integer or non-integer steps?

a) `:`

b) `seq()`

c) `rep()`

d) `sample()`

Question 7:
What does the following code output?

```r
seq(10, 1, by = -3)
```

a) `10, 7, 4, 1`

b) `10, 7, 4`

c) `1, 4, 7, 10`

d) An error occurs

Question 8:

Suppose you want to create a vector that repeats the sequence `1, 2, 3` five times. Which code will achieve this?

a) `rep(c(1, 2, 3), each = 5)`

b) `rep(c(1, 2, 3), times = 5)`

c) `rep(1:3, times = 5)`

d) `rep(1:3, each = 5)`

Question 9:

Suppose you are drawing coins from a treasure chest. There are 100 coins in this chest: 20 gold, 30 silver, and 50 bronze. Use R to draw 5 random coins from the chest. Use `set.seed(50)` to ensure reproducibility.

What will be the output of the random draw?

a) `Silver, Bronze, Bronze, Bronze, Silver`
b) `Gold, Gold, Silver, Bronze, Bronze`

c) `Gold, Bronze, Bronze, Bronze, Silver`

d) `Silver, Bronze, Gold, Bronze, Bronze`

Question 10:

What will the following code produce?

```
c(1, 2, 3) + c(4, 5)
```

a) `5, 7, 8`

b) `5, 7, 7`

c) An error due to unequal vector lengths

d) `5, 7, 9`

See the Solution to Quiz 2.1

2.5.12 Exercise 2.1.2: Vector and Factor Manipulation

Given a vector of customer feedback ratings: `c("Good", "Excellent", "Poor", "Fair", "Good", "Excellent", "Fair")`

Your Tasks:

1. Create a vector named `feedback_ratings` containing the data.
2. Convert `feedback_ratings` into an ordered factor with levels: "Poor" < "Fair" < "Good" < "Excellent".
3. Summarize the feedback ratings.
4. Identify how many customers rated "Excellent".

See the Solution to Exercise 2.1.2

2.6 Experiment 2.2: Matrices

A matrix is a two-dimensional data structure consisting of a rectangular array of elements of the same data type, organized into rows and columns. Matrices are used extensively in linear algebra and statistical computations. Figure 2.3 illustrates a typical representation of a matrix.

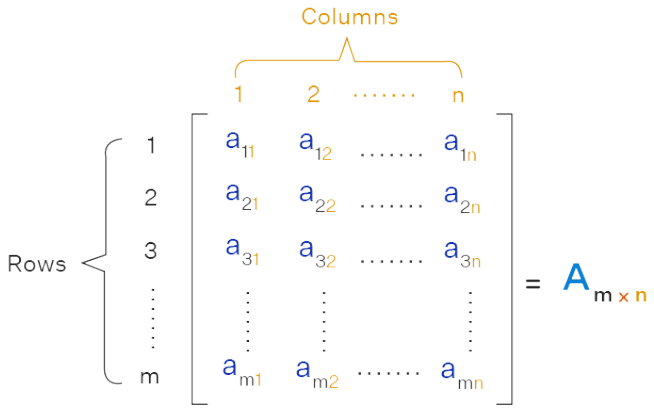


Figure 2.3: Matrix Representation in Linear Algebra

2.6.1 Creating Matrices

To create a matrix, use the `matrix()` function:

```
#|
matrix(data, nrow, ncol, byrow = FALSE)
```

where:

- `data`: The elements to be arranged in the matrix.
- `nrow`: The number of rows.
- `ncol`: The number of columns.
- `byrow`: If set to `FALSE` (the default), fills the matrix by columns; if set to `TRUE`, fills the matrix by rows.

Create the matrix `A`:

$$A = \begin{pmatrix} 1 & -2 & 5 \\ -3 & 9 & 4 \\ 5 & 0 & 6 \end{pmatrix}$$

```
A <- matrix(c(1, -2, 5, -3, 9, 4, 5, 0, 6), nrow = 3, ncol = 3, byrow = TRUE)

print(A)
```

```
#>      [,1] [,2] [,3]
#> [1,]    1   -2    5
#> [2,]   -3    9    4
#> [3,]    5    0    6
```

Create the matrix `B`:

$$B = \begin{pmatrix} 2 & -8 & 14 \\ 4 & 10 & 16 \\ 6 & 12 & 18 \end{pmatrix}$$

```
B <- matrix(c(2, 4, 6, -8, 10, 12, 14, 16, 18), nrow = 3, ncol = 3, byrow = FALSE)

print(B)
```

```
#>      [,1] [,2] [,3]
#> [1,]    2   -8   14
#> [2,]    4   10   16
#> [3,]    6   12   18
```

2.6.2 Matrices slicing

Accessing elements in a matrix is done by using `[row, column]`, between the square brackets, you indicate the position of the row and column in which the elements to access are. For example, to access the element in the first row and second column of matrix **A**, you type `A[1, 2]`. To access the element in the third row and second column of matrix **A**, you type `A[3, 2]`.

```
A[1, 2] # Element in first row, second column
```

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

```r
A[3, 2] # Element in third row, second column
```

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

2.6.3 Arithmetic Operation in Matrices

You can perform arithmetic operations on matrices. Consider the following matrices

$$A = \begin{pmatrix} 1 & -2 & 5 \\ -3 & 9 & 4 \\ 5 & 0 & 6 \end{pmatrix}$$

$$B = \begin{pmatrix} 2 & -8 & 14 \\ 4 & 10 & 16 \\ 6 & 12 & 18 \end{pmatrix}$$

```r
A <- matrix(c(1, -3, 5, -2, 9, 0, 5, 4, 6), nrow = 3, ncol = 3, byrow = FALSE)
```

```r
B <- matrix(c(2, 4, 6, 8, 10, 12, 14, 16, 18), nrow = 3, ncol = 3, byrow = FALSE)
```

Addition

```r
A + B
```

```
#>      [,1] [,2] [,3]
#> [1,]    3    6   19
#> [2,]    1   19   20
#> [3,]   11   12   24
```

Multiplication

Matrix multiplication is done using `%*%` operator:

```r
A %*% B
```

```
#>      [,1] [,2] [,3]
#> [1,]   24   48   72
#> [2,]   54  114  174
#> [3,]   46  112  178
```

2.6.4 Exercise 2.2.1: Matrix Transpose

Consider the following matrix A:

$$A = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$$

Your Task:

Find the transpose of matrix A, denoted as A^T.

> **Tip**
>
> Define matrix A, then use `t(A)` to find its transpose.

Here's a starting point for your code:

```
# Define matrix A

A <- matrix(c(...), nrow = ..., ncol = ..., byrow = TRUE)

A_transpose <- ...(A)
```

Replace the `...` with the correct values and complete the exercise!

See the Solution to Exercise 2.2.1

2.6.5 Exercise 2.2.2: Matrix Inverse Multiplication

Given the matrices A and B below:

$$A = \begin{pmatrix} 4 & 7 \\ 2 & 6 \end{pmatrix}$$

$$B = \begin{pmatrix} 3 & 5 \\ 1 & 2 \end{pmatrix}$$

Your Task:

Calculate $A^{-1} \times B$, where A^{-1} is the inverse of matrix A.

Hint:

- Use the `solve()` function in R to find the inverse of matrix A.
- Use the matrix multiplication operator `%*%` to multiply A^{-1} by B.

Here's a starting point for your code:

```
# Define matrices A and B
A <- matrix(c(...), nrow = ..., ncol = ..., byrow = TRUE)
B <- matrix(c(...), nrow = ..., ncol = ..., byrow = TRUE)

# Find the inverse of A
A_inverse <- solve(A)

# Multiply A_inverse by B
result <- A_inverse %*% B
```

Replace the `...` with the correct values for your matrices and complete the exercise!

See the Solution to Exercise 2.2.2

2.6.6 Real-World Data Scenario: Sales Data Matrix

Suppose we have sales data for three products over four regions.

```
# Sales data (units sold)

sales_data <- c(500, 600, 550, 450, 620, 580, 610, 490, 530, 610, 570, 480)

# Create a matrix

sales_matrix <- matrix(sales_data, nrow = 4, ncol = 3, byrow = TRUE)

colnames(sales_matrix) <- c("Product_A", "Product_B", "Product_C")

rownames(sales_matrix) <- c("Region_1", "Region_2", "Region_3", "Region_4")

sales_matrix
```

```
#>          Product_A Product_B Product_C
#> Region_1       500       600       550
#> Region_2       450       620       580
#> Region_3       610       490       530
#> Region_4       610       570       480
```

Addition and Subtraction:

```
# Assume a competitor's sales matrix
competitor_sales <- matrix(
  c(
    480, 590, 540, 430, 610, 570,
    600, 480, 520, 600, 560, 470
  ),
  nrow = 4, ncol = 3, byrow = TRUE
)

competitor_sales
```

```
#>      [,1] [,2] [,3]
#> [1,]  480  590  540
#> [2,]  430  610  570
#> [3,]  600  480  520
#> [4,]  600  560  470
```

```
# Calculate the difference in sales

sales_difference <- sales_matrix - competitor_sales
sales_difference
```

```
#>          Product_A Product_B Product_C
#> Region_1        20        10        10
#> Region_2        20        10        10
#> Region_3        10        10        10
#> Region_4        10        10        10
```

Matrix Multiplication:

```
# Price per product
prices <- c(20, 15, 25)

# Calculate total revenue per region

revenue_per_region <- sales_matrix %*% prices
revenue_per_region
```

```
#>            [,1]
#> Region_1 32750
#> Region_2 32800
#> Region_3 32800
#> Region_4 32750
```

Accessing Elements:

```
# Sales of Product_B in Region_2
sales_matrix["Region_2", "Product_B"] # Returns 580
```

```
#> [1] 620
```

```
# All sales for Product_C
sales_matrix[, "Product_C"]
```

```
#> Region_1 Region_2 Region_3 Region_4
#>      550      580      530      480
```

Common Pitfall

- **Dimension Mismatch:**
  - **Tip:** Ensure that the number of columns in the first matrix matches the number of rows in the second for multiplication.

2.6.7 Reflection Question 2.2.1

- In what scenarios would using a matrix be more advantageous than a data frame?

2.6.8 Practice Quiz 2.2

Question 1:

Which R function is used to find the transpose of a matrix?

a) `transpose()`

b) `t()`

c) `flip()`

d) `reverse()`

Question 2:

Given the matrix:

```
A <- matrix(1:6, nrow = 2, byrow = TRUE)
```

what is the value of `A[2, 3]`?

a) 3

b) 6

c) 5

d) 4

Question 3:

Matrix multiplication in R can be performed using the `*` operator.

a) True

b) False

Question 4:

What will be the result of adding two matrices of different dimensions in R?

a) R will perform element-wise addition up to the length of the shorter matrix.

b) An error will occur due to dimension mismatch.

c) R will recycle elements of the smaller matrix.

d) The matrices will be concatenated.

Question 5:

Which function can be used to calculate the sum of each column in a matrix `M`?

a) `rowSums(M)`

b) `colSums(M)`

c) `sum(M)`

d) `apply(M, 2, sum)`

Question 6:

Which function is used to create a matrix in R?

a) `matrix()`

b) `data.frame()`

c) `c()`

d) `list()`

See the Solution to Quiz 2.2

2.6.9 Exercise 2.2.3: Matrix Operations

Using the `sales_matrix` from the example in Section 2.6.6:

1. Calculate the total units sold per product.
2. Find the average units sold across all regions for `Product_A`.
3. Identify the region with the highest sales for `Product_C`.

See the Solution to Exercise 2.2.3

2.7 Experiment 2.3: Data frame

A data frame is a two-dimensional data structure. It resembles a table, where variables are represented as columns, and observations are rows—much like a spreadsheet or a SQL table. Data frames allow you to store columns of different data types (e.g., numeric, character, logical), making them ideal for real-world datasets.

2.7.1 Creating a Data Frame

To create a data frame, use the `data.frame()` function.

Example 1: Sales Transactions Data Frame

```r
# Sample sales transactions
transaction_id <- 1:5

product <- c("Product_A", "Product_B", "Product_C", "Product_A", "Product_B")

quantity <- c(2, 5, 1, 3, 4)

price <- c(19.99, 5.49, 12.89, 19.99, 5.49)

total_amount <- quantity * price

sales_transactions <- data.frame(
  transaction_id, product, quantity, price,
  total_amount
)

sales_transactions
```

```
#>   transaction_id   product quantity price total_amount
#> 1              1 Product_A        2 19.99        39.98
#> 2              2 Product_B        5  5.49        27.45
#> 3              3 Product_C        1 12.89        12.89
#> 4              4 Product_A        3 19.99        59.97
#> 5              5 Product_B        4  5.49        21.96
```

Notice the row numbers (`1 2 3 4 5`) displayed on the left of the console output—these are row labels. Each column in a data frame is internally represented as a vector.

Example 2: COVID 19 Data Frame

You can create a data frame for COVID-19 statistics with columns such as `states`, `confirmed_cases`, `recovered_cases`, and `death_cases`:

```r
states <- c("Lagos", "FCT", "Plateau", "Kaduna", "Rivers", "Oyo")

confirmed_cases <- c(58033, 19753, 9030, 8998, 7018, 6838)

recovered_cases <- c(56990, 19084, 8967, 8905, 6875, 6506)

death_cases <- c(439, 165, 57, 65, 101, 123)

covid_19 <- data.frame(states, confirmed_cases, recovered_cases, death_cases)
```

```
covid_19
```

```
#>    states confirmed_cases recovered_cases death_cases
#> 1   Lagos           58033           56990         439
#> 2     FCT           19753           19084         165
#> 3 Plateau            9030            8967          57
#> 4  Kaduna            8998            8905          65
#> 5  Rivers            7018            6875         101
#> 6     Oyo            6838            6506         123
```

2.7.2 Exploring Data Frames

R provides several functions to quickly explore the structure and content of data frame (`df`):

1. **`head(df)`**: Displays the first few rows.
2. **`tail(df)`**: Displays the last few rows.

Both functions also include a "header", showing the variable names in the data frame‘.

3. **`str(df)`**: Shows the structure of the data frame, including:
    - Number of observations (rows) and variables (columns).
    - Variable names and data types.
    - A preview of the data in each column.
4. **`names(df)`**: Lists the names of the columns.
5. **`nrow(df)`**: Returns the number of rows.
6. **`ncol(df)`**: Returns the number of columns.
7. **`dim(df)`**: Returns the dimensions (rows and columns) as a vector.
8. **`View(df)`**: Opens a spreadsheet-style viewer in RStudio.
9. **`summary(df)`**: Provides summary statistics for all columns.

Example: Medical Data Frame

Consider the following vectors:

```r
set.seed(2021) # Ensures reproducibility

gender <- sample(c("Male", "Female"), 120, replace = TRUE)

height <- floor(rnorm(n = 120, mean = 3, sd = 0.5))

weight <- ceiling(rnorm(n = 120, mean = 55, sd = 9))

bmi <- weight / height^2

medical_data <- data.frame(gender, height, weight, bmi)
```

Exploring the Medical Data Frame

First six observations:

```r
head(medical_data)
```

```
#>   gender height weight       bmi
#> 1   Male      3     47  5.222222
#> 2 Female      2     46 11.500000
#> 3 Female      3     58  6.444444
#> 4 Female      2     64 16.000000
#> 5   Male      3     62  6.888889
#> 6 Female      2     53 13.250000
```

Last six observations:

```r
tail(medical_data) # To get the last 6 observation
```

```
#>     gender height weight       bmi
#> 115   Male      2     54 13.500000
#> 116   Male      3     66  7.333333
#> 117   Male      3     57  6.333333
#> 118   Male      3     49  5.444444
#> 119   Male      2     51 12.750000
#> 120 Female      3     51  5.666667
```

Column names:

```r
names(medical_data)
```

```
#> [1] "gender" "height" "weight" "bmi"
```

You can also use:

```r
colnames(medical_data)
```

```
#> [1] "gender" "height" "weight" "bmi"
```

View the data in RStudio (interactive):

```r
View(medical_data)
```

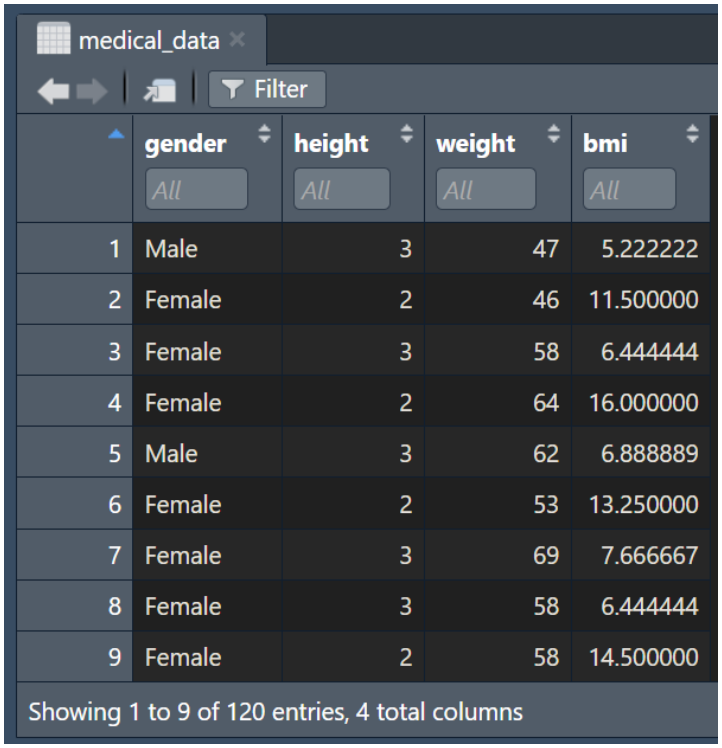


Figure 2.4: Data Frame Preview in RStudio: Gender, Height, Weight, and BMI

Descriptive statistics:

```r
summary(medical_data)
```

```
#>     gender              height          weight           bmi
#>  Length:120         Min.   :1.000   Min.   :37.00   Min.   : 2.375
#>  Class :character   1st Qu.:2.000   1st Qu.:50.00   1st Qu.: 6.333
#>  Mode  :character   Median :2.000   Median :56.00   Median :11.000
#>                     Mean   :2.433   Mean   :55.58   Mean   :12.384
#>                     3rd Qu.:3.000   3rd Qu.:62.00   3rd Qu.:14.312
#>                     Max.   :4.000   Max.   :78.00   Max.   :63.000
```

2.7.3 Built-in Datasets

R comes with a variety of built-in datasets that you can use for learning, testing, or exploring data analysis techniques. To view the available datasets, use the following command:

```r
data()
```

This will open a list of all the datasets available in the `datasets` package. The `datasets` package provides a variety of datasets on different topics, such as biology, finance, and historical events. Figure 2.5 is an example output showing some of these datasets:

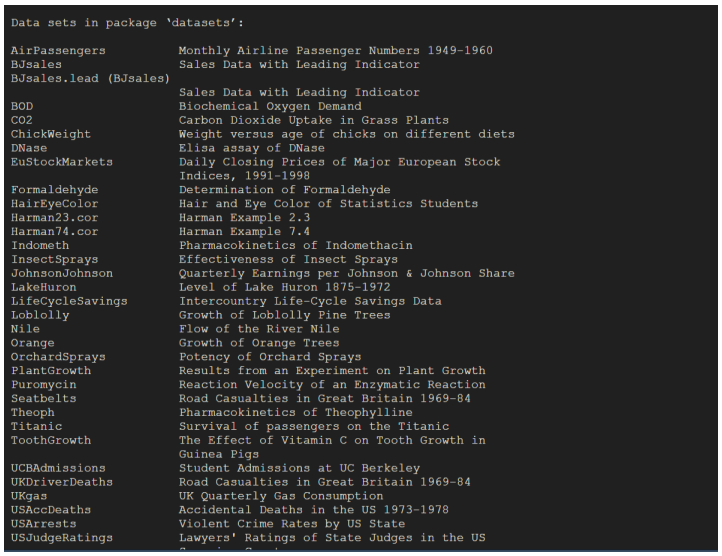


Figure 2.5: Sample Datasets Available in the R 'datasets' Package

2.7.3.1 Loading and Previewing Datasets

You can load and preview built-in datasets simply by calling their names. For example:

```
head(iris)
```

```
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#> 1          5.1         3.5          1.4         0.2  setosa
#> 2          4.9         3.0          1.4         0.2  setosa
#> 3          4.7         3.2          1.3         0.2  setosa
#> 4          4.6         3.1          1.5         0.2  setosa
#> 5          5.0         3.6          1.4         0.2  setosa
#> 6          5.4         3.9          1.7         0.4  setosa
```

This displays the first six rows of the `iris` dataset, which contains measurements of iris flowers.

```
head(airquality)
```

```
#>   Ozone Solar.R Wind Temp Month Day
#> 1    41     190  7.4   67     5   1
#> 2    36     118  8.0   72     5   2
#> 3    12     149 12.6   74     5   3
#> 4    18     313 11.5   62     5   4
#> 5    NA      NA 14.3   56     5   5
#> 6    28      NA 14.9   66     5   6
```

Here, the `airquality` dataset provides daily air quality measurements in New York from May to September 1973.

2.7.3.2 Getting Help on a Dataset

To learn more about a specific dataset, use the `?` operator followed by the dataset name. For example:

```
?airquality
```

This will bring up the documentation for the `airquality` dataset as shown in Figure 2.6, which includes details about the variables, their format, and the source of the data.

airquality {datasets} R Documentation

New York Air Quality Measurements

Description

Daily air quality measurements in New York, May to September 1973.

Usage

airquality

Format

A data frame with 153 observations on 6 variables.

[,1]	Ozone	numeric Ozone (ppb)
[,2]	Solar.R	numeric Solar R (lang)
[,3]	Wind	numeric Wind (mph)
[,4]	Temp	numeric Temperature (degrees F)
[,5]	Month	numeric Month (1–12)
[,6]	Day	numeric Day of month (1–31)

Details

Daily readings of the following air quality values for May 1, 1973 (a Tuesday) to September 30, 1973.

- Ozone: Mean ozone in parts per billion from 1300 to 1500 hours at Roosevelt Island
- Solar.R: Solar radiation in Langleys in the frequency band 4000–7700 Angstroms from 0800 to 1200 hours at Central Park
- Wind: Average wind speed in miles per hour at 0700 and 1000 hours at LaGuardia Airport
- Temp: Maximum daily temperature in degrees Fahrenheit at LaGuardia Airport.

Source

The data were obtained from the New York State Department of Conservation (ozone data) and the National Weather Service (meteorological data).

References

Chambers, J. M., Cleveland, W. S., Kleiner, B. and Tukey, P. A. (1983) *Graphical Methods for Data Analysis*. Belmont, CA: Wadsworth.

Figure 2.6: Airquality Dataset Documentation in R

2.7.4 Subsetting Data Frames

Every column in a data frame has a name. You can view the column names of a data frame, such as `iris`, by using the `names()` function:

```
names(iris)
```

```
#> [1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"
#> [5] "Species"
```

2.7.4.1 Using the $ Operator

To access a specific column by name, use the `$` operator in the format `df$colname`, where `df` is the name of the data frame and `colname` is the name of the column you want to retrieve. This operation returns the column as a vector. For example, to extract the `Sepal.Length` column from the `iris` data frame as a vector:

```
iris$Sepal.Length
```

```
#>  [1] 5.0 6.0 5.0 5.7 4.9 6.7 6.3 4.8 7.7 5.8 7.3 5.2 6.3 5.0 5.1 5.6 7.7 6.9
#> [19] 6.1 7.6
```

Similarly, to access the `Species` column:

```
iris$Species
```

```
#>  [1] setosa     versicolor virginica  setosa     virginica  setosa
#>  [7] versicolor setosa     versicolor setosa     virginica  versicolor
#> [13] setosa     virginica  setosa     versicolor virginica  setosa
#> [19] setosa     versicolor
#> Levels: setosa versicolor virginica
```

> **Note**
>
> For clarity and conciseness, we have shortened the output of both `iris$Sepal.Length` and `iris$Species` in the example outputs to save space. Both methods extract their respective columns as vectors.

Since the `$` operator produces a vector, you can directly apply vector-based functions, such as `mean()`, `sd()`, or `table()`, to perform calculations on the column data.

For example:

- To compute the mean of `Sepal.Length`:

```
mean(iris$Sepal.Length)
```

```
#> [1] 5.843333
```

- To find the frequency of each `Species`:

```
table(iris$Species)
```

```
#>
#>     setosa versicolor  virginica
#>         50         50         50
```

2.7.4.2 Using `[row, column]` Notation

You can also subset data frames using `[row, column]`, similar to matrices. In this syntax:

- The `row` argument specifies which rows to extract.
- The `column` argument specifies which columns to extract.

Here are some common examples and their interpretations, summarized in Table 2.2 below:

Table 2.2: Data Frame Slicing in R

Data Frame Slicing	Interpretation
`data[1, ]`	First row and all columns
`data[, 2]`	All rows and second column
`data[c(1, 3, 5), 2]`	Rows 1, 3, 5 and column 2 only
`data[1:3, c(1, 3)]`	First three rows and columns 1 and 3 only
`data` or `data[, ]`	All rows and all columns

Reflection Question 2.3.1

How does subsetting data frames help in data analysis?

Common Pitfalls

- **Undefined Columns Selected:**
  - **Tip:** Use `names(df)` to verify column names before subsetting.
- **Incorrect Data Types:**
  - **Tip:** Check data types with `str(df)` and convert if necessary.

2.7.5 Practice Quiz 2.3

Question 1:

Which function would you use to view the structure of a data frame, including its data types and a preview of its contents?

a) `head()`

b) `str()`

c) `summary()`

d) `names()`

Question 2:

How do you access the third row and second column of a data frame `df`?

a) `df[3, 2]`

b) `df[[3, 2]]`

c) `df$3$2`

d) `df(3, 2)`

Question 3:

In a data frame, all columns must contain the same type of data.

a) True

b) False

Question 4:

Which of the following commands would open a spreadsheet-style viewer of the data frame `df` in RStudio?

a) `View(df)`

b) `view(df)`

c) `inspect(df)`

d) `display(df)`

Question 5:

What does the `summary()` function provide when applied to a data frame?

a) Only the first few rows of the data frame.

b) Descriptive statistics for each column.

c) The structure of the data frame including data types.

d) A visual plot of the data.

Question 6:

In a data frame, all columns must be of the same data type.

a) True

b) False

See the Solution to Quiz 2.3

2.7.6 Exercise 2.3.1: Subsetting a Dataframe

Using the built-in `airquality` dataset, complete the following tasks:

- Examine the `airquality` dataset.
- Select the first three columns.
- Select rows 1 through 3 and columns 1 and 3.
- Select rows 1 through 5 and column 1.
- Select the first row.
- Select the first six rows.

See the Solution to Exercise 2.3.1

2.7.7 Exercise 2.3.2: Data Frame Manipulation

Using the `sales_transactions` data frame:

```
# Sample sales transactions
transaction_id <- 1:5

product <- c("Product_A", "Product_B", "Product_C", "Product_A", "Product_B")

quantity <- c(2, 5, 1, 3, 4)

price <- c(19.99, 5.49, 12.89, 19.99, 5.49)

total_amount <- quantity * price

sales_transactions <- data.frame(
```

```
  transaction_id, product, quantity,
  price, total_amount
)

sales_transactions
```

```
#>   transaction_id   product quantity price total_amount
#> 1              1 Product_A        2 19.99        39.98
#> 2              2 Product_B        5  5.49        27.45
#> 3              3 Product_C        1 12.89        12.89
#> 4              4 Product_A        3 19.99        59.97
#> 5              5 Product_B        4  5.49        21.96
```

1. Add a new column `discounted_price` that applies a 10% discount to `price`.
2. Filter transactions where the `total_amount` is greater than $50.
3. Calculate the average `total_amount` for `Product_B`.

See the Solution to Exercise 2.3.2

2.8 Experiment 2.4: Lists

A list is an R object that can contain elements of different types—numbers, strings, vectors, and even other lists. It's basically a container that can hold many different kinds of data structures.

You are already familiar with the notion of dimensions, such as the rows and columns in matrices and data frames. Unlike matrices or data frames, lists do not have the same concept of rows and columns. Instead, they are regarded as one-dimensional because they essentially consist of a sequence of items. You can think of it like a box full of different objects, all lined up one after the other.

2.8.1 Creating a List

To create a list, use the `list()` function:

```
# Customer Profile

customer_profile <- list(
  customer_id = 1001,
```

```
  name = "Johnny Drille",
  purchase_history = sales_transactions, # A data frame in Exercise 2.3.2
  loyalty_member = TRUE
)

customer_profile
```

```
#> $customer_id
#> [1] 1001
#>
#> $name
#> [1] "Johnny Drille"
#>
#> $purchase_history
#>   transaction_id   product quantity price total_amount
#> 1              1 Product_A        2 19.99        39.98
#> 2              2 Product_B        5  5.49        27.45
#> 3              3 Product_C        1 12.89        12.89
#> 4              4 Product_A        3 19.99        59.97
#> 5              5 Product_B        4  5.49        21.96
#>
#> $loyalty_member
#> [1] TRUE
```

Here, `customer_profile` consists of four components:

- `customer_id`: Numeric value.
- `name`: Character string.
- `purchase_history`: Data frame
- `loyalty_member`: Logical value.

To check how many elements are in a list, you can use the `length()` function. For instance:

```
length(customer_profile)
```

```
#> [1] 4
```

2.8.2 Accessing List Elements

To show the contents of a list you can simply type its name as any other object in R:

```
customer_profile
```

```
#> $customer_id
#> [1] 1001
#>
#> $name
#> [1] "Johnny Drille"
#>
#> $purchase_history
#>   transaction_id   product quantity price total_amount
#> 1              1 Product_A        2 19.99        39.98
#> 2              2 Product_B        5  5.49        27.45
#> 3              3 Product_C        1 12.89        12.89
#> 4              4 Product_A        3 19.99        59.97
#> 5              5 Product_B        4  5.49        21.96
#>
#> $loyalty_member
#> [1] TRUE
```

- **Using $ Operator:**
  The $ operator is used to access elements of a list by their names. This method is straightforward and commonly used when you know the exact name of the element you want to access.

  ```
  customer_profile$name
  ```

  ```
  #> [1] "Johnny Drille"
  ```

- **Using Double Square Brackets [[ ]]:**
  Double square brackets [[ ]] are used to extract elements from a list by their position (index) or name.

  ```
  customer_profile[[1]]
  ```

  ```
  #> [1] 1001
  ```

  ```
  customer_profile[["customer_id"]]
  ```

  ```
  #> [1] 1001
  ```

- **Using single square brackets `[]`:**
  Using single square brackets [ ] returns a list containing the element:

```
customer_profile[1]
```

```
#> $customer_id
#> [1] 1001
```

```
customer_profile["customer_id"]
```

```
#> $customer_id
#> [1] 1001
```

- **Accessing Data Frame within List:**
  When a list contains a data frame (or another list), you can access elements within that data frame by chaining the $ operator or using a combination of [[ ]] and $.

```
# Access the 'product' column in 'purchase_history'
customer_profile$purchase_history$product
```

```
#> [1] "Product_A" "Product_B" "Product_C" "Product_A" "Product_B"
```

```
# Access the amount of the second purchase
second_purchase_amount <- customer_profile$purchase_history$total_amount[2]

print(second_purchase_amount)
```

```
#> [1] 27.45
```

Reflection Question 2.4.1

- How can lists be used to organize complex data structures in R?

Common Pitfalls

- **Incorrect Indexing:**
  - **Tip:** Remember that `[ ]` returns a sublist, while `[[ ]]` returns the element itself.

2.8.3 Practice Quiz 2.4

Question 1:

Which function is used to create a list in R?

a) `c()`

b) `list()`

c) `data.frame()`

d) `matrix()`

Question 2:

Given the list:

```
L <- list(a = 1, b = "text", c = TRUE)
```

how would you access the element `"text"`?

a) `L[2]`

b) `L["b"]`

c) `L$b`

d) Both b) and c)

Question 3:

Using single square brackets `[]` to access elements in a list returns the element itself, not a sublist.

a) True

b) False

Question 4:

How can you add a new element named `d` with value `3.14` to the list `L`?

a) `L$d <- 3.14`

b) `L["d"] <- 3.14`

c) `L <- c(L, d = 3.14)`

d) All of the above

Question 5:

What will be the result of `length(L)` if

`L <- list(a = 1, b = "text", c = TRUE, d = 3.14)`?

a) 3

b) 4

c) 1

d) 0

See the Solution to Quiz 2.4

2.8.4 Exercise 2.4.1: Working with Lists

Create a list named `product_details` that contains:

- `Product ID`: 501
- `Name`: "Wireless Mouse"
- `Specifications`: A list containing `color`, `battery_life`, and `connectivity`
- `In Stock`: `TRUE`

Access each element individually and the nested list.

See the Solution to Exercise 2.4.1

2.9 Experiment 2.5: Arrays

An array is a data structure used for storing data in more than two dimensions. It's a bit like a matrix (which is two-dimensional), but arrays can extend to three dimensions or even more. In an array, all the elements must be of the same type (for example, all numeric or all character).

2.9.1 Creating Arrays

To create an array, use the `array()` function:

```
#|
array(data, dim = c(...), dimnames = NULL)
```

Where:

- **data**: The elements to be arranged in the array (all of the same type).
- **dim**: A vector specifying the dimensions of the array. For example, `c(2, 3, 4)` would create a three-dimensional array with dimensions $2 \times 3 \times 4$.
- **dimnames**: An optional argument where you can provide names for each dimension.

Example: Monthly Sales Data

Suppose we have sales data (in units sold) for two products across three regions over four months.

```
# Sales data for 2 products, across 3 regions, over 4 months

sales_vector <- c(
  50, 60, 55, 70, # Product A, Region 1, 4 months
  45, 52, 63, 65, # Product A, Region 2, 4 months
  80, 75, 70, 85, # Product A, Region 3, 4 months
  90, 95, 88, 92, # Product B, Region 1, 4 months
  55, 57, 59, 58, # Product B, Region 2, 4 months
  72, 78, 85, 80 # Product B, Region 3, 4 months
)
```

```
# Creating the 3D array (2 x 3 x 4)
sales_array <- array(
  data = sales_vector,
  dim = c(2, 3, 4),
  dimnames = list(
    Product = c("A", "B"),
    Region  = c("North", "East", "West"),
    Month   = c("Jan", "Feb", "Mar", "Apr")
  )
)

# Viewing the array
sales_array
```

```
#> , , Month = Jan
#>
#>        Region
#> Product North East West
#>       A    50   55   45
#>       B    60   70   52
#>
#> , , Month = Feb
#>
#>        Region
#> Product North East West
#>       A    63   80   70
#>       B    65   75   85
#>
#> , , Month = Mar
#>
#>        Region
#> Product North East West
#>       A    90   88   55
#>       B    95   92   57
#>
#> , , Month = Apr
#>
#>        Region
#> Product North East West
#>       A    59   72   85
#>       B    58   78   80
```

Note

1. **Data**
   The `sales_vector` contains all the numeric values we want to arrange in our array. Here, each element represents the number of units sold.

2. **Dimensions**
   We pass `dim = c(2, 3, 4)` to specify that our array will have 2 rows (for the products), 3 columns (for the regions), and 4 "layers" (for the months).

3. **Dimension Names**
   We optionally provide `dimnames` as a list of three vectors, each naming one dimension. This makes the final output more readable.

4. **Result**

The result is a 3D array in which you can access a single value or a "slice" of data by specifying the indices (for instance, `sales_array["A", "East", "Feb"]`).

Arrays are particularly useful when your data naturally extends beyond two dimensions—for instance, measuring different metrics across various categories and time periods. Just remember that all elements in an array must be of the same type (e.g., all numeric).

2.9.2 Reflection

Reflect on how **arrays** differ from other data structures in R. In what scenarios might you prefer using an array over a matrix, data frame, or list, and why?

Reflection Question 2.5.1

How does arrays differ from other data structures in R. In what scenarios might you prefer using an array over a matrix, data frame, or list, and why?

2.10 General Practice Quiz 2

Question 1

Which function is used to create a **vector** in R?

a) `vector()`

b) `c()`

c) `list()`

d) `data.frame()`

Question 2

Which function is used to create a **matrix** in R?

a) `array()`

b) `list()`

c) `matrix()`

d) `data.frame()`

Question 3

Which function is used to create an **array** in R?

a) `list()`

b) `matrix()`

c) `c()`

d) `array()`

Question 4

Which function is used to create a **list** in R?

a) `list()`

b) `c()`

c) `matrix()`

d) `data.frame()`

Question 5

A **matrix** in R must contain elements of:

a) Multiple data types (e.g., numeric and character mixed)

b) Only character type

c) Only logical type

d) The **same** type (all numeric, all logical, etc.)

Question 6

An **array** in R can be:

a) Only two-dimensional

b) Only one-dimensional

c) Two-dimensional or higher

d) Unlimited in one dimension only

Question 7

A **list** in R is considered:

a) Two-dimensional

b) One-dimensional

c) Multi-dimensional

d) A type of matrix

Question 8

Which of the following is **TRUE** about a **list**?

a) It can only contain numeric data

b) It stores data with rows and columns by default

c) It can store multiple data types in different elements

d) It must be strictly two-dimensional

Question 9

What is the most suitable structure for storing **heterogeneous data** (e.g., numbers, characters, and even another data frame) in a single R object?

a) Vector

b) Matrix

c) Array

d) List

Question 10

How do we typically check the "size" of a **list** in R?

a) `nrow()`

b) `length()`

c) `dim()`

d) `ncol()`

Question 11

Which function is used to create a **data frame** in R?

a) `data.frame()`

b) `array()`

c) `c()`

d) `list()`

Question 12

A **data frame** in R:

a) Must be strictly numeric

b) Can store different data types in each column

c) Is always one-dimensional

d) Is identical to a matrix

Question 13

If you want to assign **dimension names** to an **array**, you should use:

a) `rownames()` only

b) `colnames()` only

c) `dimnames()`

d) `names()`

Question 14

When creating a **matrix** using:

```r
matrix(1:6, nrow = 2, ncol = 3, byrow = TRUE)
```

How are the elements placed?

a) Filled by columns first

b) Filled by rows first

c) Randomly placed

d) Not possible to tell

Question 15

In an **array** with dimensions `c(2, 3, 4)`, how many elements are there in total?

a) 12

b) 18

c) 24

d) 36

See the Solution to General Quiz 2

2.11 Reflective Summary

Congratulations on completing **Lab 2**! You've expanded your R programming skills by mastering essential data structures:

- **Vectors**: The building blocks of data manipulation.
- **Matrices**: Fundamental for mathematical and statistical computations.
- **Data Frames**: Crucial for handling and analyzing real-world datasets.
- **Lists**: Versatile structures for complex data storage.

Key Takeaways:

- Understanding the appropriate data structure to use based on the data and task at hand is crucial.
- Effective data manipulation relies on mastering indexing and subsetting techniques.
- Combining different data structures allows for more complex data analyses.

What's Next?

In the next lab, you'll explore how to write your own functions in R. Functions are powerful tools that will help you streamline your code, automate tasks, and make your programs more efficient.

3 Writing Custom Function

3.1 Introduction

Welcome to **Lab 3**! In this lab, we'll explore how to write your own functions in R. Functions are essential in programming because they allow you to encapsulate code that performs specific tasks. This makes your programs more modular, readable, and easier to maintain. By designing custom functions, you can automate repetitive tasks, streamline your data analysis processes, and enhance the efficiency of your code.

3.2 Learning Objectives

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

- **Understand the Syntax of Functions in R**
  Learn how to define functions using the `function()` keyword, specify arguments, and structure the function body to perform desired operations.
- **Create Custom Functions**
  Write your own functions to perform specific data analysis tasks, allowing you to reuse code and avoid repetition.
- **Utilize Functions to Modularize and Streamline Code**
  Break down complex data analysis tasks into smaller, manageable functions to make your code more organized and maintainable.
- **Understand Variable Scope Within Functions**
  Grasp how variable scope works in R, distinguishing between local and global variables, and understand how this affects the behaviour of your functions.
- **Apply Best Practices in Function Design**
  Implement best practices such as choosing meaningful function names, including documentation with comments, handling inputs and outputs effectively, and incorporating error handling.
- **Demonstrate Understanding Through Practical Application**
  Use the functions you create in real data analysis scenarios to show how they can simplify tasks and improve code efficiency.

By completing this lab, you'll enhance your programming skills in R, enabling you to write code that is not only effective but also clean, reusable, and easy to understand. These skills are fundamental for any data analysis or data science work you'll undertake in the future.

3.3 Prerequisites

Before starting this lab, you should have:

- Completed **Lab 2** or have a basic understanding of R's data structures (vectors, matrices, data frames, and lists).
- Familiarity with basic programming concepts (variables, loops, conditionals).
- An interest in learning how to enhance your R programming skills through custom functions.

3.4 Experiment 3.1: Understanding Functions in R

A **function** is a block of code designed to perform a specific task. Functions usually take in some form of data structure—like a value, vector, or dataframe—as arguments, process it, and return a result. R has many built-in functions like `mean()`, `sum()`, and `plot()`, but creating your own functions allows you to tailor operations to your specific needs. For instance, imagine you often perform repetitive data transformations, cleaning, or statistical analysis. Writing custom functions allows you to automate these tasks, saving time and reducing potential errors.

Figure 3.1: Core Functions in R Programming

3.4.1 Types of Functions

Functions can be broadly categorized into two types:

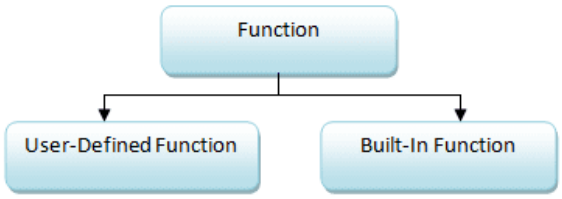


Figure 3.2: Types of Functions in R Programming

- **Built-in Functions**: These are predefined functions provided by R, e.g., `mean()`, `print()`.
- **User-defined Functions**: These are functions created by the user to perform specific tasks.

3.4.2 Why Write Your Own Function?

Creating your own functions has several advantages:

- **Code Reusability**: Functions promote code reuse and help you avoid repetition.
- **Improved Readability**: They make your code more readable and maintainable.
- **Modular Programming**: Functions allow for modular programming, where you can break down complex tasks into smaller, manageable pieces.

3.4.3 When Should You Write a Function?

Consider writing a function when:

- You find yourself repeating code.
- You need to perform a complex calculation multiple times.
- You want to make your code more organized and maintainable.

3.4.4 Creating Custom Function

A function in R has three main components:

- **Function Name**: A descriptive name that reflects the function's purpose.
- **Function Arguments**: Inputs that the function will process. This could be any type of object—such as a scalar, matrix, dataframe, vector, or logical.
- **Function Body**: The code that defines what the function does.

The general structure of a function is:

```
#|

function_name <- function(arg1, arg2, ...) {
  # Function body
  # ...
  return(result)
}
```

> Best Practice
>
> - **Naming Conventions**: Use `snake_case` and descriptive names.
> - **Comments**: Include comments to explain complex logic.
> - **Indentation**: Use consistent indentation for readability.
>
> If you create an object inside a function that you want to use outside of it, you need to return it using the `return()` function.

3.4.4.1 Calling a User-defined Function in R

You can call a user-defined function just like any built-in function, using its name. If the function accepts parameters or arguments, you pass them when calling the function.

3.4.5 Example 1: Squaring a Number

Let's start by creating a simple function to square a number. This example will introduce you to defining and using functions in R.

Defining the Function:

First, we'll define the function `square_it`. This function will take a single input, `x`, and return its square. Here's how you would write it:

```
# Function to square a number
square_it <- function(x) {
  result <- x^2
  return(result)
}
```

Now, whenever you call `square_it()` with a numerical input, it will output the square of that number.

Testing the Function

To verify that the function works as expected, try squaring a few numbers:

- Test with 12:

```
square_it(12)
```

```
#> [1] 144
```

- Test with vector, `product_prices`:

```
product_prices <- c(19.99, 5.49, 12.89, 99.99, 49.95)

square_it(product_prices)
```

```
#> [1]  399.6001   30.1401  166.1521 9998.0001 2495.0025
```

Reflection Question

- Why is it beneficial to write a function for squaring a number instead of writing `x^2` each time?

3.4.6 Example 2: Checking for Missing Values

Next, let's create a function that checks for missing values in a dataset and counts them.

Defining the Function

We'll define a function called `check_NA` as follows:

```r
# Function to check for missing values
check_NA <- function(data) {
  any_na <- anyNA(data)
  na_count <- sum(is.na(data))
  announcement <- paste("Any NA:", any_na, "| Total NA:", na_count)
  return(announcement)
}
```

Testing the Function

You can use this function to check for missing values in various datasets.

- For the `airquality` dataset:

```r
check_NA(airquality)
```

```
#> [1] "Any NA: TRUE | Total NA: 44"
```

- For the `iris` dataset:

```r
check_NA(iris)
```

```
#> [1] "Any NA: FALSE | Total NA: 0"
```

Running these commands will let you know if there are any missing values in the dataset and provide the total count of missing values.

3.5 Experiment 3.2: Advanced Function Examples

3.5.1 Example 3: Calculating the Statistical Mode

There is no built-in function in R to calculate the mode, so let's create one.

```r
statistical_mode <- function(x) {
  # Get unique values
  uniqx <- unique(x)

  # Count frequencies of each unique value
  freq <- tabulate(match(x, uniqx))
```

```r
  # Find the maximum frequency
  max_freq <- max(freq)

  # Find all values with the maximum frequency
  modes <- uniqx[freq == max_freq]

  # Handle cases
  if (length(modes) == length(uniqx)) {
    return("No mode: All values occur with equal frequency.")
  } else if (length(modes) > 1) {
    return(list("Multiple Modes" = modes, "Frequency" = max_freq))
  } else {
    return(list("Mode" = modes, "Frequency" = max_freq))
  }
}
```

Explanation

1. **Unique Values**:
   - `unique(x)` extracts distinct values from the input vector.
2. **Frequency Count**:
   - `tabulate(match(x, uniqx))` counts occurrences of each unique value.
3. **Maximum Frequency**:
   - `max(freq)` identifies the highest frequency.
4. **Multiple Modes**:
   - The function checks if more than one value has the maximum frequency.
5. **No Mode**:
   - If all unique values occur equally, the function returns a message stating there is no mode.

This **`statistical_mode()`** function will handle every possible scenario, including:

1. **Single Mode**: Returns the value with the highest frequency.
2. **Multiple Modes**: Returns all values with the highest frequency.
3. **No Mode**: Returns a message if all values appear with equal frequency (no distinct mode).

Testing the Function

- **Test 1: Single Mode**

```
calls <- c(
  0, 2, 6, 2, 2, 0, 0, 1, 1, 5, 3, 1, 0,
  2, 3, 1, 2, 1, 4, 4, 5, 0, 5, 1, 2, 2,
  2, 0, 4, 0, 6
)

statistical_mode(calls)
```

```
#> $Mode
#> [1] 2
#>
#> $Frequency
#> [1] 8
```

- **Test 2: Multiple Modes**

```
scores <- c(5, 5, 6, 6, 7, 8)

statistical_mode(scores)
```

```
#> $`Multiple Modes`
#> [1] 5 6
#>
#> $Frequency
#> [1] 2
```

- **Test 3: No Mode**

```
values <- c(1, 2, 3, 4, 5)

statistical_mode(values)
```

```
#> [1] "No mode: All values occur with equal frequency."
```

3.5.2 Example 4: Data Frame Operation Using `switch()`

Suppose we have employee data and want to perform various operations based on user input. The available operations are:

- **"summary"**: Get a summary of the data frame.

- **"add_column"**: Add a new column to the data frame.
- **"filter"**: Filter the data frame based on a specified condition.
- **"group_stats"**: Calculate group-wise statistics.

To follow along with this example, please refer to Chapter 1.8.5 for a detailed tutorial and comprehensive understanding of the `switch()` function.

Step 1: Creating a Sample Data Frame

```
#|
library(tidyverse)

# Sample employee data
staff_data <- data.frame(
  EmployeeID = 1:6,
  Name = c("Alice", "Ebunlomo", "Festus", "Othniel", "Bob", "Testimony"),
  Department = c("HR", "IT", "Finance", "Data Science", "Marketing", "Finance"),
  Salary = c(70000, 80000, 75000, 82000, 73000, 78000)
)

staff_data
```

```
#>   EmployeeID      Name   Department Salary
#> 1          1     Alice           HR  70000
#> 2          2  Ebunlomo           IT  80000
#> 3          3    Festus      Finance  75000
#> 4          4   Othniel Data Science  82000
#> 5          5       Bob    Marketing  73000
#> 6          6 Testimony      Finance  78000
```

Step 2: Defining the Function

```
# Define the function
data_frame_operation <- function(data, operation = "filter" # or any of "summary",
                                 #  "add_column", filter", "group_stats"
) {
  result <- switch(operation,

    # Case 1: Summary of the data frame
    summary = {
      print("Summary of Data Frame:")
```

```
    summary(data)
  },

  # Case 2: Add a new column 'Bonus' which is 10% of the Salary
  add_column = {
    data$Bonus <- data$Salary * 0.10
    print("Data Frame after adding 'Bonus' column:")
    data
  },

  # Case 3: Filter employees with Salary > 75,000
  filter = {
    filtered_data <- filter(data, Salary > 75000)
    print("Filtered Data Frame (Salary > 75,000):")
    filtered_data
  },

  # Case 4: Group-wise average salary
  group_stats = {
    group_summary <- data %>%
      group_by(Department) %>%
      summarize(Average_Salary = mean(Salary))
    print("Group-wise Average Salary:")
    group_summary
  },

  # Default case
  {
    print("Invalid operation. Please choose a valid option.")
    NULL
  }
)

# Return the result
return(result)
}
```

Explanation:

- **Function `data_frame_operation`:**
  - **Parameters:**

    - `data`: The data frame to operate on.
    - `operation`: A string specifying the operation to perform.
  - **Using `switch()`**:
    - Each case corresponds to a specific operation.
    - Cases that involve multiple expressions are wrapped in `{}`.
    - The last expression in the block is returned as the result of the case.
    - If no match is found, the final unnamed argument serves as the default case.
  - **Operations**:
    - **"summary"**: Provides a summary of the data frame.
    - **"add_column"**: Adds a new column `Bonus` (10% of Salary) to the data frame.
    - **"filter"**: Filters the data frame to include only employees with a salary greater than $75,000.
    - **"group_stats"**: Calculates the average salary for each department.
  - **Default Case**: Prints an error message and returns `NULL` if the operation is invalid.
  - **Return Value**: The result of the operation is returned by the function.

Step 3: Testing the Function

Let's test the function with different operations.

Example 1: Summary of the Data Frame

```
# Perform the 'summary' operation
data_frame_operation(staff_data, "summary")
```

```
#> [1] "Summary of Data Frame:"
```

```
#>    EmployeeID       Name            Department            Salary
#>  Min.   :1.00   Length:6           Length:6           Min.   :70000
#>  1st Qu.:2.25   Class :character   Class :character   1st Qu.:73500
#>  Median :3.50   Mode  :character   Mode  :character   Median :76500
#>  Mean   :3.50                                         Mean   :76333
#>  3rd Qu.:4.75                                         3rd Qu.:79500
#>  Max.   :6.00                                         Max.   :82000
```

Example 2: Add a New Column

```
# Perform the 'add_column' operation
data_frame_operation(staff_data, "add_column")
```

```
#> [1] "Data Frame after adding 'Bonus' column:"

#>   EmployeeID      Name   Department Salary Bonus
#> 1          1     Alice           HR  70000  7000
#> 2          2  Ebunlomo           IT  80000  8000
#> 3          3    Festus      Finance  75000  7500
#> 4          4   Othniel Data Science  82000  8200
#> 5          5       Bob    Marketing  73000  7300
#> 6          6 Testimony      Finance  78000  7800
```

Example 3: Filter the Data Frame

```
# Perform the 'filter' operation
data_frame_operation(staff_data, "filter")
```

```
#> [1] "Filtered Data Frame (Salary > 75,000):"

#>   EmployeeID      Name   Department Salary
#> 1          2  Ebunlomo           IT  80000
#> 2          4   Othniel Data Science  82000
#> 3          6 Testimony      Finance  78000
```

Example 4: Group-wise Statistics

```
# Perform the 'group_stats' operation
data_frame_operation(staff_data, "group_stats")
```

```
#> [1] "Group-wise Average Salary:"

#> # A tibble: 5 x 2
#>   Department   Average_Salary
#>   <chr>                <dbl>
#> 1 Data Science         82000
#> 2 Finance              76500
#> 3 HR                   70000
#> 4 IT                   80000
#> 5 Marketing            73000
```

Example 5: Invalid Operation

```
# Attempt an invalid operation
data_frame_operation(staff_data, "view")
```

```
#> [1] "Invalid operation. Please choose a valid option."
```

```
#> NULL
```

Caution

Common Pitfall:

- **Missing Libraries:** Forgetting to load required packages like `dplyr`.
- **Tip:** Include `library()` calls within the function or check if the package is installed.

3.5.3 Exercise 3.1.1: Temperature Conversion

Now, it's your turn to create a function.

Your Task: Create a function to convert Celsius (C) to Fahrenheit (F). You can use the formula:

$F = C \times 1.8 + 32$

Instructions:

1. **Define the Function**
   - Name the function `celsius_to_fahrenheit`.
   - It should take one argument, the temperature in Celsius.
2. **Implement the Formula**
   - Inside the function, apply the formula to convert Celsius to Fahrenheit.
3. **Return the Result**
   - The function should return the Fahrenheit temperature.

Test Your Function:

Use your function to convert the following Celsius temperatures to Fahrenheit:

- 100°C
- 75°C
- 120°C

For each temperature, call your function and verify that it returns the correct Fahrenheit value.

See the Solution to Exercise 3.1.1

3.5.4 Exercise 3.1.2: Pythagoras Theorem

Create a function to:

Your Task: Create a function called `pythagoras` to calculate the hypotenuse (`c`) of a right-angled triangle using Pythagoras' theorem:

$$c = \sqrt{a^2 + b^2}$$

where `a` and `b` are the lengths of the other two sides.

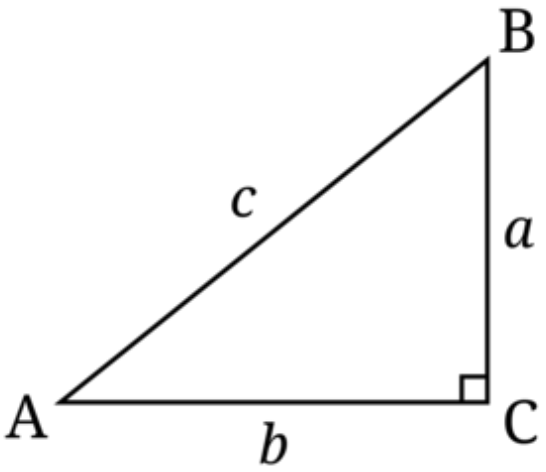


Figure 3.3: Geometric Representation: Right-Angled Triangle

Instructions:

1. **Define the Function**
    - Name the function `pythagoras`.

- It should take two arguments: `a` and `b`.

2. **Implement the Formula**
   - Inside the function, calculate the hypotenuse using the Pythagorean theorem.
3. **Return the Result**
   - The function should return the length of the hypotenuse.

Test Your Function:

Use your `pythagoras` function to calculate the hypotenuse for the following triangles:

- For $a = 4.1$ and $b = 2.6$
- For $a = 3$ and $b = 4$

Call your function with these values and verify that it returns the correct hypotenuse length.

See the Solution to Exercise 3.1.2

3.5.5 Exercise 3.1.3: Staff Data Manipulation Using `switch()`

Based on the example in Section 3.5.2, try modifying the code to include an additional operation:

- **"raise_salary"**: Increase the salary of all employees by 5%.

Instructions:

1. Add a new case to the `switch()` function for `"raise_salary"`.
2. In this case, increase the `Salary` column by 5% and return the updated data frame.
3. Test the code by setting `operation = "raise_salary"`.

Your Task:

```
# Modify the function to include 'raise_salary' operation
data_frame_operation <- function(..., operation) {
  result <- switch(operation,

    # Existing cases...

    # Case for 'raise_salary'
    raise_salary = {
      data$Salary <- data$Salary * ...
```

```
      print("Data Frame after 5% salary increase:")
      data
    },

    # Default case
    {
      print("Invalid operation. Please choose a valid option.")
      NULL
    }
  )

  # Return the result
  return(...)
}
```

Test the New Operation

```
# Perform the 'raise_salary' operation
data_frame_operation(staff_data, "---")
```

Replace the `...` with the correct values and complete the exercise!

See the Solution to Exercise 3.1.3

Best Practices in Function Design

Meaningful Function Names

- Use descriptive names that convey the function's purpose.
- Follow naming conventions (`snake_case`).

Handling Inputs and Outputs

- Validate input types and values.
- Provide clear and consistent return values.

Error Handling

- Use `stop()`, `warning()`, or `message()` to handle errors and warnings.
- Ensure that your function fails gracefully.

Example with Error Handling:

```
safe_divide <- function(a, b) {
  if (b == 0) {
    stop("Error: Division by zero is not allowed.")
  } else {
    return(a / b)
  }
}

# Testing the function
safe_divide(10, 2) # Returns 5
```

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

```
safe_divide(10, 0) # Error message
```

```
#> Error in safe_divide(10, 0): Error: Division by zero is not allowed.
```

Not validating inputs can lead to unexpected errors or incorrect results.

3.6 Experiment 3.3: Understanding Variable Scope

When writing a function, it's crucial to understand how variables behave inside and outside the function. This concept is known as **variable scope**. Variable scope determines where a variable is accessible in your code and how changes to variables within the function can affect variables outside of them.

3.6.1 Local vs. Global Variables

- **Local Variables**: These are variables that are defined within a function. They exist only during the execution of that function and are not accessible outside of it.
- **Global Variables**: These are variables that are defined outside of any function. They exist in the global environment and can be accessed by any part of your script, including inside functions (unless shadowed by a local variable of the same name).

3.6.2 How Variable Scope Works in R

In R, each function has its own environment. This means that variables created inside a function (local variables) do not interfere with variables outside the function (global variables), even if they have the same name.

Example 1: Local Variable

Let's look at an example to illustrate this:

```
variable_scope1 <- function() {
  local_var <- "I am a local variable!"
  print(local_var)
}
```

```
variable_scope1() # Prints "I am local"
```

```
#> [1] "I am a local variable!"
```

```
print(local_var) # Error: object 'local_var' not found
```

```
#> Error: object 'local_var' not found
```

In this example, `local_var` is a local variable within the `variable_scope1()` function. Trying to access `local_var` outside the function results in an error because it doesn't exist in the global environment.

Example 2: Global Variable Access

Functions in R can access global variables unless there is a local variable with the same name:

```
global_var <- "I am global"

variable_scope2 <- function() {
  print(global_var)
}

variable_scope2() # Prints "I am global"
```

```
#> [1] "I am global"
```

Here, the function `variable_scope2()` accesses the global variable `global_var` because there is no local variable named `global_var` inside the function.

3.6.3 Variable Shadowing

When a local variable has the same name as a global variable, the local variable takes precedence within the function.

```
var <- "I am global"

variable_scope3 <- function() {
  var <- "I am local"
  print(var)
}

variable_scope3() # Prints "I am local"
```

```
#> [1] "I am local"
```

```
print(var) # Prints "I am global"
```

```
#> [1] "I am global"
```

In this case, the `var` variable inside `variable_scope3()` is local and doesn't affect the global `var` variable.

> Tip
>
> **Reflection Question:**
>
> - Why is it important to understand variable scope when writing functions?

> Common Errors and Debugging Tips
>
> - Syntax Errors: Check for missing commas, parentheses, or braces.
> - Undefined Variables: Ensure all variables used in the function are defined.
> - Incorrect Return Values: Make sure the function returns what you expect.

- Variable Scope Issues: Be mindful of local vs. global variables.

Debugging Tips:

- Use `print()` statements to check intermediate results.
- Use the `debug()` function to step through your function.
- Validate inputs at the start of your function.

3.6.4 Practice Quiz 3.1

Question 1:

What is the correct way to define a function in R?

a) `function_name <- function { ... }`

b) `function_name <- function(...) { ... }`

c) `function_name <- function[ ... ] { ... }`

d) `function_name <- function(...) [ ... ]`

Question 2:

A variable defined inside a function is accessible outside the function.

a) True

b) False

Question 3:

Which of the following is NOT a benefit of writing functions?

a) Code Reusability

b) Improved Readability

c) Increased Code Complexity

d) Modular Programming

See the Solution to Quiz 3.1

3.7 Summary

In this lab, you have developed essential skills in creating custom functions:

- **Understanding** the syntax of functions in R, including how to define functions using the `function()` keyword, specify arguments, and structure the function body.
- **Creating and utilizing** your own custom functions to perform specific data analysis tasks, promoting code reuse and avoiding repetition.
- **Applying** functions to modularize and streamline your code, breaking down complex tasks into smaller, manageable pieces for better organization and maintainability.
- **Grasping** variable scope within functions, distinguishing between local and global variables, and understanding how this affects the behavior of your functions.
- **Implementing** best practices in function design, such as choosing meaningful function names, including documentation with comments, handling inputs and outputs effectively, and incorporating error handling.

These skills are fundamental for efficient programming in R and will greatly enhance your data analysis capabilities. They form a strong foundation for more advanced topics you will encounter as you continue learning. Congratulations on advancing your programming expertise!

> What's Next?
>
> In the next lab, we'll delve into managing packages, creating reproducible workflows using RStudio project, and reading data from a file.

4 Managing Packages and Workflows

4.1 Introduction

In **Lab 4**, we will explore essential practices that will enhance your efficiency and effectiveness as an R programmer. You will discover how to extend R's capabilities by installing and loading packages, how to ensure that your analyses are reproducible by using RStudio Projects, and how to proficiently import and export datasets in various formats. These skills are essential for any data analyst or data scientist, as they enable you to work with a wide range of data sources, maintain the integrity of your analyses, and share your work with others in a consistent and reliable manner.

4.2 Learning Objectives

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

- **Install and Load Packages in R**
  Learn how to find, install, and load packages from CRAN and other repositories, thereby extending the functionality of R for your data analysis tasks.
- **Ensure Reproducibility with R and RStudio Projects**
  Set up and manage RStudio Projects to organise your work, understand the concept of the working directory, and adopt best practices to make your data analyses reproducible and shareable.
- **Import and Export Datasets in Various Formats**
  Import data into R from different file types, such as CSV, Excel, and SPSS, using appropriate packages and functions. Export your data frames and analysis results to various formats for sharing or reporting.

By completing this lab, you will enhance your ability to manage and analyse data in R more efficiently. You will also ensure that your work is organised, reproducible, and ready to share with others. These foundational skills will support your development as a proficient R programmer and data analyst.

4.3 Prerequisites

Before starting this lab, you should have:

- Completed **Lab 3** or have a basic understanding of writing custom functions in R.
- Familiarity with R's data structures and basic data manipulation.
- An interest in organizing, documenting, and sharing analytical workflows efficiently.

4.4 Understanding Packages and Libraries in R

In R, a package is a collection of functions, data, and code that extends the basic functionality of R. Think of it as a specialised toolkit for particular tasks or topics. For example, packages like `tidyr` and `janitor` facilitate data wrangling, while others focus on graphics, modelling, or data import and export.

A library is a location on your computer's file system where installed packages are stored. When you install a package, it is saved in a library so that you can easily access it in future R sessions.

4.5 Compiling R Packages from Source

You may occasionally need additional tools to compile R packages from source, depending on your operating system:

- **Windows**

  Windows does not support code compilation natively. Therefore, you need Rtools, which provides the necessary software, including compilers and libraries, to build R packages from source. You can download Rtools from CRAN: https://cran.rstudio.com/bin/windows/Rtools/. After installing the appropriate version of Rtools, R will automatically detect it.

  > **Note**
  >
  > To check your R version, run the following code in your console:
  >
  > ```
  > R.version
  > ```
  >
  > To verify that Rtools is correctly installed, you can run the following code in your console:
  >
  > ```
  > Sys.which("make")
  > ```

- **Mac OS**

  On macOS, you need the Xcode Command Line Tools, which provide similar capabilities to Rtools on Windows. You can install Xcode from the Mac App Store:

  http://itunes.apple.com/us/app/xcode/id497799835?mt=12 or install the Command Line Tools directly by running:

  ```
  xcode-select --install
  ```

- **Linux**

  Most Linux distributions already come with the necessary tools for compiling packages. If additional developer tools are needed, you can install them via your package manager, usually by installing packages like `build-essential` or similar for your Linux distribution.

  > **Note**
  >
  > On Debian/Ubuntu, you can install the essential software for R package development and LaTeX (if needed for documentation) with:
  >
  > ```
  > sudo apt-get install r-base-dev texlive-full
  > ```
  >
  > To ensure all dependencies for building R itself from source are met, you can run:
  >
  > ```
  > sudo apt-get build-dep r-base-core
  > ```

4.6 Experiment 4.1: Installing and Loading Packages

As you progress in R, you will frequently need functions that are not included in the base R installation. These are provided by packages, which you can easily install and load into your R environment.

4.6.1 Installing Packages from CRAN

The Comprehensive R Archive Network (CRAN) hosts thousands of packages. To install a package from CRAN, use:

```
install.packages("package_name")
```

> **Note**
>
> Replace `package_name` with the name of the package you want to install.

For example, to install the `tidyverse` package, use:

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

Similarly, to install the `janitor` package, use:

```
install.packages("janitor")
```

> **Warning**
>
> Remember to enclose the package name in quotes—either double (`"package_name"`) or single (`'package_name'`).

4.6.2 Installing Packages from External Repositories

Some packages may not be available on CRAN but can be installed from GitHub or GitLab. First, install a helper package such as `devtools` or `remotes`:

```
install.packages("devtools")
# or
install.packages("remotes")
```

Then, to install a package from GitHub, for example `openintro` package, use:

```
devtools::install_github("OpenIntroStat/openintro")
# or
remotes::install_github("OpenIntroStat/openintro")
```

You can also install development versions of packages using these helper packages. For instance:

```
#|
remotes::install_github("datalorax/equatiomatic")
```

4.6.3 Loading Packages

Once a package has been installed, you need to load it into your R session to use its functions. You can do this by calling the `library()` function, as demonstrated in the code cell below:

```
library(package_name)
```

Here, `package_name` refers to the specific package you want to load into the R environment. For example, to load the `tidyverse` package:

```
library(tidyverse)
```

```
#> -- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
#> v dplyr     1.1.4     v readr     2.1.5
#> v forcats   1.0.0     v stringr   1.5.1
#> v ggplot2   3.5.1     v tibble    3.2.1
#> v lubridate 1.9.3     v tidyr     1.3.1
#> v purrr     1.0.2
#> -- Conflicts ------------------------------------------ tidyverse_conflicts() --
#> x dplyr::filter() masks stats::filter()
#> x dplyr::lag()    masks stats::lag()
#> i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to be
```

This command loads core `tidyverse` packages essential for most data analysis project[1]. Other installed packages can also be loaded in such manner:

```
library(janitor)

library(bulkreadr)
```

If you run this code and get the error message there is no package called `"bulkreadr"`, you'll need to first install it, then run `library()` once again.

```
install.packages("bulkreadr")

library(bulkreadr)
```

[1]It is common for a package to print out messages when you load it. These messages often include information about the package version, attached packages, or important notes from the authors. For example, when you load the tidyverse package. If you prefer to suppress these messages, you can use the `suppressMessages()` function: `suppressMessages(library(tidyverse))`

Tip

You only need to install a package once, but you must load it each time you start a new R session.

Figure 4.1: Installing vs. Loading Packages in R

4.6.4 Using Functions from a Package

When working with R packages, there are two primary ways to use a function from a package:

1. Load the Package and Call the Function Directly

You can load the package into your R session using the `library()` function, and then call the desired function by its name. For example:

```
#|
# Load the janitor package
library(janitor)

# Use the clean_names() function
clean_names(iris)
```

```
#> # A tibble: 150 x 5
#>    sepal_length sepal_width petal_length petal_width species
#>           <dbl>       <dbl>        <dbl>       <dbl> <fct>
#>  1          5.1         3.5          1.4         0.2 setosa
#>  2          4.9         3            1.4         0.2 setosa
#>  3          4.7         3.2          1.3         0.2 setosa
#>  4          4.6         3.1          1.5         0.2 setosa
#>  5          5           3.6          1.4         0.2 setosa
#>  6          5.4         3.9          1.7         0.4 setosa
#>  7          4.6         3.4          1.4         0.3 setosa
```

```
#>  8          5           3.4          1.5         0.2 setosa
#>  9          4.4         2.9          1.4         0.2 setosa
#> 10          4.9         3.1          1.5         0.1 setosa
#> # i 140 more rows
```

In this approach, you only need to load the package once in your session, and then you can use its functions without specifying the package name.

2. Use the `::` Operator to Call the Function Without Loading the Package

The `::` operator is used to call a function from a specific package without loading the entire package into your R session. This is particularly useful when:

- **Avoiding Namespace Conflicts**: If multiple packages have functions with the same name, :: ensures you use the correct one.
- **Improving Code Clarity**: It makes your code more readable by clearly indicating which package a function comes from.
- **Reducing Memory Usage**: Only the specific function is accessed, not the entire package.

The syntax is:

```
packageName::functionName(arguments)
```

where:

- `packageName`: The name of the package where the function resides.
- `functionName`: The specific function you want to use from the package.

Example

```
# Using the double colon to access clean_names() from janitor package
janitor::clean_names(iris)
```

```
#> # A tibble: 150 x 5
#>    sepal_length sepal_width petal_length petal_width species
#>           <dbl>       <dbl>        <dbl>       <dbl> <fct>
#>  1          5.1         3.5          1.4         0.2 setosa
#>  2          4.9         3            1.4         0.2 setosa
#>  3          4.7         3.2          1.3         0.2 setosa
#>  4          4.6         3.1          1.5         0.2 setosa
```

```
#>  5          5          3.6          1.4          0.2 setosa
#>  6          5.4        3.9          1.7          0.4 setosa
#>  7          4.6        3.4          1.4          0.3 setosa
#>  8          5          3.4          1.5          0.2 setosa
#>  9          4.4        2.9          1.4          0.2 setosa
#> 10          4.9        3.1          1.5          0.1 setosa
#> # i 140 more rows
```

The `clean_names()` function, in this case, returns the `iris` data frame with column names formatted in a consistent and clean style.

> **Tip**
>
> Ensure that the `janitor` package is installed before using its functions. If it's not installed, you can install it using install.packages("janitor").

As shown in Table 4.1, using `library()` attaches the entire package, while the `::` operator allows you to call specific functions without loading the entire package. This difference can significantly impact memory usage and clarity in your scripts, especially when you're only using a few functions from a large package.

Table 4.1: Comparing `library()` and `::` Operator

Aspect	`library()`	`::` Operator
Loading Behavior	Attaches the entire package to the session	Accesses specific functions without loading the package
Namespace Conflicts	Potential for conflicts if multiple packages have functions with the same name	Avoids conflicts by specifying the package
Memory Usage	Loads all exported objects, increasing memory usage	Minimal memory usage as only specific functions are accessed
Code Verbosity	Less verbose; functions can be called directly	More verbose; requires prefixing with package name
Use Cases	When using multiple functions from a package extensively	When using only a few functions or to avoid conflicts

4.6.5 Practice Quiz 4.1

Question 1:

Imagine that you want to install the shiny package from CRAN. Which command should you use?

a) `install.packages("shiny")`
b) `library("shiny")`

c) `install.packages(shiny)`

d) `require("shiny")`

Question 2:

What must you do after installing a package before you can use it in your current session?

a) Restart R

b) Run `install.packages()` again

c) Load it with `library()`

d) Convert the package into a dataset

Question 3:

If you want to install a package that is not on CRAN (e.g., from GitHub), which additional package would be helpful?

a) `installer`

b) `rio`

c) `devtools`

d) `github_install`

Question 4:

Which function would you use to update all outdated packages in your R environment?

a) `update.packages()`

b) `install.packages()`

c) `library()`

d) `require()`

Question 5:

Which function can be used to check the version of an installed package?

a) `version()`

b) `packageVersion()`

c) `libraryVersion()`

d) `install.packages()`

See the Solution to Quiz 4.1

4.7 Experiment 4.2: Ensuring Reproducibility with R and RStudio Projects

Reproducibility is vital in data science. It allows analyses to be revisited, results to be verified, and insights to be shared seamlessly, whether with collaborators or for future reference. With R and RStudio Projects, you can create a self-contained workspace that keeps your work organised and reproducible.

4.7.1 Working Directory and Paths

Your working directory is where R looks for files to read and where it saves outputs. You can find your current working directory in two ways:

1. **In the Console**: At the top of the RStudio console, the current working directory is displayed.

2. **Using Code**: Run the following command:

```
getwd()
```

```
[1] "C:/Users/Ezekiel Adebayo/Desktop/stock-market"
```

If you're not using an RStudio project, you'll need to set the working directory manually each time. For instance:

```
setwd("/path/to/your/data_analysis")
```

> **Tip**
>
> You can use the keyboard shortcut `Ctrl + Shift + H` in RStudio to quickly choose your working directory.

- **Absolute Paths**: Start from the root of your file system (e.g., `C:/Users/YourName/Documents/data.csv`). These paths are specific to your computer and should be avoided in shared scripts.
- **Relative Paths**: Refer to files relative to your working directory (e.g., `data/data.csv`). Using relative paths ensures portability and makes your scripts easier to share and reuse.

4.7.2 RStudio Projects

RStudio Projects provide a centralised environment for all your analyses: data files, scripts, figures, outputs, and documentation. This setup ensures your work remains organised, consistent, and reproducible.

Benefits of RStudio Projects

1. **Organisation**
   All project files are stored in one place, making it easy to navigate and manage.
2. **Relative Paths**
   RStudio automatically sets the working directory to the project folder, allowing you to use relative paths (e.g., `data/my_data.csv`). This eliminates the need for hardcoding absolute paths, ensuring your code works on any system.
3. **Consistency Across Systems**
   When you open your RStudio project on another computer, it recognises the same folder structure, making your work portable and adaptable.
4. **Reproducibility**
   With all components centralised and paths consistent, RStudio Projects enable others to replicate your workflow effortlessly. Everything needed to reproduce your analysis is neatly packaged and ready to go.

4.7.3 How RStudio Projects Organize Your Work

When you set up an RStudio Project, it acts as the root folder for your analysis. A recommended structure is shown in Figure 4.2 below:

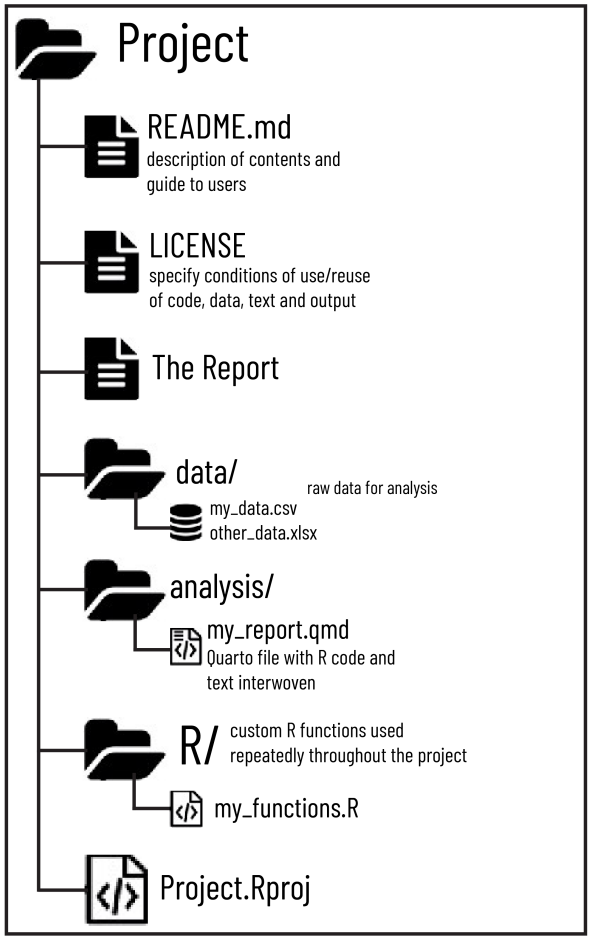


Figure 4.2: Practical Framework for Reproducible Data Analysis with RStudio Projects.

- **Top-Level Files**
  Files like `README.md` provide an overview of the project, while a `LICENSE` file outlines terms of use for the code and data. These files are essential for collaboration and transparency.
- **Data Folder (`data/`)**
  This folder contains raw data files in formats like `.csv` or `.xlsx`. Raw data is stored here in its original state to preserve reproducibility.
- **Analysis Folder (`analysis/`)**
  Scripts or dynamic documents, such as Quarto files (`.qmd`) or R Markdown files (`.Rmd`), go here. These documents combine R code, narrative text, and visualizations, serving as the heart of your analysis.
- **Custom Functions Folder (`R/`)**

Reusable R scripts containing custom functions are saved here. These can be sourced into analysis scripts to keep your workflow modular and efficient.

This structure, combined with relative paths, ensures a clean, logical, and reproducible workflow.

4.7.4 Setting Up Your RStudio Project

Let's create a new RStudio project. You can do this by following these simple steps:

4.7.4.1 Step 1: Create a New Project

1. **Go to:** `File → New Project` in RStudio

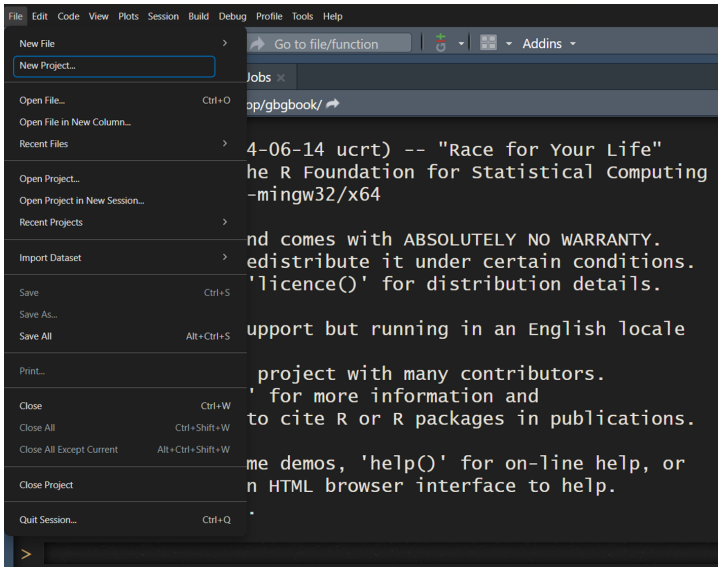


Figure 4.3: Creating a New Project in RStudio

2. **Choose:** `Existing Directory`

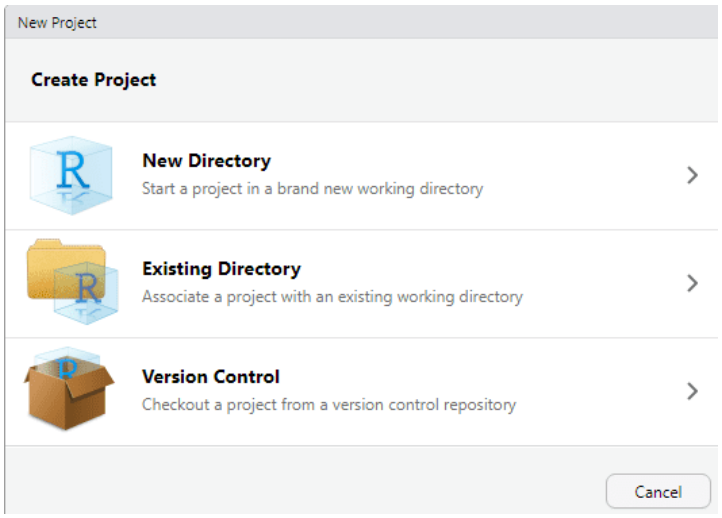


Figure 4.4: Creating a New Project in RStudio

3. **Select** the folder you want to use as your project's working directory and RStudio will create a project file (`.Rproj`).
4. **Click:** `Create Project`

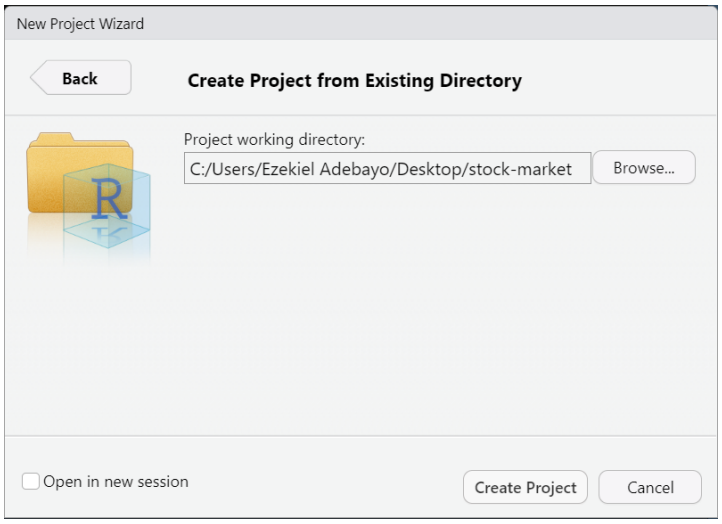


Figure 4.5: Creating a New R Project from an Existing Directory

Once you click "Create Project", you're all set! You'll be inside your new RStudio project.

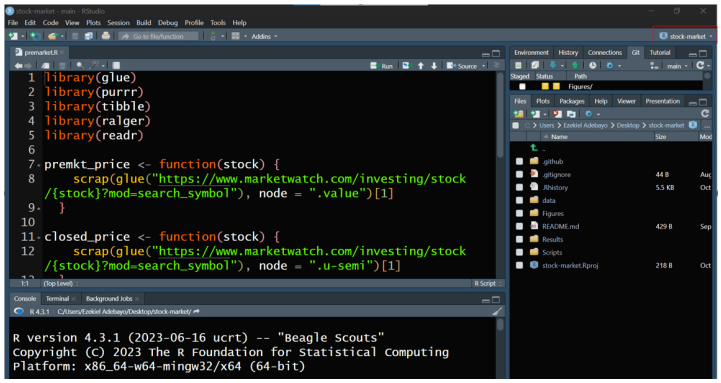


Figure 4.6: RStudio Project: Stock Market Price Scraper Using R

4.7.4.2 Step 2: Arrange Your Files

Organise your project files into the following structure:

- **`data/`**: Store raw data files here in formats such as `.csv` or `.xlsx`.
- **`analysis/`**: Place your analysis scripts or reports in this folder. For example, you might use a Quarto file (`my_report.qmd`) to combine R code, narrative text, and visualisations.
- **`R/`**: Save reusable R functions in this folder, and source them into your scripts as needed.

4.7.4.3 Step 3: Use Relative Paths

Always refer to files using paths relative to your project folder. For example:

```
data <- read.csv("data/my_data.csv")
```

4.7.4.4 Step 4: Add Documentation

Include a `README.md` file to document the purpose, structure, and usage of your project. Add a `LICENSE` file to define terms of use.

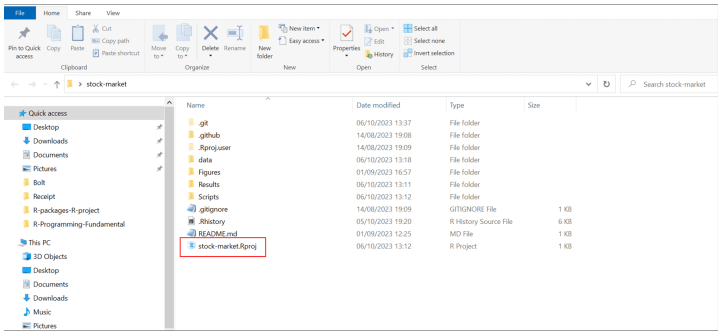


Figure 4.7: Organization of an R Project Directory

From now on, whenever you open this project (by clicking the `.Rproj` file), RStudio will automatically set your working directory, allowing you to use relative paths easily as shown in Figure 4.7

4.7.5 Practice Quiz 4.2

Question 1:

What is a key advantage of using RStudio Projects?

a) They automatically install packages.

b) They allow you to use absolute paths easily.

c) They set the working directory to the project folder, enabling relative paths.

d) They prevent package updates.

Question 2:

Which file extension identifies an RStudio Project file?

a) `.Rdata`

b) `.Rproj`

c) `.Rmd`

d) `.Rscript`

Question 3:

Why are relative paths preferable in a collaborative environment?

a) They are shorter and easier to type.

b) They change automatically when you move files.

c) They ensure that the code works regardless of the user's file system structure.

d) They are required for Git version control.

See the Solution to Quiz 4.2

4.8 Experiment 4.3: Importing and Exporting Data in R

Data import and export are vital steps in data science. With R, you can load data from spreadsheets, databases, and many other formats, and subsequently save your processed results. Figure 4.8 below illustrates some popular R packages for data import:

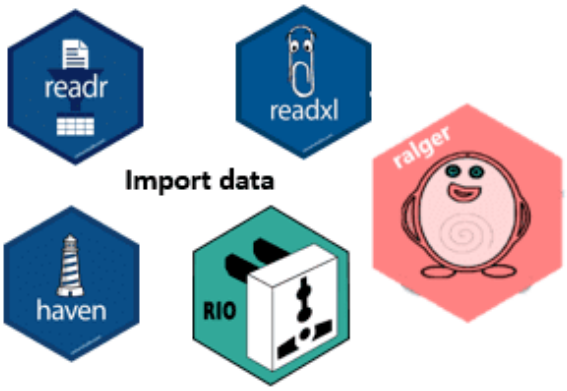


Figure 4.8: Data Import Packages in R

> **Note**
>
> Packages like `readr`, `readxl`, and `haven` are part of the tidyverse, and therefore are pre-installed when you install the tidyverse. You do not need to install these packages individually. For a complete list of tidyverse packages, see the following code:

```
tidyverse::tidyverse_packages()
```

```
#>  [1] "broom"         "conflicted"    "cli"           "dbplyr"
#>  [5] "dplyr"         "dtplyr"        "forcats"       "ggplot2"
#>  [9] "googledrive"   "googlesheets4" "haven"         "hms"
#> [13] "httr"          "jsonlite"      "lubridate"     "magrittr"
#> [17] "modelr"        "pillar"        "purrr"         "ragg"
#> [21] "readr"         "readxl"        "reprex"        "rlang"
#> [25] "rstudioapi"    "rvest"         "stringr"       "tibble"
#> [29] "tidyr"         "xml2"          "tidyverse"
```

You don't need to install any of these packages individually since they're all included with the tidyverse installation.

R offers some excellent packages that simplify the processes of importing and exporting data. In the sections below, we will explore a few commonly used packages and functions that are essential when working with data in R.

4.8.1 Flat Files

Flat files, such as CSV files, are among the most common formats for data storage and exchange. The `readr` package (a core component of the tidyverse) provides functions specifically designed to handle flat files. The two primary functions are:

- **`read_csv()`**: This function imports data from a CSV file into R as a data frame, akin to loading data directly from a spreadsheet.
- **`write_csv()`**: Once your data analysis is complete, this function exports your data frame to a CSV file. It is particularly useful for sharing your results or maintaining a backup.

Example 1: Reading CSV Data From a File

Suppose you have a file named `cleveland-heart-disease-database.csv` located in the `r-data` folder. You can import this flat file into R as follows:

```
library(tidyverse)

heart_disease_data <- read_csv("r-data/cleveland-heart-disease-database.csv")

heart_disease_data
```

```
#> # A tibble: 303 x 14
#>      age sex    `chest pain type` resting blood pressur~1 serum cholestoral in~2
#>    <dbl> <chr>  <chr>                                <dbl>                  <dbl>
#>  1    63 male   typical angina                         145                    233
```

```
#>  2    67 male   asymptomatic                       160                  286
#>  3    67 male   asymptomatic                       120                  229
#>  4    37 male   non-anginal pain                   130                  250
#>  5    41 female atypical angina                    130                  204
#>  6    56 male   atypical angina                    120                  236
#>  7    62 female asymptomatic                       140                  268
#>  8    57 female asymptomatic                       120                  354
#>  9    63 male   asymptomatic                       130                  254
#> 10    53 male   asymptomatic                       140                  203
#> # i 293 more rows
#> # i abbreviated names: 1: `resting blood pressure in mm Hg`,
#> #   2: `serum cholestoral in mg/dl`
#> # i 9 more variables: `fasting blood sugar > 120 mg/dl` <lgl>,
#> #   `resting electrocardiographic results` <dbl>,
#> #   `maximum heart rate achieved` <dbl>, `exercise induced angina` <chr>,
#> #   `ST depression` <dbl>, `slope of the peak exercise ST segment` <chr>, ...
```

> **Tip**
>
> For information on downloading the data, please refer to Appendix B.1. Should you encounter any errors when executing the code, first set up your RStudio project as described in Section 4.8.5, then re-run the code.

When you run `read_csv()`, you will notice a message detailing the number of rows and columns, the delimiter used, and information regarding the column types. This ensures that your data is read correctly. You may also specify how missing values are represented via the `na` argument. For example, setting `na = "?"` tells `read_csv()` to treat the `?` symbol as `NA` in the dataset:

```
heart_disease_data <- read_csv("r-data/cleveland-heart-disease-database.csv", na = "?")

heart_disease_data
```

```
#> # A tibble: 303 x 14
#>      age sex    `chest pain type` resting blood pressur~1 serum cholestoral in~2
#>    <dbl> <chr>  <chr>                                <dbl>                   <dbl>
#>  1    63 male   typical angina                         145                     233
#>  2    67 male   asymptomatic                           160                     286
#>  3    67 male   asymptomatic                           120                     229
#>  4    37 male   non-anginal pain                       130                     250
#>  5    41 female atypical angina                        130                     204
#>  6    56 male   atypical angina                        120                     236
#>  7    62 female asymptomatic                           140                     268
```

```
#>  8    57 female asymptomatic                          120                  354
#>  9    63 male   asymptomatic                          130                  254
#> 10    53 male   asymptomatic                          140                  203
#> # i 293 more rows
#> # i abbreviated names: 1: `resting blood pressure in mm Hg`,
#> #   2: `serum cholestoral in mg/dl`
#> # i 9 more variables: `fasting blood sugar > 120 mg/dl` <lgl>,
#> #   `resting electrocardiographic results` <dbl>,
#> #   `maximum heart rate achieved` <dbl>, `exercise induced angina` <chr>,
#> #   `ST depression` <dbl>, `slope of the peak exercise ST segment` <chr>, ...
```

Example 2: Writing to a CSV File

After processing or analysing your data, you might wish to save your results. The `write_csv()` function writes the data to disk, enabling you to share your cleaned or transformed data with others. The key arguments are:

- `x`: The data frame to be saved.
- `file`: the destination file path.

For example:

```
write_csv(heart_disease_data, "processed-cleveland-heart-disease-data.csv")
```

> **Tip**
>
> Additional arguments allow you to control the representation of missing values (using `na`) and whether to append to an existing file (using `append`). For instance:

4.8.2 Spreadsheets

Microsoft Excel is a widely used application that organises data into worksheets within a single workbook[2]. The `readxl` package is used to import Excel spreadsheets (e.g. `.xlsx` files) into R, while the `writexl` package is used to export data frames to Excel files.

The primary functions are:

- `read_xlsx()`: Imports an Excel file into R. You can specify the worksheet containing your data by using the `sheet` argument.

[2] If you or your collaborators rely on spreadsheets to organise data, we highly recommend the paper "Data Organization in Spreadsheets" by Karl Broman and Kara Woo: https://doi.org/10.1080/00031305.2017.1375989. It provides valuable guidance on best practices and efficient data structuring techniques.

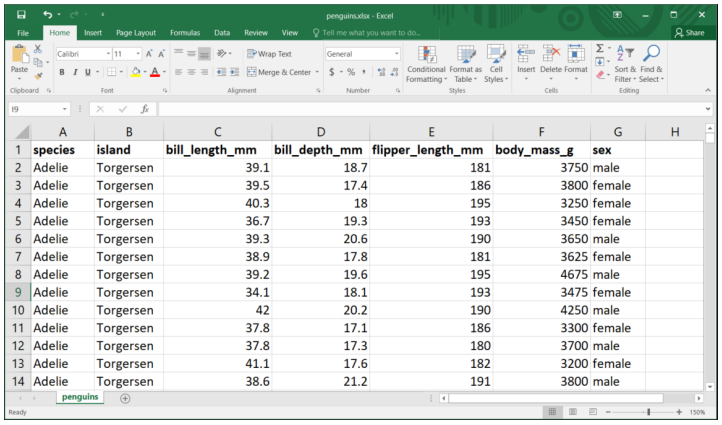
- `write_xlsx()`: Export your data frame to an Excel file—ideal for sharing your work with colleagues who prefer Excel.

Example 1: Reading Excel Spreadsheets

In this example, we import data from an Excel spreadsheet using the `readxl` package. Although `readxl` is not part of the core tidyverse, it is installed automatically with the tidyverse, so you must load it explicitly:

```
library(tidyverse)
library(readxl)
library(writexl)
```

The spreadsheet, as shown in Figure 4.9, can be downloaded from https://docs.google.com/spreadsheets/d/107H-n59gDw0QoIktksU9wv6Iro4TGUAOgmQJW6pb19Y



	A	B	C	D	E	F	G	H
1	species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g	sex	
2	Adelie	Torgersen	39.1	18.7	181	3750	male	
3	Adelie	Torgersen	39.5	17.4	186	3800	female	
4	Adelie	Torgersen	40.3	18	195	3250	female	
5	Adelie	Torgersen	36.7	19.3	193	3450	female	
6	Adelie	Torgersen	39.3	20.6	190	3650	male	
7	Adelie	Torgersen	38.9	17.8	181	3625	female	
8	Adelie	Torgersen	39.2	19.6	195	4675	male	
9	Adelie	Torgersen	34.1	18.1	193	3475	female	
10	Adelie	Torgersen	42	20.2	190	4250	male	
11	Adelie	Torgersen	37.8	17.1	186	3300	female	
12	Adelie	Torgersen	37.8	17.3	180	3700	male	
13	Adelie	Torgersen	41.1	17.6	182	3200	female	
14	Adelie	Torgersen	38.6	21.2	191	3800	male	

Figure 4.9: Spreadsheet called penguins.xlsx in Excel.

The first argument of `read_xlsx()` is the file path:

```
penguins <- read_xlsx("r-data/penguins.xlsx")
```

This function reads the file as a tibble[3]:

```
penguins
```

[3]A tibble is a modern version of an R data frame that provides cleaner printing and more consistent behavior, especially within the tidyverse ecosystem.

```
#> # A tibble: 337 x 7
#>    species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
#>    <chr>   <chr>              <dbl>         <dbl>             <dbl>       <dbl>
#>  1 Adelie  Torgersen           39.1          18.7               181        3750
#>  2 Adelie  Torgersen           39.5          17.4               186        3800
#>  3 Adelie  Torgersen           40.3          18                 195        3250
#>  4 Adelie  Torgersen           36.7          19.3               193        3450
#>  5 Adelie  Torgersen           39.3          20.6               190        3650
#>  6 Adelie  Torgersen           38.9          17.8               181        3625
#>  7 Adelie  Torgersen           39.2          19.6               195        4675
#>  8 Adelie  Torgersen           34.1          18.1               193        3475
#>  9 Adelie  Torgersen           42            20.2               190        4250
#> 10 Adelie  Torgersen           37.8          17.1               186        3300
#> # i 327 more rows
#> # i 1 more variable: sex <chr>
```

This dataset contains data on 337 penguins and includes seven variables for each species.

Example 2: Reading Worksheets

Spreadsheets may contain multiple worksheets. Figure 4.10 illustrates an Excel workbook with several sheets. The data, sourced from the ggplot2 package, is available at https://docs.google.com/spreadsheets/d/1izO0mHu3L9AMySQUXGDn9GPs1n-VwGFSEoAKGhqVQh0. Each worksheet contains information on diamond prices for different cuts.

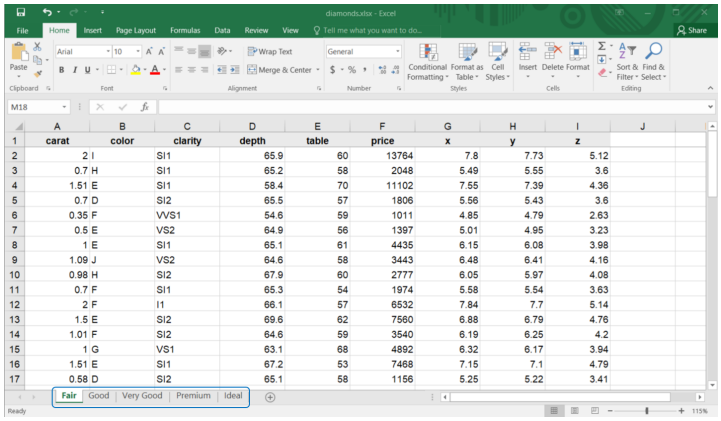


Figure 4.10: Spreadsheet called diamond.xlsx in Excel.

You can read a specific worksheet by using the `sheet` argument in `read_xlsx()`. By default, the first worksheet is read.

```r
diamonds_fair <- read_xlsx("r-data/diamonds.xlsx", sheet = "Fair")

diamonds_fair
```

```
#> # A tibble: 60 x 9
#>    carat color clarity depth table price     x     y     z
#>    <dbl> <chr> <chr>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#>  1  2    I     SI1      65.9    60 13764  7.8   7.73  5.12
#>  2  0.7  H     SI1      65.2    58  2048  5.49  5.55  3.6
#>  3  1.51 E     SI1      58.4    70 11102  7.55  7.39  4.36
#>  4  0.7  D     SI2      65.5    57  1806  5.56  5.43  3.6
#>  5  0.35 F     VVS1     54.6    59  1011  4.85  4.79  2.63
#>  6  0.5  E     VS2      64.9    56  1397  5.01  4.95  3.23
#>  7  1    E     SI1      65.1    61  4435  6.15  6.08  3.98
#>  8  1.09 J     VS2      64.6    58  3443  6.48  6.41  4.16
#>  9  0.98 H     SI2      67.9    60  2777  6.05  5.97  4.08
#> 10  0.7  F     SI1      65.3    54  1974  5.58  5.54  3.63
#> # i 50 more rows
```

If numerical data is read as text because the string "NA" is not automatically recognised as a missing value, you may correct this by specifying the `na` argument:

```r
read_excel("r-data/diamonds.xlsx", sheet = "Fair", na = "NA")
```

```
#> # A tibble: 60 x 9
#>    carat color clarity depth table price     x     y     z
#>    <dbl> <chr> <chr>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#>  1  2    I     SI1      65.9    60 13764  7.8   7.73  5.12
#>  2  0.7  H     SI1      65.2    58  2048  5.49  5.55  3.6
#>  3  1.51 E     SI1      58.4    70 11102  7.55  7.39  4.36
#>  4  0.7  D     SI2      65.5    57  1806  5.56  5.43  3.6
#>  5  0.35 F     VVS1     54.6    59  1011  4.85  4.79  2.63
#>  6  0.5  E     VS2      64.9    56  1397  5.01  4.95  3.23
#>  7  1    E     SI1      65.1    61  4435  6.15  6.08  3.98
#>  8  1.09 J     VS2      64.6    58  3443  6.48  6.41  4.16
#>  9  0.98 H     SI2      67.9    60  2777  6.05  5.97  4.08
#> 10  0.7  F     SI1      65.3    54  1974  5.58  5.54  3.63
#> # i 50 more rows
```

Another approach is to use `excel_sheets()` to list all the worksheets in an Excel file and then import only the ones you need.

```
excel_sheets("r-data/diamonds.xlsx")
```

```
#> [1] "Fair"      "Good"      "Very Good" "Premium"   "Ideal"
```

Once the worksheet names are known, they can be imported individually:

```
diamonds_fair <- read_excel("r-data/diamonds.xlsx", sheet = "Fair")

diamonds_good <- read_excel("r-data/diamonds.xlsx", sheet = "Good")

diamonds_very_good <- read_excel("r-data/diamonds.xlsx", sheet = "Very Good")

diamonds_premium <- read_excel("r-data/diamonds.xlsx", sheet = "Premium")

diamonds_ideal <- read_excel("r-data/diamonds.xlsx", sheet = "Ideal")
```

In this instance, the complete diamonds dataset is distributed across five worksheets that share the same columns but differ in the number of rows. You can inspect their dimensions using:

```
dim(diamonds_fair)
```

```
#> [1] 60  9
```

```
dim(diamonds_good)
```

```
#> [1] 49  9
```

```
dim(diamonds_very_good)
```

```
#> [1] 42  9
```

```
dim(diamonds_premium)
```

```
#> [1] 49  9
```

```
dim(diamonds_ideal)
```

```
#> [1] 60  9
```

You can also combine the worksheets into one data frame by using `bind_rows()`:

```
diamonds <- bind_rows(
  diamonds_fair,
  diamonds_good,
  diamonds_very_good,
  diamonds_premium,
  diamonds_ideal
)

diamonds
```

```
#> # A tibble: 260 x 9
#>    carat color clarity depth table price     x     y     z
#>    <dbl> <chr> <chr>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#>  1  2    I     SI1      65.9    60 13764  7.8   7.73  5.12
#>  2  0.7  H     SI1      65.2    58  2048  5.49  5.55  3.6
#>  3  1.51 E     SI1      58.4    70 11102  7.55  7.39  4.36
#>  4  0.7  D     SI2      65.5    57  1806  5.56  5.43  3.6
#>  5  0.35 F     VVS1     54.6    59  1011  4.85  4.79  2.63
#>  6  0.5  E     VS2      64.9    56  1397  5.01  4.95  3.23
#>  7  1    E     SI1      65.1    61  4435  6.15  6.08  3.98
#>  8  1.09 J     VS2      64.6    58  3443  6.48  6.41  4.16
#>  9  0.98 H     SI2      67.9    60  2777  6.05  5.97  4.08
#> 10  0.7  F     SI1      65.3    54  1974  5.58  5.54  3.63
#> # i 250 more rows
```

An alternative method to import all worksheets from a workbook is provided by the `read_excel_workbook()` function from the `bulkreadr` package. This function reads the data from every sheet in an Excel workbook and returns a single appended data frame:

```
library(bulkreadr)

diamonds <- read_excel_workbook("r-data/diamonds.xlsx")

diamonds
```

```
#> # A tibble: 260 x 9
#>    carat color clarity depth table price     x     y     z
#>    <dbl> <chr> <chr>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#>  1  2    I     SI1      65.9    60 13764  7.8   7.73  5.12
#>  2  0.7  H     SI1      65.2    58  2048  5.49  5.55  3.6
#>  3  1.51 E     SI1      58.4    70 11102  7.55  7.39  4.36
#>  4  0.7  D     SI2      65.5    57  1806  5.56  5.43  3.6
#>  5  0.35 F     VVS1     54.6    59  1011  4.85  4.79  2.63
#>  6  0.5  E     VS2      64.9    56  1397  5.01  4.95  3.23
#>  7  1    E     SI1      65.1    61  4435  6.15  6.08  3.98
#>  8  1.09 J     VS2      64.6    58  3443  6.48  6.41  4.16
#>  9  0.98 H     SI2      67.9    60  2777  6.05  5.97  4.08
#> 10  0.7  F     SI1      65.3    54  1974  5.58  5.54  3.63
#> # i 250 more rows
```

You can also specify the `.id` argument in `read_excel_workbook()` to add an output column that identifies the source of each row (using either the sheet names or their positions):

```
diamonds <- bulkreadr::read_excel_workbook("r-data/diamonds.xlsx", .id = "sheet")

diamonds
```

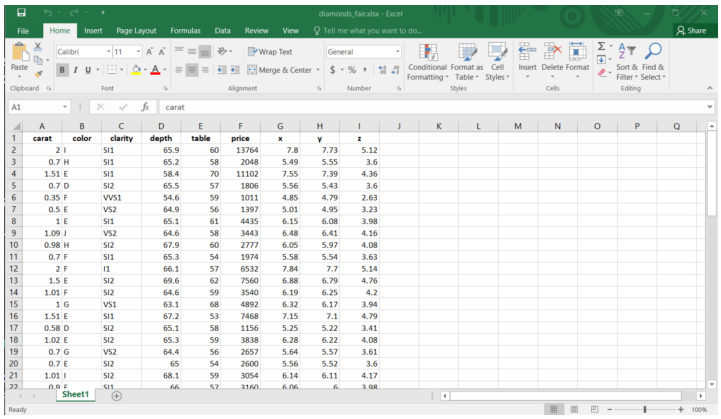
```
#> # A tibble: 260 x 10
#>    sheet carat color clarity depth table price     x     y     z
#>    <chr> <dbl> <chr> <chr>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#>  1 Fair   2    I     SI1      65.9    60 13764  7.8   7.73  5.12
#>  2 Fair   0.7  H     SI1      65.2    58  2048  5.49  5.55  3.6
#>  3 Fair   1.51 E     SI1      58.4    70 11102  7.55  7.39  4.36
#>  4 Fair   0.7  D     SI2      65.5    57  1806  5.56  5.43  3.6
#>  5 Fair   0.35 F     VVS1     54.6    59  1011  4.85  4.79  2.63
#>  6 Fair   0.5  E     VS2      64.9    56  1397  5.01  4.95  3.23
#>  7 Fair   1    E     SI1      65.1    61  4435  6.15  6.08  3.98
#>  8 Fair   1.09 J     VS2      64.6    58  3443  6.48  6.41  4.16
#>  9 Fair   0.98 H     SI2      67.9    60  2777  6.05  5.97  4.08
#> 10 Fair   0.7  F     SI1      65.3    54  1974  5.58  5.54  3.63
#> # i 250 more rows
```

Example 3: Writing Data to an Excel File

After processing your Excel data in R, you may wish to export the cleaned data. The `write_xlsx()` function from the `writexl` package enables you to save your data frame as a new Excel file:

```r
write_xlsx(diamonds_fair, path = "r-data/diamonds_fair.xlsx")
```

Figure 4.11 shows the resulting Excel file. By default, column names are included and bolded. You can disable these features by setting the `col_names` and `format_headers` arguments to `FALSE`.



carat	color	clarity	depth	table	price	x	y	z
2	I	SI1	65.9	60	13764	7.8	7.73	5.12
0.7	H	SI1	65.2	58	2048	5.49	5.55	3.6
1.51	E	SI1	58.4	70	11102	7.55	7.39	4.36
0.7	D	SI2	65.5	57	1806	5.56	5.43	3.6
0.35	F	VVS1	54.6	59	1011	4.85	4.79	2.63
0.5	E	VS2	64.9	56	1397	5.01	4.95	3.23
1	E	SI1	65.1	61	4435	6.15	6.08	3.98
1.09	J	VS2	64.6	58	3443	6.48	6.41	4.16
0.98	H	SI2	67.9	60	2777	6.05	5.97	4.08
0.7	F	SI1	65.3	54	1974	5.58	5.54	3.63
2	F	I1	66.1	57	6532	7.84	7.7	5.14
1.5	E	SI2	69.6	62	7560	6.88	6.79	4.76
1.01	F	SI2	64.6	59	3540	6.19	6.25	4.2
1	G	VS1	63.1	68	4892	6.32	6.17	3.94
1.51	E	SI1	67.2	53	7468	7.15	7.1	4.79
0.58	D	SI2	65.1	58	1156	5.25	5.22	3.41
1.02	E	SI2	65.3	59	3838	6.28	6.22	4.08
0.7	G	VS2	64.4	56	2657	5.64	5.57	3.61
0.7	E	SI2	65	54	2600	5.56	5.52	3.6
1.01	I	SI2	68.1	59	3054	6.14	6.11	4.17

Figure 4.11: Spreadsheet View of the File diamond-fair.xlsx in Excel.

4.8.3 Labelled Data

Labelled data commonly originates from specialised statistical software such as SPSS, Stata, or SAS. These datasets often include value labels that describe the meaning of each code. R can import these files using the `haven` package. Like the `readxl`, `haven` is not part of the core tidyverse, but is installed automatically with it; however, you must load it explicitly.

For example:

- SPSS
  - `read_sav()`: This function imports data from SPSS files (files with `.sav` extension) into R.
  - `write_sav()`: Exports a data frame from R back to SPSS format.
- Stata
  - `read_dta()`: For Stata users, this function imports Stata files into R.
  - `write_dta()`: Similarly, this function lets you export data frames to Stata format.
- SAS

- `read_sas()` reads .sas7bdat + .sas7bcat files and `read_xpt()` reads SAS transport files (versions 5 and 8).
- `write_xpt()` writes SAS transport files (versions 5 and 8).

Example 1: Reading an SPSS File

Suppose you have an SPSS file named `wages.sav`. You can import it using `haven` as follows:

```
library(tidyverse)
library(haven)

wages <- read_sav("r-data/wages.sav")

wages
```

```
#> # A tibble: 400 x 9
#>       id  educ south                 sex     exper  wage occup   marr    ed
#>    <dbl> <dbl> <dbl+lbl>             <dbl+l> <dbl> <dbl> <dbl+l> <dbl+l> <dbl+l>
#>  1     3    12 0 [does not live in ~ 0 [Mal~    17  7.5  6 [Oth~ 1 [Mar~ 2 [Hig~
#>  2     4    13 0 [does not live in ~ 0 [Mal~     9 13.1  6 [Oth~ 0 [Not~ 3 [Som~
#>  3     5    10 1 [lives in South]    0 [Mal~    27  4.45 6 [Oth~ 0 [Not~ 1 [Les~
#>  4    12     9 1 [lives in South]    0 [Mal~    30  6.25 6 [Oth~ 0 [Not~ 1 [Les~
#>  5    13     9 1 [lives in South]    0 [Mal~    29 20.0  6 [Oth~ 1 [Mar~ 1 [Les~
#>  6    14    12 0 [does not live in ~ 0 [Mal~    37  7.3  6 [Oth~ 1 [Mar~ 2 [Hig~
#>  7    17    11 0 [does not live in ~ 0 [Mal~    16  3.65 6 [Oth~ 0 [Not~ 1 [Les~
#>  8    20    12 0 [does not live in ~ 0 [Mal~     9  3.75 6 [Oth~ 0 [Not~ 2 [Hig~
#>  9    21    11 1 [lives in South]    0 [Mal~    14  4.5  6 [Oth~ 1 [Mar~ 1 [Les~
#> 10    23     6 1 [lives in South]    0 [Mal~    45  5.75 6 [Oth~ 1 [Mar~ 1 [Les~
#> # i 390 more rows
```

In some cases, haven's `read_sav()` may not fully process labelled variables. In such instances, you can use the `bulkreadr` package for a more seamless import:

```
library(bulkreadr)

# Import SPSS data with automatic label conversion

wages_data <- read_spss_data("r-data/wages.sav")

wages_data
```

```
#> # A tibble: 400 x 9
#>       id  educ south                 sex    exper  wage occup marr        ed
#>    <dbl> <dbl> <fct>                 <fct> <dbl> <dbl> <fct> <fct>       <fct>
#>  1     3    12 does not live in South Male     17  7.5  Other Married     High ~
#>  2     4    13 does not live in South Male      9 13.1  Other Not married Some ~
#>  3     5    10 lives in South         Male     27  4.45 Other Not married Less ~
#>  4    12     9 lives in South         Male     30  6.25 Other Not married Less ~
#>  5    13     9 lives in South         Male     29 20.0  Other Married     Less ~
#>  6    14    12 does not live in South Male     37  7.3  Other Married     High ~
#>  7    17    11 does not live in South Male     16  3.65 Other Not married Less ~
#>  8    20    12 does not live in South Male      9  3.75 Other Not married High ~
#>  9    21    11 lives in South         Male     14  4.5  Other Married     Less ~
#> 10    23     6 lives in South         Male     45  5.75 Other Married     Less ~
#> # i 390 more rows
```

Example 2: Writing to a SPSS File

After performing analyses or modifications on your SPSS data, you may wish to export the results back to an SPSS file:

```
wages_data |> write_sav("wages.sav")
```

Example 3: Reading Stata File

Similarly, if you have a Stata dataset (for example, `automobile.dta`), you can import it using `bulkreadr`:

```
# Import Stata data with bulkreadr

automobile <- read_stata_data("r-data/automobile.dta")

glimpse(automobile)
```

```
#> Rows: 32
#> Columns: 11
#> $ mpg  <dbl> 21.0, 21.0, 22.8, 21.4, 18.7, 18.1, 14.3, 24.4, 22.8, 19.2, 17.8,~
#> $ cyl  <dbl> 6, 6, 4, 6, 8, 6, 8, 4, 4, 6, 6, 8, 8, 8, 8, 8, 8, 4, 4, 4, 4, 8,~
#> $ disp <dbl> 160.0, 160.0, 108.0, 258.0, 360.0, 225.0, 360.0, 146.7, 140.8, 16~
#> $ hp   <dbl> 110, 110, 93, 110, 175, 105, 245, 62, 95, 123, 123, 180, 180, 180~
#> $ drat <dbl> 3.90, 3.90, 3.85, 3.08, 3.15, 2.76, 3.21, 3.69, 3.92, 3.92, 3.92,~
#> $ wt   <dbl> 2.620, 2.875, 2.320, 3.215, 3.440, 3.460, 3.570, 3.190, 3.150, 3.~
#> $ qsec <dbl> 16.46, 17.02, 18.61, 19.44, 17.02, 20.22, 15.84, 20.00, 22.90, 18~
```

```
#> $ vs   <dbl> 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0,~
#> $ am   <dbl> 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0,~
#> $ gear <dbl> 4, 4, 4, 3, 3, 3, 3, 4, 4, 4, 4, 3, 3, 3, 3, 3, 3, 4, 4, 4, 3, 3,~
#> $ carb <dbl> 4, 4, 1, 1, 2, 1, 4, 2, 2, 4, 4, 3, 3, 3, 4, 4, 4, 1, 2, 1, 1, 2,~
```

Example 4: Writing to a Stata File

If you wish to export a data frame to a Stata format, you can use haven's write function:

```
automobile |> write_dta("automobile-data.dta")
```

> The `rio` Package
>
> The `rio` package complements other data-importing packages by providing a one-stop solution for data import and export in R. It handles a wide variety of file types—CSV, Excel, SPSS, Stata, and more—so you don't need to remember different functions for each format.
>
> - **`import()`**: Automatically detects the file type and imports your data into R, simplifying the process of reading data from diverse sources.
> - **`export()`**: Export your data frame to various file formats with ease, whether you are creating a CSV file, an Excel workbook, or a file for statistical software.
>
> For further details on the extensive capabilities of the `rio` package, please refer to the rio documentation.

Example 1: Data Import with `rio` Package

In this example, we load the `telco-customer-churn.csv` file from the `r-data` folder using the `import()` function. This function automatically detects the file type and loads the data accordingly.

```
library(rio)

# Import data from a CSV file
telco_customer_churn <- import("r-data/telco-customer-churn.csv")
```

After importing the data, you may wish to perform some basic cleaning. For instance, you might standardise variable names using the `clean_names()` function from the `janitor` package:

```r
library(janitor)

# Clean variable names
telco_customer_churn <- telco_customer_churn |>
  clean_names()

# Glimpse the structure of the data
telco_customer_churn |>
  glimpse()
```

```
#> Rows: 7,043
#> Columns: 21
#> $ customer_id       <chr> "7590-VHVEG", "5575-GNVDE", "3668-QPYBK", "7795-CFOC~
#> $ gender            <chr> "Female", "Male", "Male", "Male", "Female", "Female"~
#> $ senior_citizen    <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0~
#> $ partner           <chr> "Yes", "No", "No", "No", "No", "No", "No", "No", "Ye~
#> $ dependents        <chr> "No", "No", "No", "No", "No", "No", "Yes", "No", "No~
#> $ tenure            <int> 1, 34, 2, 45, 2, 8, 22, 10, 28, 62, 13, 16, 58, 49, ~
#> $ phone_service     <chr> "No", "Yes", "Yes", "No", "Yes", "Yes", "Yes", "No",~
#> $ multiple_lines    <chr> "No phone service", "No", "No", "No phone service", ~
#> $ internet_service  <chr> "DSL", "DSL", "DSL", "DSL", "Fiber optic", "Fiber op~
#> $ online_security   <chr> "No", "Yes", "Yes", "Yes", "No", "No", "No", "Yes", ~
#> $ online_backup     <chr> "Yes", "No", "Yes", "No", "No", "No", "Yes", "No", "~
#> $ device_protection <chr> "No", "Yes", "No", "Yes", "No", "Yes", "No", "No", "~
#> $ tech_support      <chr> "No", "No", "No", "Yes", "No", "No", "No", "No", "Ye~
#> $ streaming_tv      <chr> "No", "No", "No", "No", "No", "Yes", "Yes", "No", "Y~
#> $ streaming_movies  <chr> "No", "No", "No", "No", "No", "Yes", "No", "No", "Ye~
#> $ contract          <chr> "Month-to-month", "One year", "Month-to-month", "One~
#> $ paperless_billing <chr> "Yes", "No", "Yes", "No", "Yes", "Yes", "Yes", "No",~
#> $ payment_method    <chr> "Electronic check", "Mailed check", "Mailed check", ~
#> $ monthly_charges   <dbl> 29.85, 56.95, 53.85, 42.30, 70.70, 99.65, 89.10, 29.~
#> $ total_charges     <dbl> 29.85, 1889.50, 108.15, 1840.75, 151.65, 820.50, 194~
#> $ churn             <chr> "No", "No", "Yes", "No", "Yes", "Yes", "No", "No", "~
```

Example 2: Exporting Data with `rio` Package

After importing your data, it is often necessary to review and adjust variable types. In this example, all character columns (which represent categorical variables) are converted to factors, except for the `customer_id` column which remains as a character to preserve its unique identifier status. Once the data is transformed (see Chapter 5 for further discussion on data transformation), we filter the data to include only records where `dependents` is `"Yes"` and then export the result to an Excel file.

```r
telco_customer_churn <- telco_customer_churn %>%
  mutate(across(where(is.character), as.factor)) %>%
  mutate(customer_id = as.character(customer_id))

# Filter data and export to an Excel file
telco_customer_churn |>
  filter(dependents == "Yes") |>
  export("telco_customer_churn.xlsx")
```

In this manner, `rio` offers a streamlined process for both importing and exporting data, reducing the need to juggle multiple packages or recall numerous functions.

4.8.4 Web Scraping

Web scraping is the process of extracting data from websites, and R provides several tools to facilitate this task. The `ralger` package is one such tool, offering a streamlined approach to retrieving and parsing data from web pages. Whether you are collecting data for text analysis, monitoring website updates, or simply gathering information from the web, `ralger` simplifies the process.

4.8.4.1 Key Features of the `ralger` Package

- **Simplified Data Retrieval**: `ralger` allows you to quickly download the HTML content of a web page without requiring multiple packages.
- **CSS Selector Support**: The package enables you to extract specific elements from a webpage by utilising CSS selectors.
- **Integrated Parsing Functions**: Once the HTML is retrieved, `ralger` provides functions to parse and manipulate the data efficiently.

Example 1: Extracting a Non-Table HTML Data

When web content is provided in HTML format, you can use the `tidy_scrap()` function from the `ralger` package to extract data into a tidy data frame. This function returns a data frame based on the arguments you supply. The key arguments are:

- **link**: The URL of the website you wish to scrape.
- **nodes**: A vector of CSS selectors corresponding to the HTML elements you want to extract. These elements will form the columns of your data frame.
- **colnames**: A vector of names to assign to the columns, which should match the order of the selectors specified in the **nodes** argument.

- **clean**: If set to `TRUE`, the function will clean the tibble's columns.
- **askRobot**: If enabled, the function will consult the site's `robots.txt` file to verify whether scraping is permitted.

In this example, we will scrape the Hacker News website (https://news.ycombinator.com). We aim to extract a tidy data frame containing the following elements:

- The story title.
- The site name (if available).
- The score of the story.
- Additional subline information.
- The username of the poster.

Here is how you can achieve this using the `tidy_scrap()` function with improved column names:

```
library(ralger)

# Define the URL for Hacker News
url <- "https://news.ycombinator.com/"

# Define the CSS selectors for the elements to extract
nodes <- c(".titleline > a", ".sitestr", ".score", ".subline a+ a", ".hnuser")

# Define descriptive column names for the resulting data frame
colnames <- c("title", "site", "score", "subline", "username")

# Extract the data from all pages using tidy_scrap()
news_data <- tidy_scrap(
  link = url,
  nodes = nodes,
  colnames = colnames,
  clean = TRUE
)
```

```
#> Warning in (function (..., deparse.level = 1) : number of rows of result is not
#> a multiple of vector length (arg 2)
```

```
news_data
```

```
#> # A tibble: 30 x 5
#>    title                                                site  score subline username
#>    <chr>                                                <chr> <chr> <chr>   <chr>
#>  1 Minding the gaps: A new way to draw separators ~ wind~ 89 p~ 33 com~ Sigmund~
#>  2 How I accepted myself into Canada's largest AI ~ fast~ 127 ~ 40 com~ fastcall
#>  3 Austral: A Systems Language with Linear Types a~ borr~ 83 p~ 10 com~ yamrzou
#>  4 'Dark oxygen': a deep-sea discovery that has sp~ phys~ 17 p~ 8 comm~ pseudol~
#>  5 'More Than a Hint' That Dark Energy Isn't What ~ nyti~ 55 p~ 42 com~ Hooke
#>  6 Hunyuan3D-2-Turbo: fast high-quality shape gene~ gith~ 103 ~ 20 com~ dvrp
#>  7 How fast the days are getting longer (2023)      joe-~ 538 ~ 189 co~ antogni~
#>  8 Bolt3D: Generating 3D Scenes in Seconds          szym~ 206 ~ 31 com~ jasonda~
#>  9 Diagrams AI can, and cannot, generate            ilog~ 86 p~ 14 com~ billyp-~
#> 10 SoftBank Group to Acquire Ampere Computing for ~ grou~ 83 p~ 41 com~ geerlin~
#> # i 20 more rows
```

Tip

In this code:

- The `tidy_scrap()` function is called with the Hacker News URL.
- The **nodes** argument specifies the CSS selectors for the elements we wish to extract. The nodes vector contains the CSS selectors:
  - ".titleline > a": extracts the story title,
  - ".sitestr": extracts the site name,
  - ".score": extracts the story score,
  - ".subline a+ a": extracts additional subline information,
  - ".hnuser": extracts the username.
- The **colnames** argument assigns clear and descriptive names to the columns—namely `"title"`, `"site"`, `"score"`, `"subline"`, and `"username"`.
- The function returns a tidy data frame in which all columns are of character class; you may convert these to other types as required for your analysis.

If you wish to scrape multiple list pages, you can use `tidy_scrap()` in conjunction with `paste0()`. Suppose you want to scrape pages 1 through 5 of Hacker News:

```
# Define the URL for Hacker News
url <- "https://news.ycombinator.com/"

# Create a vector of URLs for pages 1 to 5
```

```
links <- paste0(url, "?p=", seq(1, 5, 1))

# Extract the data from all pages using tidy_scrap()
news_data <- tidy_scrap(
  link = links,
  nodes = nodes,
  colnames = colnames
)
```

```
#> Warning in (function (..., deparse.level = 1) : number of rows of result is not
#> a multiple of vector length (arg 2)
```

```
news_data
```

```
#> # A tibble: 150 x 5
#>    title                                              site  score subline username
#>    <chr>                                              <chr> <chr> <chr>   <chr>
#>  1 Minding the gaps: A new way to draw separators ~ wind~ 89 p~ 33 com~ Sigmund~
#>  2 How I accepted myself into Canada's largest AI ~ fast~ 127 ~ 40 com~ fastcall
#>  3 Austral: A Systems Language with Linear Types a~ borr~ 83 p~ 10 com~ yamrzou
#>  4 'Dark oxygen': a deep-sea discovery that has sp~ phys~ 17 p~ 8 comm~ pseudol~
#>  5 'More Than a Hint' That Dark Energy Isn't What ~ nyti~ 55 p~ 42 com~ Hooke
#>  6 Hunyuan3D-2-Turbo: fast high-quality shape gene~ gith~ 103 ~ 20 com~ dvrp
#>  7 How fast the days are getting longer (2023)      joe-~ 539 ~ 189 co~ antogni~
#>  8 Bolt3D: Generating 3D Scenes in Seconds          szym~ 207 ~ 31 com~ jasonda~
#>  9 Diagrams AI can, and cannot, generate            ilog~ 86 p~ 14 com~ billyp-~
#> 10 SoftBank Group to Acquire Ampere Computing for ~ grou~ 84 p~ 41 com~ geerlin~
#> # i 140 more rows
```

> **Note**
>
> Since Hacker News is a dynamic website, the content may change over time. Therefore, if you rerun the same lines of code at a later time, the extracted results may differ from those obtained previously.

Example 2: Extracting an HTML Table

The `ralger` package also includes a function called `table_scrap()` for extracting HTML tables from a web page. Suppose you want to extract an HTML table from a page listing the highest lifetime gross revenues in the cinema industry. You can use the following code:

```r
# Extract an HTML table from the specified URL

url <- "https://www.boxofficemojo.com/chart/top_lifetime_gross/?area=XWW"

lifetime_gross <- table_scrap(link = url)
```

```
#> Warning: The `fill` argument of `html_table()` is deprecated as of rvest 1.0.0.
#> i An improved algorithm fills by default so it is no longer needed.
#> i The deprecated feature was likely used in the rvest package.
#>   Please report the issue at <https://github.com/tidyverse/rvest/issues>.
```

```r
# Display the extracted table
lifetime_gross
```

```
#> # A tibble: 200 x 4
#>     Rank Title                                    `Lifetime Gross`  Year
#>    <int> <chr>                                    <chr>            <int>
#>  1     1 Avatar                                   $2,923,710,708    2009
#>  2     2 Avengers: Endgame                        $2,799,439,100    2019
#>  3     3 Avatar: The Way of Water                 $2,320,250,281    2022
#>  4     4 Titanic                                  $2,264,812,968    1997
#>  5     5 Star Wars: Episode VII - The Force Awakens $2,071,310,218  2015
#>  6     6 Avengers: Infinity War                   $2,052,415,039    2018
#>  7     7 Spider-Man: No Way Home                  $1,952,732,181    2021
#>  8     8 Ne Zha 2                                 $1,889,217,312    2025
#>  9     9 Inside Out 2                             $1,698,863,816    2024
#> 10    10 Jurassic World                           $1,671,537,444    2015
#> # i 190 more rows
```

> **Note**
>
> If you are dealing with a web page that contains multiple HTML tables, you can use the `choose` argument with `table_scrap()` to target a specific table. For more advanced use cases and customisation options, please refer to the package documentation.

By incorporating web scraping into your data analysis workflow, you can dynamically collect data from the web and integrate it with your existing analyses, thereby broadening the scope of your data-driven insights.

4.8.5 Bringing It All Together

When working on a project, it is best practice to organise your data and code within a dedicated RStudio project. Store your data files (be they flat files, spreadsheets, or labelled files) in a folder (for example, `data/`), and use relative paths when reading and writing data. This approach promotes reproducibility and clarity. Let us now practise importing data using the `gapminder.csv` file.

Before We Begin:

1. **Create a Directory**
   Create a new folder on your desktop called `Experiment 4.3`.
2. **Download Data**
   Visit Google Drive to download the `r-data` folder (refer to Appendix B.1 for additional information). Once downloaded, unzip the folder and move it into your `Experiment 4.3` folder.
3. **Create an RStudio Project**
   Open RStudio and set up a new project as follows:
   - Go to **File > New Project**.
   - Select **Existing Directory** and navigate to your `Experiment 4.3` folder.
     This project setup will organise your work and ensure that everything related to this experiment is contained in one place.

Your project structure should resemble the one shown in Figure 4.12:

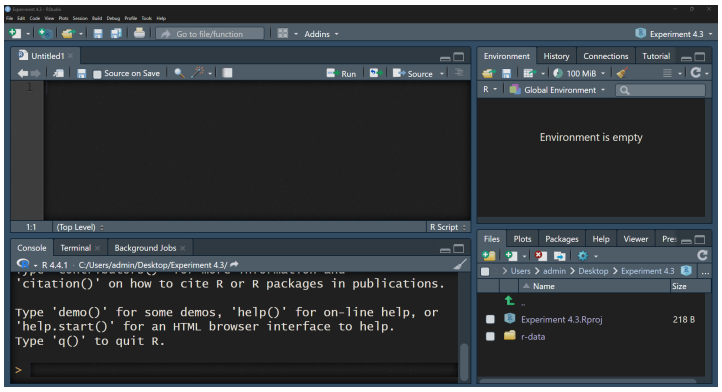


Figure 4.12: Starting a New R Project in RStudio

Now, let us import the `gapminder.csv` file from the `r-data` folder into R using the tidyverse package for convenient data manipulation and visualisation:

```r
library(tidyverse)

# Load the gapminder data
gapminder <- read_csv("r-data/gapminder.csv")
```

```
#> Rows: 1704 Columns: 6
#> -- Column specification --------------------------------------------------------
#> Delimiter: ","
#> chr (2): country, continent
#> dbl (4): year, lifeExp, pop, gdpPercap
#>
#> i Use `spec()` to retrieve the full column specification for this data.
#> i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```r
# Explore the data
names(gapminder)
```

```
#> [1] "country"   "continent" "year"      "lifeExp"   "pop"       "gdpPercap"
```

```r
dim(gapminder)
```

```
#> [1] 1704    6
```

```r
head(gapminder)
```

```
#> # A tibble: 6 x 6
#>   country     continent  year lifeExp      pop gdpPercap
#>   <chr>       <chr>     <dbl>   <dbl>    <dbl>     <dbl>
#> 1 Afghanistan Asia       1952    28.8  8425333      779.
#> 2 Afghanistan Asia       1957    30.3  9240934      821.
#> 3 Afghanistan Asia       1962    32.0 10267083      853.
#> 4 Afghanistan Asia       1967    34.0 11537966      836.
#> 5 Afghanistan Asia       1972    36.1 13079460      740.
#> 6 Afghanistan Asia       1977    38.4 14880372      786.
```

```r
summary(gapminder)
```

```
#>    country           continent              year         lifeExp
#>  Length:1704        Length:1704        Min.   :1952   Min.   :23.60
#>  Class :character   Class :character   1st Qu.:1966   1st Qu.:48.20
#>  Mode  :character   Mode  :character   Median :1980   Median :60.71
#>                                        Mean   :1980   Mean   :59.47
#>                                        3rd Qu.:1993   3rd Qu.:70.85
#>                                        Max.   :2007   Max.   :82.60
#>       pop              gdpPercap
#>  Min.   :6.001e+04   Min.   :   241.2
#>  1st Qu.:2.794e+06   1st Qu.:  1202.1
#>  Median :7.024e+06   Median :  3531.8
#>  Mean   :2.960e+07   Mean   :  7215.3
#>  3rd Qu.:1.959e+07   3rd Qu.:  9325.5
#>  Max.   :1.319e+09   Max.   :113523.1
```

After running this script, your dataset should be loaded into RStudio and ready for exploration. Your script should resemble the one in Figure 4.13:

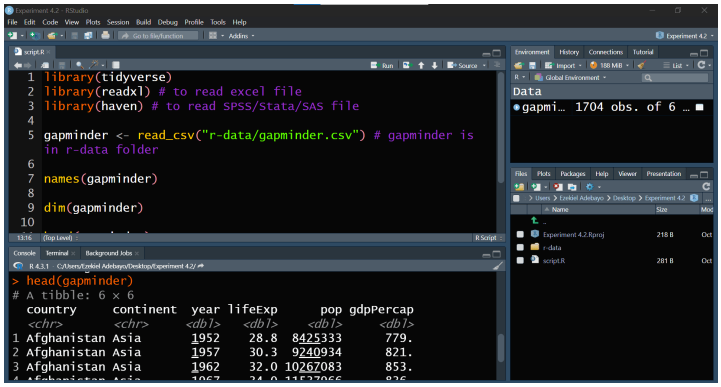


Figure 4.13: Loading and Exploring Data in RStudio

This setup ensures that everything required for your analysis is neatly organised, reproducible, and ready for you to begin analysing the gapminder data in R.

4.8.6 Practice Quiz 4.3

Question 1:

Which package is commonly used to read CSV files into R as tibbles?

a) `readxl`

b) `haven`

c) `readr`

d) `writexl`

Question 2:

If you need to import an Excel file, which function would you likely use?

a) `read_csv()`

b) `read_xlsx()`

c) `read_sav()`

d) `read_dta()`

Question 3:

Which package would you use to easily handle a wide variety of data formats without memorising specific functions for each?

a) `rio`

b) `haven`

c) `janitor`

d) `readxl`

Question 4:

After cleaning and analysing your data, which function would you use to write the results to a CSV file?

a) `write_xlsx()`

b) `export()`

c) `write_csv()`

d) `import()`

See the Solution to Quiz 4.3

4.8.7 Exercise 4.3.1: Medical Insurance Data

In this exercise, you'll explore the `medical-insurance.xlsx` file located in the `r-data` folder. You can download this file from Google Drive. This dataset contains medical insurance information for various individuals. Below is an overview of each column:

1. **User ID**: A unique identifier for each individual.
2. **Gender**: The individual's gender ('Male' or 'Female').
3. **Age**: The age of the individual in years.
4. **AgeGroup**: The age bracket the individual falls into.
5. **Estimated Salary**: An estimate of the individual's yearly salary.
6. **Purchased**: Indicates whether the individual has purchased medical insurance (1 for Yes, 0 for No).

Your Tasks:

1. **Importing and Basic Handling**:
   - Create a new script and import the data from the Excel file.
   - How would you import this data if it's in SPSS format?
   - Use the `clean_names()` function from the `janitor` package to make variable names consistent and easy to work with.
   - Can you display the first three rows of the dataset?
   - How many rows and columns does the dataset have?
2. **Understanding the Data**:
   - What are the column names in the dataset?
   - Can you identify the data types of each column?
3. **Basic Descriptive Statistics**:
   - What is the average age of the individuals in the dataset?
   - What's the range of the estimated salaries?

4.9 Reflective Summary

In **Lab 4**, you have acquired essential skills to enhance your efficiency and effectiveness as an R programmer:

- **Installing and Loading Packages**: You learned how to find, install, and load packages from CRAN and external repositories like GitHub.
- **Reproducible Workflows with RStudio Projects**: You discovered the importance of organizing your work within RStudio Projects.
- **Importing and Exporting Data**: You practiced importing and exporting data in various formats (CSV, Excel, SPSS) using packages like `readr`, `readxl`, and `haven`.

These skills are fundamental for efficient data analysis, helping you manage diverse data sources, maintain integrity in your analyses, and collaborate more effectively. Congratulations on progressing as an R programmer and data analyst!

> What's Next?
>
> In the next lab, we'll delve into data transformation where will reshape raw data into a more useful format for data analysis.

Part II

Data Analytics

5 Data Transformation

5.1 Introduction

Welcome to Lab 5! In this lab, we will focus on one of the most important steps in the data analysis process: data transformation. Real-world data rarely arrives in perfect, analysis-ready form. Before generating insights, visualising patterns, or constructing models, we must **transform** the data: cleaning, reshaping, and summarising it into a more meaningful structure.

In this lab, we will explore:

- The native pipe operator `|>` to create a smooth, readable pipeline of data operations.
- The core `dplyr` verbs—`select()`, `filter()`, `mutate()`, `arrange()`, and `summarise()`—to efficiently manipulate and refine your data.
- Strategies for grouping and summarising data to extract patterns and trends.
- Techniques for identifying and handling missing values responsibly.

These transformations form a vital step in any data workflow, ensuring that your datasets are well-prepared for subsequent analysis or visualisation.

5.2 Learning Objectives

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

- **Streamline Your Code Using the Pipe Operator `|>`**
  Connect multiple data operations into a logical sequence, enhancing code readability and reducing nesting complexity.
- **Perform Key Data Transformation Tasks Using `dplyr`**
  Utilise functions like `select()`, `filter()`, `arrange()`, `mutate()`, and `summarise()` toreshape, refine, and improve datasets for analysis.
- **Gain Insights Through Summarisation and Grouping**
  Group data and apply aggregation functions to extract meaningful summaries and trends.

- **Handle and Impute Missing Data**
  Identify missing values in your datasets, understand their impact, and apply suitable strategies (e.g., removal, mean/median imputation) to maintain data integrity.
- **Prepare Data for Analysis and Visualisation**
  Clean and structure your datasets so they are ready for modelling, plotting, and communicating results effectively.

By completing this lab, you will master data transformation and become more confident in dealing with messy, real-world datasets and advancing towards deeper analyses.

5.3 Prerequisites

Before starting this lab, you should have:

- Completed **Lab 4** or have a solid understanding of organising R projects and managing packages.
- Familiarity with loading data into R and conducting basic data checks (e.g., using `glimpse()`, `head()`).
- Interest in refining data, ensuring it is tidy, structured, and ready for more advanced analyses.

5.4 What is Data Transformation?

Data transformation is the process of reshaping raw data into a more useful format. Real-world data is rarely perfect for immediate analysis. It often contains extra variables, missing values, or is structured in a way that is not conducive to answering your research questions. Data transformation includes:

- Selecting relevant parts of the dataset: Focusing on the rows and columns you actually need.
- Creating new variables: Deriving meaningful metrics from existing data.
- Summarising information: Computing averages, totals, or other aggregate statistics to distil complex information into digestible summaries.
- Organising data into a meaningful structure: Arranging your dataset so that the relationships between variables and observations are clear.

You can think of data transformation like preparing ingredients before cooking: you wash, chop, and measure everything out so that when you start cooking, you can focus on creating your dish without interruptions or confusion.

5.5 Real-World Scenario: Preparing Data for Analysis

Imagine that you work as a data analyst at a wildlife research centre. You have received a raw dataset containing information on various species: their body measurements, diets, sleep patterns, and more. Before you can analyse ecological relationships, test hypotheses, or build models, you need to clean and organise this data. Using the techniques in this lab, you can:

- **Select** only the columns necessary for your investigation.
- **Filter** out rows that are not relevant to your study.
- **Mutate** the dataset to create new metrics or correct errors.
- **Arrange** the rows to highlight the largest or smallest values.
- **Summarise** the data by group to identify patterns by species or habitat.
- **Handle missing values** so that they do not distort your findings.

By applying these transformations, you ensure that your data is analysis-ready.

5.6 Experiment 5.1: The Pipe Operator |>

One of the best tools to simplify your R code is the pipe operator. Traditionally, the `<%>` operator from the `magrittr` package has been widely used for this purpose. However, starting from R version 4.1.0, R introduced a native pipe operator `|>`. The pipe operator allows you to chain functions together in a linear, logical sequence, rather than nesting them inside one another. Using pipes makes your code more readable and helps you think through your data transformations step-by-step.

In this lab, we will be using the base pipe operator `|>`, which functions similarly to the `magrittr` `<%>` operator[1]. Imagine you have data frame, `data`, and you want to perform multiple operations on it, such as applying functions `foo` and `bar` in sequence. Without a pipe, you might write as:

```
bar(foo(data))
```

This is harder to read than:

[1]The `|>` operator (called a pipe) means "and then." It passes the result of one function to the next.

```
data |>
  foo() |>
  bar()
```

In the piped version, you start with `data`, then say "and then apply `foo()`," and then "and then apply `bar()`," which feels more intuitive and mirrors how we naturally describe processes in words.

How to configure native pipe operator

To configure RStudio to insert the base pipe operator `|>` instead of `%>%` when pressing Ctrl/Cmd + Shift + M, navigate to the Tools menu, select Global Options..., then go to the Code section. In the Code options, check the box labelled Use native pipe operator, |> (requires R 4.1+).

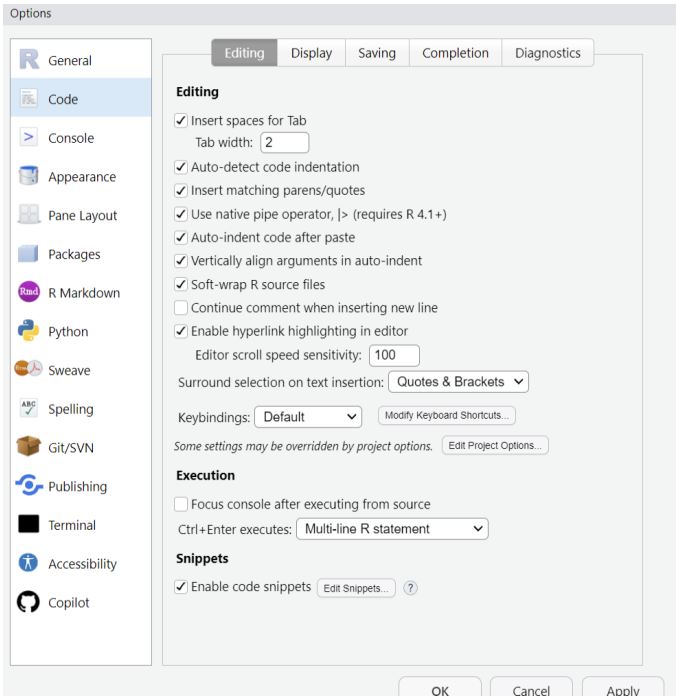


Figure 5.1: To insert |>, make sure the "Use native pipe operator" option is checked

How Does the Pipe Operator Work?

The pipe operator `|>` takes the output of one function and passes it as the first argument to the next function.

Example 1

For instance, consider:

```
iris |> head()
```

```
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#> 1          5.1         3.5          1.4         0.2  setosa
#> 2          4.9         3.0          1.4         0.2  setosa
#> 3          4.7         3.2          1.3         0.2  setosa
#> 4          4.6         3.1          1.5         0.2  setosa
#> 5          5.0         3.6          1.4         0.2  setosa
#> 6          5.4         3.9          1.7         0.4  setosa
```

This is exactly the same as:

```
head(iris)
```

```
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#> 1          5.1         3.5          1.4         0.2  setosa
#> 2          4.9         3.0          1.4         0.2  setosa
#> 3          4.7         3.2          1.3         0.2  setosa
#> 4          4.6         3.1          1.5         0.2  setosa
#> 5          5.0         3.6          1.4         0.2  setosa
#> 6          5.4         3.9          1.7         0.4  setosa
```

Example 2:

Here's another example combining multiple functions:

```
x <- 4.234

x |>
  sqrt() |>
  log() |>
  round(2)
```

```
#> [1] 0.72
```

This sequence is equivalent to the nested version:

```r
x <- 4.234

round(log(sqrt(x)), 2)
```

```
#> [1] 0.72
```

> **Reflection Question:**
>
> How does using the pipe operator enhance clarity, compared to nested function calls, especially when performing multiple operations on the same dataset?

By using pipes, you avoid writing nested code and make the flow of your data transformation much clearer.

5.6.1 Practice Quiz 5.1

Question 1:

What is the primary purpose of the pipe operator (`|>` or `%>%`) in R?

a) To run code in parallel.

b) To nest functions inside one another.

c) To pass the output of one function as the input to the next, improving code readability.
d) To automatically clean missing data.

Question 2:

Consider the following R code snippets:

```r
numbers <- c(2, 4, 6)

# Nested function version:
result1 <- round(sqrt(sum(numbers)))

# Pipe operator version:
result2 <- numbers |> sum() |> sqrt() |> round()
```

For a new R learner, is the pipe operator version generally more readable than the nested function version?

a) True
b) False

Question 3:

What is the output of the following R code?

```
result <- c(5, 10, 15)
result |> mean()
```

a) 10

b) 15

c) 5

d) 30

Question 4:

Which of the following code snippets correctly uses the pipe operator to apply the `sqrt()` function to the sum of numbers from 1 to 4?

a) `sqrt(sum(1:4))`

b) `1:4 |> sum() |> sqrt()`

c) `sum(1:4) |> sqrt`

d) `1:4 |> sqrt() |> sum()`

Question 5:

What will be the output of the following code?

```
result <- letters
result |> head(3)
```

a) `c("a", "b", "c")`

b) `c("x", "y", "z")`

c) `c("A", "B", "C")`

d) An error is thrown.

See the Solution to Quiz 5.1

5.7 Experiment 5.2: Data Manipulation with dplyr

In data analysis, we often encounter datasets that aren't in the ideal format for our needs. Data comes in all shapes and sizes, and making sense of it requires effective manipulation. This is where data manipulation becomes essential—a fundamental skill that allows you to transform and summarize data efficiently.

The dplyr package, part of the tidyverse, is designed to make data manipulation in R more approachable, efficient, and intuitive. Think of dplyr as your Swiss Army knife for taming messy datasets. It simplifies tasks like filtering, summarizing, grouping, and transforming data. The best part? Its syntax is easy to read and write, almost like having a conversation with your data.

Why Use dplyr?

- Simplicity: Provides straightforward functions that are easy to learn and remember, lowering the barrier to effective data manipulation.
- Efficiency: Optimized for performance, it handles large datasets swiftly, saving you time and computational resources.
- Readability: Code written with dplyr is often more readable and easier to maintain, which is especially beneficial when collaborating with others or revisiting your own work.
- Integration: Works seamlessly with other tidyverse packages like ggplot2 and tidyr, allowing for a cohesive and efficient data analysis workflow.

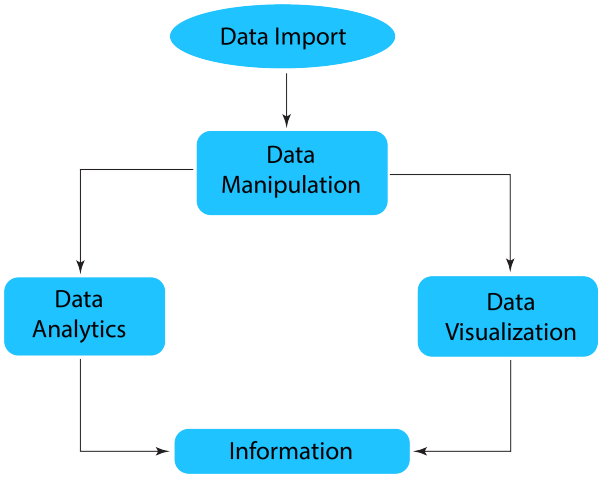


Figure 5.2: Data Exploration and Analysis Workflow

Getting Started

First, ensure you have the `dplyr` package installed and loaded. If you haven't installed it yet, you can install the `tidyverse`, which includes `dplyr`.

```
# Install the tidyverse package (if not already installed)
install.packages("tidyverse")

# Load the tidyverse package
library(tidyverse)
```

Core `dplyr` Verbs

The core functions in `dplyr` are often referred to as "verbs" because they describe actions you perform on your data:

- `select()`: Choose variables (columns) based on their names or column positions.
- `mutate()`: Create new columns or modify existing ones.
- `filter()`: Select rows based on specific conditions.
- `arrange()`: Reorder rows based on column values.
- `summarise()`: Reduce multiple values down to a summary statistic.
- `group_by()`: Group data by one or more variables for grouped operations.

When `summarise()` is paired with `group_by()`, it allows you to get a summary row for each group in the data frame.

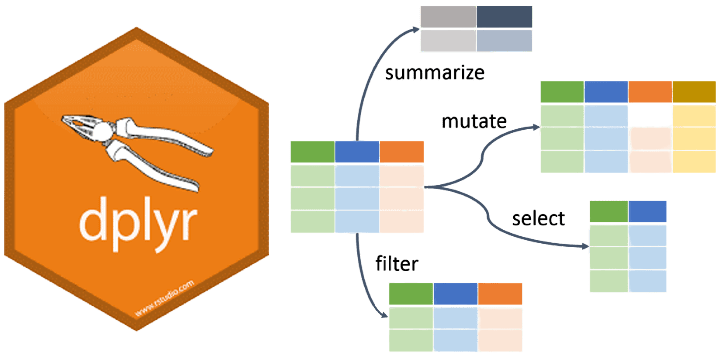


Figure 5.3: Key Data Manipulation Functions in dplyr

Additional useful functions include:

- `rename()`: Rename columns.
- `distinct()`: Find unique rows.

- `count()`: Count unique values of a variable.

Using Pipes with `dplyr` functions

One of the key strengths of `dplyr` is its ability to integrate seamlessly with the pipe operator (`%>%` or `|>`), enabling a clean, readable, and intuitive workflow for data manipulation. As illustrated in Figure 5.4, the pipe operator acts as a connector, allowing you to chain multiple **dplyr** functions in a logical sequence. This approach makes your code easier to read and follow, mirroring the flow of a conversation about your data.

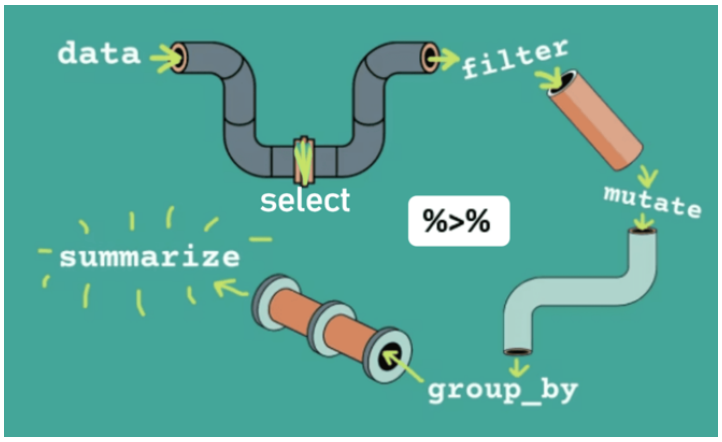


Figure 5.4: Data Transformation Pipeline in dplyr

5.7.1 Working with the `dplyr` Verbs

Let's take a deeper dive into each `dplyr` verb and understand not just how to use them but also what makes them powerful. Remember, the five core verbs—`filter()`, `select()`, `mutate()`, `arrange()`, and `summarize()`—are like tools in a toolbox. Each has a specific purpose, but together, they allow you to transform data seamlessly.

Example Datasets

We'll start our exploration by working with two fascinating datasets: the **`penguins`** dataset from the `palmerpenguins` package[2] and the **`msleep`** dataset from the `ggplot2` package. These datasets provide rich, real-world data that will help you practice and apply data manipulation in this book.

1. **The `penguins` Dataset**

[2]If you haven't installed it yet, you can do so with `install.packages("palmerpenguins")` and load it using `library(palmerpenguins)`.

The `penguins` dataset[3] contains detailed body measurements for 344 penguins from three different species—Adélie, Chinstrap, and Gentoo—found on three islands in the Palmer Archipelago of Antarctica. This dataset includes variables such as:

- **Species**: The penguin species.
- **Island**: The island where each penguin was observed.
- **Bill Length and Depth**: Measurements of the penguin's bill (beak).
- **Flipper Length**: The length of the penguin's flippers.
- **Body Mass**: The weight of the penguin.
- **Sex**: The gender of the penguin.

```
penguins <- palmerpenguins::penguins

penguins |> glimpse()
```

```
#> Rows: 344
#> Columns: 8
#> $ species           <fct> Adelie, Adelie, Adelie, Adelie, Adelie, Adelie, Adel~
#> $ island            <fct> Torgersen, Torgersen, Torgersen, Torgersen, Torgerse~
#> $ bill_length_mm    <dbl> 39.1, 39.5, 40.3, NA, 36.7, 39.3, 38.9, 39.2, 34.1, ~
#> $ bill_depth_mm     <dbl> 18.7, 17.4, 18.0, NA, 19.3, 20.6, 17.8, 19.6, 18.1, ~
#> $ flipper_length_mm <int> 181, 186, 195, NA, 193, 190, 181, 195, 193, 190, 186~
#> $ body_mass_g       <int> 3750, 3800, 3250, NA, 3450, 3650, 3625, 4675, 3475, ~
#> $ sex               <fct> male, female, female, NA, female, male, female, male~
#> $ year              <int> 2007, 2007, 2007, 2007, 2007, 2007, 2007, 2007, 2007~
```

2. **The `msleep` Dataset**

Our second dataset, `msleep`, comes from the `ggplot2` package and contains information on the sleep habits of 83 different mammals. This dataset includes 11 variables, such as:

- **name**: The common name of the mammal.
- **genus**: The taxonomic genus of the mammal.
- **vore**: The dietary category of the mammal. Possible values include:
  - `"carni"`: Carnivore (meat-eating)
  - `"herbi"`: Herbivore (plant-eating)

[3]Horst AM, Hill AP, Gorman KB (2020). palmerpenguins: Palmer Archipelago (Antarctica) penguin data. R package version 0.1.0. https://allisonhorst.github.io/palmerpenguins/. doi: 10.5281/zenodo.3960218.

  - `"omni"`: Omnivore (eating both plants and meat)
  - `"insecti"`: Insectivore (eating insects)
- **order**: The taxonomic order to which the mammal belongs (e.g., *Primates*, *Carnivora*).
- **conservation**: The conservation status of the species, indicating its level of threat or endangerment. Possible values include:
  - `"lc"`: Least Concern
  - `"nt"`: Near Threatened
  - `"vu"`: Vulnerable
  - `"en"`: Endangered
  - `"cr"`: Critically Endangered
  - `"domesticated"`: Domesticated species
- **sleep total**: Total amount of sleep per day (in hours).
- **sleep rem**: Amount of REM sleep per day.
- **sleep cycle**: Length of the sleep cycle.
- **awake**: The number of hours the mammal spends awake each day (calculated as `24 - sleep_total`).
- **brain weight**: The brain weight of the animal.
- **body weight**: The body weight of the animal.

```
msleep <- ggplot2::msleep

msleep |> glimpse()
```

```
#> Rows: 83
#> Columns: 11
#> $ name         <chr> "Cheetah", "Owl monkey", "Mountain beaver", "Greater shor~
#> $ genus        <chr> "Acinonyx", "Aotus", "Aplodontia", "Blarina", "Bos", "Bra~
#> $ vore         <chr> "carni", "omni", "herbi", "omni", "herbi", "herbi", "carn~
#> $ order        <chr> "Carnivora", "Primates", "Rodentia", "Soricomorpha", "Art~
#> $ conservation <chr> "lc", NA, "nt", "lc", "domesticated", NA, "vu", NA, "dome~
#> $ sleep_total  <dbl> 12.1, 17.0, 14.4, 14.9, 4.0, 14.4, 8.7, 7.0, 10.1, 3.0, 5~
#> $ sleep_rem    <dbl> NA, 1.8, 2.4, 2.3, 0.7, 2.2, 1.4, NA, 2.9, NA, 0.6, 0.8, ~
#> $ sleep_cycle  <dbl> NA, NA, NA, 0.1333333, 0.6666667, 0.7666667, 0.3833333, N~
#> $ awake        <dbl> 11.9, 7.0, 9.6, 9.1, 20.0, 9.6, 15.3, 17.0, 13.9, 21.0, 1~
```

```
#> $ brainwt      <dbl> NA, 0.01550, NA, 0.00029, 0.42300, NA, NA, NA, 0.07000, 0~
#> $ bodywt       <dbl> 50.000, 0.480, 1.350, 0.019, 600.000, 3.850, 20.490, 0.04~
```

> **Tip**
>
> The `glimpse()` function allows us to quickly view the structure of a data frame in a concise and readable format without printing the entire dataset. It serves as a more user-friendly alternative to the `str()` function, making it easier to understand the overall composition of your data at a glance.

5.7.2 `select()` – Picking Specific Columns

The **`select()`** function allows you to extract specific columns from a dataset, focusing on only the features relevant to your analysis. It's like creating a spotlight for just the features you need.

Key Points:

1. Select Columns by Name or Position:
   - You can specify columns by their names or their positions (e.g., 1, 2, 3).
   - This is useful when you only need a few specific columns from a large dataset.
2. Use Helper Functions for Pattern Matching:
   - `starts_with("prefix")`: Selects columns whose names begin with a specific prefix.
   - `ends_with("suffix")`: Selects columns whose names end with a specific suffix.
   - `contains("substring")`: Selects columns whose names contain a specific substring.
3. Deselect Columns Using the Minus (`-`) Sign:
   - You can drop columns by prefixing their names, indices, or helper functions with `-`.
   - This is useful when you want to keep most columns but exclude specific ones.
4. Use `where()` in `select()`:
   - This allows you to select columns programmatically or apply operations to subsets of columns.

5.7.2.1 Selecting Columns by Name:

Suppose we want to extract the `name`, `vore`, and `sleep_total` from the `msleep` data:

```
msleep |>
  select(name, sleep_total, vore)
```

```
#> # A tibble: 83 x 3
#>    name                          sleep_total vore
#>    <chr>                               <dbl> <chr>
#>  1 Cheetah                              12.1 carni
#>  2 Owl monkey                           17   omni
#>  3 Mountain beaver                      14.4 herbi
#>  4 Greater short-tailed shrew           14.9 omni
#>  5 Cow                                   4   herbi
#>  6 Three-toed sloth                     14.4 herbi
#>  7 Northern fur seal                     8.7 carni
#>  8 Vesper mouse                          7   <NA>
#>  9 Dog                                  10.1 carni
#> 10 Roe deer                              3   herbi
#> # i 73 more rows
```

5.7.2.2 Selecting Columns by Position:

You can use the position of columns to select them, for example, the first three columns:

```
msleep |>
  select(1:3)
```

```
#> # A tibble: 83 x 3
#>    name                          genus       vore
#>    <chr>                         <chr>       <chr>
#>  1 Cheetah                       Acinonyx    carni
#>  2 Owl monkey                    Aotus       omni
#>  3 Mountain beaver               Aplodontia  herbi
#>  4 Greater short-tailed shrew    Blarina     omni
#>  5 Cow                           Bos         herbi
#>  6 Three-toed sloth              Bradypus    herbi
#>  7 Northern fur seal             Callorhinus carni
#>  8 Vesper mouse                  Calomys     <NA>
#>  9 Dog                           Canis       carni
```

```
#> 10 Roe deer                    Capreolus   herbi
#> # i 73 more rows
```

5.7.2.3 Selecting Columns Using Patterns:

1. **Columns Starting with a Prefix**: Select columns starting with "sleep":

```
msleep |>
  select(starts_with("sleep"))
```

```
#> # A tibble: 83 x 3
#>    sleep_total sleep_rem sleep_cycle
#>          <dbl>     <dbl>       <dbl>
#>  1        12.1      NA         NA
#>  2        17         1.8       NA
#>  3        14.4       2.4       NA
#>  4        14.9       2.3        0.133
#>  5         4         0.7        0.667
#>  6        14.4       2.2        0.767
#>  7         8.7       1.4        0.383
#>  8         7        NA         NA
#>  9        10.1       2.9        0.333
#> 10         3        NA         NA
#> # i 73 more rows
```

2. **Columns Ending with a Suffix**: Select columns ending with "wt":

```
msleep |>
  select(ends_with("wt"))
```

```
#> # A tibble: 83 x 2
#>     brainwt  bodywt
#>       <dbl>   <dbl>
#>  1 NA        50
#>  2  0.0155    0.48
#>  3 NA         1.35
#>  4  0.00029   0.019
#>  5  0.423   600
#>  6 NA         3.85
#>  7 NA        20.5
#>  8 NA         0.045
```

```
#>  9  0.07     14
#> 10  0.0982   14.8
#> # i 73 more rows
```

3. **Columns Containing a Substring**: Select columns that contain the word "con":

```
msleep |>
  select(contains("con"))
```

```
#> # A tibble: 83 x 1
#>    conservation
#>    <chr>
#>  1 lc
#>  2 <NA>
#>  3 nt
#>  4 lc
#>  5 domesticated
#>  6 <NA>
#>  7 vu
#>  8 <NA>
#>  9 domesticated
#> 10 lc
#> # i 73 more rows
```

5.7.2.4 Deselect Columns Using the Minus Sign:

If you want to keep all columns except `name` and `sleep_total`:

```
msleep |>
  select(-c(name, sleep_total))
```

```
#> # A tibble: 83 x 9
#>    genus   vore  order conservation sleep_rem sleep_cycle awake  brainwt  bodywt
#>    <chr>   <chr> <chr> <chr>            <dbl>       <dbl> <dbl>    <dbl>   <dbl>
#>  1 Acinon~ carni Carn~ lc                NA        NA      11.9 NA        50
#>  2 Aotus   omni  Prim~ <NA>              1.8       NA       7    0.0155    0.48
#>  3 Aplodo~ herbi Rode~ nt                2.4       NA       9.6 NA         1.35
#>  4 Blarina omni  Sori~ lc                2.3        0.133   9.1  0.00029   0.019
#>  5 Bos     herbi Arti~ domesticated      0.7        0.667  20    0.423   600
#>  6 Bradyp~ herbi Pilo~ <NA>              2.2        0.767   9.6 NA         3.85
#>  7 Callor~ carni Carn~ vu                1.4        0.383  15.3 NA        20.5
```

```
#>  8 Calomys <NA>  Rode~ <NA>              NA        NA      17   NA         0.045
#>  9 Canis   carni Carn~ domesticated       2.9       0.333  13.9  0.07     14
#> 10 Capreo~ herbi Arti~ lc                NA        NA      21    0.0982   14.8
#> # i 73 more rows
```

You can also use helper functions to deselect columns. For instance, drop all columns starting with "sleep"

```
msleep |>
  select(-starts_with("sleep"))
```

```
#> # A tibble: 83 x 8
#>    name                  genus vore  order conservation awake  brainwt  bodywt
#>    <chr>                 <chr> <chr> <chr> <chr>        <dbl>    <dbl>   <dbl>
#>  1 Cheetah               Acin~ carni Carn~ lc            11.9 NA         50
#>  2 Owl monkey            Aotus omni  Prim~ <NA>           7    0.0155     0.48
#>  3 Mountain beaver       Aplo~ herbi Rode~ nt             9.6 NA          1.35
#>  4 Greater short-tailed s~ Blar~ omni  Sori~ lc            9.1  0.00029    0.019
#>  5 Cow                   Bos   herbi Arti~ domesticated  20    0.423    600
#>  6 Three-toed sloth      Brad~ herbi Pilo~ <NA>           9.6 NA          3.85
#>  7 Northern fur seal     Call~ carni Carn~ vu            15.3 NA         20.5
#>  8 Vesper mouse          Calo~ <NA>  Rode~ <NA>          17   NA         0.045
#>  9 Dog                   Canis carni Carn~ domesticated  13.9  0.07      14
#> 10 Roe deer              Capr~ herbi Arti~ lc            21    0.0982    14.8
#> # i 73 more rows
```

5.7.2.5 Using `where()` with `select()`

The `where()` helper function enables you to dynamically select or modify columns in your data. When used with `select()`, it allows you to programmatically choose columns based on specific criteria. For example, to select all numeric columns, you can use `where(is.numeric)`:

```
msleep |>
  select(where(is.numeric))
```

```
#> # A tibble: 83 x 6
#>    sleep_total sleep_rem sleep_cycle awake  brainwt  bodywt
#>          <dbl>     <dbl>       <dbl> <dbl>    <dbl>   <dbl>
#>  1        12.1      NA         NA      11.9 NA         50
#>  2        17         1.8       NA       7    0.0155     0.48
```

```
#>  3        14.4       2.4      NA       9.6 NA         1.35
#>  4        14.9       2.3       0.133   9.1  0.00029   0.019
#>  5         4         0.7       0.667  20    0.423   600
#>  6        14.4       2.2       0.767   9.6 NA         3.85
#>  7         8.7       1.4       0.383  15.3 NA        20.5
#>  8         7        NA        NA      17   NA         0.045
#>  9        10.1       2.9       0.333  13.9  0.07     14
#> 10         3        NA        NA      21    0.0982   14.8
#> # i 73 more rows
```

5.7.3 `mutate()` – Creating or Modifying Columns

The `mutate()` function is like a magic wand for adding new variables or transforming existing ones. Use it whenever you need to derive new information from your dataset.

Key Points:

- You can add as many new columns as you need.
- Existing columns can be modified by overwriting them.

5.7.3.1 Creating a New Column:

Sleep is a vital physiological process, but its duration varies widely among mammals[4]. Understanding how sleep duration relates to body weight could reveal insights into the metabolic and ecological factors influencing sleep. For example:

- **Larger mammals** may have lower sleep-to-body-weight ratios due to their lower metabolic rates relative to body size.
- **Smaller mammals** might have higher sleep-to-body-weight ratios, potentially linked to their higher metabolic demands.

To explore this, we calculate the ratio of total sleep (`sleep_total`) to body weight (`bodywt`) for each species in the `msleep` dataset:

```
msleep |>
  select(name, vore, sleep_total, bodywt) |>
  mutate(sleep_to_weight = sleep_total / bodywt)
```

[4]Siegel, J. M. (2005). Clues to the functions of mammalian sleep. Nature, 437(7063), 1264-1271.

```
#> # A tibble: 83 x 5
#>    name                        vore  sleep_total  bodywt sleep_to_weight
#>    <chr>                       <chr>       <dbl>   <dbl>           <dbl>
#>  1 Cheetah                     carni        12.1  50              0.242
#>  2 Owl monkey                  omni         17     0.48         35.4
#>  3 Mountain beaver             herbi        14.4   1.35         10.7
#>  4 Greater short-tailed shrew omni          14.9   0.019       784.
#>  5 Cow                         herbi         4   600              0.00667
#>  6 Three-toed sloth            herbi        14.4   3.85          3.74
#>  7 Northern fur seal           carni         8.7  20.5           0.425
#>  8 Vesper mouse                <NA>          7     0.045       156.
#>  9 Dog                         carni        10.1  14              0.721
#> 10 Roe deer                    herbi         3    14.8           0.203
#> # i 73 more rows
```

> **Tip**
>
> This ratio provides a standardised measure to compare sleep duration across species with varying body sizes.

In the same `msleep` dataset, we suspect that any brain weight greater than 4 is likely an error. To address this, we:

1. Exclude these suspected outliers by replacing them with `NA`.
2. Retain valid brain weight values for analysis.

```
msleep |>
  select(name, brainwt) |>
  # Replace brainwt > 4 with NA
  mutate(brainwt_corrected = ifelse(brainwt > 4, NA, brainwt)) |>
  # Sort by original brain weight in descending order
  arrange(desc(brainwt))
```

```
#> # A tibble: 83 x 3
#>    name             brainwt brainwt_corrected
#>    <chr>              <dbl>             <dbl>
#>  1 African elephant   5.71             NA
#>  2 Asian elephant     4.60             NA
#>  3 Human              1.32              1.32
#>  4 Horse              0.655             0.655
#>  5 Chimpanzee         0.44              0.44
```

```
#>  6 Cow                0.423            0.423
#>  7 Donkey             0.419            0.419
#>  8 Gray seal          0.325            0.325
#>  9 Baboon             0.18             0.18
#> 10 Pig                0.18             0.18
#> # i 73 more rows
```

> **Note**
>
> 1. **`select(name, brainwt)`**: Focuses on the columns relevant to this analysis: the species' name and their brain weight.
> 2. **`mutate(brainwt_corrected = ifelse(brainwt > 4, NA, brainwt))`**:
>   - Uses `ifelse()` to create a new column, `brainwt_corrected`, where any `brainwt` above 4 is replaced with `NA`.
>   - Values 4 or below remain unchanged.
> 3. **`arrange(desc(brainwt))`**:
>   - Sorts the data by the original `brainwt` in descending order, making it easier to verify the replaced outliers. *(You will soon learn more about the arrange() function.)*

To better understand patterns in sleep behaviour, it can be helpful to categorize species into discrete groups based on their sleep duration. This makes it easier to group species for further analysis, or explore ecological or evolutionary hypotheses.

For example, we can categorize species as:

- **"Long sleepers"**: Those that sleep more than 9 hours per day.
- **"Short sleepers"**: Those that sleep 9 hours or less per day.

We achieve this categorization using the `ifelse()` function, which allows us to transform the continuous numeric variable `sleep_total` into a categorical variable, `sleep_category`.

```
msleep |>
  select(name, vore, sleep_total) |>
  mutate(sleep_category = ifelse(sleep_total > 9, "long", "short"))
```

```
#> # A tibble: 83 x 4
#>    name                      vore  sleep_total sleep_category
#>    <chr>                     <chr>       <dbl> <chr>
```

```
#>  1 Cheetah                     carni        12.1 long
#>  2 Owl monkey                  omni         17   long
#>  3 Mountain beaver             herbi        14.4 long
#>  4 Greater short-tailed shrew omni         14.9 long
#>  5 Cow                         herbi         4   short
#>  6 Three-toed sloth            herbi        14.4 long
#>  7 Northern fur seal           carni         8.7 short
#>  8 Vesper mouse                <NA>          7   short
#>  9 Dog                         carni        10.1 long
#> 10 Roe deer                    herbi         3   short
#> # i 73 more rows
```

> **Tip**
>
> The `ifelse()` function is a handy tool for converting a numeric column into a categorical (or discrete) one. As explained earlier, `ifelse()` works by taking three arguments: a logical condition, a value to return if the condition is `TRUE`, and a value to return if the condition is `FALSE`. This makes it an efficient way to create new variables or modify existing ones based on specific criteria.

This categorization opens the door to exploring broader scientific questions:

1. **What ecological or metabolic factors correlate with sleep duration?**
    - For example, are "long sleepers" more likely to be predators, herbivores, or omnivores?
2. **Do body size or brain size influence sleep duration?**
    - Smaller animals may tend to sleep more to conserve energy, while larger animals might sleep less due to lower relative metabolic rates.
3. **Are long sleepers more prevalent in certain habitats?**
    - Does living in safer environments allow for extended sleep?

5.7.3.2 Using `mutate()` and `case_when()`:

Body weight is often an important ecological indicator, and mammals can be classified into the following categories based on their weight:

- **Heavy**: Body weight $>$ 50 kg
- **Medium**: Body weight $>$ 10 kg but 50 kg

- **Light**: Body weight 10 kg

When assigning these categories, the `case_when()` function provides a more elegant and readable solution compared to using nested `ifelse()` statements. It allows you to handle multiple conditions cleanly and intuitively. Here's how it can be used:

```
msleep |>
  select(name, sleep_total, bodywt) |>
  mutate(
    bodywt_category = case_when(
      bodywt > 50 ~ "heavy",
      bodywt > 10 ~ "medium",
      TRUE ~ "light" # Default for remaining cases
    )
  )
```

```
#> # A tibble: 83 x 4
#>    name                        sleep_total  bodywt bodywt_category
#>    <chr>                             <dbl>   <dbl> <chr>
#>  1 Cheetah                            12.1  50     medium
#>  2 Owl monkey                         17     0.48  light
#>  3 Mountain beaver                    14.4   1.35  light
#>  4 Greater short-tailed shrew         14.9   0.019 light
#>  5 Cow                                 4   600     heavy
#>  6 Three-toed sloth                   14.4   3.85  light
#>  7 Northern fur seal                   8.7  20.5   medium
#>  8 Vesper mouse                        7     0.045 light
#>  9 Dog                                10.1  14     medium
#> 10 Roe deer                           3    14.8   medium
#> # i 73 more rows
```

We can combine both categorizations into a single dataset to examine potential relationships between sleep behaviour and body weight. For example, we could ask:

- **Are "light" mammals more likely to be "short sleepers"?**

To create ordered factors for better control in plots or analyses, we use `factor()` or the `forcats` package:

```
msleep |>
  select(name, sleep_total, bodywt) |>
  mutate(
```

```
    sleep_category = ifelse(sleep_total > 9, "long", "short"),
    bodywt_discr = case_when(
      bodywt > 50 ~ "heavy",
      bodywt > 10 ~ "medium",
      TRUE ~ "light"
    ),
    # Convert to ordered factors
    sleep_category = factor(sleep_category, levels = c("short", "long")),
    bodywt_discr = factor(bodywt_discr, levels = c("light", "medium", "heavy"))
  )
```

```
#> # A tibble: 83 x 5
#>    name                        sleep_total  bodywt sleep_category bodywt_discr
#>    <chr>                             <dbl>   <dbl> <fct>          <fct>
#>  1 Cheetah                            12.1  50     long           medium
#>  2 Owl monkey                         17     0.48  long           light
#>  3 Mountain beaver                    14.4   1.35  long           light
#>  4 Greater short-tailed shrew         14.9   0.019 long           light
#>  5 Cow                                 4   600     short          heavy
#>  6 Three-toed sloth                   14.4   3.85  long           light
#>  7 Northern fur seal                   8.7  20.5   short          medium
#>  8 Vesper mouse                        7     0.045 short          light
#>  9 Dog                                10.1  14     long           medium
#> 10 Roe deer                           3    14.8   short          medium
#> # i 73 more rows
```

5.7.3.3 Calculating Row-Wise Averages Using `mutate()`

When analysing mammalian sleep data, researchers may want to compare different sleep metrics for each species. For instance:

- REM sleep (`sleep_rem`) and sleep cycle duration (`sleep_cycle`) could be combined to calculate a single representative metric, such as the average of the two values.
- This average can provide a holistic view of each species' sleep patterns.

However, standard aggregation functions like `mean()` or `sum()` operate on entire columns, summarising all observations at once rather than computing values row by row.

To calculate the average of `sleep_rem` and `sleep_cycle` for each species, you can use one of the following approaches:

1. **Explicit Arithmetic**: `(sleep_rem + sleep_cycle) / 2`

```
msleep |>
  select(name, sleep_rem, sleep_cycle) |>
  mutate(avg_sleep = (sleep_rem + sleep_cycle) / 2)
```

```
#> # A tibble: 83 x 4
#>    name                       sleep_rem sleep_cycle avg_sleep
#>    <chr>                          <dbl>       <dbl>     <dbl>
#>  1 Cheetah                         NA         NA        NA
#>  2 Owl monkey                       1.8       NA        NA
#>  3 Mountain beaver                  2.4       NA        NA
#>  4 Greater short-tailed shrew       2.3        0.133     1.22
#>  5 Cow                              0.7        0.667     0.683
#>  6 Three-toed sloth                 2.2        0.767     1.48
#>  7 Northern fur seal                1.4        0.383     0.892
#>  8 Vesper mouse                    NA         NA        NA
#>  9 Dog                              2.9        0.333     1.62
#> 10 Roe deer                        NA         NA        NA
#> # i 73 more rows
```

2. **Using** `rowwise()` and `mutate()`:

```
msleep |>
  select(name, sleep_rem, sleep_cycle) |>
  rowwise() |> # Enable row-wise operations
  mutate(avg_sleep = mean(c(sleep_rem, sleep_cycle), na.rm = TRUE)) |>
  ungroup()
```

```
#> # A tibble: 83 x 4
#>    name                       sleep_rem sleep_cycle avg_sleep
#>    <chr>                          <dbl>       <dbl>     <dbl>
#>  1 Cheetah                         NA         NA       NaN
#>  2 Owl monkey                       1.8       NA         1.8
#>  3 Mountain beaver                  2.4       NA         2.4
#>  4 Greater short-tailed shrew       2.3        0.133     1.22
#>  5 Cow                              0.7        0.667     0.683
#>  6 Three-toed sloth                 2.2        0.767     1.48
#>  7 Northern fur seal                1.4        0.383     0.892
#>  8 Vesper mouse                    NA         NA       NaN
#>  9 Dog                              2.9        0.333     1.62
#> 10 Roe deer                        NA         NA       NaN
#> # i 73 more rows
```

When you use `rowwise()`, it is important to call `ungroup()` afterwards to remove row-wise grouping and revert to the default, ungrouped state of the data.

5.7.3.4 Using `mutate()` and `across()` to Modify Specific Columns

When working with datasets, it is common to encounter situations where you need to apply the same transformation to multiple columns. For instance, you might want to convert units, round numeric values, or standardise text. In such cases, the combination of `mutate()` and `across()` from the `dplyr` package is a powerful tool. These functions allow you to efficiently select and modify subsets of columns, reducing the need for repetitive code.

The `across()` function is particularly versatile. It enables you to:

1. **Select columns** using tidy selection helpers like `starts_with()`, `contains()`, or `where()`.
2. **Apply a function** to each of the selected columns, making it easy to perform bulk operations.

We will explore several practical examples using the `msleep` dataset, which contains information about the sleep patterns of mammals. These examples will demonstrate how to use `mutate()` and `across()` to perform common data transformation tasks.

Example 1: Converting Units for Sleep-Related Columns

The `msleep` dataset includes several columns related to sleep duration, measured in **hours**. Suppose you need these values in **minutes** instead. Rather than manually transforming each column, you can use `across()` to apply the conversion to all relevant columns at once [5].

```
msleep |>
  select(name, contains("sleep")) |> # Focus on sleep-related columns
  mutate(across(contains("sleep"), ~ .x * 60)) # Multiply each value by 60
```

```
#> # A tibble: 83 x 4
#>    name                          sleep_total sleep_rem sleep_cycle
#>    <chr>                               <dbl>     <dbl>       <dbl>
#>  1 Cheetah                               726        NA       NA
#>  2 Owl monkey                           1020       108       NA
#>  3 Mountain beaver                       864       144       NA
#>  4 Greater short-tailed shrew            894       138        8.00
#>  5 Cow                                   240        42       40.0
```

[5] Using the same example, the syntax for `across()` with column positions would be as follows: `msleep |> select(name, contains("sleep")) |> mutate(across(c(2, 3, 4), ~ .x * 60))`

```
#>  6 Three-toed sloth                  864       132       46.0
#>  7 Northern fur seal                 522        84       23.0
#>  8 Vesper mouse                      420        NA       NA
#>  9 Dog                               606       174       20.0
#> 10 Roe deer                          180        NA       NA
#> # i 73 more rows
```

> **Note**
>
> 1. **`select(name, contains("sleep"))`**:
>
>   - Keeps the **`name`** column (for species identification) and all columns whose names contain the word "sleep".
>
> 2. **`mutate(across(contains("sleep"), ~ .x * 60))`**:
>
>   - `across(contains("sleep"))`: Selects all columns containing "sleep" in their names.
>   - The formula ~ .x * 60 converts hours to minutes by multiplying each value by 60.
>   - Uses ~ to define an anonymous function and .x to refer to column values.

This approach is not only concise but also scalable. If new sleep-related columns are added to the dataset, the same code will automatically include them in the transformation.

Example 2: Rounding Numeric Columns

Another common task is rounding numeric values to a specified number of decimal places. For example, you might want to round all numeric columns in the **`msleep`** dataset to the nearest integer.

```
msleep |>
  mutate(across(where(is.numeric), round))
```

```
#> # A tibble: 83 x 11
#>    name   genus vore  order conservation sleep_total sleep_rem sleep_cycle awake
#>    <chr>  <chr> <chr> <chr> <chr>              <dbl>     <dbl>       <dbl> <dbl>
#>  1 Cheet~ Acin~ carni Carn~ lc                    12        NA          NA    12
#>  2 Owl m~ Aotus omni  Prim~ <NA>                  17         2          NA     7
#>  3 Mount~ Aplo~ herbi Rode~ nt                    14         2          NA    10
#>  4 Great~ Blar~ omni  Sori~ lc                    15         2           0     9
#>  5 Cow    Bos   herbi Arti~ domesticated           4         1           1    20
#>  6 Three~ Brad~ herbi Pilo~ <NA>                  14         2           1    10
```

```
#>  7 North~ Call~ carni Carn~ vu                    9         1          0    15
#>  8 Vespe~ Calo~ <NA>  Rode~ <NA>                  7        NA         NA    17
#>  9 Dog    Canis carni Carn~ domesticated         10         3          0    14
#> 10 Roe d~ Capr~ herbi Arti~ lc                    3        NA         NA    21
#> # i 73 more rows
#> # i 2 more variables: brainwt <dbl>, bodywt <dbl>
```

> **Note**
>
> **`mutate(across(where(is.numeric), round))`**:
>
> - `where(is.numeric)` selects all numeric columns.
> - The `round` function is applied to each numeric column, rounding the values to the nearest integer.

This is particularly useful when preparing data for reporting or visualisation, where overly precise values can clutter the output.

Example 3: Scaling Numeric Columns to a Range of 0 to 1

Normalising data is a common preprocessing step in data analysis, particularly when features have different scales. For instance, scaling numeric columns to a range of 0 to 1 ensures comparability across variables. This is achieved using **min-max scaling**, defined by the formula:

$$x_{\text{scaled}} = \frac{x - \min(x)}{\max(x) - \min(x)}$$

> **Tip**
>
> Where:
>
> - $\min(x)$ is the minimum value in the column.
> - $\max(x)$ is the maximum value in the column.
> - x_{scaled} is the normalised value, constrained to the interval $[0, 1]$.

To streamline this process, we encapsulate the formula into a reusable function:

```r
# Define function for min-max scaling (handles missing values)
min_max_scale <- function(x) {
  (x - min(x, na.rm = TRUE)) /
    (max(x, na.rm = TRUE) - min(x, na.rm = TRUE))
}
```

> **Note**
>
> This function:
>
> - Ignores missing values (`na.rm = TRUE`) to avoid `NA` propagation.
> - Automatically adapts to each column's unique range.

Apply the function to all numeric columns in the `msleep` dataset:

```r
msleep |>
  mutate(across(where(is.numeric), min_max_scale))
```

```
#> # A tibble: 83 x 11
#>    name   genus vore  order conservation sleep_total sleep_rem sleep_cycle awake
#>    <chr>  <chr> <chr> <chr> <chr>              <dbl>     <dbl>       <dbl> <dbl>
#>  1 Cheet~ Acin~ carni Carn~ lc                 0.567    NA          NA      0.433
#>  2 Owl m~ Aotus omni  Prim~ <NA>               0.839     0.262      NA      0.161
#>  3 Mount~ Aplo~ herbi Rode~ nt                 0.694     0.354      NA      0.306
#>  4 Great~ Blar~ omni  Sori~ lc                 0.722     0.338       0.0120 0.278
#>  5 Cow    Bos   herbi Arti~ domesticated       0.117     0.0923      0.398  0.883
#>  6 Three~ Brad~ herbi Pilo~ <NA>               0.694     0.323       0.470  0.306
#>  7 North~ Call~ carni Carn~ vu                 0.378     0.2         0.193  0.622
#>  8 Vespe~ Calo~ <NA>  Rode~ <NA>               0.283    NA          NA      0.717
#>  9 Dog    Canis carni Carn~ domesticated       0.456     0.431       0.157  0.544
#> 10 Roe d~ Capr~ herbi Arti~ lc                 0.0611   NA          NA      0.939
#> # i 73 more rows
#> # i 2 more variables: brainwt <dbl>, bodywt <dbl>
```

> **Key Considerations**
>
> - **Zero-variance columns**: If all values in a column are identical, `max(x) - min(x) = 0`, resulting in `NaN` (division by zero).
> - **Missing values**: Columns with only `NA` values will return `NaN`.

- **Interpretability**: Scaled values retain the relative relationships in the original data.

This transformation ensures all numeric values are proportionally mapped to the same range, simplifying comparisons and improving the performance of machine learning algorithms.

Example 4: Transforming Character Columns

Text data often contains extra whitespace—leading, trailing, or even multiple spaces between words. This can lead to inconsistencies in your analysis. The `str_squish()` function from the `stringr` package cleans such text by removing extraneous whitespace.

In this example, you will apply `str_squish()` to all character columns in the `msleep` dataset:

```
msleep |>
  mutate(across(where(is.character), str_squish))
```

```
#> # A tibble: 83 x 11
#>    name   genus vore  order conservation sleep_total sleep_rem sleep_cycle awake
#>    <chr>  <chr> <chr> <chr> <chr>              <dbl>     <dbl>       <dbl> <dbl>
#>  1 Cheet~ Acin~ carni Carn~ lc                  12.1      NA        NA      11.9
#>  2 Owl m~ Aotus omni  Prim~ <NA>                17         1.8      NA       7
#>  3 Mount~ Aplo~ herbi Rode~ nt                  14.4       2.4      NA       9.6
#>  4 Great~ Blar~ omni  Sori~ lc                  14.9       2.3       0.133   9.1
#>  5 Cow    Bos   herbi Arti~ domesticated         4         0.7       0.667  20
#>  6 Three~ Brad~ herbi Pilo~ <NA>                14.4       2.2       0.767   9.6
#>  7 North~ Call~ carni Carn~ vu                   8.7       1.4       0.383  15.3
#>  8 Vespe~ Calo~ <NA>  Rode~ <NA>                 7        NA        NA      17
#>  9 Dog    Canis carni Carn~ domesticated        10.1       2.9       0.333  13.9
#> 10 Roe d~ Capr~ herbi Arti~ lc                   3        NA        NA      21
#> # i 73 more rows
#> # i 2 more variables: brainwt <dbl>, bodywt <dbl>
```

Note

- **`where(is.character)`**: Selects all character columns.
- **`str_squish`**: Removes extra whitespace from the text in these columns.

For example, if a column contains the value `"  African   elephant  "`, it will be cleaned up to `"African elephant"`.

5.7.4 `filter()` – Selecting Rows Based on Conditions

The `filter()` function allows you to keep rows that meet specific criteria. Think of it as a way to zoom in on the part of your dataset that matters most.

Key Points:

- `filter()` takes a logical condition (e.g., `gender == "female"`) and keeps rows where the condition is `TRUE`.
- You can use `filter()` to subset numeric variables based on their values. Common comparison operators include:
  - `>` (greater than)
  - `>=` (greater than or equal to)
  - `<` (less than)
  - `<=` (less than or equal to)
  - `==` (equal to)
  - `!=` (not equal to)

  For details on these operators, see Chapter 1.6.4.
- To filter rows based on multiple conditions, you can use logical operators:
  - `&` or `,` (AND): Ensures all conditions are true.
  - `|` (OR): Keeps rows where at least one condition is true.
  - `!` (NOT): Negates a condition, keeping rows where it is false.

5.7.4.1 Basic row filters

Let's find all animals that sleep more than 10 hours:

```
msleep |>
  select(name, sleep_total) |>
  filter(sleep_total > 10)
```

```
#> # A tibble: 44 x 2
#>    name                        sleep_total
#>    <chr>                             <dbl>
#>  1 Cheetah                            12.1
#>  2 Owl monkey                         17
#>  3 Mountain beaver                    14.4
#>  4 Greater short-tailed shrew         14.9
#>  5 Three-toed sloth                   14.4
#>  6 Dog                                10.1
#>  7 Chinchilla                         12.5
#>  8 Star-nosed mole                    10.3
#>  9 Long-nosed armadillo               17.4
#> 10 North American Opossum             18
#> # i 34 more rows
```

To select a range of values, you can use two logical conditions. For example, to filter all animals with a total sleep time between 9 and 16 hours, you could use:

```
msleep |>
  select(name, sleep_total) |>
  filter(sleep_total >= 9, sleep_total <= 16)
```

```
#> # A tibble: 46 x 2
#>    name                        sleep_total
#>    <chr>                             <dbl>
#>  1 Cheetah                            12.1
#>  2 Mountain beaver                    14.4
#>  3 Greater short-tailed shrew         14.9
#>  4 Three-toed sloth                   14.4
#>  5 Dog                                10.1
#>  6 Guinea pig                          9.4
#>  7 Grivet                             10
#>  8 Chinchilla                         12.5
#>  9 Star-nosed mole                    10.3
#> 10 Lesser short-tailed shrew          9.1
#> # i 36 more rows
```

However, there is a more concise way to achieve the same result using the `between()` function[6]:

[6]The `between()` function simplifies the code and improves readability by combining the range condition into a single statement.

```
msleep |>
  select(name, sleep_total) |>
  filter(between(sleep_total, 9, 16))
```

```
#> # A tibble: 46 x 2
#>    name                       sleep_total
#>    <chr>                            <dbl>
#>  1 Cheetah                           12.1
#>  2 Mountain beaver                   14.4
#>  3 Greater short-tailed shrew        14.9
#>  4 Three-toed sloth                  14.4
#>  5 Dog                               10.1
#>  6 Guinea pig                         9.4
#>  7 Grivet                            10
#>  8 Chinchilla                        12.5
#>  9 Star-nosed mole                   10.3
#> 10 Lesser short-tailed shrew          9.1
#> # i 36 more rows
```

5.7.4.2 Filtering Based on Exact Character Variable Matches

When working with character variables, you can filter rows based on exact matches, exclusions, or membership in specific groups. Below are practical examples of how to handle these scenarios effectively:

1. Select Rows with an Exact Match

To filter a specific group of animals, use the `==` comparison operator. For example, to select only carnivores:

```
msleep |>
  select(name, vore, sleep_total) |>
  filter(vore == "carni")
```

```
#> # A tibble: 19 x 3
#>    name                       vore  sleep_total
#>    <chr>                      <chr>       <dbl>
#>  1 Cheetah                    carni        12.1
#>  2 Northern fur seal          carni         8.7
#>  3 Dog                        carni        10.1
#>  4 Long-nosed armadillo       carni        17.4
```

```
#>  5 Domestic cat               carni        12.5
#>  6 Pilot whale                carni         2.7
#>  7 Gray seal                  carni         6.2
#>  8 Thick-tailed opposum       carni        19.4
#>  9 Slow loris                 carni        11
#> 10 Northern grasshopper mouse carni        14.5
#> 11 Tiger                     carni        15.8
#> 12 Jaguar                    carni        10.4
#> 13 Lion                      carni        13.5
#> 14 Caspian seal              carni         3.5
#> 15 Common porpoise           carni         5.6
#> 16 Bottle-nosed dolphin      carni         5.2
#> 17 Genet                     carni         6.3
#> 18 Arctic fox                carni        12.5
#> 19 Red fox                   carni         9.8
```

2. Exclude Rows Using `!=`

To exclude rows with a specific value, use the `!=` operator. For example, to exclude omnivores:

```
msleep |>
  select(name, vore, sleep_total) |>
  filter(vore != "omni")
```

```
#> # A tibble: 56 x 3
#>    name              vore  sleep_total
#>    <chr>             <chr>       <dbl>
#>  1 Cheetah           carni        12.1
#>  2 Mountain beaver   herbi        14.4
#>  3 Cow               herbi         4
#>  4 Three-toed sloth  herbi        14.4
#>  5 Northern fur seal carni         8.7
#>  6 Dog               carni        10.1
#>  7 Roe deer          herbi         3
#>  8 Goat              herbi         5.3
#>  9 Guinea pig        herbi         9.4
#> 10 Chinchilla        herbi        12.5
#> # i 46 more rows
```

3. Filter for Multiple Values Using `%in%`

If you want to filter rows where a variable matches one of multiple values, use the `%in%` operator. For example, to select animals belonging to the orders *Primates* or *Rodentia*:

```
msleep |>
  select(name, sleep_total, order) |>
  filter(order %in% c("Primates", "Rodentia"))
```

```
#> # A tibble: 34 x 3
#>    name                       sleep_total order
#>    <chr>                            <dbl> <chr>
#>  1 Owl monkey                        17   Primates
#>  2 Mountain beaver                   14.4 Rodentia
#>  3 Vesper mouse                       7   Rodentia
#>  4 Guinea pig                         9.4 Rodentia
#>  5 Grivet                            10   Primates
#>  6 Chinchilla                        12.5 Rodentia
#>  7 African giant pouched rat          8.3 Rodentia
#>  8 Patas monkey                      10.9 Primates
#>  9 Western american chipmunk         14.9 Rodentia
#> 10 Galago                             9.8 Primates
#> # i 24 more rows
```

4. Exclude Multiple Groups Using `!%in%`

To exclude rows belonging to specific groups, negate the `%in%` operator by using `!` at the beginning of your filter. For example, to exclude animals in the orders *Rodentia*, *Carnivora*, and *Primates*:

```
msleep |>
  select(name, order, sleep_total) |>
  filter(!order %in% c("Rodentia", "Carnivora", "Primates"))
```

```
#> # A tibble: 37 x 3
#>    name                       order           sleep_total
#>    <chr>                      <chr>                 <dbl>
#>  1 Greater short-tailed shrew Soricomorpha           14.9
#>  2 Cow                        Artiodactyla            4
#>  3 Three-toed sloth           Pilosa                 14.4
#>  4 Roe deer                   Artiodactyla            3
#>  5 Goat                       Artiodactyla            5.3
#>  6 Star-nosed mole            Soricomorpha           10.3
#>  7 Lesser short-tailed shrew  Soricomorpha            9.1
```

```
#>  8 Long-nosed armadillo       Cingulata              17.4
#>  9 Tree hyrax                 Hyracoidea              5.3
#> 10 North American Opossum     Didelphimorphia        18
#> # i 27 more rows
```

> **Tip**
>
> In R, you must place the negation operator (`!`) before the variable you want to filter with when using the `%in%` operator. For example, instead of writing:
>
> ```
> filter(order !%in% c("Rodentia", "Carnivora", "Primates"))
> ```
>
> —which is invalid—write it as:
>
> ```
> filter(!order %in% c("Rodentia", "Carnivora", "Primates"))
> ```
>
> This ensures that the filter correctly excludes the specified groups.

5.7.4.3 Filtering Rows Based on Regular Expressions

The filtering methods discussed earlier work well when you are matching the entire content of a variable. However, in many cases, you may need to filter rows based on **partial matches** within a string. To achieve this, you can use functions that evaluate regular expressions and return Boolean values (`TRUE` or `FALSE`). Rows where the condition is `TRUE` will be retained.

There are two ways to do this:

1. **`grepl()`** (from base R):
    - Checks if a pattern exists in a string and returns a logical vector.
    - Often combined with `tolower()` to make matching case-insensitive.
2. **`str_detect()`** (from the `stringr` package):
    - A more intuitive function for detecting patterns in strings.
    - Often combined with `str_to_lower()` to make matching case-insensitive.
    - Part of the `tidyverse`, making it consistent with `dplyr` workflows.

> **Note**
>
> R is case-sensitive by default. For instance:
>
> - Using `filter(str_detect(name, pattern = "mouse"))` would exclude rows with `"Mouse"` because of the difference in case.
>
> To avoid missing such matches, it's a good practice to convert the text to lowercase (or uppercase) using `str_to_lower()` (or `str_to_upper()`) before performing the match.

In the following example, we filter rows where the `name` column contains the substring "mouse," regardless of case:

```
msleep |>
  select(name, sleep_total) |>
  filter(str_detect(str_to_lower(name), pattern = "mouse"))
```

```
#> # A tibble: 5 x 2
#>   name                         sleep_total
#>   <chr>                              <dbl>
#> 1 Vesper mouse                         7
#> 2 House mouse                         12.5
#> 3 Northern grasshopper mouse          14.5
#> 4 Deer mouse                          11.5
#> 5 African striped mouse                8.7
```

5.7.4.4 Filtering with Multiple Conditions

The `filter()` function not only allows filtering based on single conditions but also supports combining multiple conditions using logical operators. These operators—**`AND`**, **`OR`**, **`NOT`**, and **`XOR`**—as summarized in Table 5.1, provide the flexibility to create complex filtering logic tailored to your data analysis needs.

Table 5.1: Logical Operators for Filtering Data in R

Example	Operator	Description	Example Usage
1	`,` or `&` (AND)	Both conditions must be true for a row to be returned.	`filter(condition1, condition2)` or `filter(condition1 & condition2)`

Example	Operator	Description	Example Usage
2	`|` (OR)	At least one condition must be true for a row to be included.	`filter(condition1 |condition2)`
3	`!` (NOT)	The condition must be false for a row to be included.	`filter(!condition1)`
4	`xor()`	Only one condition must be true, and not both.	`filter(xor(condition1, condition2))`

Example 1:

Filter Animals That Are Carnivores and Sleep More Than 10 Hours

```
msleep |>
  filter(vore == "carni", sleep_total > 10)
```

```
#> # A tibble: 11 x 11
#>    name   genus vore  order conservation sleep_total sleep_rem sleep_cycle awake
#>    <chr>  <chr> <chr> <chr> <chr>              <dbl>     <dbl>       <dbl> <dbl>
#>  1 Cheet~ Acin~ carni Carn~ lc                  12.1      NA        NA      11.9
#>  2 Dog    Canis carni Carn~ domesticated        10.1       2.9       0.333  13.9
#>  3 Long-~ Dasy~ carni Cing~ lc                  17.4       3.1       0.383   6.6
#>  4 Domes~ Felis carni Carn~ domesticated        12.5       3.2       0.417  11.5
#>  5 Thick~ Lutr~ carni Dide~ lc                  19.4       6.6      NA       4.6
#>  6 Slow ~ Nyct~ carni Prim~ <NA>                11         NA        NA      13
#>  7 North~ Onyc~ carni Rode~ lc                  14.5      NA        NA       9.5
#>  8 Tiger  Pant~ carni Carn~ en                  15.8      NA        NA       8.2
#>  9 Jaguar Pant~ carni Carn~ nt                  10.4      NA        NA      13.6
#> 10 Lion   Pant~ carni Carn~ vu                  13.5      NA        NA      10.5
#> 11 Arcti~ Vulp~ carni Carn~ <NA>                12.5      NA        NA      11.5
#> # i 2 more variables: brainwt <dbl>, bodywt <dbl>
```

Example 2:

Filter Animals That Are Omnivores or Sleep Less Than 8 Hours

```
msleep |>
  filter(vore == "omni" | sleep_total < 8)
```

```
#> # A tibble: 41 x 11
#>    name   genus vore  order conservation sleep_total sleep_rem sleep_cycle awake
#>    <chr>  <chr> <chr> <chr> <chr>              <dbl>     <dbl>       <dbl> <dbl>
#>  1 Owl m~ Aotus omni  Prim~ <NA>                17         1.8      NA       7
#>  2 Great~ Blar~ omni  Sori~ lc                  14.9       2.3       0.133   9.1
#>  3 Cow    Bos   herbi Arti~ domesticated         4         0.7       0.667  20
#>  4 Vespe~ Calo~ <NA>  Rode~ <NA>                 7        NA        NA      17
#>  5 Roe d~ Capr~ herbi Arti~ lc                   3        NA        NA      21
#>  6 Goat   Capri herbi Arti~ lc                   5.3       0.6      NA      18.7
#>  7 Grivet Cerc~ omni  Prim~ lc                  10         0.7      NA      14
#>  8 Star-~ Cond~ omni  Sori~ lc                  10.3       2.2      NA      13.7
#>  9 Afric~ Cric~ omni  Rode~ <NA>                 8.3       2        NA      15.7
#> 10 Lesse~ Cryp~ omni  Sori~ lc                   9.1       1.4       0.15   14.9
#> # i 31 more rows
#> # i 2 more variables: brainwt <dbl>, bodywt <dbl>
```

Example 3:

Filter Animals That Are Not Omnivores and Sleep More Than 8 Hours

```
msleep |>
  filter(!vore == "herbi", sleep_total > 8)
```

```
#> # A tibble: 37 x 11
#>    name   genus vore  order conservation sleep_total sleep_rem sleep_cycle awake
#>    <chr>  <chr> <chr> <chr> <chr>              <dbl>     <dbl>       <dbl> <dbl>
#>  1 Cheet~ Acin~ carni Carn~ lc                  12.1      NA        NA      11.9
#>  2 Owl m~ Aotus omni  Prim~ <NA>                17         1.8      NA       7
#>  3 Great~ Blar~ omni  Sori~ lc                  14.9       2.3       0.133   9.1
#>  4 North~ Call~ carni Carn~ vu                   8.7       1.4       0.383  15.3
#>  5 Dog    Canis carni Carn~ domesticated        10.1       2.9       0.333  13.9
#>  6 Grivet Cerc~ omni  Prim~ lc                  10         0.7      NA      14
#>  7 Star-~ Cond~ omni  Sori~ lc                  10.3       2.2      NA      13.7
#>  8 Afric~ Cric~ omni  Rode~ <NA>                 8.3       2        NA      15.7
#>  9 Lesse~ Cryp~ omni  Sori~ lc                   9.1       1.4       0.15   14.9
#> 10 Long-~ Dasy~ carni Cing~ lc                  17.4       3.1       0.383   6.6
#> # i 27 more rows
#> # i 2 more variables: brainwt <dbl>, bodywt <dbl>
```

Example 4:

Filter Animals That Are Either Carnivores or Sleep More Than 16 Hours, But Not Both

```r
msleep |>
  filter(xor(vore == "carni", sleep_total > 16))
```

```
#> # A tibble: 23 x 11
#>    name   genus vore  order conservation sleep_total sleep_rem sleep_cycle awake
#>    <chr>  <chr> <chr> <chr> <chr>              <dbl>     <dbl>       <dbl> <dbl>
#>  1 Cheet~ Acin~ carni Carn~ lc                  12.1      NA         NA      11.9
#>  2 Owl m~ Aotus omni  Prim~ <NA>                17         1.8       NA       7
#>  3 North~ Call~ carni Carn~ vu                   8.7       1.4        0.383  15.3
#>  4 Dog    Canis carni Carn~ domesticated        10.1       2.9        0.333  13.9
#>  5 North~ Dide~ omni  Dide~ lc                  18         4.9        0.333   6
#>  6 Big b~ Epte~ inse~ Chir~ lc                  19.7       3.9        0.117   4.3
#>  7 Domes~ Felis carni Carn~ domesticated        12.5       3.2        0.417  11.5
#>  8 Pilot~ Glob~ carni Ceta~ cd                   2.7       0.1       NA      21.4
#>  9 Gray ~ Hali~ carni Carn~ lc                   6.2       1.5       NA      17.8
#> 10 Littl~ Myot~ inse~ Chir~ <NA>                19.9       2          0.2     4.1
#> # i 13 more rows
#> # i 2 more variables: brainwt <dbl>, bodywt <dbl>
```

> Key Tips
>
> - **Order Matters**: R evaluates conditions left to right in a `filter()` statement.
> - **Combine Freely**: You can mix `AND`, `OR`, and `NOT` to form highly specific filters.
> - **Readability**: Use parentheses for complex conditions to make them easier to understand.

For complex conditions combining `AND` (`&`), `OR` (`|`), and `NOT` (`!`), the order of evaluation is determined by standard precedence rules unless overridden with parentheses.

Precedence Order:

1. **NOT** (`!`) is evaluated first.
2. **AND** (`&` or `,`) is evaluated second.
3. **OR** (`|`) is evaluated last.

Example 5:

Filter animals that are either herbivores or omnivores and have a total sleep time of more than 10 hours.

```
msleep |>
  filter((vore == "herbi" | vore == "omni") & sleep_total > 10)
```

```
#> # A tibble: 25 x 11
#>    name   genus vore  order conservation sleep_total sleep_rem sleep_cycle awake
#>    <chr>  <chr> <chr> <chr> <chr>              <dbl>     <dbl>       <dbl> <dbl>
#>  1 Owl m~ Aotus omni  Prim~ <NA>               17         1.8      NA       7
#>  2 Mount~ Aplo~ herbi Rode~ nt                 14.4       2.4      NA       9.6
#>  3 Great~ Blar~ omni  Sori~ lc                 14.9       2.3       0.133   9.1
#>  4 Three~ Brad~ herbi Pilo~ <NA>               14.4       2.2       0.767   9.6
#>  5 Chinc~ Chin~ herbi Rode~ domesticated       12.5       1.5       0.117  11.5
#>  6 Star-~ Cond~ omni  Sori~ lc                 10.3       2.2      NA      13.7
#>  7 North~ Dide~ omni  Dide~ lc                 18         4.9       0.333   6
#>  8 Europ~ Erin~ omni  Erin~ lc                 10.1       3.5       0.283  13.9
#>  9 Patas~ Eryt~ omni  Prim~ lc                 10.9       1.1      NA      13.1
#> 10 Weste~ Euta~ herbi Rode~ <NA>               14.9      NA        NA       9.1
#> # i 15 more rows
#> # i 2 more variables: brainwt <dbl>, bodywt <dbl>
```

If the parentheses were omitted, the conditions could be misinterpreted, leading to unintended results.

5.7.4.5 Filtering Out Empty Rows

Missing values (`NA`) in datasets can disrupt analysis, so it is often necessary to remove rows with missing values in specific columns. This can be done using the `is.na()` function within a `filter()` statement.

1. Select Rows Where a Column Has `NA` Values

To display rows where the `conservation` column has missing values (`NA`):

```
msleep |>
  select(name, conservation:sleep_cycle) |>
  filter(is.na(conservation))
```

```
#> # A tibble: 29 x 5
#>    name                      conservation sleep_total sleep_rem sleep_cycle
#>    <chr>                     <chr>              <dbl>     <dbl>       <dbl>
#>  1 "Owl monkey"              <NA>                17         1.8      NA
#>  2 "Three-toed sloth"        <NA>                14.4       2.2       0.767
```

```
#>  3 "Vesper mouse"              <NA>                7        NA        NA
#>  4 "African giant pouched rat" <NA>                8.3       2        NA
#>  5 "Western american chipmunk" <NA>               14.9      NA        NA
#>  6 "Galago"                    <NA>                9.8       1.1       0.55
#>  7 "Human"                     <NA>                8         1.9       1.5
#>  8 "Macaque"                   <NA>               10.1       1.2       0.75
#>  9 "Vole "                     <NA>               12.8      NA        NA
#> 10 "Little brown bat"          <NA>               19.9       2         0.2
#> # i 19 more rows
```

2. Remove Rows Where a Column Has `NA` Values

To exclude rows where the `conservation` column is missing, negate the `is.na()` function using `!`:

```
msleep |>
  select(name, conservation:sleep_cycle) |>
  filter(!is.na(conservation))
```

```
#> # A tibble: 54 x 5
#>    name                       conservation sleep_total sleep_rem sleep_cycle
#>    <chr>                      <chr>              <dbl>     <dbl>       <dbl>
#>  1 Cheetah                    lc                  12.1      NA        NA
#>  2 Mountain beaver            nt                  14.4       2.4      NA
#>  3 Greater short-tailed shrew lc                  14.9       2.3       0.133
#>  4 Cow                        domesticated         4         0.7       0.667
#>  5 Northern fur seal          vu                   8.7       1.4       0.383
#>  6 Dog                        domesticated        10.1       2.9       0.333
#>  7 Roe deer                   lc                   3        NA        NA
#>  8 Goat                       lc                   5.3       0.6      NA
#>  9 Guinea pig                 domesticated         9.4       0.8       0.217
#> 10 Grivet                     lc                  10         0.7      NA
#> # i 44 more rows
```

3. Filtering Multiple Columns for Missing Values

To remove rows with missing values across multiple columns, you can combine multiple `is.na()` checks:

```
msleep |>
  select(name, conservation, sleep_total) |>
  filter(!is.na(conservation), !is.na(sleep_total))
```

```
#> # A tibble: 54 x 3
#>    name                       conservation sleep_total
#>    <chr>                      <chr>              <dbl>
#>  1 Cheetah                    lc                  12.1
#>  2 Mountain beaver            nt                  14.4
#>  3 Greater short-tailed shrew lc                  14.9
#>  4 Cow                        domesticated         4
#>  5 Northern fur seal          vu                   8.7
#>  6 Dog                        domesticated        10.1
#>  7 Roe deer                   lc                   3
#>  8 Goat                       lc                   5.3
#>  9 Guinea pig                 domesticated         9.4
#> 10 Grivet                     lc                  10
#> # i 44 more rows
```

4. Using `if_any()` or `if_all()` for Cleaner Filtering

The `if_any()` and `if_all()` functions from **dplyr** offer a more concise way to handle missing values across multiple columns:

- **a. Remove Rows with `NA` in Any Selected Column**

This keeps rows where at least one of the specified columns (`conservation` or `sleep_total`) is not `NA`:

```
msleep |>
  select(name, conservation, sleep_total) |>
  filter(if_any(c(conservation, sleep_total), ~ !is.na(.)))
```

```
#> # A tibble: 83 x 3
#>    name                       conservation sleep_total
#>    <chr>                      <chr>              <dbl>
#>  1 Cheetah                    lc                  12.1
#>  2 Owl monkey                 <NA>                17
#>  3 Mountain beaver            nt                  14.4
#>  4 Greater short-tailed shrew lc                  14.9
#>  5 Cow                        domesticated         4
#>  6 Three-toed sloth           <NA>                14.4
#>  7 Northern fur seal          vu                   8.7
#>  8 Vesper mouse               <NA>                 7
#>  9 Dog                        domesticated        10.1
#> 10 Roe deer                   lc                   3
#> # i 73 more rows
```

- **b. Remove Rows with `NA` in All Selected Columns**

This keeps rows where all the specified columns are not `NA`:

```
msleep |>
  select(name, conservation, sleep_total) |>
  filter(if_all(c(conservation, sleep_total), ~ !is.na(.)))
```

```
#> # A tibble: 54 x 3
#>    name                       conservation sleep_total
#>    <chr>                      <chr>              <dbl>
#>  1 Cheetah                    lc                  12.1
#>  2 Mountain beaver            nt                  14.4
#>  3 Greater short-tailed shrew lc                  14.9
#>  4 Cow                        domesticated         4
#>  5 Northern fur seal          vu                   8.7
#>  6 Dog                        domesticated        10.1
#>  7 Roe deer                   lc                   3
#>  8 Goat                       lc                   5.3
#>  9 Guinea pig                 domesticated         9.4
#> 10 Grivet                     lc                  10
#> # i 44 more rows
```

> **!** Important
>
> By filtering out rows with missing values, you ensure that your analysis is accurate, clean, and focused on complete data.

5.7.5 `arrange()` – Reordering Rows

The `arrange()` function lets you sort rows in a dataset based on the values in one or more columns. Sorting can be ascending or descending.

Key Points:

- By default, rows are sorted in **ascending** order.
- To sort in **descending** order, use the `desc()` function

Example 1: Sort by Total Sleep Time (Ascending)

Suppose you want to focus on species belonging to the orders *Rodentia*, *Carnivora*, and *Primates*. After filtering these groups, you can arrange the rows by their total sleep time (`sleep_total`) to identify species that sleep the least.

```r
# Total sleep in ascending order
msleep |>
  select(name, order, sleep_total) |>
  filter(order %in% c("Rodentia", "Carnivora", "Primates")) |>
  arrange(sleep_total)
```

```
#> # A tibble: 46 x 3
#>    name                       order     sleep_total
#>    <chr>                      <chr>           <dbl>
#>  1 Caspian seal               Carnivora         3.5
#>  2 Gray seal                  Carnivora         6.2
#>  3 Genet                      Carnivora         6.3
#>  4 Vesper mouse               Rodentia          7
#>  5 Degu                       Rodentia          7.7
#>  6 Human                      Primates          8
#>  7 African giant pouched rat  Rodentia          8.3
#>  8 Northern fur seal          Carnivora         8.7
#>  9 African striped mouse      Rodentia          8.7
#> 10 Guinea pig                 Rodentia          9.4
#> # i 36 more rows
```

Example 2: Sort by Total Sleep Time (Descending)
To identify the species that sleep the most, you can sort the same filtered dataset in descending order of `sleep_total`:

```r
# Total sleep in descending order
msleep |>
  select(name, order, sleep_total) |>
  filter(order %in% c("Rodentia", "Carnivora", "Primates")) |>
  arrange(desc(sleep_total))
```

```
#> # A tibble: 46 x 3
#>    name                           order     sleep_total
#>    <chr>                          <chr>           <dbl>
#>  1 Owl monkey                     Primates         17
#>  2 Arctic ground squirrel         Rodentia         16.6
#>  3 Golden-mantled ground squirrel Rodentia         15.9
#>  4 Tiger                          Carnivora        15.8
#>  5 Eastern american chipmunk      Rodentia         15.8
#>  6 Western american chipmunk      Rodentia         14.9
#>  7 Round-tailed muskrat           Rodentia         14.6
```

```
#>  8 Northern grasshopper mouse     Rodentia         14.5
#>  9 Mountain beaver                Rodentia         14.4
#> 10 Golden hamster                 Rodentia         14.3
#> # i 36 more rows
```

5.7.6 `slice()` – Selecting Rows by Position

The `slice()` function is a simple and efficient way to select rows based on their numerical position in your dataset[7]. Unlike functions that filter rows based on the content of columns, `slice()` simply picks rows by their numeric index. There are several helpful variants of this function that you might find useful:

- **`slice_head()`**: Selects the first n rows.
- **`slice_tail()`**: Selects the last n rows.
- **`slice_sample()`**: Randomly selects a specified number of rows.
- **`slice_min()`** and **`slice_max()`**: Select rows with the minimum or maximum values of a given variable.

Example 1: Select the First 5 Rows

Imagine you wish to quickly inspect the top entries from species belonging to the orders *Rodentia*, *Carnivora*, and *Primates*. With `slice_head()`, you can easily view the first 5 rows after filtering and selecting the relevant columns.

```
# Selecting the first 5 rows
msleep |>
  select(name, order, sleep_total) |>
  filter(order %in% c("Rodentia", "Carnivora", "Primates")) |>
  slice_head(n = 5)
```

```
#> # A tibble: 5 x 3
#>   name              order     sleep_total
#>   <chr>             <chr>           <dbl>
#> 1 Cheetah           Carnivora        12.1
#> 2 Owl monkey        Primates         17
#> 3 Mountain beaver   Rodentia         14.4
#> 4 Northern fur seal Carnivora         8.7
#> 5 Vesper mouse      Rodentia          7
```

[7] While `slice()` selects rows based on their numeric position (e.g., the first or last few rows), without reordering the entire dataset, the `arrange()` function is used to reorder all rows in the dataset according to the values of one or more specified columns.

Example 2: Select the Last 5 Rows

Similarly, if you are interested in looking at the bottom entries of the same group of species, you can use `slice_tail()` to view the last 5 rows.

```
# Selecting the last 5 rows
msleep |>
  select(name, order, sleep_total) |>
  filter(order %in% c("Rodentia", "Carnivora", "Primates")) |>
  slice_tail(n = 5)
```

```
#> # A tibble: 5 x 3
#>   name                           order     sleep_total
#>   <chr>                          <chr>           <dbl>
#> 1 Golden-mantled ground squirrel Rodentia         15.9
#> 2 Eastern american chipmunk      Rodentia         15.8
#> 3 Genet                          Carnivora         6.3
#> 4 Arctic fox                     Carnivora        12.5
#> 5 Red fox                        Carnivora         9.8
```

> **Tip**
>
> Both `slice_head()` and `slice_tail()` are very useful when you want to quickly check the beginning or end of a filtered dataset, especially when dealing with large datasets.

Example 3: Identify Top 5 Animals by Brain-to-Body Weight Ratio Using `slice_max()`

Suppose you want to identify the top five animals with the highest brain-to-body weight ratio among those that weigh more than 5 kg and have available brain weight data. First, filter the dataset to include only animals meeting these criteria. Then, calculate the ratio by dividing `brainwt` by `bodywt`. Finally, the `slice_max()` function will help you extract the five animals with the highest ratios.

```
msleep |>
  filter(bodywt > 5, !is.na(brainwt)) |>
  mutate(brain_to_body_ratio = brainwt / bodywt) |>
  slice_max(brain_to_body_ratio, n = 5)
```

```
#> # A tibble: 5 x 12
#>   name    genus vore  order conservation sleep_total sleep_rem sleep_cycle awake
#>   <chr>   <chr> <chr> <chr> <chr>              <dbl>     <dbl>       <dbl> <dbl>
```

```
#> 1 Macaque Maca~ omni  Prim~ <NA>                10.1       1.2       0.75   13.9
#> 2 Human   Homo  omni  Prim~ <NA>                 8         1.9       1.5    16
#> 3 Patas ~ Eryt~ omni  Prim~ lc                  10.9       1.1      NA      13.1
#> 4 Chimpa~ Pan   omni  Prim~ <NA>                 9.7       1.4       1.42   14.3
#> 5 Baboon  Papio omni  Prim~ <NA>                 9.4       1         0.667  14.6
#> # i 3 more variables: brainwt <dbl>, bodywt <dbl>, brain_to_body_ratio <dbl>
```

Example 4: Identify Bottom 3 Species by REM Sleep Using `slice_min()`

Suppose you want to find out which species have the least amount of REM sleep among those in the orders *Rodentia*, *Carnivora*, and *Primates*. The `slice_min()` function will help you extract the three species with the minimum REM sleep values.

```
msleep |>
  select(name, order, sleep_rem) |>
  filter(order %in% c("Rodentia", "Carnivora", "Primates")) |>
  slice_min(sleep_rem, n = 3)
```

```
#> # A tibble: 3 x 3
#>   name         order     sleep_rem
#>   <chr>        <chr>         <dbl>
#> 1 Caspian seal Carnivora       0.4
#> 2 Grivet       Primates        0.7
#> 3 Guinea pig   Rodentia        0.8
```

> **Tip**
>
> When you use `slice_min()` or `slice_max()`, the rows returned are already sorted by the specified variable (ascending for `slice_min()` and descending for `slice_max()`). You only need to use `arrange()` if you want to apply a different sorting order than the default.

5.7.7 `summarise()` – Aggregating Data

The `summarise()` function is used to create summary statistics by collapsing data into single values, such as calculating the minimum, maximum, mean, median, standard deviation, sum, or count for specific variables.

To use `summarise()`, define a new column name, followed by the `=` sign and the summary calculation:
`new_column = function(variable)`. You can include multiple summary functions within a single `summarise()` statement.

Key Points:

- Use it to compute one or more summary statistics.
- The functions `summarise()` and `summarize()` are interchangeable
- It is often used with `group_by()` to generate summaries by groups within the dataset.

Example: Summarising the Entire Dataset

To calculate the total number of animals, the average sleep time, and the maximum sleep time across all species in the `msleep` dataset:

```
msleep |>
  summarise(
    n = n(),
    average = mean(sleep_total),
    maximum = max(sleep_total)
  )
```

```
#> # A tibble: 1 x 3
#>       n average maximum
#>   <int>   <dbl>   <dbl>
#> 1    83    10.4    19.9
```

> **Note**
>
> The `summarise()` function works with various aggregate functions, including:
>
> - `n()`: Number of observations.
> - `n_distinct(var)`: Number of unique values in a variable.
> - Arithmetic functions: `sum(var)`, `max(var)`, `min(var)`.
> - Statistical functions: `mean(var)`, `median(var)`, `sd(var)`, `IQR(var)`.

In most cases, we don't just want to summarise the whole data table, but we want to get summaries by a group.

5.7.8 `group_by()` – Working with Groups

On its own, the `group_by()` function doesn't perform any operation. However, when combined with functions like `summarise()` or `mutate()`, it becomes a powerful tool for splitting data into groups and applying operations to each group separately.

Key Points:

- Groups can be based on one or multiple variables.
- After grouping, any operation is applied independently to each group.
- The results are combined into a single data frame.

Figure 5.5 below illustrates the group by strategy:

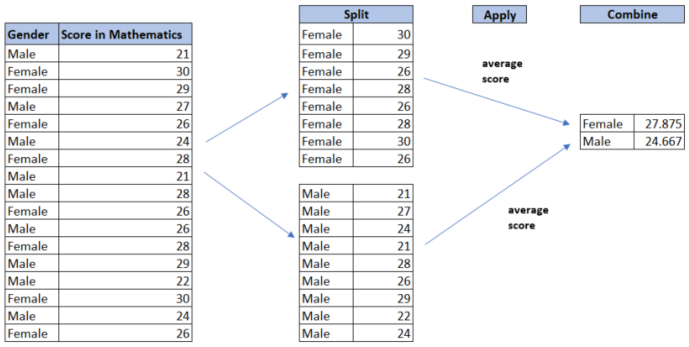


Figure 5.5: Data Aggregation and Group Operations

Example 2: Summarising by Groups

To generate summaries for specific groups, combine `summarise()` with `group_by()`:

```
msleep |>
  group_by(order) |>
  summarise(
    n = n(),
    average_sleep = mean(sleep_total, na.rm = TRUE),
    maximum_sleep = max(sleep_total, na.rm = TRUE)
  )
```

```
#> # A tibble: 19 x 4
#>    order              n average_sleep maximum_sleep
#>    <chr>          <int>         <dbl>         <dbl>
#>  1 Afrosoricida       1         15.6           15.6
#>  2 Artiodactyla       6          4.52           9.1
#>  3 Carnivora         12         10.1           15.8
#>  4 Cetacea            3          4.5            5.6
#>  5 Chiroptera         2         19.8           19.9
#>  6 Cingulata          2         17.8           18.1
#>  7 Didelphimorphia    2         18.7           19.4
#>  8 Diprotodontia      2         12.4           13.7
```

```
#>  9 Erinaceomorpha      2         10.2          10.3
#> 10 Hyracoidea          3          5.67          6.3
#> 11 Lagomorpha          1          8.4           8.4
#> 12 Monotremata         1          8.6           8.6
#> 13 Perissodactyla      3          3.47          4.4
#> 14 Pilosa              1         14.4          14.4
#> 15 Primates           12         10.5          17
#> 16 Proboscidea         2          3.6           3.9
#> 17 Rodentia           22         12.5          16.6
#> 18 Scandentia          1          8.9           8.9
#> 19 Soricomorpha        5         11.1          14.9
```

> **Note**
>
> In this example:
>
> - `n`: Number of species in each group.
> - `average_sleep`: Average sleep time for each group, ignoring `NA` values (`na.rm = TRUE`).
> - `maximum_sleep`: Maximum sleep time for each group, ignoring `NA` values.

Using `summarise()` with or without `group_by()` helps you aggregate and summarize your data efficiently, making it easier to extract meaningful insights from your dataset.

5.7.9 Combining All the Verbs:

Let's tackle a realistic problem by chaining the verbs you've learned together. For this example, we'll use the `penguins` dataset (from the `palmerpenguins` package) to answer a specific question:

Task:

Identify the top 5 penguins on Dream Island with flipper lengths exceeding 200 mm. Calculate their bill aspect ratio (bill length ÷ bill depth) and display the results sorted by this ratio.

```r
# Load the dataset
penguins <- palmerpenguins::penguins

# Perform the analysis
penguins |>
  filter(island == "Dream" & flipper_length_mm > 200) |>
```

```
  mutate(bill_aspect_ratio = bill_length_mm / bill_depth_mm) |>
  slice_max(bill_aspect_ratio, n = 5) |>
  select(species, flipper_length_mm, bill_aspect_ratio)
```

```
#> # A tibble: 5 x 3
#>   species   flipper_length_mm bill_aspect_ratio
#>   <fct>                 <int>             <dbl>
#> 1 Chinstrap               201              2.87
#> 2 Chinstrap               207              2.82
#> 3 Chinstrap               201              2.75
#> 4 Chinstrap               203              2.71
#> 5 Chinstrap               201              2.71
```

Tip

This code performs the following steps:

1. **Filter**: Retains only penguins located on Dream Island with a flipper length greater than 200 mm.
2. **Mutate**: Calculates the `bill_aspect_ratio` by dividing `bill_length_mm` by `bill_depth_mm`.
3. **Slice Max**: Selects the top 5 penguins with the highest bill aspect ratios.
4. **Select**: Displays only the columns for species, flipper length, and bill aspect ratio.

5.7.10 Exercise 5.2.1: Top 5 Carnivorous Animals

Now, let's extend your skills with a new challenge. Using the `msleep` dataset, identify the **top 5 carnivorous animals** that sleep the most and calculate their **sleep-to-weight ratio** to understand how sleep duration scales with body size.

Task: Complete the following code by replacing the placeholders (`...`) with the correct values:

```
msleep |>
  filter(vore == ...) |>
  mutate(sleep_to_weight = ... / ...) |>
  select(name, sleep_total, sleep_to_weight) |>
  slice_max(sleep_total, n = ---)
```

Instructions

1. Replace `...` with the appropriate filtering criteria and calculations.
2. Ensure the final code filters for carnivores, calculates the `sleep_to-weight` ratio, and returns the top 5 animals that sleep the most.

See the Solution to Exercise 5.2.1

5.7.11 Exploring More Functions in `dplyr`

The `dplyr` package offers a wealth of functions to simplify data manipulation, enabling you to efficiently clean, transform, and analyze data. In addition to the core verbs (`filter()`, `select()`, `mutate()`, `arrange()`, `summarize()`), there are other incredibly useful functions such as:

1. **`rename()`**: Rename columns in your dataset.
2. **`distinct()`**: Extract unique rows or values.
3. **`count()`**: Count occurrences of unique values in a variable.
4. **`relocate()`**: Reorder or reposition columns for better organization.

Table 5.2: Summary of other Functions in dplyr

Function	Purpose	Example Usage
`rename()`	Rename columns to more meaningful names	`rename(new_name = old_name)`
`distinct()`	Find unique rows or specific values	`distinct(column1, column2)`
`count()`	Count the frequency of unique values	`count(column_name)`
`relocate()`	Reorder columns for better organization	`relocate(column_name, .before = another_column)`

Let's explore each of the functions in Table 5.2 in detail using the **`msleep`** dataset from `ggplot2` and the **`penguins`** dataset from `palmerpenguins`.

5.7.11.1 `rename()` – Renaming Columns

The `rename()` function allows you to change column names to make them more meaningful or easier to work with. This is especially helpful when dealing with datasets that have poorly named columns.

Key Points:

- Syntax: `rename(new_name = old_name)`
- You can rename one or multiple columns at a time.
- The rest of the dataset remains unchanged.

Example 1: Renaming Columns in `msleep`

Rename the column `name` to `animal_name` and `sleep_total` to `total_sleep`:

```
msleep |>
  rename(
    animal_name = name,
    total_sleep = sleep_total
  )
```

```
#> # A tibble: 83 x 11
#>    animal_name  genus vore  order conservation total_sleep sleep_rem sleep_cycle
#>    <chr>        <chr> <chr> <chr> <chr>              <dbl>     <dbl>       <dbl>
#>  1 Cheetah      Acin~ carni Carn~ lc                  12.1      NA        NA
#>  2 Owl monkey   Aotus omni  Prim~ <NA>                17         1.8      NA
#>  3 Mountain be~ Aplo~ herbi Rode~ nt                  14.4       2.4      NA
#>  4 Greater sho~ Blar~ omni  Sori~ lc                  14.9       2.3       0.133
#>  5 Cow          Bos   herbi Arti~ domesticated         4         0.7       0.667
#>  6 Three-toed ~ Brad~ herbi Pilo~ <NA>                14.4       2.2       0.767
#>  7 Northern fu~ Call~ carni Carn~ vu                   8.7       1.4       0.383
#>  8 Vesper mouse Calo~ <NA>  Rode~ <NA>                 7        NA        NA
#>  9 Dog          Canis carni Carn~ domesticated        10.1       2.9       0.333
#> 10 Roe deer     Capr~ herbi Arti~ lc                   3        NA        NA
#> # i 73 more rows
#> # i 3 more variables: awake <dbl>, brainwt <dbl>, bodywt <dbl>
```

> **Note**
>
> 1. `rename(animal_name = name)` renames the `name` column to `animal_name`.
> 2. `rename(total_sleep = sleep_total)` renames `sleep_total` to `total_sleep`.
> 3. You can use this to make column names more descriptive.

Example 2:

Rename `bill_length_mm` to `bill_length` and `flipper_length_mm` to `flipper_length` in the penguins data:

```
penguins |>
  rename(
    bill_length = bill_length_mm,
    flipper_length = flipper_length_mm
  )
```

```
#> # A tibble: 344 x 8
#>    species island    bill_length bill_depth_mm flipper_length body_mass_g sex
#>    <fct>   <fct>           <dbl>         <dbl>          <int>       <int> <fct>
#>  1 Adelie  Torgersen        39.1          18.7            181        3750 male
#>  2 Adelie  Torgersen        39.5          17.4            186        3800 female
#>  3 Adelie  Torgersen        40.3          18              195        3250 female
#>  4 Adelie  Torgersen        NA            NA               NA          NA <NA>
#>  5 Adelie  Torgersen        36.7          19.3            193        3450 female
#>  6 Adelie  Torgersen        39.3          20.6            190        3650 male
#>  7 Adelie  Torgersen        38.9          17.8            181        3625 female
#>  8 Adelie  Torgersen        39.2          19.6            195        4675 male
#>  9 Adelie  Torgersen        34.1          18.1            193        3475 <NA>
#> 10 Adelie  Torgersen        42            20.2            190        4250 <NA>
#> # i 334 more rows
#> # i 1 more variable: year <int>
```

Reflection Question

How might renaming confusing column names or removing duplicates before analysis contribute to more confident and accurate inferences?

5.7.11.2 `distinct()` – Extracting Unique Rows or Values

Duplicates in datasets can distort analyses and lead to inaccurate conclusions. Identifying and removing duplicates ensures clean data that accurately represents unique observations. The `distinct()` function is a simple yet powerful tool for finding unique rows or specific combinations of values in a dataset. It is particularly useful for removing duplicate rows or understanding unique categories in a variable.

Key Points:

1. **Default Behavior**:
   By default, `distinct()` considers all columns to identify unique rows.
2. **Specific Variables**:
   You can specify one or more columns to extract unique values for specific variables.
3. **Keeping All Variables**:
   Adding `.keep_all = TRUE` while specifying columns ensures that all other variables are retained in the resulting dataset.
4. **Using `janitor::get_dupes()`**:
   The `get_dupes()` function from the **janitor** package provides an easy way to identify duplicates. It returns:
   - Full records where the specified variables have duplicates.
   - A column called `dupe_count` showing the number of rows sharing each duplicate combination.

Example 1: Counting Duplicates (`iris` Dataset)

To count the number of duplicate rows in a dataset, use the combination of `duplicated()` and `sum()` functions:

```
sum(duplicated(iris))
```

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

There is one duplicate record found in this dataset.

Example 2: Identifying and Removing Duplicates (`iris` Dataset)

To find duplicate records in the `iris` dataset

```
iris |> janitor::get_dupes()
```

```
#> No variable names specified - using all columns.
```

```
#> # A tibble: 2 x 6
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width Species   dupe_count
#>          <dbl>       <dbl>        <dbl>       <dbl> <fct>          <int>
#> 1          5.8         2.7          5.1         1.9 virginica          2
#> 2          5.8         2.7          5.1         1.9 virginica          2
```

The output shows the duplicate records in the `iris` dataset. To remove these duplicates and retain only the unique rows, you can use the `distinct()` function as follows:

```
iris |> distinct()
```

```
#> # A tibble: 149 x 5
#>    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#>           <dbl>       <dbl>        <dbl>       <dbl> <fct>
#>  1          5.1         3.5          1.4         0.2 setosa
#>  2          4.9         3            1.4         0.2 setosa
#>  3          4.7         3.2          1.3         0.2 setosa
#>  4          4.6         3.1          1.5         0.2 setosa
#>  5          5           3.6          1.4         0.2 setosa
#>  6          5.4         3.9          1.7         0.4 setosa
#>  7          4.6         3.4          1.4         0.3 setosa
#>  8          5           3.4          1.5         0.2 setosa
#>  9          4.4         2.9          1.4         0.2 setosa
#> 10          4.9         3.1          1.5         0.1 setosa
#> # i 139 more rows
```

Example 3: Finding Unique Values in a Specific Column (`msleep` Dataset)

To extract unique values from the `vore` column in the `msleep` dataset (diet types):

```
msleep |>
  distinct(vore)
```

```
#> # A tibble: 5 x 1
#>   vore
#>   <chr>
#> 1 carni
#> 2 omni
#> 3 herbi
#> 4 <NA>
#> 5 insecti
```

> **Note**
>
> The `distinct(vore)` function returns only the unique values in the `vore` column, helping you understand the categories in this variable.

Example 4: Keeping All Variables When Filtering for Uniqueness

To return unique rows based on the `vore` column while keeping all other variables:

```
msleep |>
  distinct(vore, .keep_all = TRUE)
```

```
#> # A tibble: 5 x 11
#>   name    genus vore  order conservation sleep_total sleep_rem sleep_cycle awake
#>   <chr>   <chr> <chr> <chr> <chr>              <dbl>     <dbl>       <dbl> <dbl>
#> 1 Cheetah Acin~ carni Carn~ lc                  12.1      NA         NA      11.9
#> 2 Owl mo~ Aotus omni  Prim~ <NA>                17         1.8       NA       7
#> 3 Mounta~ Aplo~ herbi Rode~ nt                  14.4       2.4       NA       9.6
#> 4 Vesper~ Calo~ <NA>  Rode~ <NA>                 7        NA         NA      17
#> 5 Big br~ Epte~ inse~ Chir~ lc                  19.7       3.9        0.117   4.3
#> # i 2 more variables: brainwt <dbl>, bodywt <dbl>
```

> **Note**
>
> Adding `.keep_all = TRUE` ensures that the uniqueness is determined by the `vore` column, but all other columns are retained in the output.

5.7.11.3 `count()` – Counting Occurrences

The `count()` function is a quick way to calculate the frequency of unique values in a column or combinations of columns. It is particularly useful for summarizing categorical variables.

Adding the `sort = TRUE` argument will automatically sort the results in descending order of frequency.

Key Points:

- By default, `count()` returns the number of occurrences for each unique value.
- Use `count(x, sort = TRUE)` to sort the results, with the largest groups appearing at the top.

Example 1

Count the number of animals in each diet category (`vore`) in the `msleep` data:

```
msleep |>
  count(vore, sort = TRUE)
```

```
#> # A tibble: 5 x 2
#>   vore        n
#>   <chr>   <int>
#> 1 herbi      32
#> 2 omni       20
#> 3 carni      19
#> 4 <NA>        7
#> 5 insecti     5
```

Explanation:

1. `count(vore)` calculates how many times each diet type (`vore`) appears.
2. The output has two columns: `vore` and `n` (the count).

Example 2

Count the number of penguins on each island in the penguins data:

```
penguins |>
  count(island)
```

```
#> # A tibble: 3 x 2
#>   island        n
#>   <fct>     <int>
#> 1 Biscoe      168
#> 2 Dream       124
#> 3 Torgersen    52
```

5.7.11.4 `relocate()` – Reordering Columns

The `relocate()` function allows you to rearrange columns for better readability or logical grouping using the same syntax as `select()` to make it easy to move blocks of columns at once. It doesn't remove or modify columns, only changes their position.

Key Points:

- Use `relocate(column_name, .before = ...)` to move a column before a specific column.
- Use `relocate(column_name, .after = ...)` to move a column after a specific column.

Example 1 : Reordering Columns in `msleep`

Move `bodywt` to appear after the `sleep_total` column:

```
msleep |>
  relocate(bodywt, .after = sleep_total)
```

```
#> # A tibble: 83 x 11
#>    name             genus vore  order conservation sleep_total  bodywt sleep_rem
#>    <chr>            <chr> <chr> <chr> <chr>              <dbl>   <dbl>     <dbl>
#>  1 Cheetah          Acin~ carni Carn~ lc                  12.1  50          NA
#>  2 Owl monkey       Aotus omni  Prim~ <NA>                17     0.48        1.8
#>  3 Mountain beaver  Aplo~ herbi Rode~ nt                  14.4   1.35        2.4
#>  4 Greater short-t~ Blar~ omni  Sori~ lc                  14.9   0.019       2.3
#>  5 Cow              Bos   herbi Arti~ domesticated         4   600           0.7
#>  6 Three-toed sloth Brad~ herbi Pilo~ <NA>                14.4   3.85        2.2
#>  7 Northern fur se~ Call~ carni Carn~ vu                   8.7  20.5         1.4
#>  8 Vesper mouse     Calo~ <NA>  Rode~ <NA>                 7     0.045      NA
#>  9 Dog              Canis carni Carn~ domesticated        10.1  14           2.9
#> 10 Roe deer         Capr~ herbi Arti~ lc                   3    14.8        NA
#> # i 73 more rows
#> # i 3 more variables: sleep_cycle <dbl>, awake <dbl>, brainwt <dbl>
```

Example 2:

You can also relocate variables based on their data type:

```
penguins |> relocate(where(is.factor), .before = bill_length_mm)
```

```
#> # A tibble: 344 x 8
#>    species island    sex    bill_length_mm bill_depth_mm flipper_length_mm
#>    <fct>   <fct>     <fct>           <dbl>         <dbl>             <int>
#>  1 Adelie  Torgersen male             39.1          18.7               181
#>  2 Adelie  Torgersen female           39.5          17.4               186
#>  3 Adelie  Torgersen female           40.3          18                 195
#>  4 Adelie  Torgersen <NA>             NA            NA                  NA
#>  5 Adelie  Torgersen female           36.7          19.3               193
#>  6 Adelie  Torgersen male             39.3          20.6               190
#>  7 Adelie  Torgersen female           38.9          17.8               181
#>  8 Adelie  Torgersen male             39.2          19.6               195
#>  9 Adelie  Torgersen <NA>             34.1          18.1               193
#> 10 Adelie  Torgersen <NA>             42            20.2               190
```

```
#> # i 334 more rows
#> # i 2 more variables: body_mass_g <int>, year <int>
```

Combining These Functions

Let's combine `rename()`, `distinct()`, `count()`, and `relocate()` to solve a real-world problem.

Example: Cleaning and Organizing `msleep` Data

1. Rename columns for clarity.
2. Remove duplicate rows.
3. Reorganize columns for better readability.
4. Count occurrences of diet types.

```
msleep |>
  rename(
    animal_name = name,
    diet_type = vore
  ) |>
  distinct() |>
  relocate(diet_type, .before = animal_name) |>
  count(diet_type, sort = TRUE)
```

```
#> # A tibble: 5 x 2
#>   diet_type     n
#>   <chr>     <int>
#> 1 herbi        32
#> 2 omni         20
#> 3 carni        19
#> 4 <NA>          7
#> 5 insecti       5
```

5.7.12 Practice Quiz 5.2

Question 1:

Which function would you use in `dplyr` to randomly select a specified number of rows from a dataset?

a) `sample(n = 5)`

b) `slice_sample(n = 5)`

c) `filter_sample()`

d) `mutate_sample()`

Question 2:

To calculate the average `sleep_total` for each `vore` category, which combination of functions is most appropriate?

a) `group_by(vore) |> select(sleep_total) |> summarise(mean(sleep_total))`

b) `select(vore, sleep_total) |> summarise(mean(sleep_total)) |> group_by(vore)`

c) `group_by(vore) |> summarise(avg_sleep = mean(sleep_total, na.rm = TRUE))`

d) `filter(vore) |> mutate(avg_sleep = mean(sleep_total))`

Question 3:

To extract rows with the maximum value of a specified variable, which function is appropriate in `dplyr`?

a) `slice_max()`

b) `slice_min()`

c) `mutate()`

d) `select()`

Question 4:

Which `dplyr` function would you use if you want to create a new column called `weight_ratio` by dividing `bodywt` by `mean_bodywt`?

a) `filter()`

b) `select()`

c) `mutate()`

d) `arrange()`

Question 5:

Suppose you need to identify the top 3 penguins with the highest bill aspect ratio from the `penguins` dataset after calculating it in a new column. Which of the following code snippets is the most concise and appropriate?

a)

```
penguins |>
  mutate(bill_aspect_ratio = bill_length_mm / bill_depth_mm) |>
  arrange(desc(bill_aspect_ratio)) |>
  head(3)
```

b)

```
penguins |>
  mutate(bill_aspect_ratio = bill_length_mm / bill_depth_mm) |>
  slice_max(bill_aspect_ratio, n = 3)
```

c) Both a and b are equally concise and valid.

d) Neither a nor b is valid.

Question 6:

Given the following code, which is the correct equivalent using the pipe operator?

```
result <- arrange(filter(select(msleep, name, sleep_total), sleep_total > 8), sleep_total)
```

a) `msleep |> select(name, sleep_total) |> filter(sleep_total > 8) |> arrange(sleep_total)`

b) `msleep |> filter(sleep_total > 8) |> select(name, sleep_total) |> arrange(sleep_total)`

c) `select(msleep, name, sleep_total) |> filter(sleep_total > 8) |> arrange(sleep_total)`

d) `msleep |> arrange(sleep_total) |> filter(sleep_total > 8) |> select(name, sleep_total)`

Question 7:

Which of the following correctly applies a log transformation to numeric columns **only**?

a)

```
mutate_all(log)
```

b)

```
mutate(across(everything(), log))
```

c)

```
mutate(select(where(is.numeric), log))
```

d)

```
mutate(across(where(is.numeric), log))
```

Question 8:

What does `mutate(across(everything(), as.character))` do?

a) Converts all character columns to numeric.

b) Converts all columns in the dataset to character type.

c) Applies a conditional transformation to numeric columns.

d) Filters out non-character values.

Question 9:

To extract the rows with the minimum value of a specified variable, which `dplyr` function should you use?

a) `slice_min()`

b) `slice_max()`

c) `arrange()`

d) `filter()`

Question 10:

If you want to reorder the rows of `msleep` by `sleep_total` in ascending order and then only show the top 5 rows, which code snippet is correct?

a) `msleep |> arrange(sleep_total) |> head(5)`

b) `msleep |> head(5) |> arrange(sleep_total)`

c) `msleep |> summarise(sleep_total) |> head(5)`

d) `msleep |> select(sleep_total) |> arrange(desc(sleep_total)) |> head(5)`

See the Solution to Quiz 5.2

5.7.13 Exercise 5.2.2: Analysing the Penguins Dataset

Let's put your skills into practice with a modified `penguins` dataset. First, you'll need to create a new RStudio project called `Experiment 5.1`.

1. **Importing and Inspecting Data**
   - Locate the `penguins.xlsx` file in the `r-data` directory. If you don't already have the file, you can download it from Google Drive.
   - Import the data into R.
   - Use `glimpse(penguins)` to get an overview.
   - How many rows and columns are there?
2. **Filtering Data**
   - How many penguins are from the `Biscoe` island?
   - Extract data for penguins with a body mass greater than 4,500 grams.
3. **Arranging Data**
   - Arrange the data in descending order based on flipper length.
   - Find the top 5 penguins with the highest body mass.
4. **Selecting and Mutating**
   - Select only the columns `species`, `island`, and `sex`.
   - Remove the `sex` column from the dataset.
   - Convert the flipper length from millimeters to meters and create a new column `flipper_length_m`

     To convert millimeters to meters, you simply divide the number of millimeters by 1,000. Here's the conversion formula:

$$\text{flipper\_length\_m} = \frac{\text{flipper\_length\_mm}}{1000}$$

- Create a new column `BMI` calculated as:

$$\text{BMI} = \frac{\text{body\_mass\_g}}{\text{flipper\_length\_m}^2}$$

5. **Summarizing and Grouping**
    - Calculate the average body mass of all penguins.
    - Group the data by `species` and find the average body mass for each species.
6. **Combining Operations**
    - Filter penguins from the **Dream** island and summarize the average bill length for each species from this island.

5.7.14 Exercise 5.2.3: Data Analyst Candidate Assessment

In this exercise, you'll work with a real dataset of medical insurance records similar to those of a well-known health insurance company in the country. We want to see how you clean, transform and analyse data in a practical, real-world context as a Data Analyst. Please follow the instructions below and document your process along the way.

Dataset Overview

You're provided with a dataset containing medical insurance records for various individuals. Here's what each column represents:

- **User ID**: A unique identifier for each individual.
- **Gender**: The individual's gender ('Male' or 'Female').
- **Age**: The age of the individual in years.
- **AgeGroup**: The age bracket into which the individual falls.
- **Estimated Salary**: An estimate of the individual's annual salary.
- **Purchased**: Indicates whether the individual has purchased medical insurance ('purchased' or 'not-purchased').

Data Import Instructions

1. Locate the `medical_insurance.xlsx` file in the `r-data` directory. If you don't have it yet, you can download it from Google Drive.
2. Import the dataset into R using the `readxl` package.

Tasks

1. Data Transformation

- **Purchased Column Conversion:**
  Convert the `Purchased` column values to binary: use `1` for 'purchased' and `0` for 'not-purchased'.
- **Creating Salary Brackets:**
  Add a new column called `SalaryBracket` based on the `Estimated Salary`:
  - **Low**: Salary $< 30{,}000$
  - **Medium**: Salary between 30,000 and 70,000
  - **High**: Salary $> 70{,}000$

2. Analysis and Insights

- **Insurance Purchase Analysis:**
  Calculate and present:
  - The percentage of individuals who have purchased insurance, broken down by **Gender**.
  - The percentage of individuals who have purchased insurance, broken down by **AgeGroup**.
- **Salary Bracket Purchase Rate:**
  Determine which `SalaryBracket` shows the highest rate of insurance purchases.

> **Note**
>
> Take your time to work through these tasks carefully. We're looking forward to seeing how you apply your analytical skills to solve real-world data challenges. Good luck!

5.8 Experiment 5.3: Dealing with Missing Data

Missing data is common in real-world scenarios. Values may be missing because of measurement errors, data entry mistakes, or unavailability of certain information. It is crucial to detect and handle missing values properly, as they can bias results or cause errors in your analysis. R provides several functions to help you deal with missing data.

5.8.1 Recognising Missing Values

In R, missing values are represented by `NA`. Identifying these missing values is crucial for accurate data analysis. Here are some functions to check for missing data:

- **`is.na()`**: Returns a logical vector indicating which elements are `NA`.

```
x <- c(1, 2, NA, 4, NA, 6)

is.na(x)
```

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

- **`anyNA()`**: Checks if there are any `NA` values in an object. It returns `TRUE` if there is at least one `NA`, and `FALSE` otherwise.

```
anyNA(x)
```

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

Let's apply the `anyNA()` function to a sample `salary_data` data frame:

```
salary_data <- data.frame(
  Name = c("Alice", "Francisca", "Fatima", "David"),
  Age = c(25, NA, 30, 35),
  Salary = c(50000, 52000, NA, 55000)
)

salary_data
```

```
#>        Name Age Salary
#> 1     Alice  25  50000
#> 2 Francisca  NA  52000
#> 3    Fatima  30     NA
#> 4     David  35  55000
```

In this data frame, Francisca's age and Fatima's salary are missing. We can use `anyNA()` to check whether there are any missing values in the entire data frame:

```
anyNA(salary_data)
```

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

Since the output is `TRUE`, we know there are missing values. We can also check specific columns for missing values. For example, let's check the `Age` column:

```
anyNA(salary_data$Age)
```

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

And similarly, we can check the `Name` column:

```
anyNA(salary_data$Name)
```

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

- **`complete.cases()`**: Identifies rows in a dataset that have no missing values (`NA`). It evaluates each row and checks whether it is "complete" (i.e., contains no `NA` values). It returns a logical vector where:
  - **`TRUE`** indicates that a row has no missing values (all columns are complete).
  - **`FALSE`** indicates that a row contains at least one missing value.

For example, using our sample `salary_data` data frame:

```
salary_data
```

```
#>        Name Age Salary
#> 1     Alice  25  50000
#> 2 Francisca  NA  52000
#> 3    Fatima  30     NA
#> 4     David  35  55000
```

```r
complete.cases(salary_data)
```

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

> **Note**
>
> This indicates that:
>
> - Row 1 (`Alice`): No missing values, so it is **complete** (`TRUE`).
> - Row 2 (`Francisca`): Missing value in the `Age` column, so it is **not complete** (`FALSE`).
> - Row 3 (`Fatima`): Missing value in the `Salary` column, so it is **not complete** (`FALSE`).
> - Row 4 (`David`): No missing values, so it is **complete** (`TRUE`).

5.8.2 Summarising Missing Data

After identifying that your dataset contains missing values, it's essential to quantify them to understand the extent of the issue. Summarizing missing data helps you decide how to handle these gaps appropriately. To count the total number of missing values in your entire dataset, you can use the `sum()` function combined with `is.na()`. Remember the `is.na()` function returns a logical vector where each element is `TRUE` if the corresponding value in the dataset is `NA`, and `FALSE` otherwise. Summing this logical vector gives you the total count of missing values because `TRUE` is treated as 1 and `FALSE` as 0 in arithmetic operations.

Example:

Suppose you have a sampled `airquality` dataset:

```r
airquality_data <- data.frame(
  Ozone = c(41, 36, 12, 18, NA, 28, 23, 19, 8, NA),
  Solar.R = c(190, 118, 149, 313, NA, NA, 299, 99, 19, 194),
  Wind = c(7.4, 8, 12.6, 11.5, 14.3, 14.9, 8.6, 13.8, 20.1, 8.6),
  Temp = c(67, 72, 74, 62, 56, 66, 65, 59, 61, 69),
  Month = c(5, 5, 5, NA, NA, NA, 5, 5, 5, 5),
  Day = c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
)

airquality_data
```

```
#>    Ozone Solar.R Wind Temp Month Day
#> 1     41     190  7.4   67     5   1
#> 2     36     118  8.0   72     5   2
#> 3     12     149 12.6   74     5   3
#> 4     18     313 11.5   62    NA   4
#> 5     NA      NA 14.3   56    NA   5
#> 6     28      NA 14.9   66    NA   6
#> 7     23     299  8.6   65     5   7
#> 8     19      99 13.8   59     5   8
#> 9      8      19 20.1   61     5   9
#> 10    NA     194  8.6   69     5  10
```

To count the total number of missing values in this dataset, you would use:

```
sum(is.na(airquality_data))
```

```
#> [1] 7
```

There are 7 missing values in the entire data frame.

Missing Values Per Column:

```
colSums(is.na(airquality_data))
```

```
#>   Ozone Solar.R    Wind    Temp   Month     Day
#>       2       2       0       0       3       0
```

This output indicates:

- `Ozone` column has 2 missing values.
- `Solar.R` column has 2 missing values.
- `Wind` column has 0 missing values.
- `Temp` column has 0 missing values.
- `Month` column has 3 missing values.
- `Day`column has 0 missing values.

For a column-wise summary, you can also use the `inspect_na()` function from the `inspectdf` package.

```
inspectdf::inspect_na(airquality_data)
```

```
#> # A tibble: 6 x 3
#>   col_name   cnt  pcnt
#>   <chr>    <int> <dbl>
#> 1 Month        3    30
#> 2 Ozone        2    20
#> 3 Solar.R      2    20
#> 4 Wind         0     0
#> 5 Temp         0     0
#> 6 Day          0     0
```

5.8.3 Strategies for Dealing with Missing Data

Managing missing data is a critical step in data preprocessing. There are several strategies available, depending on the nature of the data and the goals of the analysis.

5.8.3.1 Remove Missing Values

You can remove rows with missing values using `na.omit()` function:

```
cleaned_data <- na.omit(airquality_data)
cleaned_data
```

```
#>   Ozone Solar.R Wind Temp Month Day
#> 1    41     190  7.4   67     5   1
#> 2    36     118  8.0   72     5   2
#> 3    12     149 12.6   74     5   3
#> 7    23     299  8.6   65     5   7
#> 8    19      99 13.8   59     5   8
#> 9     8      19 20.1   61     5   9
```

> **Warning**
>
> This method is simple and effective when the proportion of missing data is small. However, it can result in a significant loss of data if many rows contain missing values.

5.8.3.2 Replace Missing Values

Replace missing values with a default value using **`replace_na()`** from the **`tidyr`** package:

```
library(tidyverse) # tidyr is part of tidyverse
dataset <- airquality_data %>%
  replace_na(list(Ozone = 17, Month = 3))

dataset
```

```
#>    Ozone Solar.R Wind Temp Month Day
#> 1     41     190  7.4   67     5   1
#> 2     36     118  8.0   72     5   2
#> 3     12     149 12.6   74     5   3
#> 4     18     313 11.5   62     3   4
#> 5     17      NA 14.3   56     3   5
#> 6     28      NA 14.9   66     3   6
#> 7     23     299  8.6   65     5   7
#> 8     19      99 13.8   59     5   8
#> 9      8      19 20.1   61     5   9
#> 10    17     194  8.6   69     5  10
```

> **Note**
>
> This approach ensures that missing values are replaced consistently, allowing for meaningful analysis without introducing bias.

5.8.3.3 Impute Missing Values

Missing values can be replaced with statistical measures such as the mean, median, or mode. For example, missing values in the `Ozone` column can be replaced with the mean, while missing values in the `Month` column can be replaced with the median:

```
library(dplyr)

airquality_data <- airquality_data %>%
  mutate(
    Ozone = ifelse(is.na(Ozone), mean(Ozone, na.rm = TRUE), Ozone),
    Month = ifelse(is.na(Month), median(Month, na.rm = TRUE), Month)
  )
```

```r
# View the resulting dataset
airquality_data
```

```
#>     Ozone Solar.R Wind Temp Month Day
#> 1  41.000     190  7.4   67     5   1
#> 2  36.000     118  8.0   72     5   2
#> 3  12.000     149 12.6   74     5   3
#> 4  18.000     313 11.5   62     5   4
#> 5  23.125      NA 14.3   56     5   5
#> 6  28.000      NA 14.9   66     5   6
#> 7  23.000     299  8.6   65     5   7
#> 8  19.000      99 13.8   59     5   8
#> 9   8.000      19 20.1   61     5   9
#> 10 23.125     194  8.6   69     5  10
```

Tip

If you don't want to use `ifelse`, you can achieve the same result using the `coalesce()` function from **dplyr**. `coalesce()` replaces `NA` values by providing a fallback value.

```r
library(dplyr)

airquality_data <- airquality_data %>%
  mutate(
    Ozone = coalesce(Ozone, mean(Ozone, na.rm = TRUE)),
    Month = coalesce(Month, median(Month, na.rm = TRUE))
  )

# View the resulting dataset
airquality_data
```

```
#>     Ozone Solar.R Wind Temp Month Day
#> 1  41.000     190  7.4   67     5   1
#> 2  36.000     118  8.0   72     5   2
#> 3  12.000     149 12.6   74     5   3
#> 4  18.000     313 11.5   62     5   4
#> 5  23.125      NA 14.3   56     5   5
#> 6  28.000      NA 14.9   66     5   6
#> 7  23.000     299  8.6   65     5   7
#> 8  19.000      99 13.8   59     5   8
#> 9   8.000      19 20.1   61     5   9
#> 10 23.125     194  8.6   69     5  10
```

- `coalesce()` is a simpler and more concise alternative when dealing with missing values.
- `coalesce(Ozone, mean(Ozone, na.rm = TRUE))` replaces `NA` values in `Ozone` with the computed mean.
- `coalesce(Month, median(Month, na.rm = TRUE))` does the same for `Month` using the median.

For more advanced imputation methods, you can use specialised packages like `mice` or `Hmisc`. Additionally, the bulkreadr package simplifies the process with the `fill_missing_values()` function:

- Impute Specific Columns:

```
library(bulkreadr)

fill_missing_values(airquality_data,
  selected_variables = c("Ozone", "Solar.R"),
  method = "mean"
)
```

```
#>     Ozone Solar.R Wind Temp Month Day
#> 1  41.000 190.000  7.4   67     5   1
#> 2  36.000 118.000  8.0   72     5   2
#> 3  12.000 149.000 12.6   74     5   3
#> 4  18.000 313.000 11.5   62     5   4
#> 5  23.125 172.625 14.3   56     5   5
#> 6  28.000 172.625 14.9   66     5   6
#> 7  23.000 299.000  8.6   65     5   7
#> 8  19.000  99.000 13.8   59     5   8
#> 9   8.000  19.000 20.1   61     5   9
#> 10 23.125 194.000  8.6   69     5  10
```

- Impute All Columns in the Data Frame:

```
fill_missing_values(airquality_data, method = "median")
```

```
#>     Ozone Solar.R Wind Temp Month Day
#> 1  41.000   190.0  7.4   67     5   1
#> 2  36.000   118.0  8.0   72     5   2
#> 3  12.000   149.0 12.6   74     5   3
#> 4  18.000   313.0 11.5   62     5   4
#> 5  23.125   169.5 14.3   56     5   5
#> 6  28.000   169.5 14.9   66     5   6
```

```
#> 7  23.000   299.0  8.6   65     5   7
#> 8  19.000    99.0 13.8   59     5   8
#> 9   8.000    19.0 20.1   61     5   9
#> 10 23.125   194.0  8.6   69     5  10
```

> **Warning**
>
> This approach helps to preserve the structure of the dataset and minimises data loss. However, it can introduce bias if the chosen statistic (e.g., mean or median) does not accurately represent the underlying data.

5.8.3.4 Flag Missing Data

You can create a new column to flag rows with missing values:

```
airquality_data %>% mutate(missing_flag = !complete.cases(.))
```

```
#>     Ozone Solar.R Wind Temp Month Day missing_flag
#> 1  41.000     190  7.4   67     5   1        FALSE
#> 2  36.000     118  8.0   72     5   2        FALSE
#> 3  12.000     149 12.6   74     5   3        FALSE
#> 4  18.000     313 11.5   62     5   4        FALSE
#> 5  23.125      NA 14.3   56     5   5         TRUE
#> 6  28.000      NA 14.9   66     5   6         TRUE
#> 7  23.000     299  8.6   65     5   7        FALSE
#> 8  19.000      99 13.8   59     5   8        FALSE
#> 9   8.000      19 20.1   61     5   9        FALSE
#> 10 23.125     194  8.6   69     5  10        FALSE
```

> **Tip**
>
> Using `!complete.cases(.)` ensures the `missing_flag` column accurately identifies rows with missing values (`TRUE`) while marking rows without missing values as `FALSE`. This binary flag is useful for quickly filtering or inspecting incomplete data.

Alternatively, instead of a binary flag, you can add a column that shows the number of missing values in each row:

```
airquality_data$missing_count <- rowSums(is.na(airquality_data))

airquality_data
```

```
#>     Ozone Solar.R Wind Temp Month Day missing_count
#> 1  41.000     190  7.4   67     5   1             0
#> 2  36.000     118  8.0   72     5   2             0
#> 3  12.000     149 12.6   74     5   3             0
#> 4  18.000     313 11.5   62     5   4             0
#> 5  23.125      NA 14.3   56     5   5             1
#> 6  28.000      NA 14.9   66     5   6             1
#> 7  23.000     299  8.6   65     5   7             0
#> 8  19.000      99 13.8   59     5   8             0
#> 9   8.000      19 20.1   61     5   9             0
#> 10 23.125     194  8.6   69     5  10             0
```

> **Tip**
>
> Adding a column like `missing_count` provides a numeric indicator of the total number of missing values in each row. This approach is particularly helpful when you need to assess the extent of missingness across the dataset or prioritise rows for further investigation

> **Reflection Question**
>
> Consider the potential biases introduced by removing all rows with missing values. In which scenarios would you prefer imputation over removal?

5.8.4 Practice Quiz 5.3

Question 1:

Which function in R checks if there are any missing values in an object?

a) `is.na()`

b) `anyNA()`

c) `complete.cases()`

d) `na.omit()`

Question 2:

Which approach removes any rows containing `NA` values?

a) `na.omit()`

b) `replace_na()`

c) `complete.cases()`

d) `anyNA()`

Question 3:

If you decide to impute missing values in a column using the median, what is one potential advantage of using the median rather than the mean?

a) The median is always easier to compute.

b) The median is more affected by outliers than the mean.

c) The median is less influenced by extreme values and may provide a more robust estimate.

d) The median will always be exactly halfway between the min and max values.

Question 4:

How would you replace all `NA` values in character columns with `"Unknown"`?

a)

```
mutate(across(where(is.character), ~ replace_na(., "Unknown")))
```

b)

```
mutate_all(~ replace_na(., "Unknown"))
```

c)

```
mutate(across(where(is.character), na.omit))
```

d)

```
mutate(across(where(is.character), replace(. == NA, "Unknown")))
```

Question 5:

What does the `anyNA()` function return?

a) The number of missing values in an object.

b) `TRUE` if there are any missing values in the object; otherwise, `FALSE`.

c) A logical vector of missing values in each row.

d) A subset of the data frame without missing values.

Question 6:

You want to create a new column in a data frame that flags rows with missing values as `TRUE`. Which code achieves this?

a) `df$new_col <- !complete.cases(df)`
b) `df$new_col <- complete.cases(df)`

c) `df$new_col <- anyNA(df)`

d) `df$new_col <- is.na(df)`

Question 7:

Before removing rows with missing values, what is an important consideration?

a) Whether the missing values are randomly distributed across the data.

b) Whether the dataset is stored in a data frame.

c) Whether missing values exist in every column.

d) Whether the missing values are encoded as `NA`.

Question 8:

Why should the proportion of missing data in a row or column be considered before removing it?

a) Removing rows or columns with minimal missing values may lead to excessive data loss.

b) Columns with missing values cannot be visualized.

c) Rows with missing values are always irrelevant.

d) Rows with missing values should never be analyzed.

Question 9:

If a dataset has 50% missing values in a column, what is a common approach to handle this situation?

a) Replace missing values with the column mean.

b) Remove the column entirely.

c) Replace missing values with zeros.

d) Leave the missing values as they are.

Question 10:

What does the following Tidyverse-style code do?

```
library(dplyr)

airquality_data <- airquality_data %>%
  mutate(Ozone = if_else(is.na(Ozone), mean(Ozone, na.rm = TRUE), Ozone))
```

a) Removes rows where `Ozone` is missing.

b) Replaces missing values in `Ozone` with the mean of the column.

c) Flags rows where `Ozone` is missing.

d) Deletes the `Ozone` column if it has missing values.

See the Solution to Quiz 5.3

5.8.5 Exercise 5.3.1: Handling Missing Data in the Television Company Dataset

This exercise will test your data cleaning skills using the `data-tv-company.csv` dataset, located in the `r-data` directory. If you do not already have the file, you can download it from Google Drive.

Dataset Metadata

This dataset was collected by a small television company seeking to understand the factors that influence how viewers rate the company. It includes viewer ratings and related measures. The variables in the dataset are as follows:

- **regard**: Viewer rating of the television company (higher ratings indicate greater regard).
- **gender**: The gender with which the viewer identifies.
- **views**: The number of views.

- **online**: The number of times bonus online material was accessed.
- **library**: The number of times the online library was browsed.
- **Show1** to **Show4**: Scores for four different shows.

Tasks

1. **Importing and Inspecting Data**
    - Locate the `data-tv-company.csv` file in the `r-data` directory.
    - Import the data into your R environment.
    - Inspect the dataset for any missing values.
2. **Strategies for Dealing with Missing Data**
    - Demonstrate at least four different methods for handling missing data in this dataset.
    - Apply these methods to the imported data.
    - Evaluate the methods and select the best approach based on your analysis.

See the Solution to Exercise 5.3.1

5.9 Reflective Summary

In **Lab 5**, you developed essential skills in data transformation with R, including::

- **The Pipe Operator `|>`**: You learned to link functions in a logical sequence, enhancing code readability.
- **Data Manipulation with `dplyr`**: You used core verbs—`select()`, `filter()`, `mutate()`, `arrange()`, and `summarise()`—to reshape and refine your data.
- **Summarisation and Grouping**: By using `group_by()` and `summarise()`, you aggregated data to uncover patterns and derive insights.
- **Handling Missing Data**: You learned to detect and manage missing values, ensuring the quality of your analysis.

These techniques form a crucial step in the data analysis pipeline, enabling you to approach complex datasets with confidence and produce meaningful insights.

What's Next?

In the next lab, we will explore tidy data and joins. You will learn to reshape datasets and merge diverse data sources, converting raw data into structured, analysis-ready formats. This will pave the way for deeper insights and more efficient workflows.

6 Tidy Data and Joins

6.1 Introduction

Welcome to Lab 6! In Lab 5, you explored advanced data transformation techniques—including the pipe operator, key `dplyr` functions, and methods for handling missing data. Now, we will move on to organising and transforming your data using tidy data principles.

Tidy data involves arranging your dataset so that each variable occupies its own column, each observation its own row, and each type of observational unit its own table. Although tidying data may require some upfront effort, this structure greatly simplifies subsequent analysis and improves efficiency. With the tidyverse tools at your disposal, you will spend less time cleaning data and more time uncovering insights.

If you have ever struggled with reshaping datasets, merging data from multiple sources, or applying complex transformations, this lab is for you. The skills you acquire here are essential for real-world data analysis and will significantly enhance your proficiency in R.

6.2 Learning Objectives

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

- **Understand Tidying Data:**
  Comprehend what tidy data is and why it is crucial for effective analysis.
- **Reshape Data:**
  Convert datasets between wide and long formats using functions like `pivot_longer()` and `pivot_wider()` to prepare your data for analysis and visualisation.
- **Separate and Unite Columns:**
  Use `separate()` to split columns and `unite()` to combine them, thereby improving the structure and usability of your dataset.
- **Combine Datasets Effectively:**
  Master various join operations in `dplyr` to merge datasets seamlessly, resolving issues such as mismatched keys and duplicates.

By completing this lab, you will be equipped to transform messy, real-world datasets into analysis-ready formats, paving the way for insightful visualisations, robust models, and confident data analysis.

6.3 Prerequisites

Before you begin this lab, you should have:

- Completed **Lab 5** or have a basic understanding of data manipulation.
- A working knowledge of R's data structures.
- Familiarity with the tidyverse packages, particularly dplyr.

6.4 The Principles of Tidy Data

Tidy data is all about maintaining a clear and consistent structure. It is organised in such a way that:

- Each variable forms a column.
- Each observation forms a row.
- Each cell contains a single value.

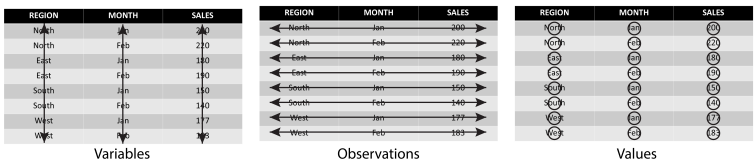


Figure 6.1: Understanding Tidy Data: Variables, Observations, and Values

As illustrated in Figure 6.1, this structure makes it much easier to analyse and visualise data. Tools like tidyr and dplyr work best with data that adheres to these principles, minimising errors and streamlining the analysis process. In essence, tidy data provides the foundation for efficient data manipulation and reproducible research.

6.5 Experiment 6.1: Reshaping Data with `tidyr`

Since most real-world datasets are not tidy, they are often collected in a wide format, which can complicate detailed analysis. For instance, consider a sales manager who records monthly sales figures for each region in separate columns. Table 6.1 illustrates this wide-format layout:

Table 6.1: Regional Sales Data in Wide Format

Month	North	East	South	West
Jan	200	180	150	177
Feb	220	190	140	183
Mar	210	200	160	190

For many analyses—such as trend identification and visualisation—it is advantageous to reshape this data into a long format where each row represents a unique combination of region and month. Table 6.2 shows the transformed data:

Table 6.2: Regional Sales Data in Long Format

Region	Month	Sales
North	Jan	200
North	Feb	220
North	Mar	210
East	Jan	180
East	Feb	190
East	Mar	200
South	Jan	150
South	Feb	140
South	Mar	160
West	Jan	177
West	Feb	183
West	Mar	190

This long format is particularly useful for time series analysis and visualisation, as it consolidates all pertinent information into common columns. The `tidyr` package, a core component of the tidyverse, provides efficient functions to perform these transformations.

6.5.1 Reshaping Data from Wide to Long Using `pivot_longer()`

The `pivot_longer()` function gathers multiple columns into two key-value columns. This process is essential when columns represent variables that you wish to convert into rows. Figure 6.2 illustrates this transformation:

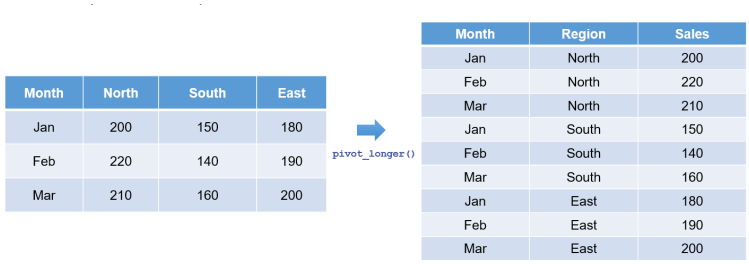


Figure 6.2: Transforming Data from Wide to Long Format Using `pivot_longer()`

The general syntax of `pivot_longer()` is:

```
pivot_longer(
  cols = <columns to reshape>,
  names_to = <column for variable names>,
  values_to = <column for values>
)
```

Where:

- `cols`: A tidy-select expression specifying the columns to pivot into longer format.
- `names_to`: A character vector indicating the name of the new column (or columns) that will store the original column names in `cols`.
- `values_to`: A string specifying the name of the new column that will store the corresponding values.

Example: Converting a Wide Dataset to Long Format

In the following example, we will convert the data in Table 6.1 into a long format using `pivot_longer()`:

```
library(tidyverse)

# Create the data frame
sales_data_wide <- data.frame(
```

```
  Month = c("Jan", "Feb", "Mar"),
  North = c(200, 220, 210),
  East  = c(180, 190, 200),
  South = c(150, 140, 160),
  West  = c(177, 183, 190)
)

sales_data_wide
```

```
#> # A tibble: 3 x 5
#>   Month North  East South  West
#>   <chr> <dbl> <dbl> <dbl> <dbl>
#> 1 Jan     200   180   150   177
#> 2 Feb     220   190   140   183
#> 3 Mar     210   200   160   190
```

```
sales_data_long <- sales_data_wide |>
  pivot_longer(
    cols = c(North, East, South, West),
    names_to = "Region",
    values_to = "Sales"
  )

sales_data_long
```

```
#> # A tibble: 12 x 3
#>    Month Region Sales
#>    <chr> <chr>  <dbl>
#>  1 Jan   North    200
#>  2 Jan   East     180
#>  3 Jan   South    150
#>  4 Jan   West     177
#>  5 Feb   North    220
#>  6 Feb   East     190
#>  7 Feb   South    140
#>  8 Feb   West     183
#>  9 Mar   North    210
#> 10 Mar   East     200
#> 11 Mar   South    160
#> 12 Mar   West     190
```

The dataset is now prepared for further analysis, such as plotting sales figures by region or calculating averages.

6.5.2 Reshaping Data from Long to Wide Using `pivot_wider()`

Conversely, the `pivot_wider()` function reverses the process by spreading key-value pairs into separate columns. This approach is particularly useful when you need to summarise data or create compact tables. Figure 6.3 demonstrates this transformation:

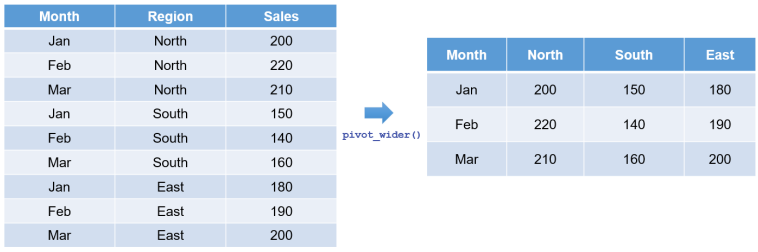


Figure 6.3: Transforming Data from Long to Wide Format Using `pivot_wider()`

The general syntax of `pivot_wider()` is:

```
pivot_wider(
  names_from = <column for new column names>,
  values_from = <column for values>
)
```

Where:

- `names_from`: A tidy-select expression specifying the column(s) from which to derive the new column names.
- `values_from`: A tidy-select expression specifying the column(s) from which to retrieve the corresponding values.

Example: Converting a Long Dataset to Wide Format

Using the long-format sales data (`sales_data_long`), created earlier, we can convert it back to a wide format:

```
sales_data_long
```

```
#> # A tibble: 12 x 3
#>    Month Region Sales
#>    <chr> <chr>  <dbl>
#>  1 Jan   North    200
#>  2 Jan   East     180
#>  3 Jan   South    150
#>  4 Jan   West     177
#>  5 Feb   North    220
#>  6 Feb   East     190
#>  7 Feb   South    140
#>  8 Feb   West     183
#>  9 Mar   North    210
#> 10 Mar   East     200
#> 11 Mar   South    160
#> 12 Mar   West     190
```

```
sales_data_wide <- sales_data_long |>
  pivot_wider(
    names_from = "Region",
    values_from = "Sales"
  )

sales_data_wide
```

```
#> # A tibble: 3 x 5
#>   Month North  East South  West
#>   <chr> <dbl> <dbl> <dbl> <dbl>
#> 1 Jan     200   180   150   177
#> 2 Feb     220   190   140   183
#> 3 Mar     210   200   160   190
```

This transformation demonstrates how effortlessly data can be reshaped to meet the requirements of various analytical approaches.

6.5.3 Practice Quiz 6.1

Question 1:

Consider the following data frame:

```r
sales_data_wide <- data.frame(
  Month = c("Oct", "Nov", "Dec"),
  North = c(180, 190, 200),
  East  = c(177, 183, 190),
  South = c(150, 140, 160),
  West  = c(200, 220, 210)
)
```

Which function would you use to convert this wide-format dataset into a long-format dataset?

a) `pivot_long()`

b) `pivot_wider()`

c) `separate()`

d) `pivot_longer()`

Question 2:

In the `pivot_longer()` function, if you want the original column names ("North", "East", "South", "West") to appear in a new column called "Region", which argument would you use?

a) `cols`

b) `names_to`

c) `values_to`

d) `names_prefix`

Question 3:

Given the same data frame, which argument in `pivot_longer()` specifies the name of the new column that stores the sales figures?

a) `names_to`

b) `values_to`

c) `cols`

d) `values_drop_na`

Question 4:

What is the primary purpose of using `pivot_wider()`?

a) To convert long-format data into wide format

b) To combine two data frames

c) To split a column into multiple columns

d) To remove missing values

Question 5:

If you apply `pivot_longer()` on `sales_data_wide` without specifying `cols`, what is likely to happen?

a) All columns will be pivoted, including the identifier column "Month", leading to an undesired result.

b) Only numeric columns will be pivoted.

c) The function will automatically ignore non-numeric columns.

d) An error will be thrown immediately.

Question 6:

Which package provides the functions `pivot_longer()` and `pivot_wider()`?

a) dplyr

b) tidyr

c) ggplot2

d) readr

Question 7:

The functions `pivot_longer()` and `pivot_wider()` are inverses of each other, allowing you to switch between wide and long formats easily.

a) True

b) False

Question 8:

In the following code snippet, what is the role of the `cols = c(North, East, South, West)` argument?

```
sales_data_long <- sales_data_wide |>
  pivot_longer(
    cols = c(North, East, South, West),
    names_to = "Region",
    values_to = "Sales"
  )
```

a) It tells `pivot_longer()` which columns to keep as they are.

b) It specifies the columns to be pivoted from wide to long format.

c) It defines the new column names for the output.

d) It removes missing values from these columns.

Question 9:

After reshaping the data to long format, which of the following is a potential advantage?

a) Easier to merge with other datasets

b) Simplified time series analysis and visualisation

c) Increased redundancy in the dataset

d) Reduced number of observations

Question 10:

Which of the following best describes tidy data?
a) Each variable forms a column and each observation a row
b) Data is merged from multiple sources
c) Data is automatically plotted
d) Missing values are always removed

See the Solution to Quiz 6.1

6.5.4 Exercise 6.1.1: Tidying the Pew Religion and Income Survey Data

This exercise tests your data tidying skills using the `religion_income` dataset, which can be found in the `r-data` directory. If you do not already have the file, you may download it from Google Drive.

Dataset Metadata

The Pew Research Trust's 2014 survey compiled this dataset to examine the relationship between religious affiliation and income in the United States. It shows the proportions of sampled individuals from each religious tradition who fall into various income bands (e.g., `<$10k`, `$10k-$30k`, etc.). The dataset includes:

- **religion**: The name of the religion.
- **Income Range Columns**: Multiple columns corresponding to various income brackets (e.g. `<$10k`, `$10k-$30k`, etc.). Each column represents the number of respondents falling within that income category. Some columns may also include descriptors such as "Don't know/refused".

Tasks

1. **Importing and Inspecting Data**
    - Locate the `religion_income.csv` file in the `r-data` directory.
    - Import the data into your R environment.
    - Inspect the structure of the dataset and familiarise yourself with its variables.
2. **Data Tidying**
    - Reshape the dataset so that the various income range columns are gathered into two new variables: one for the income range (e.g. `income_range`) and another for the corresponding number of respondents (e.g. `count`).
    - Rename the resulting columns as necessary to ensure they are clear and descriptive.
    - Create a summary table that displays the total number of respondents per income range across all religions.
    - Produce a bar plot to visualise the distribution of respondents across the different income ranges.

Happy tidying!

See the Solution to Exercise 6.1.1

6.6 Experiment 6.2: Splitting and Combining Columns

In this section, we will explore two essential functions from the tidyr package-`separate()` and `unite()`—that help restructure your data effectively.

6.6.1 Splitting Columns with `separate()`

The `separate()` function splits a single column into multiple columns. It offers several variations, each tailored to different splitting methods:

- `separate_wider_delim()`: Splits a column using a specified delimiter.
- `separate_wider_position()`: Splits a column at fixed widths.
- `separate_wider_regex()`: Splits a column using regular expressions.

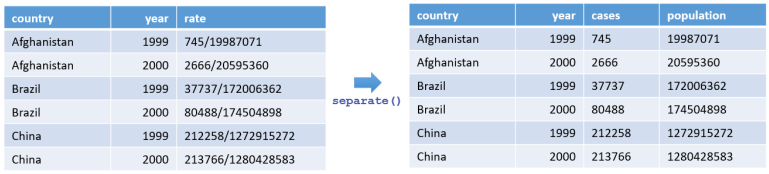


Figure 6.4: Splitting a Single Column into Multiple Columns Using `separate()`

The general syntax is:

```
separate(data, col = <column to split>, into = <new columns>, sep = <separator>, remove = <l
```

Where:

- `data`: The data frame to be transformed.
- `col`: The column to be split.
- `into`: A character vector specifying the names of the new columns.
- `sep`: A string or regular expression indicating where to split the column. If not specified, it defaults to splitting at non-alphanumeric characters.
- `remove`: A logical flag (default `TRUE`) indicating whether to remove the original column after splitting.
- `convert`: A logical flag (default `FALSE`); if `TRUE`, it automatically converts the new columns to appropriate data types using `type.convert()`.

Example 1:

Below is a data frame showing tuberculosis (TB) rates in Afghanistan, Brazil, and China from 1999 to 2000. The `"rate"` column contains both cases and population values combined by a slash (`/`).

```
library(tidyverse)

# Create the data frame
tb_cases <- tibble(
  country = c("Afghanistan", "Afghanistan", "Brazil", "Brazil", "China", "China"),
  year = c(1999, 2000, 1999, 2000, 1999, 2000),
  rate = c(
    "745/19987071", "2666/20595360", "37737/172006362", "80488/174504898",
    "212258/1272915272", "213766/1280428583"
  )
)

tb_cases
```

```
#> # A tibble: 6 x 3
#>   country      year rate
#>   <chr>       <dbl> <chr>
#> 1 Afghanistan  1999 745/19987071
#> 2 Afghanistan  2000 2666/20595360
#> 3 Brazil       1999 37737/172006362
#> 4 Brazil       2000 80488/174504898
#> 5 China        1999 212258/1272915272
#> 6 China        2000 213766/1280428583
```

Since the `"rate"` column combines both cases and population, we can split it into two separate columns, `"cases"` and `"population"`, using `separate_wider_delim()`:

```
tb_cases |>
  separate_wider_delim(cols = rate, names = c("cases", "population"), delim = "/")
```

```
#> # A tibble: 6 x 4
#>   country      year cases  population
#>   <chr>       <dbl> <chr>  <chr>
#> 1 Afghanistan  1999 745    19987071
#> 2 Afghanistan  2000 2666   20595360
#> 3 Brazil       1999 37737  172006362
```

```
#> 4 Brazil       2000 80488  174504898
#> 5 China        1999 212258 1272915272
#> 6 China        2000 213766 1280428583
```

> **!** Important
>
> This approach is particularly helpful when your data arrive in a combined format that needs to be split for meaningful analysis.

6.6.2 Combining Columns with `unite()`

Conversely, the `unite()` function merges multiple columns into one.

Figure 6.5: Combining Multiple Columns into One Using `unite()`

Its syntax is:

```
unite(data, col = <name of new column>, ..., sep = <separator>, remove = <logical flag>)
```

Where:

- `data`: The data frame to be transformed.
- `col`: The name of the new column that will contain the combined values.
- `...`: The columns to combine.
- `sep`: The separator to insert between values (defaults to `"_"` if not specified).
- `remove`: A logical flag (default `TRUE`) indicating whether to remove the original columns after merging.

Example 2:

Imagine your TB dataset has separate `"century"` and `"year"` columns that you want to combine into a single `"year"` column:

```r
# Create the data frame
tb_cases <- tibble(
  country = c("Afghanistan", "Afghanistan", "Brazil", "Brazil", "China", "China"),
  century = c("19", "20", "19", "20", "19", "20"),
  year = c("99", "00", "99", "00", "99", "00"),
  rate = c("745/19987071", "2666/20595360", "37737/172006362", "80488/174504898", "212258/12
)

tb_cases
```

```
#> # A tibble: 6 x 4
#>   country     century year  rate
#>   <chr>       <chr>   <chr> <chr>
#> 1 Afghanistan 19      99    745/19987071
#> 2 Afghanistan 20      00    2666/20595360
#> 3 Brazil      19      99    37737/172006362
#> 4 Brazil      20      00    80488/174504898
#> 5 China       19      99    212258/1272915272
#> 6 China       20      00    213766/1280428583
```

You can merge `"century"` and `"year"` into a single column, also named `"year"`, by specifying no separator:

```r
tb_cases |> unite(year, century, year, sep = "")
```

```
#> # A tibble: 6 x 3
#>   country     year  rate
#>   <chr>       <chr> <chr>
#> 1 Afghanistan 1999  745/19987071
#> 2 Afghanistan 2000  2666/20595360
#> 3 Brazil      1999  37737/172006362
#> 4 Brazil      2000  80488/174504898
#> 5 China       1999  212258/1272915272
#> 6 China       2000  213766/1280428583
```

> **!** Important
>
> This operation tidies your dataset by reducing two related columns into one, making future analysis more straightforward.

6.6.3 Practice Quiz 6.2

Question 1:

Given the following tibble:

```
tb_cases <- tibble(
  country = c("Brazil", "Brazil", "China", "China"),
  year = c(1999, 2000, 1999, 2000),
  rate = c("37737/172006362", "80488/174504898", "212258/1272915272", "213766/1280428583")
)
```

Which function would you use to split the `"rate"` column into two separate columns for cases and population?

a) `separate()`

b) `unite()`

c) `pivot_longer()`

d) `pivot_wider()`

Question 2:

Which argument in `separate()` allows automatic conversion of new columns to appropriate data types?

a) `remove`

b) `auto`
c) `convert`
d) `into`

Question 3:

Which function would you use to merge two columns into one, for example, combining separate "century" and "year" columns?

a) `separate()`

b) `unite()`

c) `pivot_longer()`

d) `pivot_wider()`

Question 4:

In the `separate()` function, what does the `sep` argument define?

a) The new column names

b) The delimiter at which to split the column

c) The data frame to be merged

d) The columns to remove

Question 5:

Consider the following data frame:

```
tb_cases <- tibble(
  country = c("Afghanistan", "Brazil", "China"),
  century = c("19", "19", "19"),
  year = c("99", "99", "99")
)
```

Which code correctly combines "century" and "year" into a single column "year" without any separator?

a) `tb_cases |> unite(year, century, year, sep = "")`

b) `tb_cases |> separate(year, into = c("century", "year"), sep = "")`

c) `tb_cases |> unite(year, century, year, sep = "_")`

d) `tb_cases |> pivot_longer(cols = c(century, year))`

Question 6:

When using `separate()`, how can you retain the original column after splitting it?

a) Set `remove = FALSE`

b) Set `convert = TRUE`

c) Use `unite()` instead

d) Omit the `sep` argument

Question 7:

Which variant of `separate()` would you use to split a column at fixed character positions?

a) `separate_wider_delim()`

b) `separate_wider_regex()`

c) `separate_wider_position()`

d) `separate()`

Question 8:

By default, the `unite()` function removes the original columns after combining them.

a) True

b) False

Question 9:

What is the main benefit of using `separate()` on a column that combines multiple data points (e.g. "745/19987071")?

a) It facilitates the conversion of string data into numeric data automatically.

b) It simplifies further analysis by splitting combined information into distinct, analysable components.

c) It merges the data with another dataset.

d) It increases data redundancy.

Question 10:

Which argument in `unite()` determines the character inserted between values when combining columns?

a) `separator`

b) `sep`

c) `col`

d) `delimiter`

See the Solution to Quiz 6.2

6.6.4 Exercise 6.2.1: Transforming the Television Company Dataset

This exercise tests your data cleaning skills using the `television-company-data.csv` dataset, which can be found in the `r-data` directory. If you do not already have the file, you may download it from Google Drive.

Dataset Metadata

This dataset was gathered by a small television company aiming to understand the factors influencing viewer ratings of the company. It contains viewer ratings and related metrics. The variables are as follows:

- `regard`: Viewer rating of the television company (higher ratings indicate greater regard).
- `gender`: The gender with which the viewer identifies.
- `views`: The number of views.
- `online`: The number of times bonus online material was accessed.
- `library`: The number of times the online library was browsed.
- `Shows`: Scores for four different shows, each separated by a comma.

Tasks

1. **Importing and Inspecting Data**
    - Locate the `television-company-data.csv` file in the `r-data` directory.
    - Import the data into your R environment.
2. **Data Tidying**
    - Create four new columns from the `Shows` column, naming them `Show1` through `Show4`.
    - Create a new variable, `mean_show`, calculated as the mean of `Show1` to `Show4`.
    - Examine how the mean show scores vary by `gender`.

See the Solution to Exercise 6.2.1

Table 6.3: Relational Keys in Action: Linking Datasets with Primary and Foreign Keys

(a) Students Table

student_id	name	age
1	Alice	20
2	Bob	22
3	Charlie	21

(b) Exam Scores Table

student_id	score
1	85
2	90
4	78

6.7 Experiment 6.3: Combining Datasets with Joins

When you're working with real-world data, it's common to have information scattered across multiple tables. In order to get a full picture and answer the questions you are interested in, you often need to merge these datasets. In this section, we will dive into how you can accomplish this in R using joins, a fundamental concept in both database management and data analysis.

6.7.1 The Role of Keys

Before you begin joining datasets, it's crucial to understand what keys are and why they matter. In relational databases, a key is a column (or set of columns) that uniquely identifies each row in your dataset. When you join tables, keys determine how rows are matched across your datasets.

- **Primary Key**: Think of this as the unique identifier for each record in a table. For example, in students table (Table 6.3a), the `student_id` is the primary key because it uniquely identifies each student.
- **Foreign Key**: This is a column in another table that links back to the primary key of the first table. In our case, the `student_id` in an exam scores table (Table 6.3b) would act as a foreign key, connecting scores to the corresponding students.

To illustrate, consider these two datasets in Table 6.3:

In the Students table, the `student_id` column is the primary key that uniquely identifies each student. In the Exam Scores table, `student_id` is used as a foreign key to refer back to the Students table. We will use these two tables to illustrate the different types of joins.

6.7.2 Types of Joins

Joins are the backbone of relational data analysis, and the `dplyr` offers a family of join functions to merge tables, each with a specific way of handling matches and non-matches. They all follow a similar pattern:

```
join_function(x, y, by = "key_column")
```

Where:

- `x`: The first table (the "left" table).
- `y`: The second table (the "right table").
- `by`: The column(s) to match on.

Let's explore the main types: inner, left, right, and full joins. We wll use `students` and `scores` to see how each one works.

6.7.2.1 Inner Join

An inner join keeps only the rows where the key matches in both tables. If there's no match, the row gets dropped. It's the strictest join—think of it as finding the overlap between two circles in a Venn diagram.

The syntax is:

```
inner_join(x, y, by = "key_column")
```

Example 1: Matching Students and Scores

Suppose you are a teacher who only wants to see data for students who are enrolled and took the exam. Using Tables 6.3a and 6.3b:

```
students <- data.frame(student_id = 1:3, name = c("Alice", "Bob", "Charlie"), age = 20:22)

scores <- data.frame(student_id = c(1, 2, 4), score = c(85, 90, 78))

# Inner join
students |> inner_join(scores, by = "student_id")
```

```
#>   student_id  name age score
#> 1          1 Alice  20    85
#> 2          2   Bob  21    90
```

Tip

What's Happening:

- Alice (student_id 1) and Bob (student_id 2) appear because they're in both tables.
- Charlie (student_id 3) is missing—he didn't take the exam, so there's no match in scores.
- Student_id 4 is also out—it's in `scores` but not `students`.

This join is perfect when you need complete data from both sides, like calculating averages for students with scores.

Note

Think about how the result might change if we used `left_join()` instead?

6.7.2.2 The Left Join

A left join keeps every row from the left table (`x`) and brings along matching rows from the right table (`y`). If there's no match, you get `NA` in the columns from `y`.

The syntax is:

```
left_join(x, y, by = "key_column")
```

Example 2: Report Cards for All Students

Imagine you are preparing report cards and need every student listed, even if they missed the exam:

```
# Left join
students |> left_join(scores, by = "student_id")
```

```
#>   student_id    name age score
#> 1          1   Alice  20    85
#> 2          2     Bob  21    90
#> 3          3 Charlie  22    NA
```

Tip

What's Happening:

- All three students from students are here because it's the left table.
- Alice and Bob have their scores (85 and 90).
- Charlie gets an NA for score—he's enrolled but didn't take the test.
- Student_id 4 from scores is excluded—it's not in the left table.

Use this when the left table is your priority, like ensuring every student gets a report card.

Warning

Consider what happens if the `id` column contains duplicates in either `df1` or `df2`. This could result in more rows than expected in the joined dataset.

6.7.2.3 The Right Join

A right join flips it around: it keeps all rows from the right table (`y`) and matches them with the left table (`x`), filling in `NA` where needed.

The syntax is:

```
right_join(x, y, by = "key_column")
```

Example 3: All Exam Scores

Now, suppose the exam office wants every score reported, even for students not in your class list:

```
# Right join

students |> right_join(scores, by = "student_id")
```

```
#>   student_id  name age score
#> 1          1 Alice  20    85
#> 2          2   Bob  21    90
#> 3          4  <NA>  NA    78
```

Tip

What's Happening:

- All three scores from scores are included because it's the right table.
- Alice and Bob match up with their info from students.
- Student_id 4 has a score (78) but no name or age (NA)—they're not in students.

This is handy when the right table drives the analysis, like auditing all exam results.

6.7.2.4 The Full Join

A full join keeps everything—every row from both tables, matching where possible and using NA for gaps. It's the most inclusive join.

The syntax is:

```
full_join(x, y, by = "key_column")
```

Example 4: Complete Audit

For an audit, you want every student and every score, matched or not:

```
# Full join

students |> full_join(scores, by = "student_id")
```

```
#>   student_id    name age score
#> 1          1   Alice  20    85
#> 2          2     Bob  21    90
#> 3          3 Charlie  22    NA
#> 4          4    <NA>  NA    78
```

Tip

What's Happening:

- Every student_id from both tables is here.
- Alice and Bob are fully matched.

- Charlie has no score (NA in score).
- Student_id 4 has a score but no student info (NA in name and age).

This join is your go-to when you can't afford to lose any data, like reconciling records.

6.7.3 Joins with Different Key Names

What happens when the keys don't have the same name? Real datasets often throw this curveball—the primary key in one table might be `club_code`, while the foreign key in another is `group_id`. No worries—`dplyr` can still connect them.

Let's try a new example:

```
clubs <- tibble(club_code = c("A01", "B02", "C03"), club_name = c("Chess Club", "Robotics Tea

members <- tibble(member_id = 101:103, name = c("Alice", "Bob", "Charlie"), group_id = c("A0
```

Check them out:

```
clubs
```

```
#> # A tibble: 3 x 2
#>   club_code club_name
#>   <chr>     <chr>
#> 1 A01       Chess Club
#> 2 B02       Robotics Team
#> 3 C03       Art Society
```

```
members
```

```
#> # A tibble: 3 x 3
#>   member_id name    group_id
#>       <int> <chr>   <chr>
#> 1       101 Alice   A01
#> 2       102 Bob     B02
#> 3       103 Charlie D04
```

Here, `club_code` is the primary key in `clubs`, and `group_id` is the foreign key in `members`. They're different names, but their values (like "A01" and "B02") match up.

Example:

You are a club coordinator and want to see who's in which club, despite the naming mismatch. Use an inner join with a named vector in `by`:

```
clubs |> inner_join(members, by = c("club_code" = "group_id"))
```

```
#> # A tibble: 2 x 4
#>   club_code club_name     member_id name
#>   <chr>     <chr>             <int> <chr>
#> 1 A01       Chess Club          101 Alice
#> 2 B02       Robotics Team       102 Bob
```

Tip

What's Happening:

- The `by = c("club_code" = "group_id")` tells R to match `club_code` from `clubs` with `group_id` from members.
- Alice (A01) and Bob (B02) match their clubs.
- Charlie (D04) is dropped—there's no D04 in `clubs`.

This trick works with any join type. For instance, a full join would include Charlie and the unmatched C03 club:

```
clubs |> full_join(members, by = c("club_code" = "group_id"))
```

```
#> # A tibble: 4 x 4
#>   club_code club_name     member_id name
#>   <chr>     <chr>             <int> <chr>
#> 1 A01       Chess Club          101 Alice
#> 2 B02       Robotics Team       102 Bob
#> 3 C03       Art Society          NA <NA>
#> 4 D04       <NA>                103 Charlie
```

Joins are your superpower for combining datasets in R. With `inner_join()`, you get the overlap; `left_join()` prioritizes the left table; `right_join()` favours the `right`; and `full_join()` keeps it all. Plus, you can handle differently named keys with a simple tweak to `by`.

6.7.4 Practice Quiz 6.3

Question 1:

Given the following data frames:

```
df1 <- data.frame(id = 1:4, name = c("Ezekiel", "Bob", "Samuel", "Diana"))

df2 <- data.frame(id = c(2, 3, 5), score = c(85, 90, 88))
```

Which join would return only the rows with matching `id` values in both data frames?
a) `left_join()`
b) `right_join()`
c) `inner_join()`
d) `full_join()`

Question 2:

Using the same data frames, which join function retains all rows from `df1` and fills unmatched rows with `NA`?

a) `left_join()`

b) `inner_join()`

c) `right_join()`

d) `full_join()`

Question 3:

Which join function ensures that all rows from `df2` are preserved, regardless of matches in `df1`?

a) `left_join()`

b) `inner_join()`

c) `full_join()`
d) `right_join()`

Question 4:

What does a full join return when applied to `df1` and `df2`?

a) Only matching rows

b) All rows from both data frames, with `NA` for unmatched entries

c) Only rows from `df1`

d) Only rows from `df2`

Question 5:

In a join operation, what is the purpose of the `by` argument?

a) It specifies the common column(s) used to match rows between the data frames

b) It orders the data frames

c) It selects which rows to retain

d) It converts keys to numeric values

Question 6:

If `df1` contains duplicate values in the key column, what is a likely outcome of an inner join with `df2`?

a) The joined data frame may contain more rows than either original data frame due to duplicate matches.

b) The join will remove all duplicates automatically.

c) The function will return an error.

d) The duplicate rows will be merged into a single row.

Question 7:

An inner join returns all rows from both data frames, regardless of whether there is a match.

a) True

b) False

Question 8:

Consider the following alternative key columns:

```r
df1 <- data.frame(studentID = 1:4, name = c("Alice", "Bob", "Charlie", "Diana"))

df2 <- data.frame(id = c(2, 3, 5), score = c(85, 90, 88))
```

How can you join these two data frames when the key column names differ?

a) Rename one column before joining.

b) Use `by = c("studentID" = "id")` in the join function.
c) Use an inner join without specifying keys.

d) Convert the keys to factors.

Question 9:

What is a 'foreign key' in the context of joining datasets?

a) A column in one table that uniquely identifies each row.

b) A column in one table that refers to the primary key in another table.
c) A column that has been split into multiple parts.

d) A column that is combined using `unite()`.

Question 10:

Which join function would be most appropriate if you want a complete union of two datasets, preserving all rows from both?

a) `full_join()`
b) `inner_join()`

c) `left_join()`

d) `right_join()`

See the Solution to Quiz 6.3

6.7.5 Exercise 6.3.1: Relational Analysis with the NYC Flights 2013 Dataset

This exercise tests your relational analysis skills using the `nycflights13` dataset, which is available as an R package. To access it, install and load the package.

Dataset Metadata

The `nycflights13` dataset contains information on all 336,776 outbound flights from New York City airports (Newark Liberty International Airport (EWR), John F. Kennedy International Airport (JFK) and LaGuardia Airport (LGA)) in 2013, compiled from various sources, including the U.S. Bureau of Transportation Statistics. It is designed to explore the relationships between flight details, aircraft information, airport data, weather conditions and airline carriers. The dataset comprises five main tables:

- **flights:** Details of each flight, including departure and arrival times, delays and identifiers such as `tailnum` (aircraft tail number), `origin` (departure airport), `dest` (destination airport) and `carrier` (airline code). This table contains 336,776 rows and 19 columns.
- **planes:** Information about aircraft, such as the manufacturer, model and year built, linked by the `tailnum` key. This table contains 3,322 rows and 9 columns.
- **airports:** Data on airports, including location and name, linked by the `faa` code (used in `flights$origin` and `flights$dest`). This table contains 1,458 rows and 8 columns.
- **weather:** Hourly weather data for New York City airports, linked by `origin` and `time_hour`. This table contains 26,115 rows and 15 columns.
- **airlines:** Airline names and their carrier codes, linked by `carrier`. This table contains 16 rows and 2 columns.

Tasks

1. **Importing and Inspecting Data**
    - Install and load the `nycflights13` package in your R environment.
    - Access the `flights` and `planes` tables, and inspect their structure to familiarise yourself with their variables.
    - Identify the common key (`tailnum`) that links flights and planes, and note any potential mismatches (for example, flights without corresponding plane data).
2. **Relational Analysis with Joins**
    - Perform an `inner_join` between `flights` and `planes` using the `tailnum` key. How many rows are in the result, and why might this differ from the number of rows in `flights`?
    - Perform a `left_join` between `flights` and `planes` using the `tailnum` key. Compare the number of rows with the original `flights` table, and explain what happens to flights without matching plane data.

- Perform a `right_join` between `flights` and `planes` using the `tailnum` key. How does this differ from the `left_join` result, and what does it reveal about planes not used in flights?
- Perform a `full_join` between `flights` and `planes` using the `tailnum` key. Describe how this result combines information from both tables, including cases with no matches.
- Create a summary table showing the number of flights per aircraft manufacturer (from `planes$manufacturer`) after performing a `left_join`. Handle missing values appropriately (for example, label flights with no plane data as "Unknown").
- Produce a bar plot to visualise the distribution of flights across the top five aircraft manufacturers based on your summary table.

Happy joining!

See the Solution to Exercise 6.3.1

6.8 Reflective Summary

In **Lab 6**, you have acquired advanced data transformation skills essential for effective data analysis:

- **Reshaping Data**: You learned to convert datasets between wide and long formats using `pivot_longer()` and `pivot_wider()`. Mastering these techniques is foundational for conducting time series analyses and creating visualisations.
- **Separating and Uniting Columns:** You explored how to split a single column into multiple columns with `separate()` and combine several columns into one with `unite()`, thereby enhancing the structure of your data.
- **Combining Datasets:** You became familiar with various join operations in **dplyr**, enabling you to merge datasets seamlessly and manage issues such as mismatched keys and duplicates.

> What's Next?
>
> In the next lab, we will delve into data visualisation where will transform raw data into a visual language that reveals patterns, highlights trends, and conveys stories in ways that are easy to understand and interpret.

7 Data Visualisation

7.1 Introduction

In Lab 7, we embark on an exploration into the transformative realm of data visualisation using R. Our approach is twofold. First, we introduce ggplot2—a powerful package built on the Grammar of Graphics that enables you to craft professional, layered, and highly customisable visualisations. Second, we examine Base R graphics, a built-in solution that requires no additional packages and is perfectly suited for quick, exploratory plots. Although ggplot2 offers exceptional flexibility and elegance, Base R graphics remain indispensable for rapid analysis; they are straightforward, intuitive, and ideally suited to simple or preliminary visualisations.

Through this lab, you will learn how to effectively map data variables to visual properties, build complex plots with ggplot2, and harness the power of R's native plotting functions. This dual approach empowers you to choose the most appropriate method for your analytical needs.

7.2 Learning Objectives

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

- **Understand the Grammar of Graphics Framework:**
  Grasp how ggplot2's structured, layered approach enables the step-by-step construction of complex plots.
- **Create a Range of Visualisations with ggplot2:**
  Develop scatter plots, bar charts, histograms, boxplots, and more to represent your data effectively.
- **Customise Visual Elements in ggplot2:**
  Adjust themes, colours, labels, scales, and facets to enhance both clarity and visual appeal.
- **Utilise Base R Graphics for Exploratory Analysis:**
  Construct quick, function-based plots using built-in functions such as `plot()`, `hist()`, `boxplot()`, `barplot()`, and `pie()`, and apply customisations directly through graphical parameters.

- **Integrate Data Manipulation with Visualisation:**
  Combine data preparation tools like `dplyr` with both ggplot2 and Base R graphics to develop seamless and insightful workflows.

By completing this lab, you will become proficient in visualising data and communicating your findings effectively. Let us now transform raw data into impactful stories!

7.3 What is Data Visualization?

Data visualisation is both an art and a science—it is the practice of representing data through graphical means, such as charts, graphs, and maps. By transforming numerical or textual information into visual formats, we can uncover patterns, trends, and insights that might be hidden in raw data. This process breathes life into data, turning abstract numbers into compelling stories that are easy to understand and share.

Figure 7.1: Data Scientist Analyzing Large-Scale Data[^lab7-1].

In today's data-driven world, the ability to visualise data effectively is an essential skill across various industries—be it data science, finance, education, or healthcare. As the volume and complexity of data continue to grow, visualisation provides the means to make sense of it all and to share insights in a compelling and accessible manner.

Visual representations often prove more effective than descriptive statistics or tables for analysing data, as they allow us to:

- **Identify Patterns and Trends**: Spot relationships within the data that may not be immediately apparent.

- **Understand Distributions**: Clearly see how data is spread out, where concentrations or gaps exist.
- **Detect Outliers**: Quickly identify data points that deviate markedly from the rest of the dataset.
- **Communicate Insights**: Present findings in a manner that is both engaging and easy for diverse audiences to grasp.

By leveraging data visualisation, we enhance our capacity to analyse complex datasets and to communicate our discoveries with clarity.

7.4 Importance of Data Visualisation

Data visualisation plays a pivotal role in the analytical process for several reasons:

1. **Simplifies Complex Data**:
   Large datasets can be overwhelming when viewed in their raw form. Visualisation distils and structures this data, rendering it comprehensible at a glance. For instance, a line chart can succinctly illustrate trends over time that would be challenging to discern from a mere table of numbers.
2. **Reveals Patterns and Trends**:
   Visual tools help in identifying relationships within the data, such as correlations between variables or changes over time. This often leads to the generation of new insights and hypotheses—for example, a scatter plot may reveal a positive correlation between hours studied and exam scores.
3. **Supports Decision Making**:
   Visual evidence provides a persuasive basis for conclusions and recommendations. Decision-makers can rapidly grasp complex information and make informed choices, especially when key performance indicators are highlighted on a well-designed dashboard.
4. **Engages the Audience**:
   Visuals are naturally more engaging than raw numbers or text. They capture attention and enhance the persuasiveness of presentations by effectively conveying complex information in a digestible format.
5. **Facilitates Communication**:
   Visualisation transcends language barriers and simplifies the communication of intricate ideas. It fosters collaboration across disciplines by providing a common visual language.

7.5 Choosing the Right Visualization

Selecting the appropriate type of visualisation is critical for effectively communicating your data's story. Consider the following factors:

1. **Define Your Objective**:
   Determine whether you want to compare values, illustrate composition, understand distribution, or analyse trends over time.
2. **Understand Your Data**:
   Identify whether your variables are categorical, numerical, or time-series, and decide whether you are interested in relationships between variables, distributions, or outliers.
3. **Know Your Audience**:
   Tailor your visualisation to the background and expertise of your audience—ensure that the chosen visualisation is neither too complex nor overly simplistic.
4. **Consider Practical Constraints**:
   Think about the medium of presentation (digital, print, or verbal), and assess data quality and quantity. Large datasets may need aggregation, and lower-quality data may restrict the types of visualisations available.
5. **Aesthetics and Clarity**:
   Employ colour, shape, and size judiciously to enhance comprehension without overwhelming the viewer. Avoid clutter by keeping designs clean and focused.
6. **Ethical Representation**:
   Ensure that scales and representations are accurate and truthful, maintaining credibility and avoiding misleading interpretations.

By carefully weighing these considerations, you can select visualisations that not only effectively present your data but also resonate with your audience.

7.6 Types of Data Visualisation Analysis

Data visualisation can be broadly classified into three categories based on the number of variables analysed: univariate, bivariate, and multivariate.

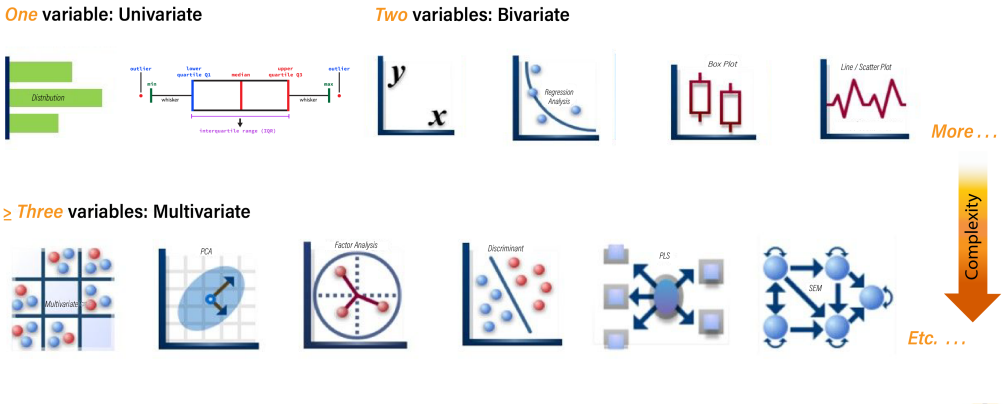


Figure 7.2: Types of Data Analysis: Univariate, Bivariate, and Multivariate Techniques

Each category offers a unique lens through which to interpret your data, allowing you to uncover different insights.

- **Univariate Analysis**
  Univariate analysis involves examining a single variable at a time. This approach helps you understand the distribution, central tendency, and variability of the data. For example, you might create a histogram to explore the age distribution within a population. This visualisation will reveal patterns such as skewness, clustering, and the presence of outliers, enabling you to gain a clear understanding of the variable's overall behaviour.
- **Bivariate Analysis**
  Bivariate analysis focuses on exploring the relationship between two variables. This type of analysis is particularly useful for identifying associations or correlations between variables. For instance, a scatter plot can be used to investigate the relationship between advertising spend and sales revenue. By plotting one variable against the other, you can observe trends, clusters, or even potential causal relationships, providing deeper insight into how the variables interact.
- **Multivariate Analysis**
  Multivariate analysis extends beyond two variables to examine the interplay among three or more variables simultaneously. This approach is invaluable when dealing with complex data sets where multiple factors may be influencing an outcome. For example, a bubble chart or parallel coordinates plot might be employed to evaluate factors affecting customer satisfaction by analysing service quality, price, and brand reputation all at once. This holistic view helps you capture the multidimensional nature of your data, revealing intricate relationships that might otherwise go unnoticed.

Understanding the type of analysis you wish to perform will guide you in selecting the most appropriate visualisation techniques to extract the insights you need.

7.7 Common Data Visualization Techniques

While there is an abundance of graphs and charts, mastering the core types will equip you with the essential tools for most analytical tasks.

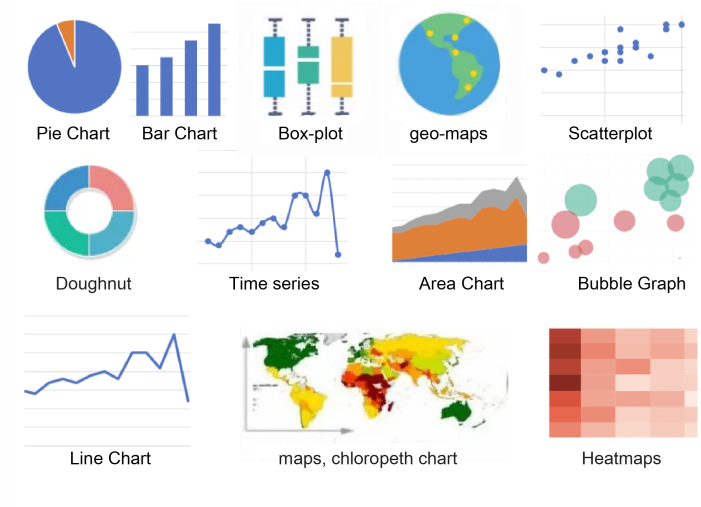


Figure 7.3: Common Data Visualisation Techniques

Let's now explore some of the most frequently used visualisation techniques.

7.7.1 Bar Chart

A bar chart represents categorical data using rectangular bars, with the length of each bar proportional to the corresponding value. Bars can be plotted vertically or horizontally.

When to Use:

- Comparing quantities across different categories.
- Illustrating rankings or frequencies.
- Displaying discrete data.

Example Uses:

- Comparing sales figures across regions.
- Showing student enrolment numbers across courses.
- Visualising survey responses by category.

Key Features:

- Categories typically appear on the x-axis, while values are on the y-axis.
- Bars are spaced to emphasise that the data is discrete.

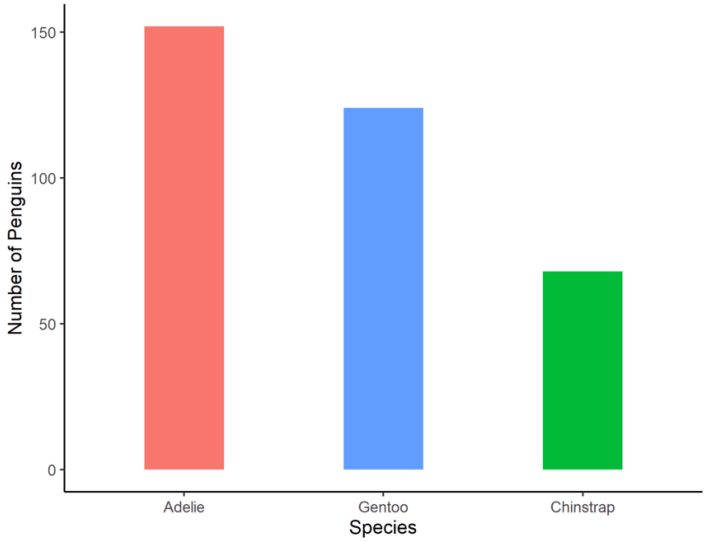


Figure 7.4: Bar Chart Illustration

7.7.2 Histogram

A histogram groups continuous data into bins, displaying the frequency of data points within each bin.

When to Use:

- Understanding the distribution of continuous data.
- Identifying patterns such as skewness, modality, or outliers.
- Assessing the probability distribution of a dataset.

Example Uses:

- Displaying the distribution of ages in a population.
- Showing the frequency of test scores among students.
- Analysing the spread of housing prices.

Key Features:

- The x-axis represents the continuous data divided into bins.
- The y-axis indicates frequency or count.
- Adjacent bars reflect the continuous nature of the data.

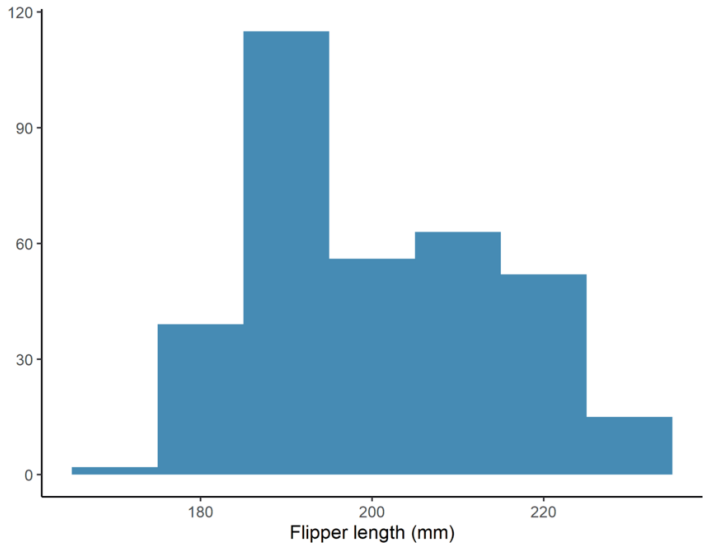


Figure 7.5: Histogram Illustration

7.7.3 Circular charts

A circular chart is a type of statistical graphic represented in a circular format to illustrate numerical proportions. A pie chart (Figure 7.6a) and a doughnut chart (Figure 7.6b) are examples of circular charts. Each slice or segment represents a category's contribution to the whole, making it easy to visualize parts of a whole in a compact form.

When to Use:

- Showing parts of a whole.
- Representing percentage or proportional data.
- Comparing categories within a dataset where the total represents 100%.
- When there are a limited number of categories (ideally less than six).

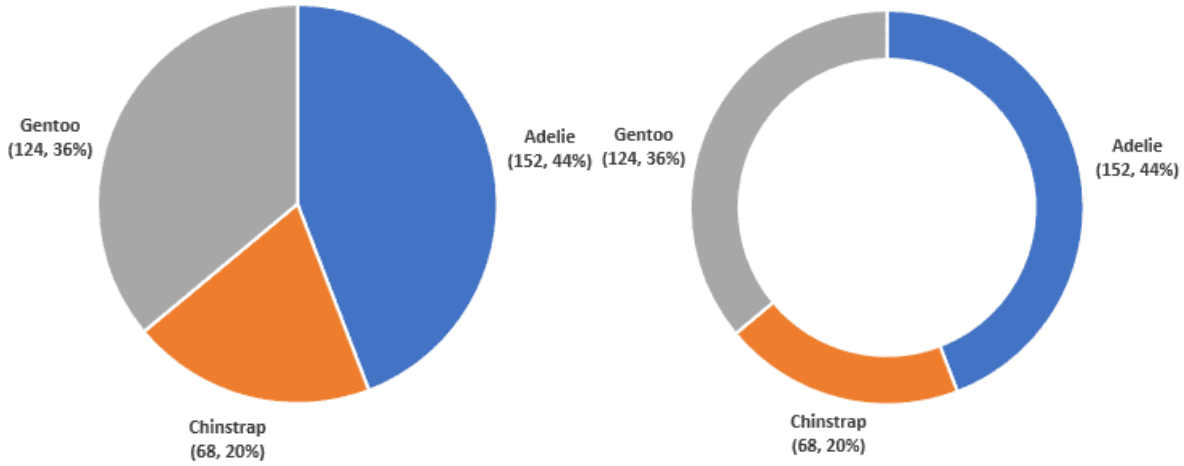
Example Uses:

- Displaying market share of different companies.
- Illustrating budget allocations across departments.

- Showing survey results for single-choice questions.

Key Features:

- The circle represents the entire dataset.
- Slices are proportional to each category's contribution.
- Doughnut charts provide additional central space for extra labelling or data.



(a) Pie Chart Illustration

(b) Doughnut Chart Illustration

Figure 7.6: Circular Statistical Charts

> **Note**
>
> Circular charts can be challenging to interpret with many small or similarly sized slices. In such cases, consider using bar charts or stacked charts for clarity.

7.7.4 Scatter Plot

A scatter plot uses Cartesian coordinates to display values for two numerical variables. Each point represents an observation, allowing you to explore relationships or correlations between variables.

When to Use:

- Exploring relationships or correlations between two continuous variables.
- Detecting patterns, trends, clusters, or outliers.

Example Uses:

- Examining the relationship between study hours and exam scores.
- Analysing the correlation between advertising spend and sales revenue.
- Investigating associations between temperature and energy consumption.

Key Features:

- One variable is plotted on the x-axis, and another on the y-axis.
- Data points are distributed in two-dimensional space.
- A trend line can be added to highlight the overall relationship.

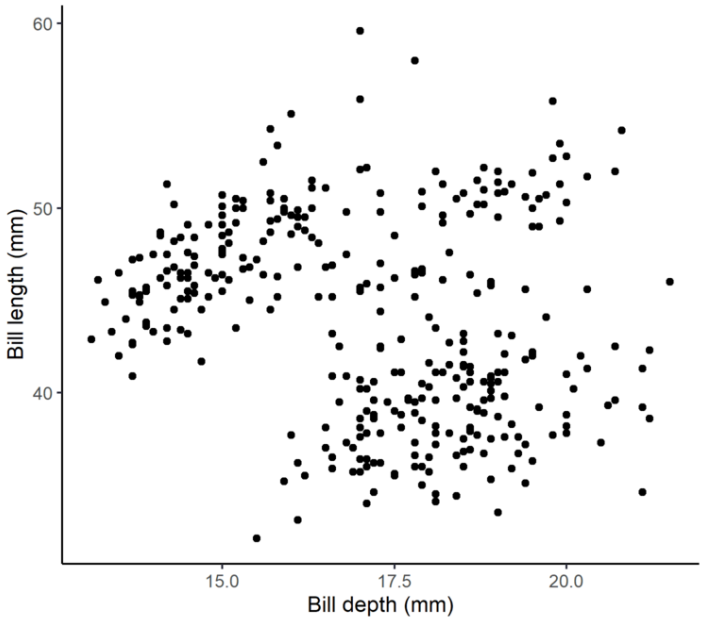


Figure 7.7: Scatter Plot Illustration

7.7.5 Box and Whisker Plot

A box plot summarises a dataset by displaying its median, quartiles, and potential outliers along a number line.

When to Use:

- Comparing distributions across different categories.
- Identifying central tendency, spread, and skewness.

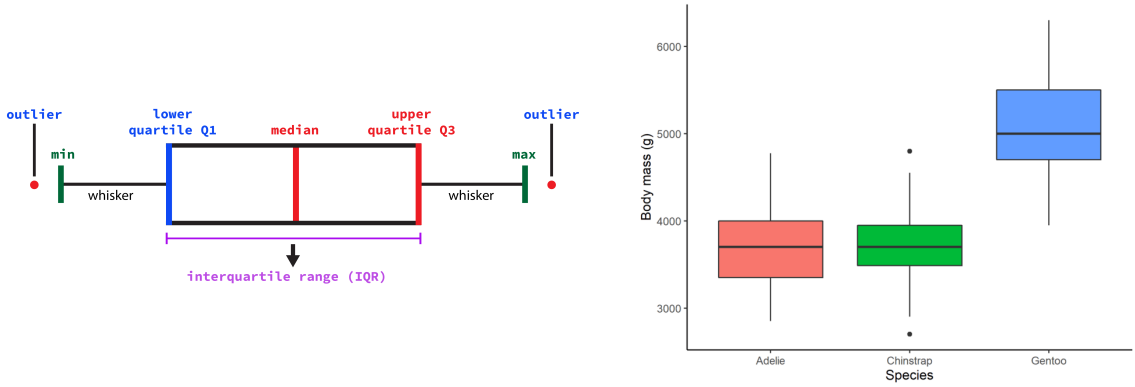
- Highlighting outliers.

Example Uses:

- Comparing test scores across classrooms..
- Analysing the spread of salaries in different industries
- Visualising delivery times from various suppliers.

Key Features:

- The box shows the interquartile range (IQR), from the first quartile (Q1) to the third quartile (Q3).
- The line inside the box indicates the median.
- Whiskers extend to the minimum and maximum values within 1.5 × IQR.
- Points outside the whiskers represent outliers.



(a) Boxplot Explaining the Statistical Components (b) Boxplots Comparing Body Mass Across Species

Figure 7.8: Box Plot Illustration

7.7.6 Line Chart

A line chart displays information as a series of data points called 'markers' connected by straight line segments. It is commonly used to visualise data that changes over time.

When to Use:

- Tracking changes or trends over intervals (e.g. time).
- Comparing multiple time series.

- Showing continuous data progression.

Example Uses:

- Monitoring stock prices.
- Showing temperature changes throughout the day.
- Illustrating website traffic trends.

Key Features:

- The x-axis represents time or sequential data.
- The y-axis shows quantitative values.
- Different lines may represent various categories or groups.

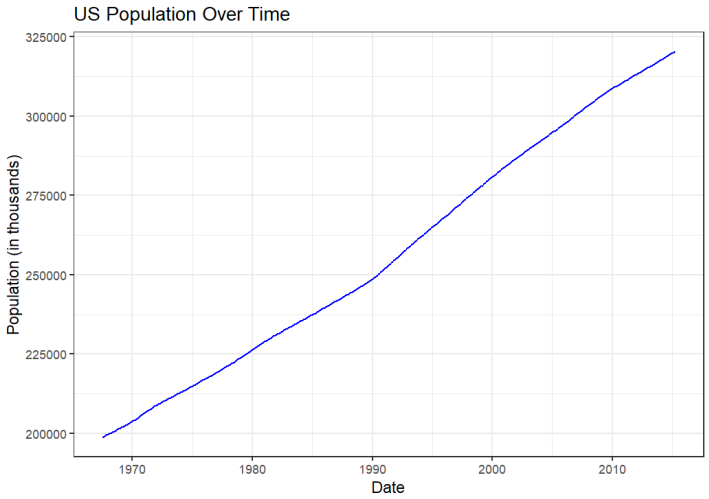


Figure 7.9: Line Chart Illustration

7.7.7 Areas chart

An area chart is similar to a line chart but fills the area below the line, emphasising the magnitude of values over time.

When to Use:

- Showing cumulative totals over time.
- Visualizing part-to-whole relationships.
- Comparing multiple quantities over time.

Example Uses:

- Displaying total sales over months.
- Visualising population growth.
- Comparing energy consumption by source.

Key Features:

- Time or sequential data on the x-axis.
- Quantitative values on the y-axis.
- The area beneath the line is filled, highlighting cumulative magnitude.

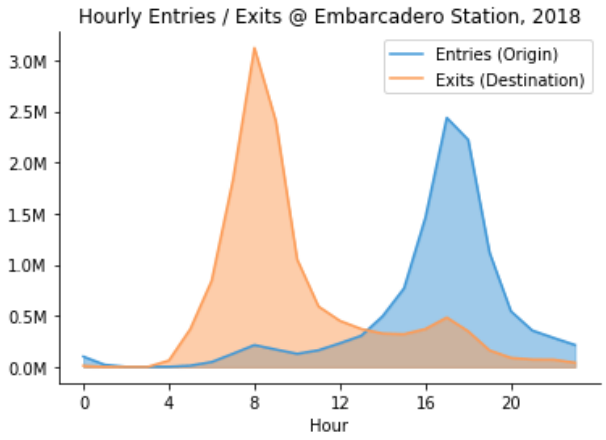


Figure 7.10: Area Chart Illustration

> **Tip**
>
> These core visualisation techniques form the foundation of data storytelling. Mastering them equips you with the tools necessary for most day-to-day analytical tasks. Remember, successful data visualisation is not only about selecting the right chart type, but also about achieving clarity, accuracy, and effective communication of your intended message.

7.8 Experiment 7.1: Data Visualization with `ggplot2`

R offers several systems for making graphs, but `ggplot2` stands out as one of the most elegant and versatile tools for creating high-quality visualisations. As part of the `tidyverse`, ggplot2 is

built upon the principles of the Grammar of Graphics—a systematic framework for describing and constructing graphs. This approach enables you to map data variables to visual properties in a coherent manner, allowing for the creation of a wide variety of statistical graphics.

Advantages of Using `ggplot2`

Let's explore the key benefits that make ggplot2 a preferred choice for data visualisation in R.

- **Consistency and Grammar**: Its structured, layered approach simplifies the process of building complex plots.
- **Customisation**: Nearly every aspect of a plot can be tailored to your specific needs.
- **Extension**: `ggplot2` is highly extensible, with additional packages such as `ggthemes`, `ggrepel`, and `plotly` available for further customisation and interactivity.
- **Professional Quality**: It produces publication-ready graphics that are ideal for reports, presentations, and academic papers.

7.8.1 Understanding the Grammar of Graphics

At its core, the Grammar of Graphics breaks down a graphic into semantic components:

1. **Data**: The dataset to be visualised.
2. **Aesthetics (`aes()`)**: The mappings between data variables and visual properties such as position, colour, size, shape, and transparency. For example:
    - `x`: Variable on the x-axis
    - `y`: Variable on the y-axis
    - `fill`: Fill color for areas like bars
    - `color`: Colour of points, lines, or areas
    - `size`: Size of points or lines
    - `shape`: Shape of points
    - `alpha`: Transparency level
    - `group`: Grouping variable for series of points
    - `facet`: For creating small multiples

3. **Geometric Objects (or geoms):** These are the visual building blocks in ggplot2 that define the type of plot being created. They determine how data points are visually represented by specifying the form of the plot elements. Each `geom_` function corresponds to a specific type of chart, allowing you to create a diverse range of plots. Examples include:

    - `geom_point()` for a scatter plot
    - `geom_smooth()` for adding trend lines or smoothing curves on a scatter plot
    - `geom_bar()` for a bar chart
    - `geom_col()` for bar charts using precomputed values
    - `geom_histogram()` for a histogram
    - `geom_boxplot()` for a boxplot
    - `geom_violin()` for a violin plot
    - `geom_freqpoly()` for a frequency polygon
    - `geom_line()` for a line chart
    - `geom_area()` for an area chart

Other Layers:

Additional layers enhance or modify your plot, allowing for customization and refinement:

4. **Statistical Transformations (`stats`)**: Computations applied to the data before plotting, such as summarising or smoothing data. For example, `stat_smooth()` adds a smoothed line to a scatter plot.
5. **Scales**: These control how data values are translated into aesthetic values, including axis ranges and colour gradients.
6. **Coordinate Systems**: Define the space in which the data is plotted, such as Cartesian coordinates (`coord_cartesian()`), polar coordinates (`coord_polar()`), or flipped coordinates (`coord_flip()` - to swap the x and y axes).
7. **Facets**: Create multiple panels (small multiples) by splittting the data based on one or more variables, using `facet_wrap()` or `facet_grid()`.
8. **Themes**: Customise non-data elements like backgrounds, gridlines, and text using prebuilt themes such as `theme_minimal()`, `theme_bw()`, or `theme_classic()`, or by modifying individual elements with `theme()`.
9. **Labels**: Add titles, axis labels, legend titles, and annotations with the `labs()` function.

7.8.2 Building Plots with `ggplot2`

To create a plot using ggplot2, start with the `ggplot()` function, specifying your data and aesthetic mappings, then add layers with the `+` operator. For example:

```
ggplot(data = <DATA>, aes(<MAPPINGS>)) +
  <GEOM_FUNCTION> +
  <OTHER_LAYERS>
```

Alternatively, you can use the pipe operator:

```
data |> ggplot(aes(<MAPPINGS>)) +
  <GEOM_FUNCTION> +
  <OTHER_LAYERS>
```

> **Tip**
>
> For a detailed breakdown of ggplot2 components, refer to Section 7.8.1.

7.8.3 Example Datasets

We begin our visualisation journey using five widely recognised datasets: `mtcars`, `iris`, `diamonds`, `economics`, and `heart`.

1. **The `mtcars` Dataset**

The built-in `mtcars` dataset contains information on fuel consumption and various automobile design and performance features for 32 car models from the 1973–74 era. It is ideal for exploring relationships such as those between weight, horsepower, and fuel efficiency.

```
library(tidyverse)

mtcars |> glimpse()
```

```
#> Rows: 32
#> Columns: 11
#> $ mpg  <dbl> 21.0, 21.0, 22.8, 21.4, 18.7, 18.1, 14.3, 24.4, 22.8, 19.2, 17.8,~
#> $ cyl  <dbl> 6, 6, 4, 6, 8, 6, 8, 4, 4, 6, 6, 8, 8, 8, 8, 8, 8, 4, 4, 4, 4, 8,~
#> $ disp <dbl> 160.0, 160.0, 108.0, 258.0, 360.0, 225.0, 360.0, 146.7, 140.8, 16~
#> $ hp   <dbl> 110, 110, 93, 110, 175, 105, 245, 62, 95, 123, 123, 180, 180, 180~
#> $ drat <dbl> 3.90, 3.90, 3.85, 3.08, 3.15, 2.76, 3.21, 3.69, 3.92, 3.92, 3.92,~
```

```
#> $ wt   <dbl> 2.620, 2.875, 2.320, 3.215, 3.440, 3.460, 3.570, 3.190, 3.150, 3.~
#> $ qsec <dbl> 16.46, 17.02, 18.61, 19.44, 17.02, 20.22, 15.84, 20.00, 22.90, 18~
#> $ vs   <dbl> 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0,~
#> $ am   <dbl> 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0,~
#> $ gear <dbl> 4, 4, 4, 3, 3, 3, 3, 4, 4, 4, 4, 3, 3, 3, 3, 3, 3, 4, 4, 4, 3, 3,~
#> $ carb <dbl> 4, 4, 1, 1, 2, 1, 4, 2, 2, 4, 4, 3, 3, 3, 4, 4, 4, 1, 2, 1, 1, 2,~
```

Before visualisation, it is important to transform the data appropriately. For example, we will convert variables such as `carb`, `cyl`, and `vs` to factors because they represent categorical information rather than continuous numerical values. Converting these variables to factors ensures they are treated as discrete categories during analysis and visualisation, allowing for more accurate grouping and comparison.

```
mtcars <- mtcars |> mutate(
  across(c(carb, cyl, vs), as.factor)
)
```

2. **The `iris` Dataset**

The built-in `iris` dataset includes measurements of sepal length, sepal width, petal length, and petal width for 150 iris flowers, representing three different species. This classic dataset is widely used for classification and clustering tasks.

```
iris |> glimpse()
```

```
#> Rows: 150
#> Columns: 5
#> $ Sepal.Length <dbl> 5.1, 4.9, 4.7, 4.6, 5.0, 5.4, 4.6, 5.0, 4.4, 4.9, 5.4, 4.~
#> $ Sepal.Width  <dbl> 3.5, 3.0, 3.2, 3.1, 3.6, 3.9, 3.4, 3.4, 2.9, 3.1, 3.7, 3.~
#> $ Petal.Length <dbl> 1.4, 1.4, 1.3, 1.5, 1.4, 1.7, 1.4, 1.5, 1.4, 1.5, 1.5, 1.~
#> $ Petal.Width  <dbl> 0.2, 0.2, 0.2, 0.2, 0.2, 0.4, 0.3, 0.2, 0.2, 0.1, 0.2, 0.~
#> $ Species      <fct> setosa, setosa, setosa, setosa, setosa, setosa, setosa, s~
```

3. **The `diamonds` Dataset**

Provided by ggplot2, the `diamonds` dataset includes detailed information on nearly 54,000 diamonds, such as carat, cut, colour, and clarity, prices. It is a valuable resource for exploring the relationships between diamond quality factors and price.

```
diamonds |> glimpse()
```

```
#> Rows: 53,940
#> Columns: 10
#> $ carat   <dbl> 0.23, 0.21, 0.23, 0.29, 0.31, 0.24, 0.24, 0.26, 0.22, 0.23, 0.~
#> $ cut     <ord> Ideal, Premium, Good, Premium, Good, Very Good, Very Good, Ver~
#> $ color   <ord> E, E, E, I, J, J, I, H, E, H, J, J, F, J, E, E, I, J, J, J, I,~
#> $ clarity <ord> SI2, SI1, VS1, VS2, SI2, VVS2, VVS1, SI1, VS2, VS1, SI1, VS1, ~
#> $ depth   <dbl> 61.5, 59.8, 56.9, 62.4, 63.3, 62.8, 62.3, 61.9, 65.1, 59.4, 64~
#> $ table   <dbl> 55, 61, 65, 58, 58, 57, 57, 55, 61, 61, 55, 56, 61, 54, 62, 58~
#> $ price   <int> 326, 326, 327, 334, 335, 336, 336, 337, 337, 338, 339, 340, 34~
#> $ x       <dbl> 3.95, 3.89, 4.05, 4.20, 4.34, 3.94, 3.95, 4.07, 3.87, 4.00, 4.~
#> $ y       <dbl> 3.98, 3.84, 4.07, 4.23, 4.35, 3.96, 3.98, 4.11, 3.78, 4.05, 4.~
#> $ z       <dbl> 2.43, 2.31, 2.31, 2.63, 2.75, 2.48, 2.47, 2.53, 2.49, 2.39, 2.~
```

4. **The `economics` Dataset**

This dataset, also from `ggplot2`, comprises US economic time series data, including variables like unemployment, personal savings rate, and inflation over several decades. It is excellent for time series analyses and exploring economic trends.

```
economics |> glimpse()
```

```
#> Rows: 574
#> Columns: 6
#> $ date     <date> 1967-07-01, 1967-08-01, 1967-09-01, 1967-10-01, 1967-11-01, ~
#> $ pce      <dbl> 506.7, 509.8, 515.6, 512.2, 517.4, 525.1, 530.9, 533.6, 544.3~
#> $ pop      <dbl> 198712, 198911, 199113, 199311, 199498, 199657, 199808, 19992~
#> $ psavert  <dbl> 12.6, 12.6, 11.9, 12.9, 12.8, 11.8, 11.7, 12.3, 11.7, 12.3, 1~
#> $ uempmed  <dbl> 4.5, 4.7, 4.6, 4.9, 4.7, 4.8, 5.1, 4.5, 4.1, 4.6, 4.4, 4.4, 4~
#> $ unemploy <dbl> 2944, 2945, 2958, 3143, 3066, 3018, 2878, 3001, 2877, 2709, 2~
```

5. **The `heart` Dataset**

Derived from the Framingham Heart Study, the `heart` dataset contains 5,209 observations with 17 variables that capture essential cardiovascular health information. Variables include participant status, cause of death (if applicable), age at CHD diagnosis, sex, age at the start of observation, height, weight, blood pressure measurements, metropolitan relative weight, smoking habits, serum cholesterol, and categorical statuses for cholesterol, blood pressure, weight, and smoking. This dataset is ideal for exploring risk factors associated with heart disease. The `heart.xlsx` file is available in the `r-data` directory. If you do not have it yet, you can download it from Google Drive.

```
library(readxl)
library(janitor)
heart <- read_xlsx("r-data/heart.xlsx", sheet = 1)
heart <- heart |> clean_names()
heart |> glimpse()
```

```
#> Rows: 5,209
#> Columns: 17
#> $ status         <chr> "Dead", "Dead", "Alive", "Alive", "Alive", "Alive", "Al~
#> $ death_cause    <chr> "Other", "Cancer", NA, NA, NA, NA, NA, "Other", NA, "Ce~
#> $ age_ch_ddiag   <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, 57, 55, 79,~
#> $ sex            <chr> "Female", "Female", "Female", "Female", "Male", "Female~
#> $ age_at_start   <dbl> 29, 41, 57, 39, 42, 58, 36, 53, 35, 52, 39, 33, 33, 57,~
#> $ height         <dbl> 62.50, 59.75, 62.25, 65.75, 66.00, 61.75, 64.75, 65.50,~
#> $ weight         <dbl> 140, 194, 132, 158, 156, 131, 136, 130, 194, 129, 179, ~
#> $ diastolic      <dbl> 78, 92, 90, 80, 76, 92, 80, 80, 68, 78, 76, 68, 90, 76,~
#> $ systolic       <dbl> 124, 144, 170, 128, 110, 176, 112, 114, 132, 124, 128, ~
#> $ mrw            <dbl> 121, 183, 114, 123, 116, 117, 110, 99, 124, 106, 133, 1~
#> $ smoking        <dbl> 0, 0, 10, 0, 20, 0, 15, 0, 0, 5, 30, 0, 0, 15, 30, 10, ~
#> $ age_at_death   <dbl> 55, 57, NA, NA, NA, NA, NA, 77, NA, 82, NA, NA, NA, NA,~
#> $ cholesterol    <dbl> NA, 181, 250, 242, 281, 196, 196, 276, 211, 284, 225, 2~
#> $ chol_status    <chr> NA, "Desirable", "High", "High", "High", "Desirable", "~
#> $ bp_status      <chr> "Normal", "High", "High", "Normal", "Optimal", "High", ~
#> $ weight_status  <chr> "Overweight", "Overweight", "Overweight", "Overweight",~
#> $ smoking_status <chr> "Non-smoker", "Non-smoker", "Moderate (6-15)", "Non-smo~
```

We transformed the variable `smoking_status` as ordered factor:

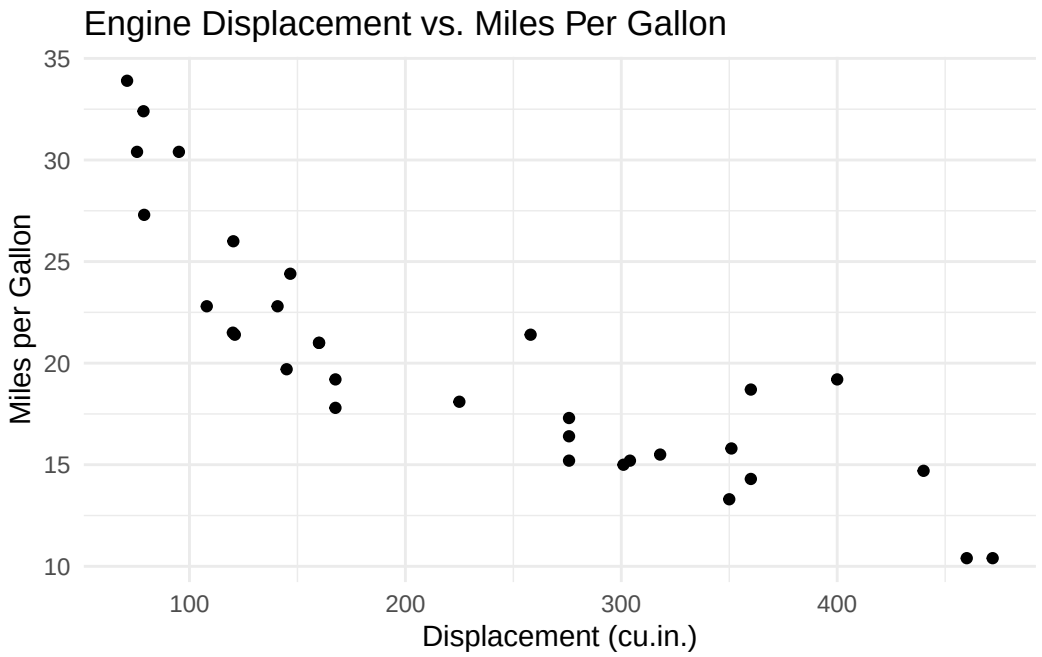
```
heart <- heart |> mutate(smoking_status = factor(smoking_status, levels = c(
  "Non-smoker", "Light (1-5)", "Moderate (6-15)", "Heavy (16-25)",
  "Very Heavy (> 25)"
)))
```

Creating Visualisations with ggplot2

7.8.4 Creating a Scatter Plot

Suppose you wish to explore the relationship between engine displacement and miles per gallon using the `mtcars` dataset. You can create a scatter plot as follows:

```r
ggplot(data = mtcars, aes(x = disp, y = mpg)) +
  geom_point() +
  labs(
    title = "Engine Displacement vs. Miles Per Gallon",
    x = "Displacement (cu.in.)",
    y = "Miles per Gallon"
  ) +
  theme_minimal()
```



In this example:

- **Data**: `mtcars`
- **Aesthetics**: `x = disp`, `y = mpg`
- **Geometric Object**: `geom_point()` adds points to represent each car.
- **Labels**: `labs()` adds a title and axis labels.
- **Theme**: `theme_minimal()` provides a clean, minimalist background.

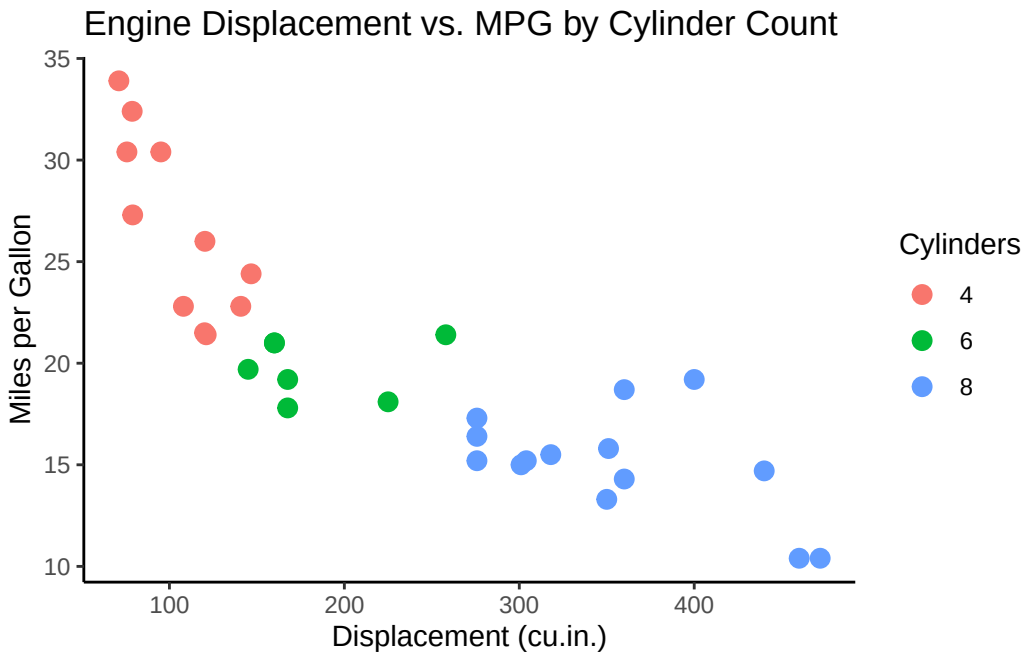
Interpretation

The scatter plot clearly shows an inverse relationship: cars with higher displacement generally have lower fuel efficiency (mpg).

7.8.4.1 Customizing Aesthetics and Geoms

For example, to distinguish between cylinder groups, you can map variable `cyl` to colour:

```
ggplot(data = mtcars, aes(x = disp, y = mpg, color = cyl)) +
  geom_point(size = 3) +
  labs(
    title = "Engine Displacement vs. MPG by Cylinder Count",
    x = "Displacement (cu.in.)",
    y = "Miles per Gallon",
    color = "Cylinders"
  ) +
  theme_classic()
```



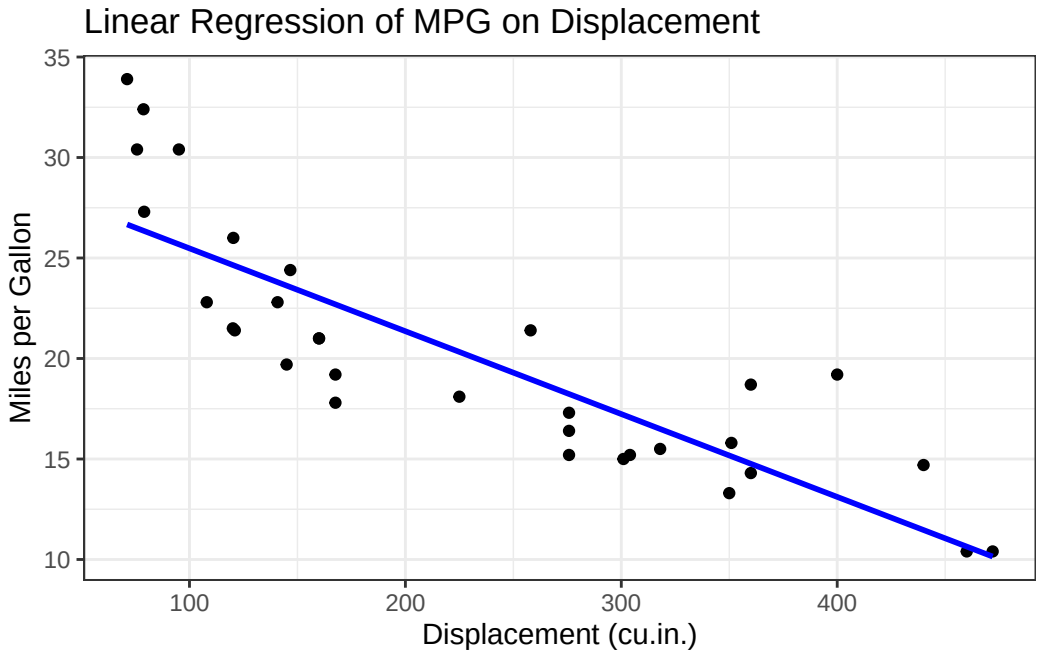
> **Tip**
>
> The `color` aesthetic maps the number of cylinders (`cyl`) to different colors, allowing you to distinguish groups within the data.

7.8.4.2 Incorporating Regression Line

To add a regression line to a scatter plot, you can use `geom_smooth()`:

```
ggplot(data = mtcars, aes(x = disp, y = mpg)) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE, color = "blue") +
  labs(
    title = "Linear Regression of MPG on Displacement",
    x = "Displacement (cu.in.)",
    y = "Miles per Gallon"
  ) +
  theme_bw()
```

```
#> `geom_smooth()` using formula = 'y ~ x'
```

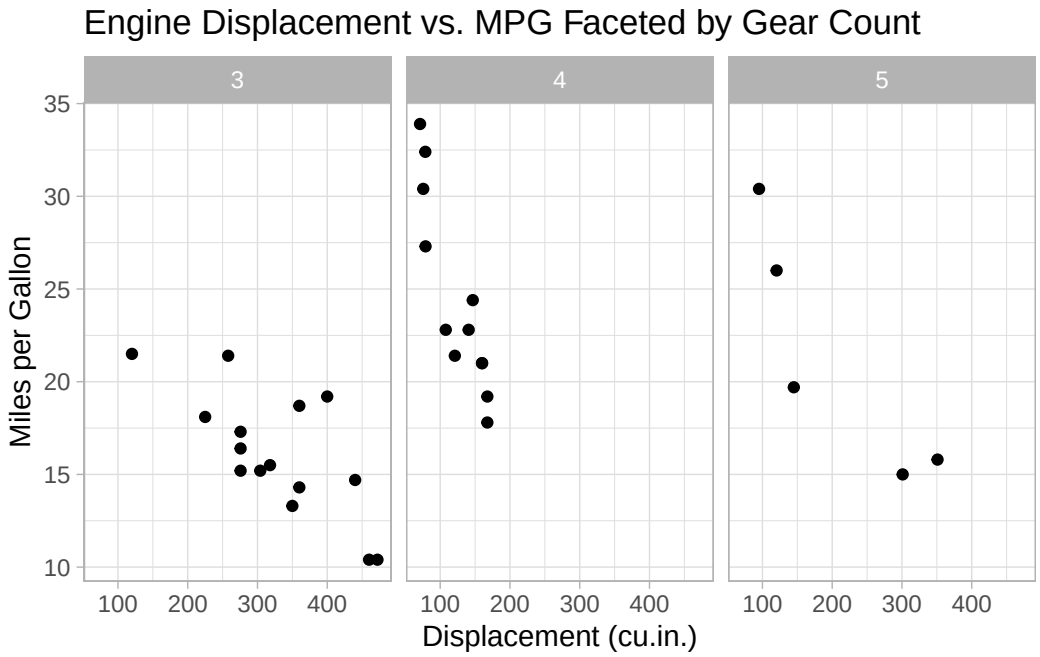


> **Tip**
>
> The `geom_smooth()` function adds a linear regression line to your scatter plot, offering valuable insight into the overall trend.

7.8.4.3 Faceting for Multi-Panel Plots

Faceting enables you to split data into subsets, displaying each in its own panel. For example, the following code facets the scatter plot by `gear`:

```
ggplot(data = mtcars, aes(x = disp, y = mpg)) +
  geom_point() +
  facet_wrap(~gear) +
  labs(
    title = "Engine Displacement vs. MPG Faceted by Gear Count",
    x = "Displacement (cu.in.)",
    y = "Miles per Gallon"
  ) +
  theme_light()
```

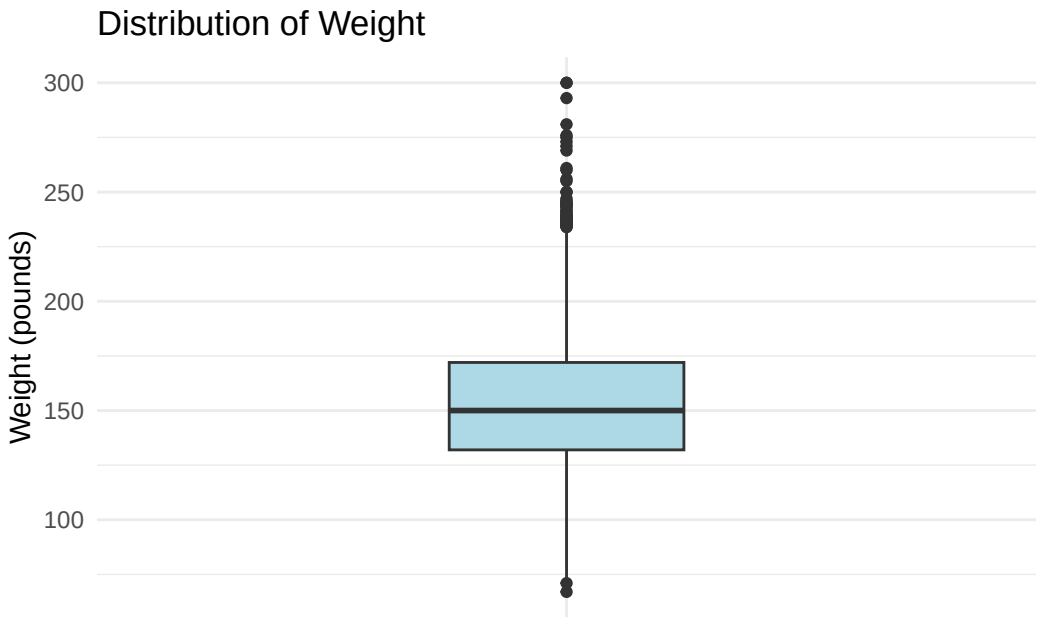


> **Tip**
>
> This code creates a scatter plot for each unique value of `gear`, allowing for easy comparison across groups.

7.8.5 Creating Boxplots

Suppose you wish to examine the distribution of `weight` in the `heart` dataset using a boxplot. You can create it as follows:

```r
# Boxplot of a single continuous variable
heart |>
  ggplot(aes(x = "", y = weight)) +
  geom_boxplot(fill = "lightblue", width = 0.3) +
  labs(
    title = "Distribution of Weight",
    x = NULL,
    y = "Weight (pounds)"
  ) +
  theme_minimal()
```



In this example::

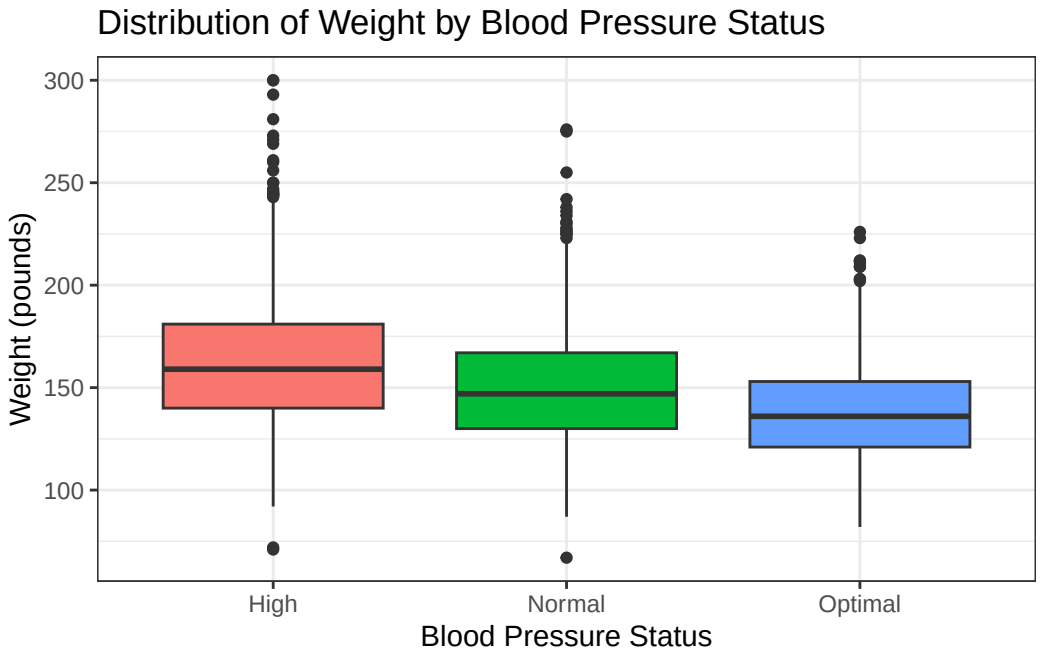
- **Data**: `heart` dataset.
- **Aesthetic**: `y = weight` defines the continuous variable.
- **Geometric Object**: `geom_boxplot()` draws a boxplot showing the median, quartiles, and outliers.
- **Labels**: `labs()` adds a title and y-axis label.
- **Theme**: `theme_minimal()` simplifies the background for readability.

Interpretation

The majority of participants weigh between approximately 130 and 190 lbs, with a median weight near 160 lbs. Multiple outliers above 220 lbs indicate a right-skewed distribution, suggesting that a subset of participants weighs substantially more than the central range.

You can also use a grouped boxplot to compare weight across different blood pressure statuses:

```
# Boxplot of a continuous variable grouped by a categorical variable
heart |>
  ggplot(aes(x = bp_status, y = weight, fill = bp_status)) +
  geom_boxplot(show.legend = FALSE) +
  labs(
    title = "Distribution of Weight by Blood Pressure Status",
    x = "Blood Pressure Status",
    y = "Weight (pounds)"
  ) +
  theme_bw()
```



In this example:

- **Data**: `heart` dataset.

- **Aesthetics**:
  - `x = bp_status`: Categorical grouping variable.
  - `y = weight`: Continuous variable to analyze.
  - `fill = bp_status`: Colours boxes by blood pressure status.
- **Geometric Object**: `geom_boxplot()` creates separate boxplots for each species.
- **Labels**: `labs()` clarifies the title and axes.

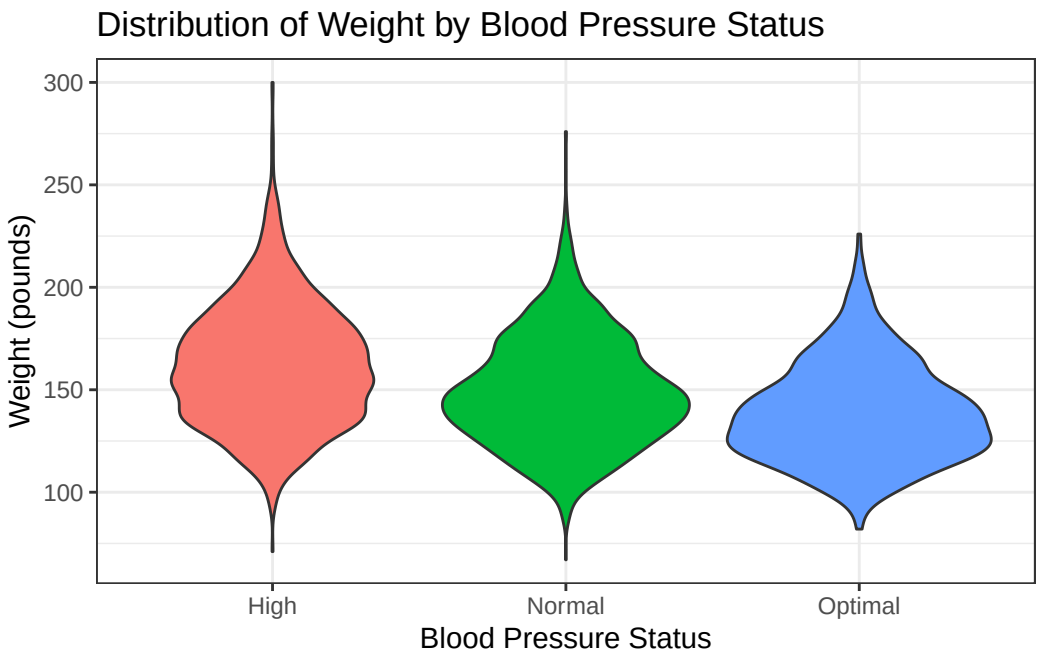
Interpretation

Based on the box plot, weight varies within each blood pressure status (High, Normal, Optimal), as shown by the spread of each box and whiskers; however, the location of these distributions are different, visually suggesting weight is not the same across all blood pressure statuses, with 'High' status tending towards higher weights and 'Optimal' status towards lower weights.

Note

Consider using `geom_violin()` instead of `geom_boxplot()` when you wish to display the full distribution of the data.

```
heart |>
  ggplot(aes(x = bp_status, y = weight, fill = bp_status)) +
  geom_violin(show.legend = FALSE) +
  labs(
    title = "Distribution of Weight by Blood Pressure Status",
    x = "Blood Pressure Status",
    y = "Weight (pounds)"
  ) +
  theme_bw()
```

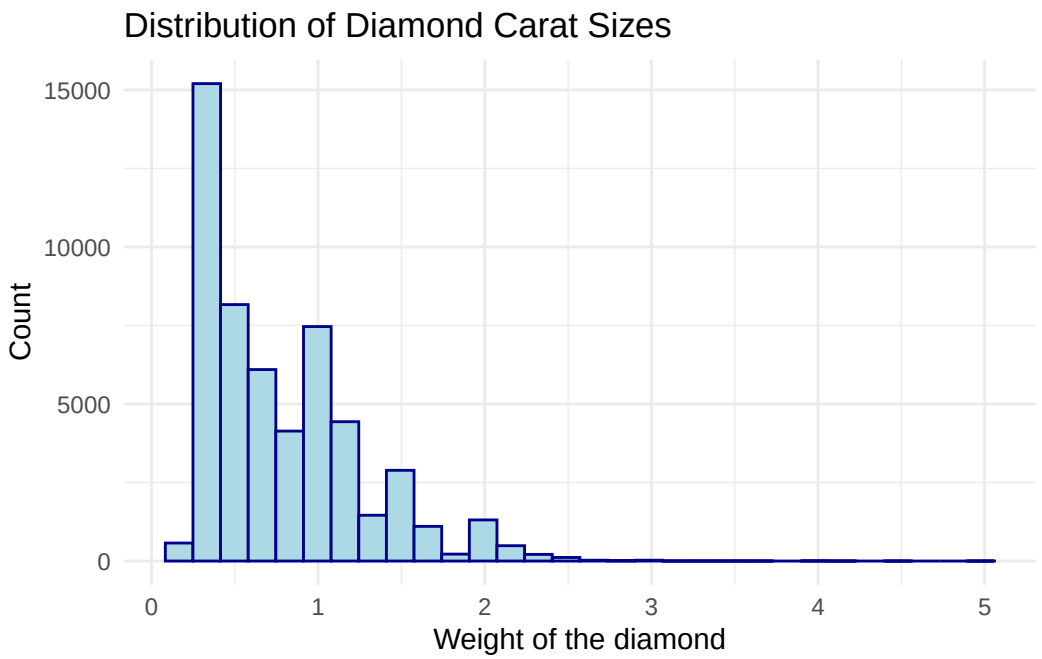


Violin plots not only summarise the quartiles and median but also reveal the density of the data, highlighting features such as multimodality or skewness, which can offer deeper insights into the underlying distribution.

7.8.6 Creating a Histogram

Suppose you want to plot the distribution of diamond carat sizes in the `diamond` dataset:

```
# Create a histogram of carat values
diamonds |>
  ggplot(aes(x = carat)) +
  geom_histogram(
    fill = "lightblue",
    color = "darkblue"
  ) +
  labs(
    title = "Distribution of Diamond Carat Sizes",
    x = "Weight of the diamond",
    y = "Count"
  ) +
  theme_minimal()
```



In this example:

- **Data**: `diamonds` dataset.
- **Aesthetic**: `x = carat` maps the continuous carat values to the x-axis.
- **Geometric Object**:
  - `geom_histogram()` bins the data and plots frequencies.
  - `fill` and `color` customise bar appearance.
- **Labels**: `labs()` adds a title and axis labels.
- **Theme**: `theme_minimal()` simplifies the background.

Interpretation

The histogram reveals a right-skewed distribution, indicating that smaller carat sizes (e.g., 0.2–1.0) are more common, while larger diamonds (e.g., >2.0 carats) are rare. Peaks around common sizes (e.g., 0.3, 0.7 carats) reflect market preferences or production trends.

Note

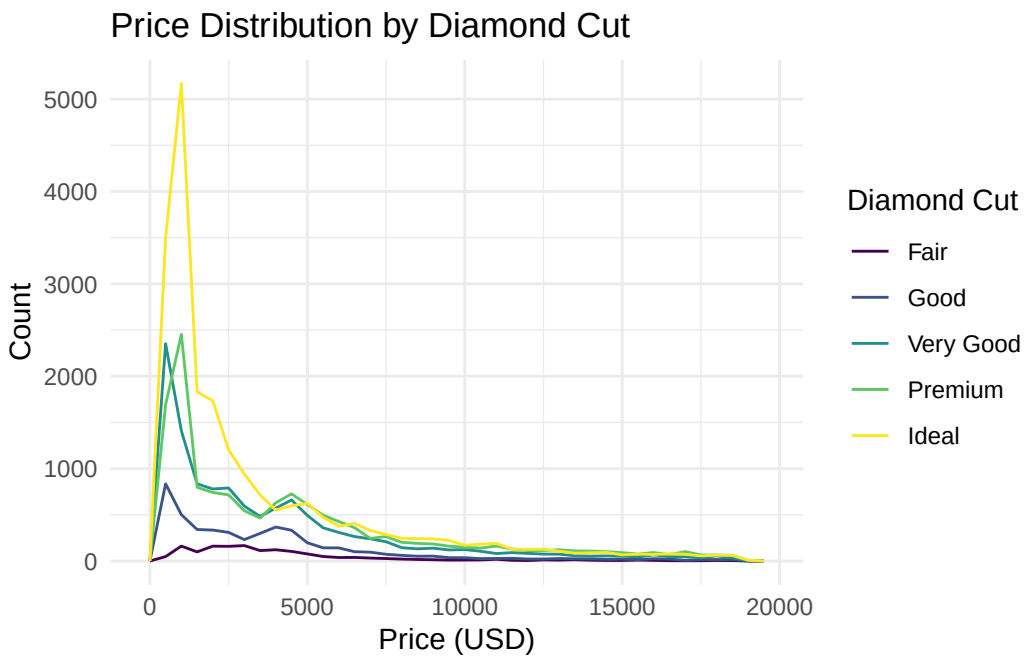
Adjust `binwidth` to balance detail and clarity. For example:

```
geom_histogram(
    binwidth = 0.1,
    fill = "lightblue",
    color = "darkblue"
  )
```

7.8.7 Creating Frequency Polygons

Frequency polygons are ideal for overlaying multiple distributions. For example, to compare diamond price distributions by cut:

```
diamonds |>
  ggplot(aes(x = price, colour = cut)) +
  geom_freqpoly(binwidth = 500) +
  labs(
    title = "Price Distribution by Diamond Cut",
    x = "Price (USD)",
    y = "Count",
    colour = "Diamond Cut"
  ) +
  theme_minimal()
```



In this example::

- **Data**: `diamonds` dataset.
- **Aesthetics**:
  - `x = price`: Maps price to the x-axis.
  - `colour = cut`: Colours lines by diamond cut (Fair, Good, etc.).
- **Geometric Object**:
  - `geom_freqpoly(binwidth = 500)` creates smoothed frequency lines.
- **Labels**: `labs()` clarifies axes and legend.

Interpretation

The frequency polygon highlights:

- **Price trends**: Higher cuts (e.g., Ideal, Premium) dominate mid-to-high price ranges.
- **Overlap**: Lower-quality cuts (Fair, Good) cluster in lower price brackets.
- **Granularity**: `binwidth = 500` balances noise and trend visibility.

> **Note**
>
> Use frequency polygons instead of stacked histograms when comparing subgroups—overlaid lines reduce visual clutter and improve comparability.

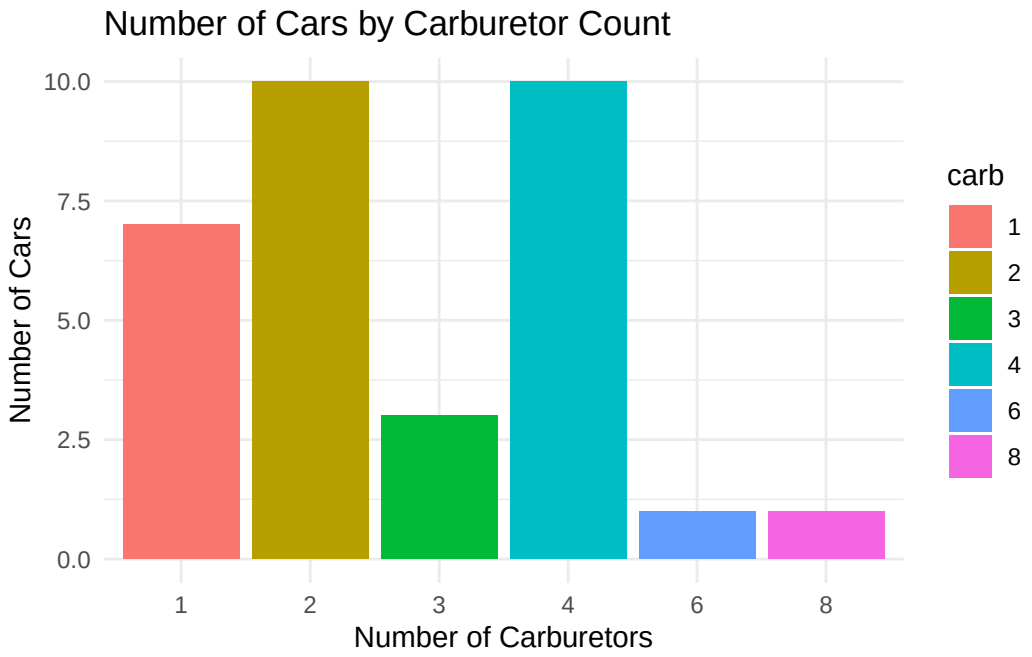
7.8.8 Creating Bar Charts

There are two primary geoms for creating bar charts in ggplot2: `geom_bar()` and `geom_col()`.

7.8.8.1 Bar Charts with Observation Counts

To create a bar chart where each bar represents the count of observations in a category, use the `geom_bar()` function. It automatically counts observations in each category and scales the bar heights accordingly. For example, to visualise the distribution of carburetor counts in the `mtcars` dataset:

```
mtcars |>
  ggplot(aes(x = carb, fill = carb)) +
  geom_bar() +
  labs(
    title = "Number of Cars by Carburetor Count",
    x = "Number of Carburetors",
    y = "Number of Cars"
  ) +
  theme_minimal()
```



In this example::

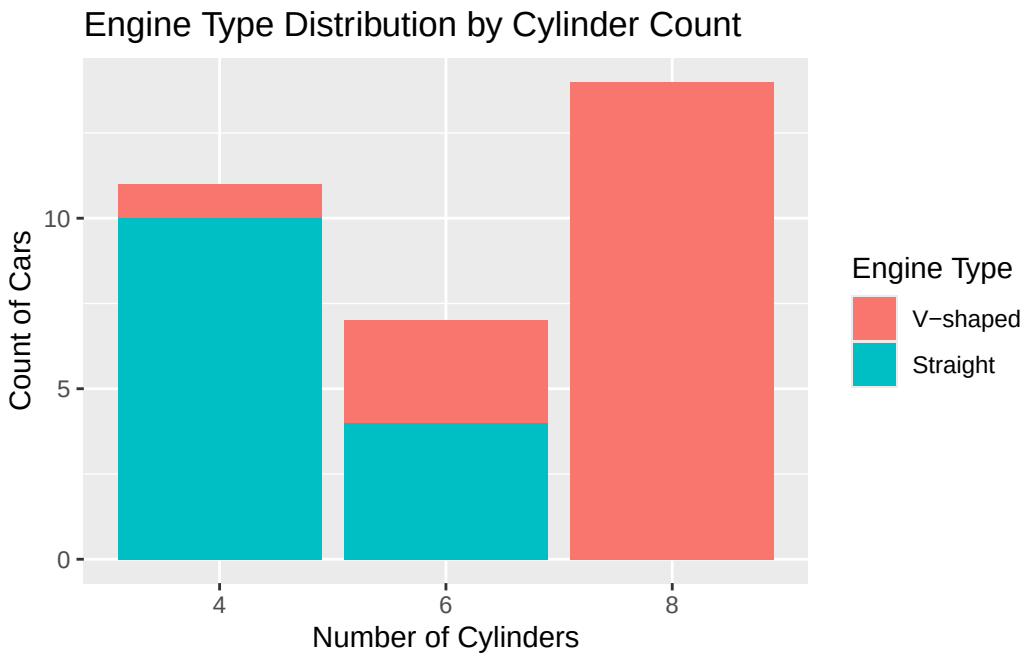
- **Data**: The `mtcars` dataset.
- **Aesthetics**: `x = carb` defines categories; `fill = carb` colors bars by carburetor count.
- **Geometric Object**: `geom_bar()` generates bars with heights proportional to counts. `show.legend = FALSE` removes redundant legend.
- **Labels**: `labs()` adds descriptive titles and axis labels.
- **Theme**: `theme_minimal()` simplifies the background for readability.

7.8.8.2 Stacked and Clustered Bar Charts

Stacked bar charts are useful for showing subgroup distributions. For example, to visualise engine type (`vs`) distribution across cylinder counts (`cyl`):

```
mtcars |> ggplot(aes(x = cyl, fill = vs)) +
  geom_bar() +
  labs(
    title = "Engine Type Distribution by Cylinder Count",
    x = "Number of Cylinders",
    y = "Count of Cars",
    fill = "Engine Type"
```

```
  ) +
  scale_fill_discrete(labels = c("V-shaped", "Straight"))
```



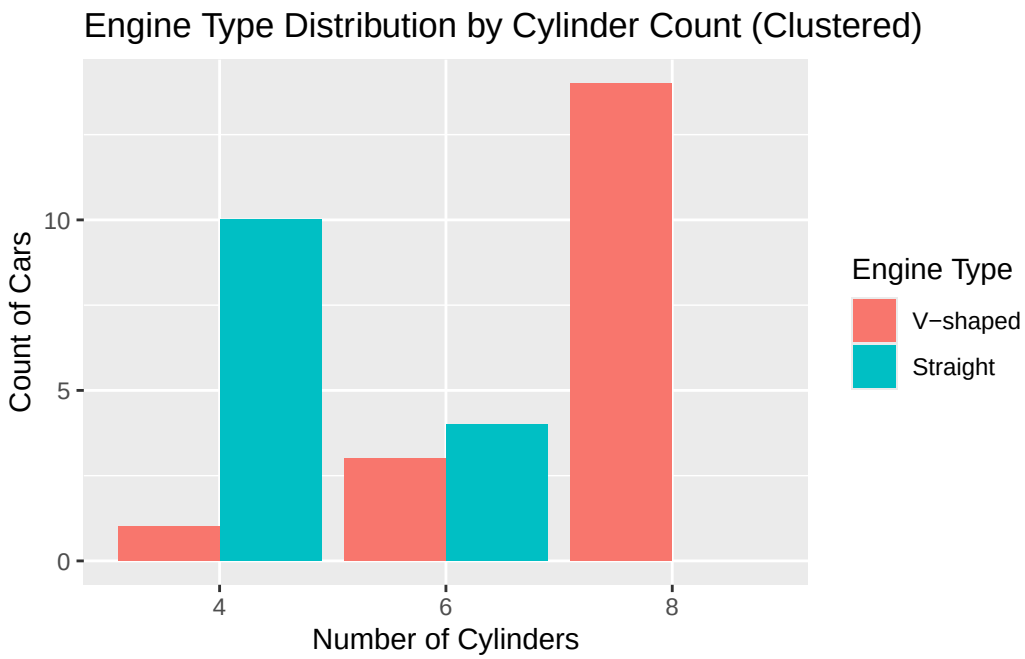
In this example::

- **Stacking**: `geom_bar()` automatically stacks subgroups when a `fill` aesthetic is mapped (here, `vs`).
- **Labels**: `scale_fill_discrete()` clarifies the engine types ("V-shaped" for `0`, "Straight" for `1`).

> **Tip**
>
> Clustered bar charts enable direct comparison of subgroups across categories by placing bars side-by-side. For example, to visualise engine type (vs) distributions across cylinder counts (cyl) with grouped bars:

```
mtcars |> ggplot(aes(x = cyl, fill = vs)) +
  geom_bar(position = position_dodge(preserve = "single")) +
  labs(
    title = "Engine Type Distribution by Cylinder Count (Clustered)",
    x = "Number of Cylinders",
    y = "Count of Cars",
```

```
    fill = "Engine Type"
  ) +
  scale_fill_discrete(labels = c("V-shaped", "Straight"))
```



In this example::

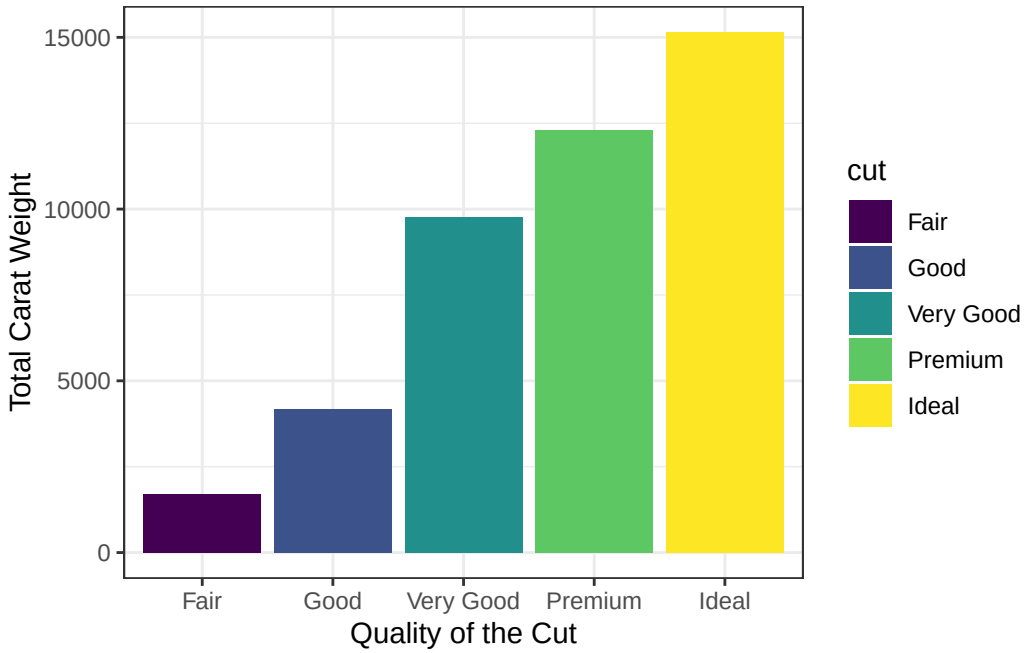
- **Data**: `mtcars` dataset.
- **Aesthetics**: `x = cyl` defines cylinder groups; `fill = vs` colours bars by engine type.
- **Geometric Object**: `geom_bar(position = position_dodge(...))` groups bars side-by-side instead of stacking.
  - `preserve = "single"` ensures consistent bar widths even if some subgroups are missing.
- **Labels**: `scale_fill_discrete()` clarifies engine type labels.

> **Note**
>
> Clustered bars make it easier to directly compare V-shaped vs. straight engines within each cylinder group.

7.8.8.3 Bar Charts with Precomputed Values

When the bar heights represent explicit values in your dataset (rather than counts of observations), use `geom_col()`. This function requires both `x` and `y` aesthetics, where `y` corresponds to precomputed values (e.g., sums, averages, or other aggregated metrics). For example, to visualise the total carat weight of diamonds grouped by cut quality:

```
diamonds |>
  ggplot(aes(x = cut, y = carat, fill = cut)) +
  geom_col() +
  labs(
    x = "Quality of the Cut",
    y = "Total Carat Weight"
  ) +
  theme_bw()
```



In this example::

- **Data**: `diamonds` dataset (requires aggregation if `carat` is not pre-summarised).
- **Aesthetics**:
  - `x = cut`: Categorises bars by diamond cut quality.
  - `y = carat`: Uses raw carat values (summed automatically per group).

  - `fill = carat`

- **Geometric Object**: `geom_col()` plots bars with heights proportional to `y`.
- **Labels**: `labs()` clarifies axis titles.

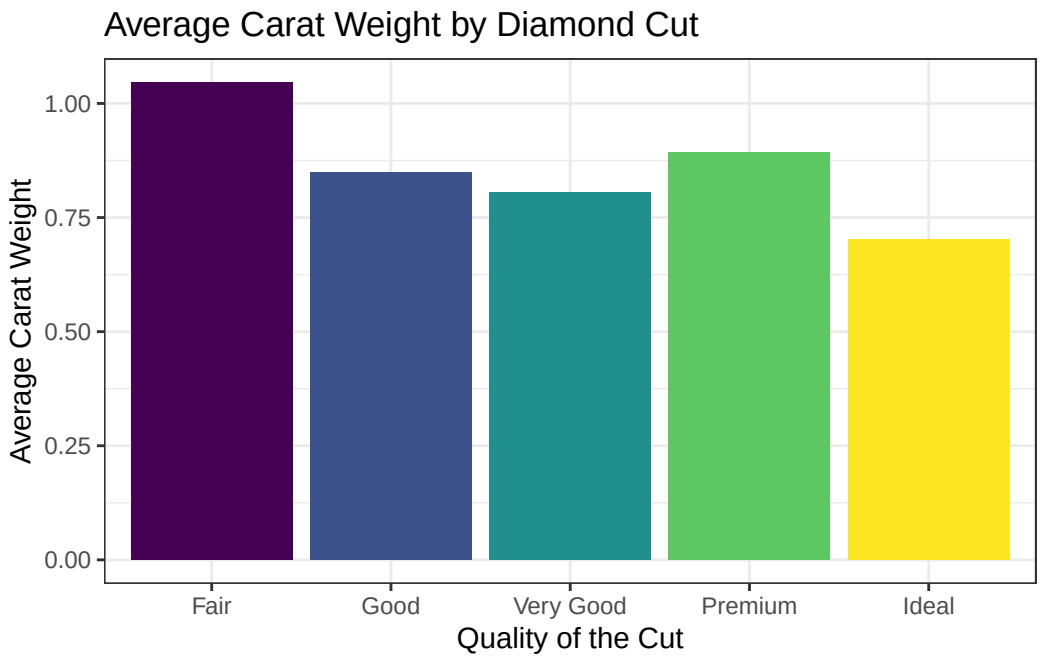
Interpretation:

This chart shows the total carat weight of diamonds per cut. For example, "Ideal" cuts have a higher total carat weight because they are more prevalent in the dataset.

⚠ Data preparation

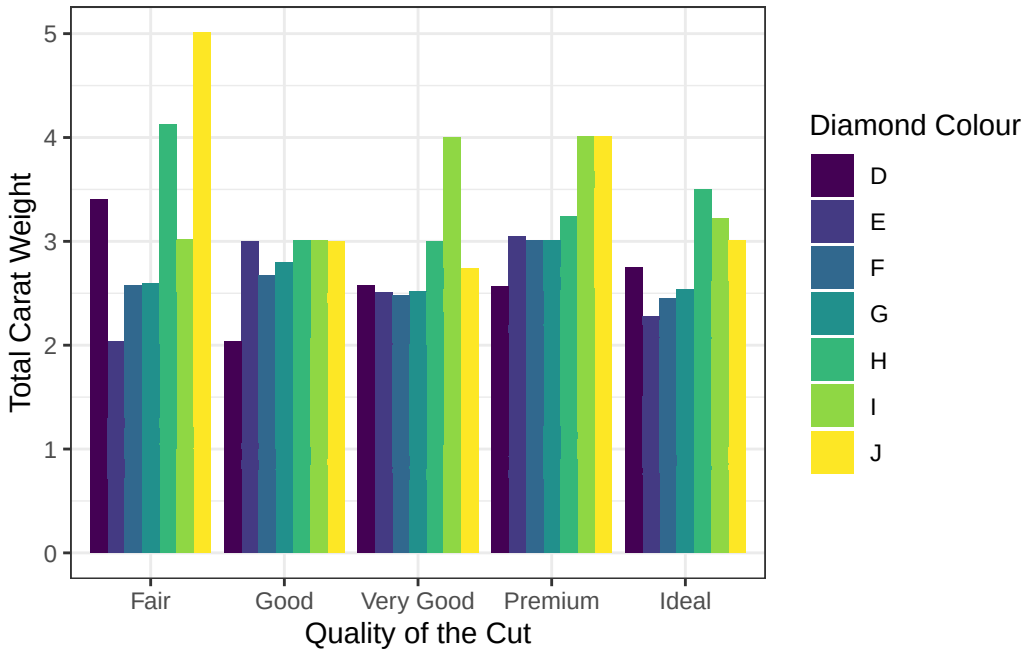
`geom_col()` assumes `y` values are precomputed. To plot group means, summarise data first:

```
# Precompute total carat weight for each cut
diamonds_summary <- diamonds |>
  group_by(cut) |>
  summarise(mean_carat = mean(carat))

# Create the bar chart with labels
diamonds_summary |>
  ggplot(aes(x = cut, y = mean_carat, fill = cut)) +
  geom_col(show.legend = FALSE) + # Remove legend
  labs(
    title = "Average Carat Weight by Diamond Cut",
    x = "Quality of the Cut",
    y = "Average Carat Weight"
  ) +
  theme_bw()
```



For grouped comparisons with precomputed values, for example, comparing diamond carat weights across cuts and subdividing by diamond colour:

```
diamonds |>
  ggplot(aes(x = cut, y = carat, fill = color)) +
  geom_col(position = position_dodge()) +
  labs(
    x = "Quality of the Cut",
    y = "Total Carat Weight",
    fill = "Diamond Colour"
  ) +
  theme_bw()
```



In this example::

- **Aesthetics**:
  - `fill = color`: Subdivides bars by diamond colour (D–J).
- **Positioning**:
  - `position_dodge()` places bars side-by-side for direct subgroup comparison.

Interpretation:

The clustered bars reveal that carat weight distribution varies across diamond colour grades (D–J) within each cut category. Higher-quality cuts such as "Ideal" and "Premium" correlate with lighter colour grades (D–F), which contribute disproportionately to total carat weight, while lower-quality cuts ("Fair", "Good") show a broader representation across mid-range colours (G–J).

7.8.8.4 Bar Charts for Data Presented in a Table

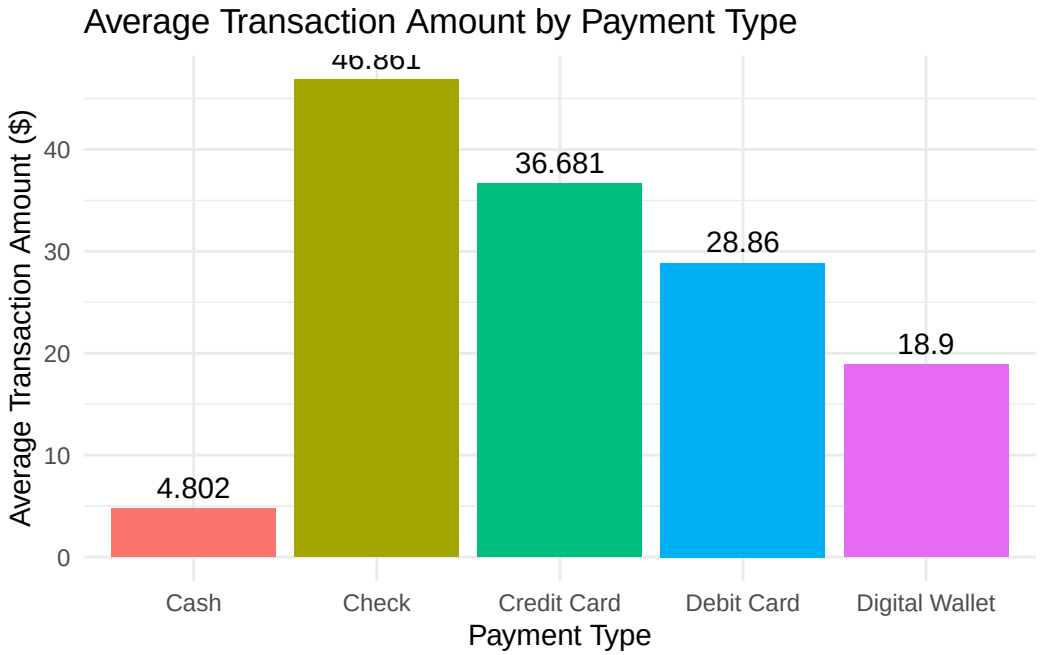
Imagine you are a data analyst for a retail company that accepts multiple payment methods. The following Table 7.1 shows the average transaction amount for each payment type:

Table 7.1: Average Transaction Amount by Payment Type

Payment Type	Average Transaction
Check	46.861
Credit Card	36.681
Debit Card	28.860
Digital Wallet	18.900
Cash	4.802

To visualise this, you can create a column plot with `geom_col()`:

```
# Create a data frame with the payment data
payment_data <- data.frame(
  payment_type = c("Check", "Credit Card", "Debit Card", "Digital Wallet", "Cash"),
  avg_transaction = c(46.861, 36.681, 28.860, 18.900, 4.802)
)

payment_data |>
  ggplot(aes(x = payment_type, y = avg_transaction, fill = payment_type)) +
  geom_col(show.legend = FALSE) + # Remove legend
  geom_text(
    aes(label = avg_transaction), # Add bar labels
    vjust = -0.5, # Position labels above bars
    colour = "black"
  ) +
  labs(
    title = "Average Transaction Amount by Payment Type",
    x = "Payment Type",
    y = "Average Transaction Amount ($)"
  ) +
  theme_minimal()
```



In this example::

- **Data**: A custom data frame (`payment_data`) containing payment types and their corresponding average transaction amounts.
- **Aesthetics**:
  - `x = payment_type`: Categorises bars by payment method.
  - `y = avg_transaction`: Uses precomputed average transaction values.
  - `fill = payment_type`: Fills bars with different colours for each payment method.
- **Geometric Object**: `geom_col()` plots bars with heights proportional to the provided average transaction values.
- **Bar Labels**:
  - `geom_text(aes(label = avg_transaction))`: Adds average transaction values above each bar.
  - `vjust = -0.5`: Positions labels slightly above the bars.
  - `colour = "black"`: Ensures labels are visible against the bars.
- **Labels**: `labs()` provides a descriptive title and axis labels.

Interpretation

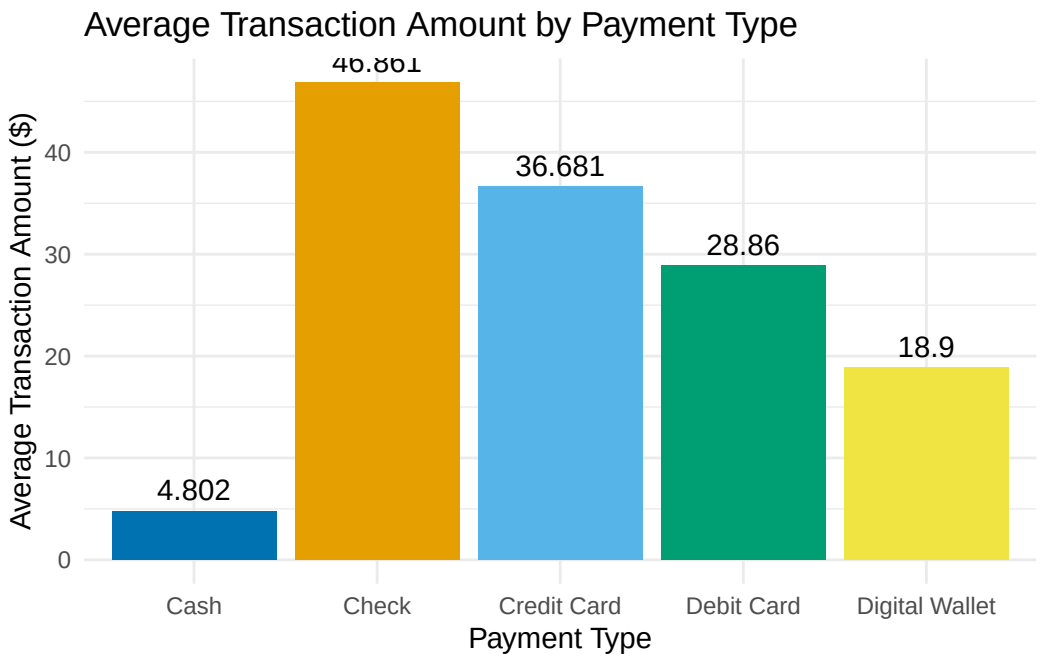
This chart clearly shows that customers using checks have the highest average transaction amount, while cash transactions are the lowest—offering valuable insight into customer spending habits.

Tip

Additional customisation options include adjusting label positioning with `vjust` or manually setting bar colours using `scale_fill_manual()`. For example:

- Adjust `vjust` to fine-tune label positioning (e.g., `vjust = 1.5` for labels inside bars).
- Use `scale_fill_manual()` to assign specific colours to bars. For example:

```
payment_data |>
  ggplot(aes(x = payment_type, y = avg_transaction, fill = payment_type)) +
  geom_col(show.legend = FALSE) +
  geom_text(
    aes(label = avg_transaction),
    vjust = -0.5,
    colour = "black"
  ) +
  labs(
    title = "Average Transaction Amount by Payment Type",
    x = "Payment Type",
    y = "Average Transaction Amount ($)"
  ) +
  # Assign custom colours for each payment type
  scale_fill_manual(values = c(
    "Check" = "#E69F00",
    "Credit Card" = "#56B4E9",
    "Debit Card" = "#009E73",
    "Digital Wallet" = "#F0E442",
    "Cash" = "#0072B2"
  )) +
  theme_minimal()
```



7.8.8.5 Faceting for Multi-Panel Plots

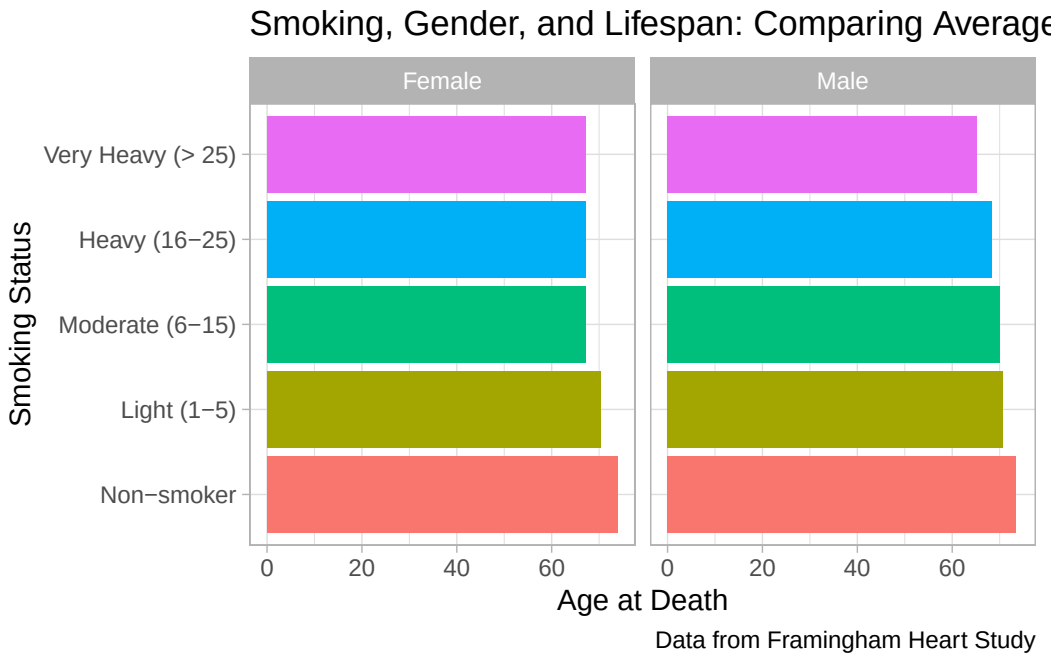
Faceting enables you to split data into subsets, displaying each in its own panel. For example, the following code facets the barchat plot by `gender` in the `heart` data:

```
avg_age_death_by_smoking_sex <- heart |>
  filter(!is.na(smoking_status)) |>
  group_by(smoking_status, sex) |>
  summarise(avg_age_at_death = mean(age_at_death, na.rm = TRUE))
```

```
#> `summarise()` has grouped output by 'smoking_status'. You can override using
#> the `.groups` argument.
```

```
avg_age_death_by_smoking_sex |> ggplot(aes(x = avg_age_at_death, y = smoking_status, fill = s
  geom_col(show.legend = FALSE) +
  facet_wrap(~sex) +
  labs(
    title = "Smoking, Gender, and Lifespan: Comparing Average Age at Death",
    x = "Age at Death",
    y = "Smoking Status",
```

```
    caption = "Data from Framingham Heart Study",
  ) +
  theme_light()
```



> **Tip**
>
> This code creates a scatter plot for each unique value of `gear`, allowing for easy comparison across groups.

To summarise, the key differences between `geom_bar()` and `geom_col()` are illustrated in the following Table 7.2:

Table 7.2: Key Differences: `geom_bar()` vs. `geom_col()`

Function	Use Case	Aesthetics Required
`geom_bar()`	Count observations per category	`x` only
`geom_col()`	Plot precomputed values per category	`x` and `y`

7.8.9 Creating a Line Chart

Using the `economics` dataset, we can plot unemployment trends:

```r
economics |>
  ggplot(aes(x = date, y = unemploy)) +
  geom_line(color = "blue") +
  labs(
    title = "Unemployment Trends Over Time",
    x = "Date",
    y = "Number of Unemployed Individuals"
  ) +
  theme_bw()
```

In this example:

- **Data:** US economic time series data.
- **Aesthetics:**

  `x = date`: Maps time to the x-axis.

  `y = unemploy`: Maps unemployment counts to the y-axis.
- **Geometric Object:**
  - `geom_area()` fills the area under the line, emphasising cumulative magnitude.
  - `fill = "lightblue"` sets the area colour.

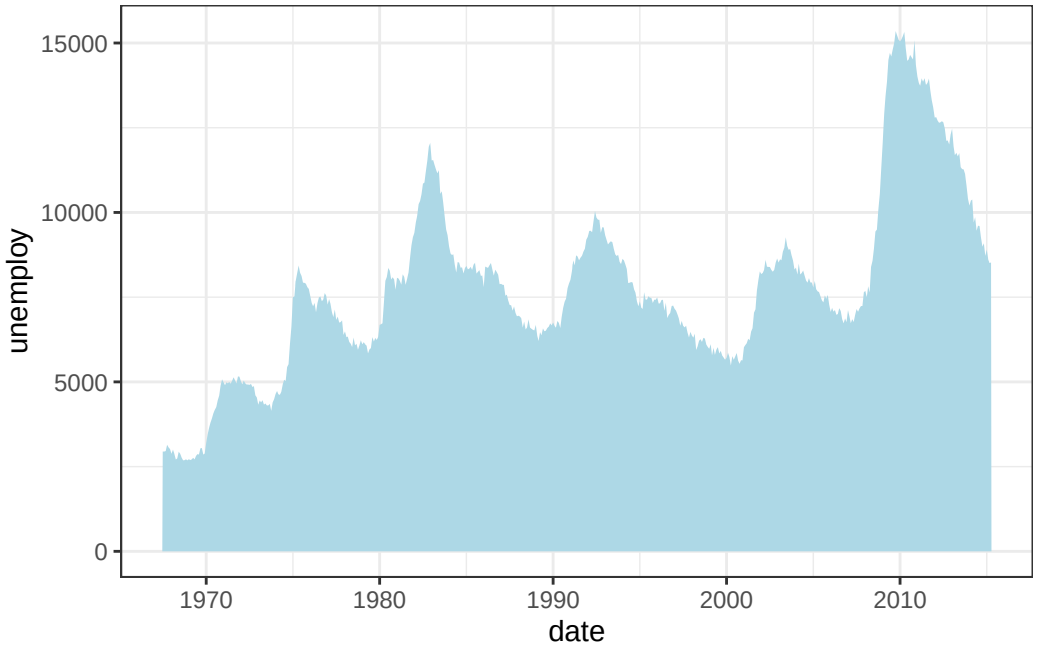
- **Labels:** `labs()` adds a title and axis labels.
- **Theme:** `theme_bw()` applies a black-and-white theme for a clear, classic look.

> Tip
>
> This line chart displays the trend in unemployment over time, allowing us to observe how the number of unemployed individuals changes across different periods.

7.8.10 Creating an Area Chart

Using the same economics data, create an area plot using the `geom_area()` function to display the number of unemployed individuals over time.

```
economics |> ggplot(aes(x = date, y = unemploy)) +
  geom_area(fill = "lightblue") +
  theme_bw()
```



In this example:

- **Data:** `economics` dataset (US economic time series)
- **Aesthetics:**

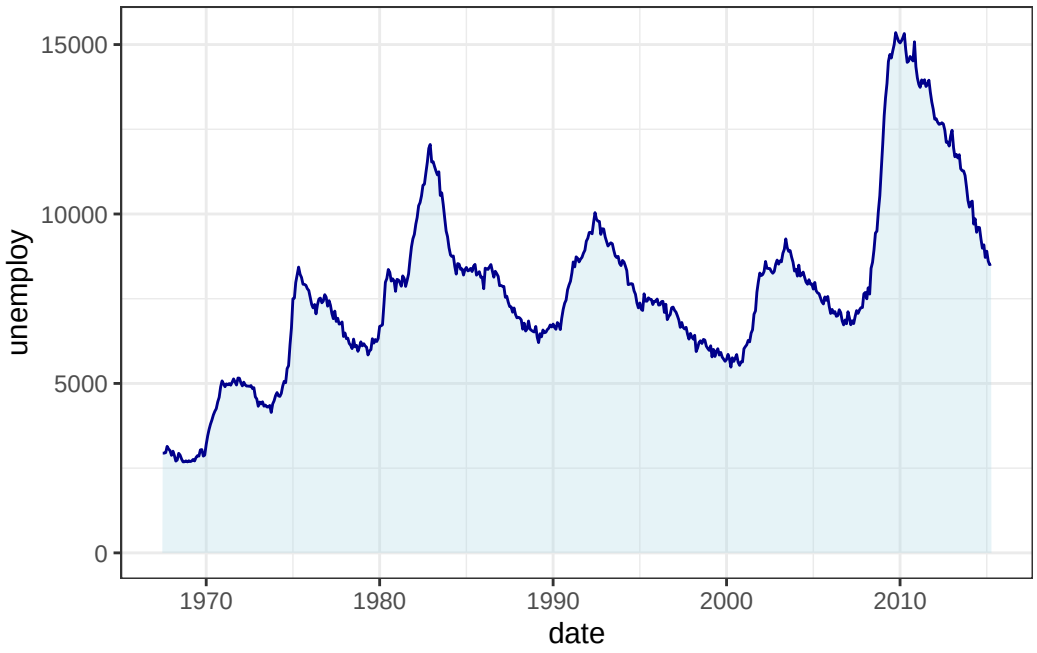
- `x = date`: Maps time to the x-axis.
- `y = unemploy`: Maps unemployment counts to the y-axis.
- **Geometric Object:**
- `geom_area()` fills the area under the line, emphasising cumulative magnitude.
- `fill = "lightblue"` sets the area colour.
- **Labels:** None are explicitly added here, so the default axis labels (`date` and `unemploy`) will be used.
- **Theme:** `theme_bw()` applies a clean black-and-white background.

> Interpretation
>
> The area chart not only illustrates long-term unemployment trends but also highlights economic cycles—peaks during recessions and troughs during recovery periods.

You can further customise the chart by:

- Adjusting the `alpha` parameter to control area transparency (e.g., `alpha = 0.5` for semi-transparency).
- Overlaying `geom_line()` on `geom_area()` for dual emphasis:

```
economics |>
  ggplot(aes(x = date, y = unemploy)) +
  geom_area(fill = "lightblue", alpha = 0.3) +
  geom_line(colour = "darkblue", linewidth = 0.5) + # Adds a trend line
  theme_bw()
```



7.8.11 Saving Your Plots

Once you have created a visually appealing plot with ggplot2, you may wish to save it as an image file for use in reports or presentations. The `ggsave()` function makes this simple:

```
diamonds |>
  ggplot(aes(x = cut, y = carat, fill = color)) +
  geom_col(position = position_dodge()) +
  labs(x = "Quality of the cut", y = "Weight of the diamond") +
  ggthemes::theme_economist()
```


```
ggsave(filename = "diamonds-plot.png")
```

> Tip
>
> - `filename = "diamonds-plot"`: Specifies the name and format of the output file.
> - By default, `ggsave()` saves the most most recently created plot in your working directory.

Customizing the Output:

For reproducible results and consistent dimensions, always adjust dimensions and resolution to ensure your plot meets publication or presentation standards.

```
ggsave(
  filename = "diamonds-plot.png",
  width = 8, # Width in inches
  height = 6, # Height in inches
  units = "in", # Units for width and height (can be "in", "cm", or "mm")
  dpi = 300 # Resolution in dots per inch
)
```

Tip

- `width` and `height`: Set the size of the image.
- `units`: Specify the units of measurement.
- `dpi`: Controls the resolution; 300 dpi is standard for high-quality images.

The `ggsave()` function lets you export your plots to formats such as PNG, PDF, or JPEG. For more advanced options, consult the official documentation:

```
?ggsave
```

7.8.12 Practice Quiz 7.1

Question 1:

Which principle is the foundation of `ggplot2`'s structured approach to building graphs?

a) The Aesthetic Mapping Principle

b) The Facet Wrapping Technique

c) The Grammar of Graphics

d) The Scaling Transformation Theory

Question 2:

In a `ggplot2` plot, which of the following best describes the role of `aes()`?

a) It specifies the dataset to be plotted.

b) It defines statistical transformations to apply to the data.

c) It maps data variables to visual properties, like colour or size.

d) It sets the coordinate system for the plot.

Question 3:

If you want to display the distribution of a single continuous variable and identify its modality and skewness, which geom is most appropriate?

a) `geom_point()`

b) `geom_bar()`

c) `geom_histogram()`

d) `geom_col()`

Question 4:

When creating a boxplot to show the variation of a continuous variable across multiple categories, what do the "whiskers" typically represent?

a) The median value and the mean value.

b) The full range of the data, excluding outliers.

c) One standard deviation above and below the mean.

d) The maximum and minimum values after applying a 1.5 * IQR rule.

Question 5:

You have a dataset with a categorical variable `Region` and a continuous variable `Sales`. You want to compare total sales across different regions. Which geom and aesthetic mapping would be most appropriate?

a) `geom_bar(aes(x = Region))`, which internally counts the occurrences of each region.

b) `geom_col(aes(x = Region, y = Sales))`, which uses the actual `Sales` values for the bar heights.

c) `geom_line(aes(x = Region, y = Sales))`, connecting points across regions.

d) `geom_area(aes(x = Region, y = Sales))`, to show cumulative totals over regions.

Question 6:

If you want to add a smoothing line (e.g., a regression line) to a scatter plot created with `geom_point()`, which geom should you use and with what parameter to fit a linear model without confidence intervals?

a) `geom_smooth(method = "lm", se = FALSE)`

b) `geom_line(stat = "lm", se = TRUE)`

c) `geom_line(method = "regress", se = FALSE)`

d) `geom_smooth(method = "reg", confint = FALSE)`

Question 7:

Consider you have a factor variable `cyl` representing the number of cylinders in the `mtcars` dataset. If you want to create multiple plots (small multiples) for each value of `cyl`, which ggplot2 function can you use?

a) `facet_wrap(~ cyl)`

b) `facet_side(~ cyl)`

c) `group_by(cyl)` followed by multiple `geom_point()` calls

d) `geom_facet(cyl)`

Question 8:

Which of the following statements about `ggsave()` is true?

a) `ggsave()` must be called before creating any plots for it to work correctly.

b) `ggsave()` saves the last plot displayed, and you can control the output format by specifying the file extension.

c) `ggsave()` cannot control the width, height, or resolution of the output image.

d) `ggsave()` only saves plots as PDF files.

Question 9:

What is the purpose of setting `group` aesthetics in a ggplot, for example in a line plot?

a) To change the colour scale of all elements.

b) To ensure that discrete categories are grouped together for transformations like smoothing.

c) To define which points belong to the same series, enabling lines to connect points within groups instead of mixing data across categories.

d) To modify only the legend titles and labels.

Question 10:

When customizing themes, which of the following options is NOT directly controlled by a `theme()` function in ggplot2?

a) Axis text size, angle, and colour.

b) Background grid lines and panel background.

c) The raw data values in the dataset.

d) The plot title alignment and style.

See the Solution to Quiz 7.1

7.8.13 Exercise 7.1.1: Data Analysis and Visualization with Medical Insurance Data

For this exercise, you will use Rstudio Project, call it `Experiment 7.1` and medical insurance data. These questions and tasks will give you hands-on experience with the key functionalities of `dplyr` and `ggplot2`, reinforcing your learning and understanding of both data manipulation and visualization in R.

1. Data Manipulation using `dplyr`:

a. Locate the `medical_insurance.xlsx` file in the `r-data` directory. If you don't already have the file, you can download it from Google Drive.

b. Import the data into R.

c. How many individuals have purchased medical insurance? Use `dplyr` to filter and count.

d. What is the average estimated salary for males and females? Use `group_by()` and `summarise()`.

e. How many individuals in the age group 20-30 have not purchased medical insurance? Use `filter()`.

f. Which age group has the highest number of non-purchasers? Use `group_by()` and `summarise()`.

g. For each gender, find the mean, median, and maximum estimated salary. Use `group_by()`, `summarise` and appropriate statistical functions.

2. Data Visualization using `ggplot2`:

a. Create a histogram of the ages of the individuals. Use `geom_histogram()`.

b. Plot a bar chart that shows the number of purchasers and non-purchasers. Use `geom_bar()`.

c. Create a boxplot to visualize the distribution of estimated salaries for males and females. Use `geom_boxplot()`.

d. Generate a scatter plot of age versus estimated salary. Color the points by their "Purchased" status. This will give insights into the relationship between age, salary, and the decision to purchase insurance. Use `geom_point()`.

e. Overlay a density plot on the scatter plot created in (d) to better understand the concentration of data points. Use `geom_density_2d()`.

3. Combining `dplyr` and `ggplot2`:

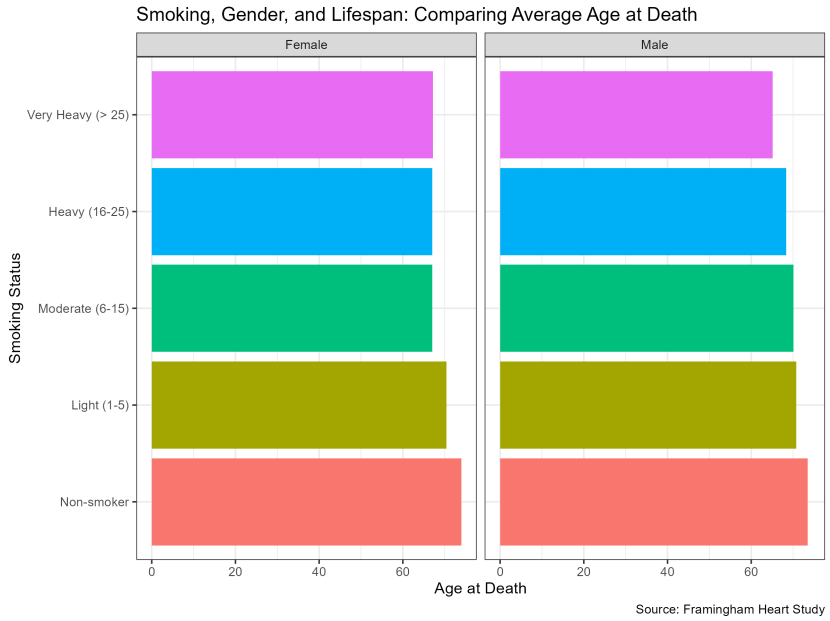
a. Filter the data to only include those who haven't purchased insurance and then create a histogram of their ages.

b. Group the data by gender and then plot the average estimated salary for each gender using a bar chart.

c. For each age, calculate the percentage of individuals who have purchased insurance and then plot this as a line graph against age.

7.8.14 Exercise 7.1.2: Reproducing the Smoking, Gender, and Lifespan Chart

In this exercise you will use the `heart` dataset from the Framingham Heart Study (located in the `r-data` directory) to replicate a chart that compares the average age at death by smoking status for both males and females.



The example chart is a horizontal bar chart, divided into two panels—one for each sex. In each panel, individuals are grouped by their smoking intensity, which is classified into the following categories: *Non-smoker*, *Light (1–5)*, *Moderate (6–15)*, *Heavy (16–25)*, and *Very Heavy (>25)*. The x-axis shows the average age at death, and the data source is the Framingham Heart Study.

1. Data Description

- The `heart` dataset contains cardiovascular health data from the Framingham Heart Study, including variables such as `sex`, `age_at_death`, and `smoking_status`.
- The smoking status is classified into the following categories: *Non-smoker*, *Light (1–5)*, *Moderate (6–15)*, *Heavy (16–25)*, and *Very Heavy (>25)*.
- Note that some variables (for example, `smoking_status`) may contain missing values.

2. Objective

- **Filter** the dataset to remove any rows with a missing `smoking_status`.
- **Group** the data by `smoking_status` and `sex`.
- **Calculate** the mean `age_at_death` for each group.
- **Visualise** the results using **ggplot2** by creating a bar chart that:
  - Displays the average age at death on the x-axis.
  - Shows the smoking status on the y-axis.

- Uses faceting to create separate panels for females and males.
- Utilises colour/fill to differentiate between the smoking categories.
- Includes a clear title and appropriate labels for both axes.

3. Deliverables

- Provide a brief description of your approach (no code is required here, just an explanation of your rationale).
- Produce a bar chart that closely replicates the example above, and save it as an image file (PNG or PDF).
- Write a short summary of what the chart reveals about the relationship between smoking status, sex, and average age at death.

> **Tip**
>
> To complete this exercise, you will need to use both **dplyr** (for data manipulation) and **ggplot2** (for visualisation). Focus on computing the mean of `age_at_death` for each group defined by `smoking_status` and `sex`.

Good luck, and enjoy exploring the data!

See the Solution to Exercise 7.1.2

7.9 Experiment 7.2: Data Visualisation Using Base R Graphics

Data visualisation using Base R graphics offers a built-in, efficient approach that requires no additional packages. Although ggplot2 offers extensive flexibility and elegance, Base R graphics remain indispensable for quick and exploratory analyses, especially when producing simple or preliminary plots.

Base R uses a function-based approach that gives you direct control over graphical elements. Unlike ggplot2's layered grammar, each plot type in Base R is created with a dedicated function, making it ideal for rapid exploratory analysis or for producing simple static plots.

7.9.1 Advantages of Using Base R

Let us explore the key benefits that make Base R a practical choice for data analysis in R:

- **No Dependencies**: Base R plotting functions are part of the core R installation, so there is no need for additional packages.

- **Quick and Simple**: These functions are ideal for exploratory data analysis or when a fast visualisation is required.
- **Full Control**: You have direct access to low-level graphical parameters, allowing for highly customised plots.

7.9.2 Core Plotting Functions

Base R provides a suite of primary functions that offer a straightforward and flexible method to create and customise graphics. These include:

- `plot()`: a generic function for creating scatterplots, line graphs, and more.
- `hist()`: for displaying data distributions in the form of histograms.
- `boxplot()`: for creating box-and-whisker plots to summarise and compare data.
- `barplot()`: for generating bar charts of categorical data.
- `pie()`: for illustrating proportions using pie charts.

> Tip
>
> These functions form the backbone of many data visualisation tasks in Base R and can be easily combined with customisations to enhance your analysis.

7.9.3 Customising Plots in Base R

A major strength of Base R plotting is its extensive customisation through built-in graphical parameters. Unlike specialised libraries that employ layered functions and themes, customisations in Base R are applied directly within plotting functions or via the `par()` function to control global settings.

Some key graphical parameters include:

- **Colour (`col`)**:
  The `col` parameter sets colours for plot elements, including points, lines, bars, and text. You can specify colours by name (e.g. `"blue"`, `"red"`), numeric codes, or hexadecimal codes (`"#FF0000"`). For instance, specifying `col = "blue"` in `plot()` sets points or lines to blue. You may also supply a vector of colours to differentiate groups.

- **Point Symbols (`pch`)**
  The `pch` parameter defines the style of points in scatterplots or line plots. It accepts numerical codes (0–25), each corresponding to different symbols (circles, triangles, crosses, etc.). For instance, `pch = 19` creates solid circular points, which are often used for clarity. Alternatively, character symbols (e.g., `pch = "*"`) to add more stylistic customisation.
- **Line Width and Type (`lwd` and `lty`)**
  The `lwd` parameter controls the thickness of lines (e.g., in line charts or regression lines), with higher numeric values creating thicker lines. The default is typically 1, but you can adjust this for visibility or emphasis. Meanwhile, `lty` controls line patterns—solid lines (`lty = 1`), dashed (`lty = 2`), dotted (`lty = 3`), or combinations thereof (1 through 6).
- **Scaling of Element Sizes (`cex`)**
  The `cex` parameter scales the size of plot elements such as points, labels, and text annotations. A value greater than 1 increases element size for enhanced visibility, while a value less than 1 reduces it, ensuring that plots remain readable in presentations or printed reports.
- **Axes and Text Labels (`main`, `xlab`, `ylab`)**
  Meaningful and clear labelling significantly improves plot readability. The parameters include:
  - **`main`**: Sets the main title at the top of the plot, summarising its purpose.
  - **`xlab` and `ylab`**: Define the labels for the horizontal and vertical axes, providing essential context for the numeric scales.
- **Adjusting Margins and Layout (`par()`)**
  For more extensive customisation, the `par()` function adjusts global graphical settings, such as margins (`mar`), axis text orientation (`las`), font sizes (`font`), and layout arrangements (`mfrow`). For example, setting `par(mfrow = c(2, 2))` divides your graphical device into a 2x2 grid, allowing multiple plots in a single output.

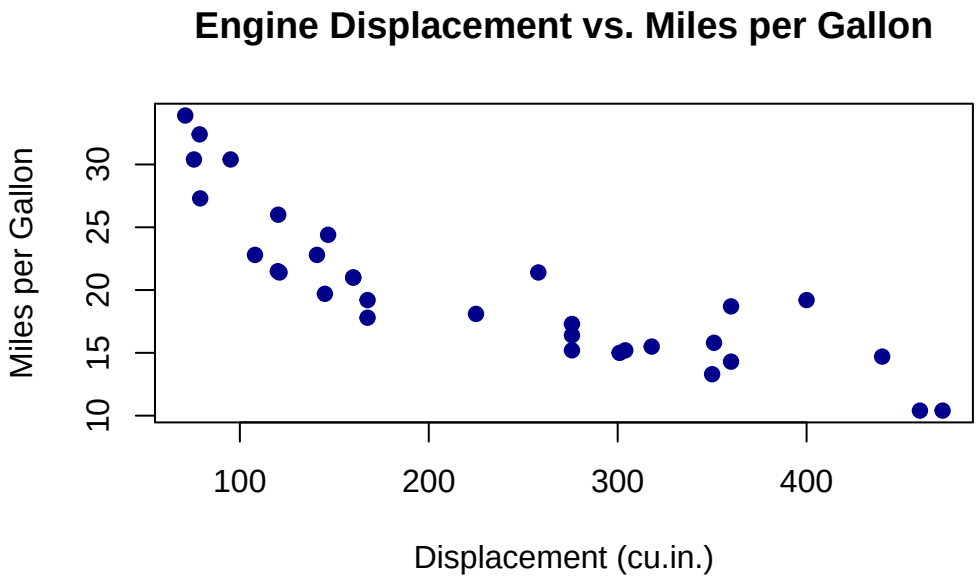
> **Note**
>
> Customising these parameters directly within Base R plotting functions offers precise control over your visualisations, ensuring that your plots are both informative and aesthetically pleasing.

We will reproduce the visualisation examples from Experiment 7.1 to demonstrate how similar plots can be constructed using R's built-in graphics functions and to highlight the available customisation options.

7.9.4 Creating a Scatter Plot with Base R

Using the `mtcars` dataset, we can visualise the relationship between engine displacement and miles per gallon:

```
plot(mtcars$disp, mtcars$mpg, # X-Y plotting style for scatter plot
  main = "Engine Displacement vs. Miles per Gallon",
  xlab = "Displacement (cu.in.)",
  ylab = "Miles per Gallon",
  pch = 19, col = "darkblue"
)
```

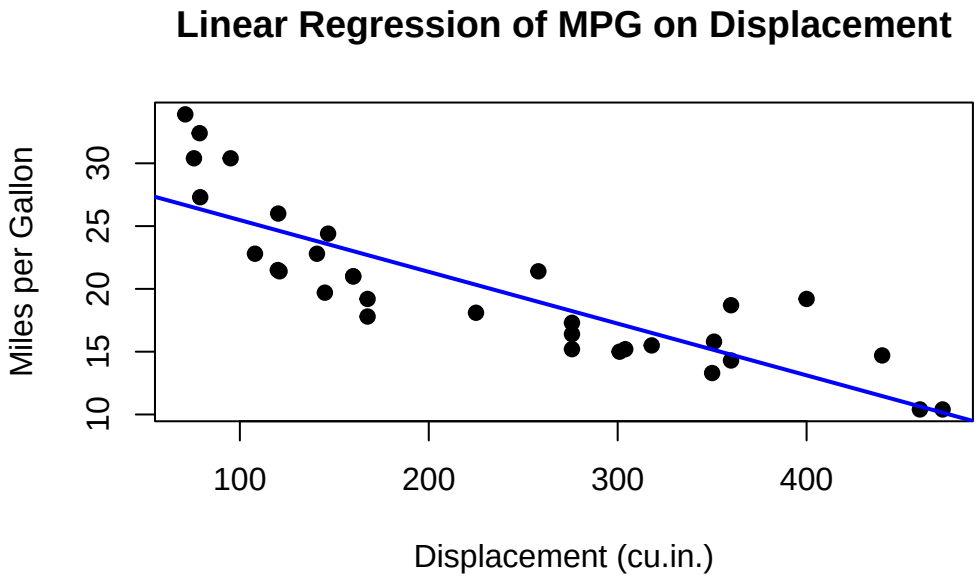


Adding a Regression Line:

You can further enhance the scatter plot by adding a regression line:

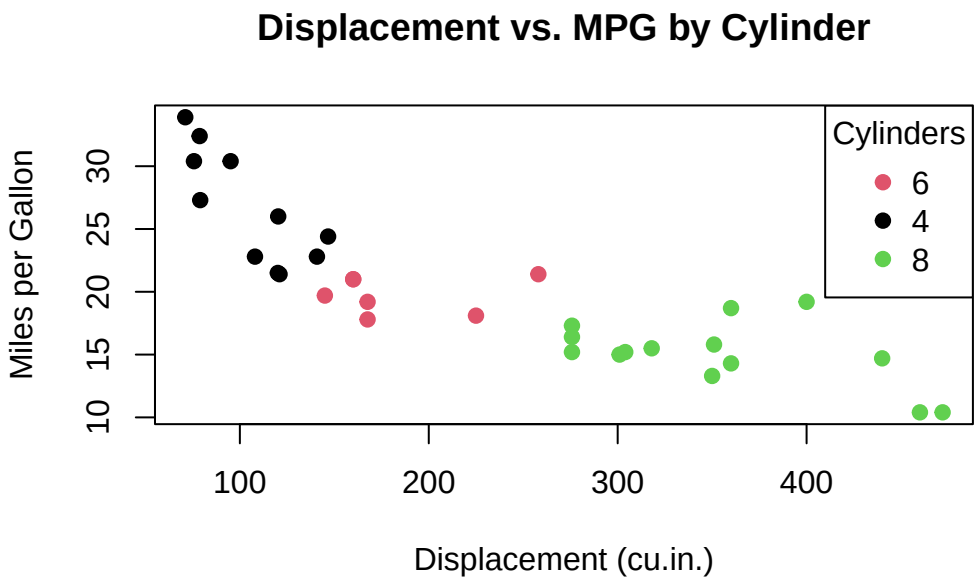
```
# Base scatter plot using a formula interface
plot(mpg ~ disp,
  data = mtcars,
  main = "Linear Regression of MPG on Displacement",
  xlab = "Displacement (cu.in.)",
  ylab = "Miles per Gallon",
  pch = 19
```

```
)

# Adding a linear regression line
abline(lm(mpg ~ disp, data = mtcars), col = "blue", lwd = 2)
```



For a scatter plot that incorporates a categorical variable (e.g. cylinder) to distinguish colours:

```
plot(mpg ~ disp,
  col = cyl, data = mtcars, pch = 19,
  main = "Displacement vs. MPG by Cylinder",
  xlab = "Displacement (cu.in.)",
  ylab = "Miles per Gallon"
)
legend("topright",
  legend = unique(mtcars$cyl),
  col = unique(mtcars$cyl), pch = 19, title = "Cylinders"
)
```

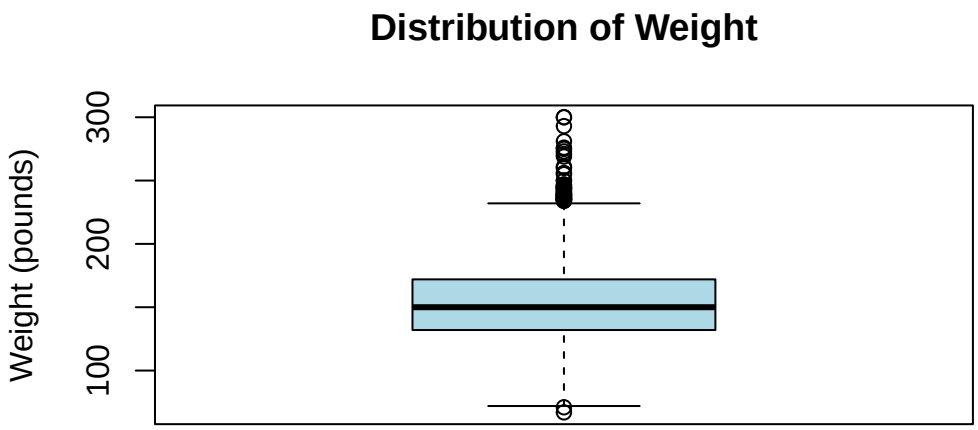


> **Tip**
>
> Experiment with different `pch` and `col` values to improve the clarity of your scatter plots, especially when dealing with overlapping data points.

7.9.5 Creating Boxplots in Base R

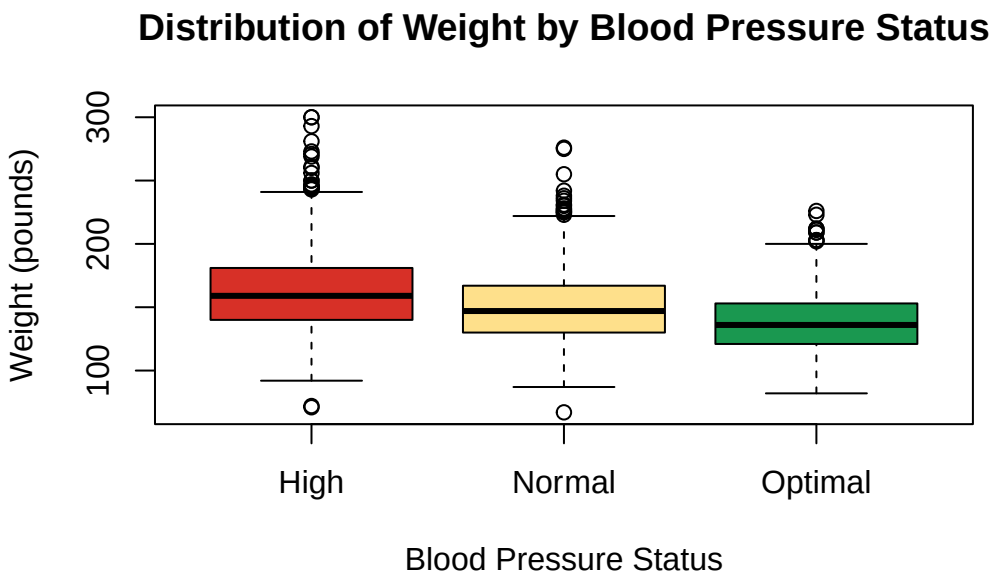
Boxplots are invaluable for visualising the distribution of a variable. For instance, using the `heart` dataset, we can examine the distribution of weight.

```
boxplot(heart$weight,
  main = "Distribution of Weight",
  ylab = "Weight (pounds)",
  col = "lightblue"
)
```



We can also visualise weight by blood pressure status:

```
boxplot(weight ~ bp_status,
  data = heart,
  col = c("#D73027", "#FEE08B", "#1A9850"),
  main = "Distribution of Weight by Blood Pressure Status",
  xlab = "Blood Pressure Status",
  ylab = "Weight (pounds)"
)
```

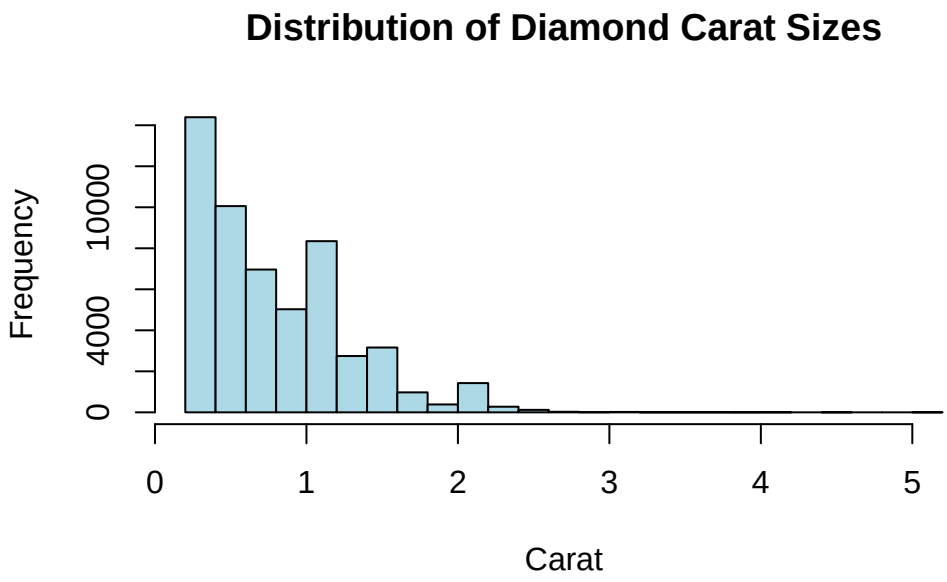


> **Note**
>
> When comparing groups using boxplots, ensure that your grouping variable is appropriately formatted (e.g. as a factor) for clear interpretation.

7.9.6 Creating a Histogram in Base R

To display the distribution of diamond carat sizes, we use the `hist()` function:

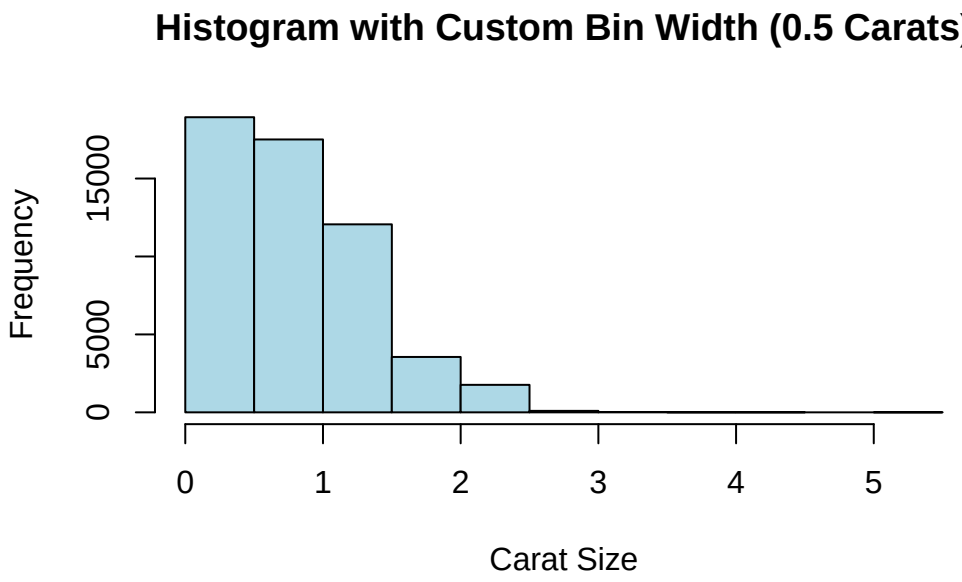
```
hist(diamonds$carat,
  breaks = 30, col = "lightblue",
  main = "Distribution of Diamond Carat Sizes",
  xlab = "Carat",
  ylab = "Frequency"
)
```



Adjusting Bin-width

Base R utilises the `breaks` argument to control bin-width. For example, to use a bin-width of 0.5 carats, ensure the breaks span the entire range of your data:

```
hist(diamonds$carat,
  breaks = seq(0, max(diamonds$carat) + 0.5, by = 0.5),
  col = "lightblue",
  xlab = "Carat Size",
  main = "Histogram with Custom Bin Width (0.5 Carats)"
)
```

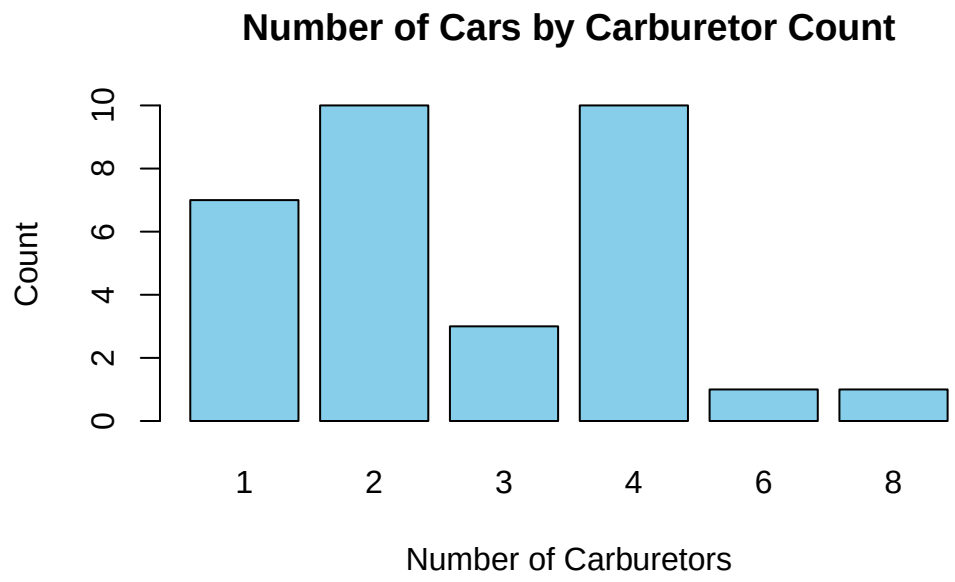


> **Tip**
>
> Adjust the `breaks` argument to fine-tune the granularity of your histogram and capture the nuances in your data distribution.

7.9.7 Creating Bar Charts in Base R

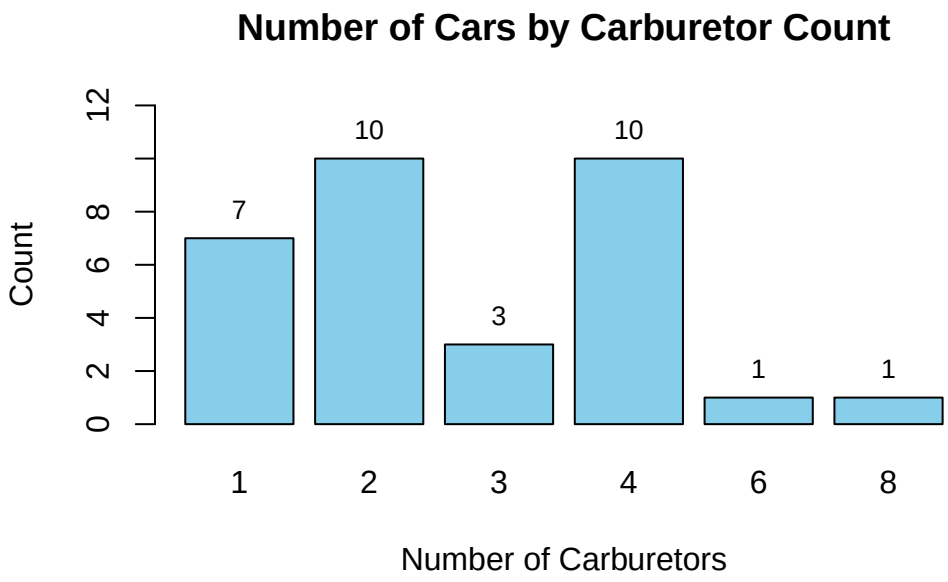
A simple bar chart can be created to display the number of cars by carburettor count:

```
barplot(table(mtcars$carb),
  main = "Number of Cars by Carburetor Count",
  xlab = "Number of Carburetors",
  ylab = "Count",
  col = "skyblue"
)
```



You can also add value labels above each bar to indicate the count:

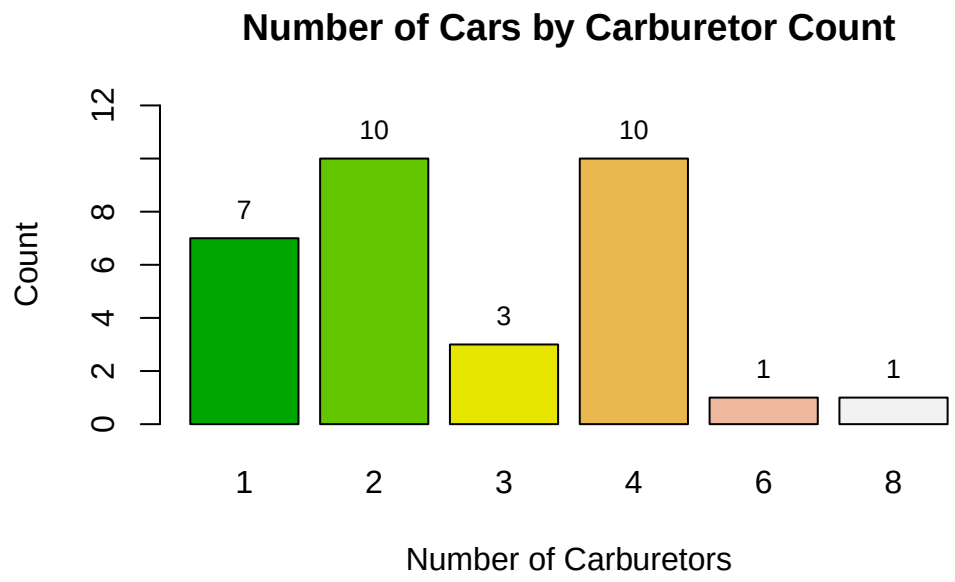
```r
# Create a table of counts for each carburetor count
counts <- table(mtcars$carb)

# Generate the bar plot and store the midpoints of the bars
bar_midpoints <- barplot(counts,
  main = "Number of Cars by Carburetor Count",
  ylim = c(0, 12),
  xlab = "Number of Carburetors",
  ylab = "Count",
  col = "skyblue"
)

# Add value labels above each bar
text(
  x = bar_midpoints, y = counts,
  labels = counts,
  pos = 3, # Position text above the bar
  cex = 0.8, # Adjust text size as needed
  col = "black"
)
```



> **Tip**
>
> The `ylim()` parameter sets the lower and upper limits of the y-axis so that the value labels are clearly visible, particularly for the bars representing 2 and 4 carburettors.

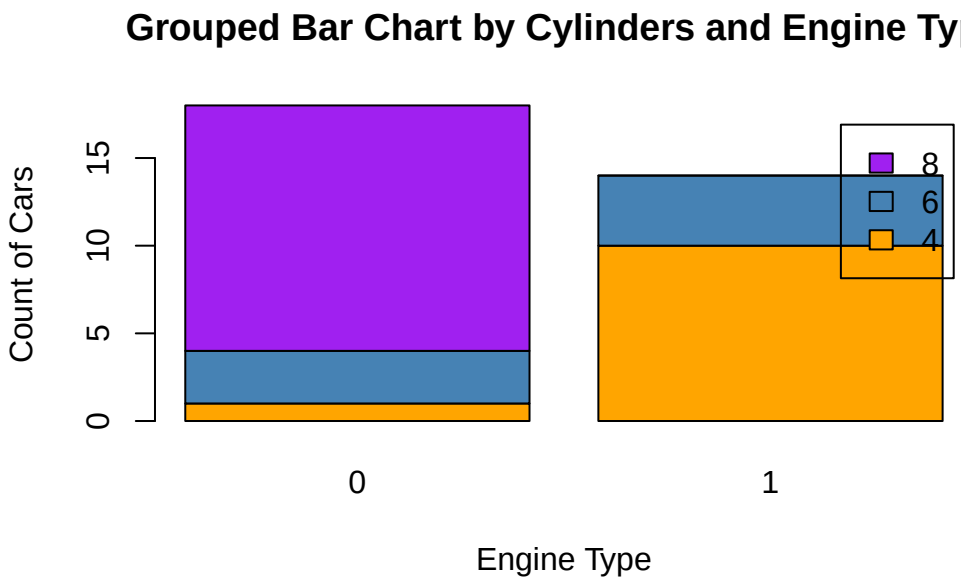
If desired, you can colour each bar differently by using a vector of colours. For example, you might use `terrain.colors(n)` or `rainbow(n)`, where n is the number of colours required:

```
bar_midpoints <- barplot(table(mtcars$carb),
  main = "Number of Cars by Carburetor Count",
  ylim = c(0, 12),
  xlab = "Number of Carburetors",
  ylab = "Count",
  col = terrain.colors(6)
)
text(
  x = bar_midpoints, y = table(mtcars$carb),
  labels = table(mtcars$carb),
  pos = 3, cex = 0.8, col = "black"
)
```



Grouped bar charts allow you to visualise relationships between two categorical variables. For example:

```
counts <- table(mtcars$cyl, mtcars$vs)
barplot(counts,
  beside = FALSE,
  col = c("orange", "steelblue", "purple"),
  legend = rownames(counts),
  main = "Grouped Bar Chart by Cylinders and Engine Type",
  xlab = "Engine Type",
  ylab = "Count of Cars"
)
```

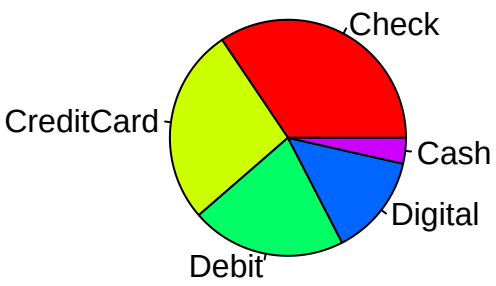


> **Tip**
>
> Set `beside = TRUE` to create a clustered bar chart, which makes it easier to compare subgroups directly.

7.9.8 Creating Pie and Doughnut Charts in Base R

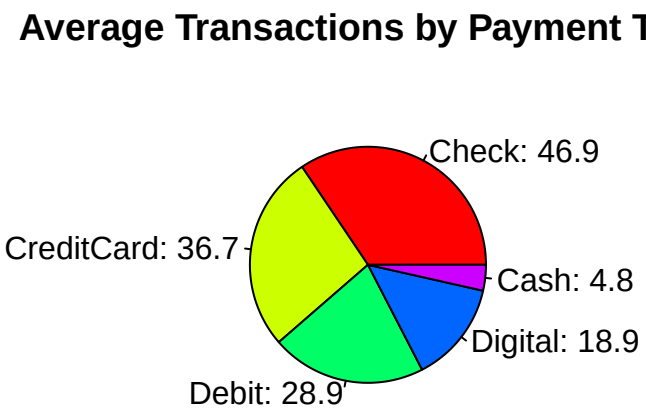
Using the payment method dataset presented earlier in Table 7.1, we can visualise the distribution of average transaction amounts with a pie chart:

```
payments <- c(
  Check = 46.861, CreditCard = 36.681, Debit = 28.860,
  Digital = 18.9, Cash = 4.802
)

pie(payments,
  main = "Average Transactions by Payment Type",
  col = rainbow(length(payments))
)
```



> **Tip**
>
> ou can enhance the clarity of this visualisation by adding value labels to each sector of the pie chart. This is done by specifying the `labels` argument; for instance, combining each payment method's name with its corresponding value creates informative labels for each slice.

```
payments <- c(
  Check = 46.861, CreditCard = 36.681, Debit = 28.860,
  Digital = 18.9, Cash = 4.802
)
# Create labels combining the payment method and its value
pie_labels <- paste(names(payments), round(payments, 1), sep = ": ")

pie(payments,
  main = "Average Transactions by Payment Type",
  col = rainbow(length(payments)),
  labels = pie_labels
)
```



> **Note**
>
> Doughnut charts are not directly supported by Base R graphics. However, you can create them using custom functions or additional packages, such as `plotrix`.

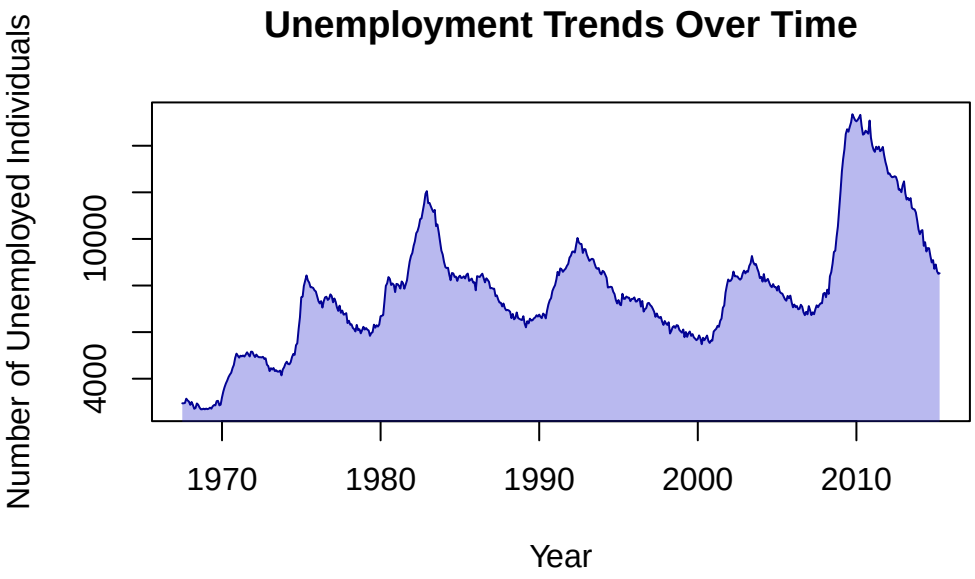
7.9.9 Creating Line and Area Charts

Using the `economics` dataset, we can display unemployment trends over time with a line chart:

```
plot(unemploy ~ date,
  data = economics, type = "l",
  main = "Unemployment Trends Over Time",
  xlab = "Year",
  ylab = "Unemployed",
  col = "blue", lwd = 2
)
```

To create an area chart, we fill the area beneath the line:

```
plot(unemploy ~ date,
  data = economics, type = "l",
  main = "Unemployment Trends Over Time",
  xlab = "Year",
  ylab = "Number of Unemployed Individuals",
  col = "darkblue"
)

polygon(c(economics$date, rev(economics$date)),
  c(economics$unemploy, rep(0, length(economics$unemploy))),
  col = rgb(0.1, 0.1, 0.8, 0.3), border = NA
)
```



> **Tip**
>
> The `polygon()` function is a powerful tool for filling areas under curves, which can help emphasise trends in your data.

7.9.10 Saving Plots

To save your Base R plots, wrap your plotting code with a graphics device function like `png()`, `jpeg()`, or `pdf()`. For example:

```
png(filename = "scatterplot.png", width = 800, height = 600)

plot(mpg ~ disp,
  data = mtcars,
  main = "Linear Regression of MPG on Displacement",
  xlab = "Displacement (cu.in.)",
  ylab = "Miles per Gallon",
  pch = 19
)

# Adding a linear regression line
abline(lm(mpg ~ disp, data = mtcars), col = "blue", lwd = 2)
```

```
dev.off()
```

> **Caution**
>
> Always remember to close the graphics device with `dev.off()` to finalise the output file.

7.9.11 Practice Quiz 7.2

Question 1:

Which of the following is a key advantage of using Base R graphics for exploratory data analysis?

a) They require additional packages.

b) They offer a quick, function-based approach with no dependencies

c) They utilise a layered grammar for complex plotting.

d) They automatically produce interactive visualisations.

Question 2:

Which function is the generic function in Base R for creating scatterplots, line graphs, and other basic plots?

a) `hist()`

b) `plot()`

c) `boxplot()`

d) `barplot()`

Question 3:

Which function in Base R is specifically used to display data distributions as histograms?

a) `pie()`

b) `plot()`

c) `hist()`

d) `boxplot()`

Question 4:

What is the purpose of the `breaks` argument in the `hist()` function?

a) To set the colour of the bars.

b) To determine the bin width for the histogram

c) To label the axes.

d) To specify the main title.

Question 5:

Which graphical parameter in Base R is used to specify the colour of plot elements?

a) `pch`

b) `lty`

c) `col`

d) `cex`

Question 6:

The `pch` parameter in Base R plots is used to control:

a) The type of point symbol displayed

b) The line thickness.

c) The overall scaling of plot elements.

d) The arrangement of multiple plots.

Question 7:

Which function in Base R is used to adjust global graphical settings, such as margins and layout arrangements?

a) `plot()`

b) `par()`

c) `hist()`

d) `boxplot()`

Question 8:

In a Base R scatter plot, which function is used to add a regression line?

a) `lines()`

b) `abline()`

c) `curve()`

d) `segments()`

Question 9:

What is one of the main reasons Base R graphics are considered advantageous over ggplot2 for certain tasks?

a) They require no additional packages since they are built into R

b) They offer more extensive theme options.

c) They are better suited for interactive visualisations.

d) They automatically manage data transformations.

Question 10:

When saving a Base R plot using the `png()` function, what is the purpose of calling `dev.off()` afterwards?

a) To display the saved plot.

b) To open the saved file in a new window.

c) To close the graphics device and finalise the output file

d) To reset all graphical parameters.

See the Solution to Quiz 7.2

7.10 Reflective Summary

In Lab 7, you have acquired essential data visualisation skills that enable you to transform raw data into compelling visual narratives:

- **Building Complex Visualisations with ggplot2:**
  You learned how to harness the power of the Grammar of Graphics to create layered, professional plots—from scatter plots to histograms and boxplots—that effectively communicate trends and patterns. You now understand how to map data variables to visual properties, adjust themes, and add annotations.
- **Quick and Efficient Plotting with Base R Graphics:**
  You explored R's built-in plotting functions, such as `plot()`, `hist()`, `boxplot()`, `barplot()`, and `pie()`, which are ideal for rapid exploratory analysis. Customisation through graphical parameters like `col`, `pch`, `lwd`, `lty`, and `cex` enables you to fine-tune your visualisations for clarity and impact.
- **Customising Visual Elements for Clarity and Impact:**
  Mastering both ggplot2 and Base R graphics has equipped you to tailor visual elements—such as colours, labels, scales, and themes—so your plots are not only informative but also visually engaging.
- **Integrating Data Manipulation and Visualisation:**
  You practiced combining data transformation techniques with visualisation tools to create seamless analytical workflows, empowering you to extract meaningful insights from complex datasets.

What's Next?

In the next lab, we will delve into statistical fundamentals, where you will build on these skills by exploring core statistical concepts, distinguishing between qualitative and quantitative data, understanding scales of measurement, and calculating descriptive statistics to further empower your data analysis and interpretation.

Part III

Statistical Thinking

8 Statistical Concept

8.1 Introduction

Welcome to Lab 8, where we introduce the foundational principles of statistics and equip you with the skills to analyse and interpret data effectively. These core concepts form the building blocks of statistical reasoning, essential for summarising data, understanding variability, and making informed decisions.

Statistics is the science of turning data into knowledge. It provides a structured approach to discerning patterns, testing hypotheses, and drawing meaningful conclusions. Whether you work in medicine, business, education, or any other field, statistical thinking is an invaluable skill—it helps you make sense of variability, measure uncertainty, and ultimately, make decisions based on data.

Mastering statistical concepts goes beyond memorising formulas. It involves developing a mindset that allows you to critically evaluate data and apply the appropriate methods to answer questions and solve problems.

8.2 Learning Objectives

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

- **Understand the Core Concepts of Data Science**
  Recognise the fundamental principles of data science and how they are applied to extract meaningful insights from raw data.
- **Differentiate Between Types of Data**
  Identify and distinguish between qualitative and quantitative data, and understand their relevance in statistical analysis.
- **Perform Key Descriptive Statistical Analyses**
  Calculate and interpret measures of central tendency (mean, median, mode) and measures of variability (range, variance, standard deviation).
- **Create Visual Representations of Data**
  Use R to produce visualisations such as boxplots and interpret the five-number summary to communicate your findings effectively.

- **Understand the Four Scales of Measurement**
  Apply nominal, ordinal, interval, and ratio scales to different types of data and select appropriate statistical techniques for analysis.

By completing this lab, you will develop a strong foundation in statistical concepts, enabling you to confidently approach more advanced topics in data analysis and data science.

8.3 What is Data?

Data is everywhere—it is embedded in nearly every aspect of modern life. Whether you are recording the number of steps you take each day, capturing sales in a shop, or measuring rainfall in a region, data is constantly being collected. Hospitals maintain clinical records, teachers mark student attendance, and governments conduct censuses, all of which generate valuable data.

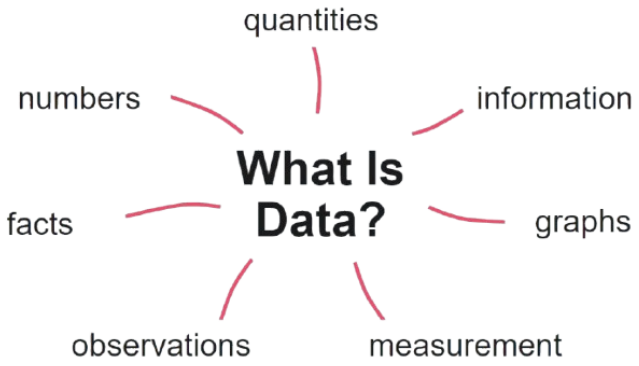


Figure 8.1: Understanding Data: Key Elements and Their Connections

However, raw data on its own is meaningless. It consists of unprocessed facts—numbers, words, images, or sounds. For example:

- A list of temperatures recorded each hour.
- The number of books borrowed from a library in a day.
- The names of students enrolled in a school.

When data is analysed and processed, it transforms into information—contextualised and meaningful insights that help us make decisions.

8.3.1 Why is Data Important?

Data forms the foundation for decision-making. Organisations use data to:

- Evaluate progress.
- Solve problems.
- Identify trends.
- Make predictions about the future.

In today's world, understanding data is not merely an advantage; it is a necessity. The ability to collect, analyse, and interpret data is a critical skill in almost every profession.

8.3.2 Types of Data

Data can be broadly classified into qualitative and quantitative types, each with unique characteristics and roles in analysis.

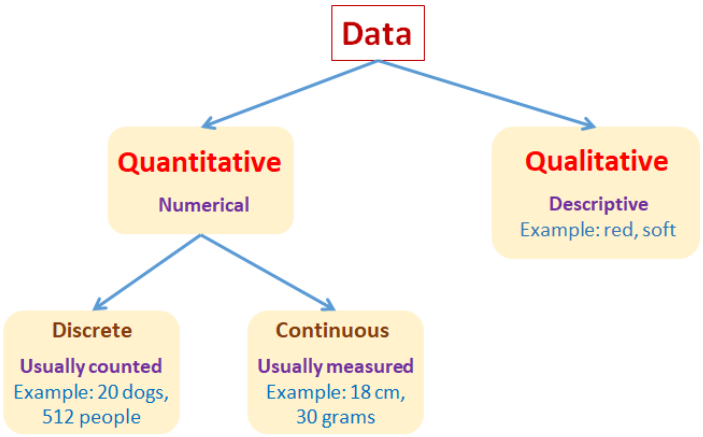


Figure 8.2: Classification of Data Types

8.3.2.1 Qualitative data

Qualitative data is descriptive, focusing on characteristics or categories rather than numerical values. It answers questions such as "What kind?" or "Which type?"

Examples include:

- Gender: Male, Female

- Blood Type: A, B, AB, O
- Opinion: Agree, Neutral, Disagree

Key Features:

- Non-numerical: It is not measured in numbers.
- Categorical: It is organised into distinct groups or categories.
- Applications: Commonly used in surveys, classifications, and descriptive studies.

8.3.2.2 Quantitative data

Quantitative data is numerical and answers questions like "How much?" or "How many?"

Examples include:

- Age: 21 years
- Revenue: $10,000
- Distance: 5 kilometres

Key Features:

- Measurable: It can be counted or measured.
- Mathematical Operations: It is suitable for statistical and mathematical analysis.

Quantitative data is further divided into discrete and continuous.

1. **Discrete data**

Discrete data consists of distinct, countable values and cannot take fractional or decimal forms. Essentially, if you're counting something in whole numbers, you are dealing with discrete data.

Key Features:

- Countable: Values are obtained by counting (e.g. 1, 2, 3, ...).
- No Intermediate Values: There are no fractions or decimals between the values.
- Finite or Infinite: Although it can be finite, theoretically, it can continue indefinitely.

Examples include:

- The number of students in a classroom.
- The number of goals scored in a football match.

- The number of books on a shelf.

> **Practical Applications**
>
> Imagine you are surveying how many pets each household owns. The results—0, 1, 2 pets, etc.—are discrete because you are counting individual units. Discrete data is useful for analysing totals, frequencies, and trends.

2. **Continuous Data**

Continuous data arises from measurement and can take on an infinite number of values within a given range.

Key Features:

- Measurable: It is obtained by measuring (e.g. height, weight, time).
- Infinite Possibilities: It can assume any value within a range.
- Decimals and Fractions:: Values are not restricted to whole numbers.

Examples include::

- A person's height: 160.5 cm, 172.8 cm.
- The weight of a package: 5.75 kg, 12.3 kg.
- Time taken to complete a task: 3.25 hours, 7.8 seconds.

> **Practical Applications**
>
> If you measure the rainfall over a month, each reading (e.g. 12.3 mm or 45.8 mm) is continuous data, allowing for detailed analyses such as calculating averages, variances, and trends over time.

Table 8.1 summarises the key differences between discrete and continuous data, helping you choose the appropriate analytical methods.

Table 8.1: Comparison of Discrete and Continuous Data

Aspect	**Discrete Data**	**Continuous Data**
Nature	Countable	Measurable
Values	Whole numbers only	Can include fractions and decimals
Range	Finite or infinite, but distinct steps	Infinite possibilities within a range

Aspect	Discrete Data	Continuous Data
Examples	Number of people, cars, or items	Weight, height, time, or temperature
Use in Statistics	Frequency counts, categorical trends	Averages, variances, and precise analysis

Activity

Think of three examples of qualitative and quantitative data from your own life and jot them down.

8.3.3 Sources of data

Data comes from two primary sources:

1. **Primary Data**
   Collected first-hand for a specific purpose.
2. **Secondary Data**
   Obtained from pre-existing sources such as reports, journals, or online databases.

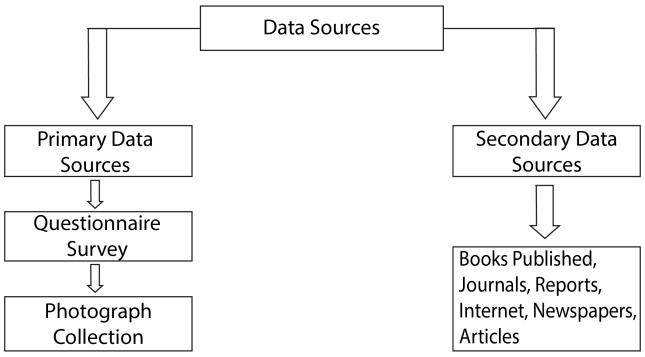


Figure 8.3: Classification of Data Sources

8.3.3.1 Primary Data

Primary data is collected first-hand for a specific purpose. It is raw, original, and hasn't been processed. Examples include:

- Surveys
- Experiments
- Interviews

8.3.3.2 Secondary Data

Secondary data comes from pre-existing sources, such as reports, journals, or online databases. This data has been processed or used for other purposes.

8.3.4 Practice Quiz 8.0

Question 1:

Data that focuses on characteristics or qualities rather than numbers is known as:

a) Quantitative data

b) Discrete data

c) Qualitative data

d) Continuous data

Question 2:

Which of the following is an example of discrete data?

a) The height of students in a class

b) The number of cars in a parking lot

c) The amount of rainfall in a day

d) The time taken to complete a task

Question 3:

Quantitative data that can take on any value within a given range is referred to as:

a) Categorical data

b) Nominal data

c) Discrete data

d) Continuous data

Question 4:

Qualitative data differs from quantitative data because qualitative data:

a) Can only be expressed with numbers

b) Has meaningful mathematical operations

c) Describes categories or groups

d) Is always collected from secondary sources

Question 5:

Primary data refers to data that:

a) Has been previously published by others

b) Comes directly from observation or experiment

c) Is always collected online

d) Is obtained only from government agencies

Question 6:

A list of colours observed in a garden (e.g., red, yellow, green) is an example of:

a) Quantitative continuous data

b) Quantitative discrete data

c) Qualitative data

d) Secondary data

Question 7:

Which of the following statements is true?

a) Data is always meaningful without analysis

b) Data, once processed, is known as information

c) Data and information are identical concepts

d) Information is just another term for data collection

Question 8:

A measurement like "23 people attended the seminar" is an example of:

a) Qualitative data

b) Continuous data

c) Discrete data

d) Nominal scale data

Question 9:

Data collected for the first time for a specific research purpose is known as:

a) Secondary data

b) Primary data

c) Nominal data

d) Discrete data

Question 10:

A researcher using census data from a national statistics bureau is working with:

a) Primary data

b) Secondary data

c) Continuous data

d) Nominal data

See the Solution to Quiz 8.0

8.4 Experiment 8.1: Statistical Thinking

Statistical thinking aligns your approach with the fundamental principles of statistics, enabling you to make better decisions under uncertainty. Essentially, understanding statistics is vital for making sound decisions in any field. By mastering basic statistical methods, you learn when to apply the right tools to solve problems and think critically about data.

8.4.1 Population Data versus Sample Data

Population Data

A population is the complete set of elements under study. For example:

- All the heights of graduating students at Harvard University (USA)
- All the weights of adult women in Lagos (Nigeria)
- All the ages of students at the University of Oxford (UK)
- All the undergraduate students at the University of Cape Town (South Africa)
- All the countries in the European Union

If data is available for every element in the population, it is referred to as a census. However, collecting data on an entire population is often impractical or too expensive, so we typically rely on samples—manageable subsets that represent the whole.

Sample Data

A sample is a subset or fraction of the population. For example:

- 150 fish randomly sampled from the Amazon River (Brazil)
- 5 top-performing students selected from each Ivy League university (USA)
- 10 West African countries selected from all African nations
- 200 randomly chosen households from Lagos (Nigeria)
- 50 giraffes selected from the entire Serengeti population (Tanzania)

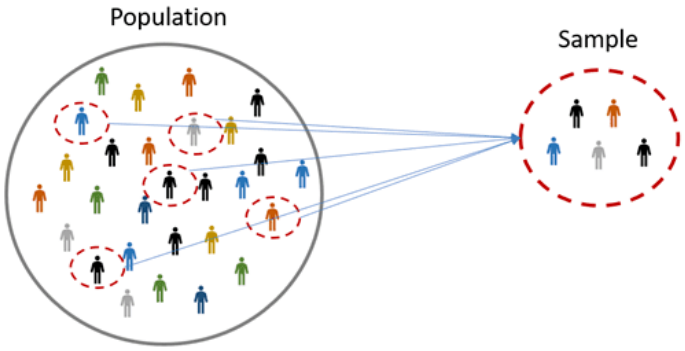


Figure 8.4: Population and Sample Illustration

8.4.2 Parameters and Statistics

Parameter

Parameters are characteristics of a population. They are descriptive measures for an entire population and are typically denoted using Greek letters. For example, the population mean is represented by μ (pronounced "mu"), the variance by σ^2 (sigma squared), the standard deviation by σ (sigma), and the proportion by P.

Statistic

Statistics are characteristics of a sample. They are descriptive measures for a sample and are typically denoted using Roman letters. For example, the sample mean is represented by $\bar{x}$, the variance by s^2, the standard deviation by s, and the proportion by p. These sample statistics are used to estimate the unknown population parameters.

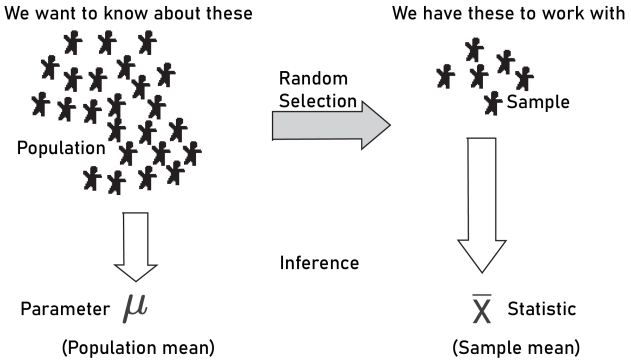


Figure 8.5: Relationship Between Population and Sample

8.4.3 Descriptive Statistics

Descriptive statistics summarise and present the key features of a dataset, enabling you to understand its central tendency, variability, and the shape of its distribution. In this lab, you will use R to calculate these measures.

8.4.3.1 Measures of Central Tendency

The most common measures of central tendency are the mean, median, and mode. They are sometimes called measures of location or averages.

- **Mean**

The arithmetic mean, or simply the mean, is the sum of all elements divided by the number of elements. The sample mean is denoted by $\bar{x}$ and the population mean by μ.

$$\bar{x} = \frac{x_1 + x_2 + \cdots + x_n}{n} \quad \text{and} \quad \mu = \frac{x_1 + x_2 + \cdots + x_N}{N}$$

Example 1: Calculating the Sample Mean

Consider the ages of a study group:

14, 25, 17, 21, 11, 17, 22, 25, 16, and 13.

The sample mean is calculated as follows:

$$\bar{x} = \frac{14 + 25 + 17 + 21 + 11 + 17 + 22 + 25 + 16 + 13}{10} = \frac{181}{10} = 18.1$$

The sample mean age is 18.1 years.

Calculating the Mean in R

```
age <- c(14, 25, 17, 21, 11, 17, 22, 25, 16, 13)

mean(age)
```

```
#> [1] 18.1
```

Example 2:

Professor Francisca, the Vice-Chancellor of Thomas Adewumi University (Kwara, Nigeria) and a Professor of Computer Science, is known for her generosity. Each week, she awards monetary prizes (in dollars) to the best student in the weekly Computer Science assignment for the DTS 204 module. Below are the prize amounts she has given out:

495, 503, 503, 498, 503, 505, 503, 500, 501, 489, 498, 488, 499, 497, 508, 507, 507, 509, 508, and 503.

Calculate the mean amount:

```
money <- c(
  495, 503, 503, 498, 503, 505, 503, 500, 501, 489, 498, 488, 499,
  497, 508, 507, 507, 509, 508, 503
)

mean(money)
```

```
#> [1] 501.2
```

The mean monetary prize is $501.2.

- **Median**

The median is the middle value when the data is arranged in ascending or descending order.

> **i** Note
>
> When the number of observations is odd, the median is the single middle value; when even, it is the average of the two middle values.

Example 1:

Consider the scores of thirteen students who registered for *R for Data Science (CS 202)*:

7, 15, 10, 9, 18, 6, 21, 12, 16, 13, 5, 23, 2.

To determine the median, the data is first sorted in ascending order:

1	2	3	4	5	6	**7**	8	9	10	11	12	13
2	5	6	7	9	10	**12**	13	15	16	18	21	23

Since there are 13 observations (an odd number), the median is the 7th value, which is 12.

Calculating the median in R

The median can also be calculated programmatically using the `median()` function:

```
performance <- c(7, 15, 10, 9, 18, 6, 21, 12, 16, 13, 5, 23, 2)

median(performance)
```

```
#> [1] 12
```

Example 2:

Sixteen students registered for *Introduction to Data Science (CS 212)*, and their scores are:

13, 18, 10, 9, 21, 13, 23, 23, 2, 2, 7, 21, 15, 18, 20, 7.

The median is calculated similarly in R:

```
performance <- c(13, 18, 10, 9, 21, 13, 23, 2, 23, 2, 7, 21, 15, 18, 20, 7)

median(performance)
```

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

- **Mode**

The mode is the value that occurs most frequently in a dataset. For example, consider Sophia, a parent in Lagos, recording the number of steps her baby takes each day. The recorded data is as follows:

0, 2, 6, 2, 2, 0, 0, 1, 1, 5, 3, 1, 0, 2, 3, 1, 2, 1, 4, 4, 5, 0, 5, 1, 2, 2, 2, 0, 4, 0, 6.

To gain insight into this data, one can first create a frequency table that shows how many times each number of steps occurs:

Table 8.3: Frequency of Daily Baby Steps Recorded by Sophia

Number of calls	Frequency
0	7
1	6
2	**8**
3	2
4	3
5	3

Number of calls	Frequency
6	2

This table clearly shows that the value 2 occurs 8 times, which is more frequent than any other number in the dataset. Therefore, the mode of the baby steps data is 2.

Calculating the mode in R

R does not have a built-in function for the mode of a numeric vector, but you can define a custom function:

```
statistical_mode <- function(x) {
  uniqx <- unique(x)
  uniqx[which.max(tabulate(match(x, uniqx)))]
}
```

Explanation:

1. **`unique(x)`** creates a vector of the unique values in **`x`**.
2. **`match(x, uniqx)`** returns a vector of the positions of the elements of **`x`** in **`uniqx`**.
3. **`tabulate(match(x, uniqx))`** counts the number of times each unique value appears in **`x`**.
4. **`which.max(...)`** finds the position of the maximum count.
5. **`uniqx[which.max(...)]`** returns the unique value corresponding to the maximum count, i.e., the mode.

The `statistical_mode()` can be customized further for specific use cases (see Chapter 3.5.1).

Applying this function to the baby steps data:

```
baby_steps <- c(
  0, 2, 6, 2, 2, 0, 0, 1, 1, 5, 3, 1, 0, 2,
  3, 1, 2, 1, 4, 4, 5, 0, 5, 1, 2, 2, 2,
  0, 4, 0, 6
)

statistical_mode(baby_steps)
```

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

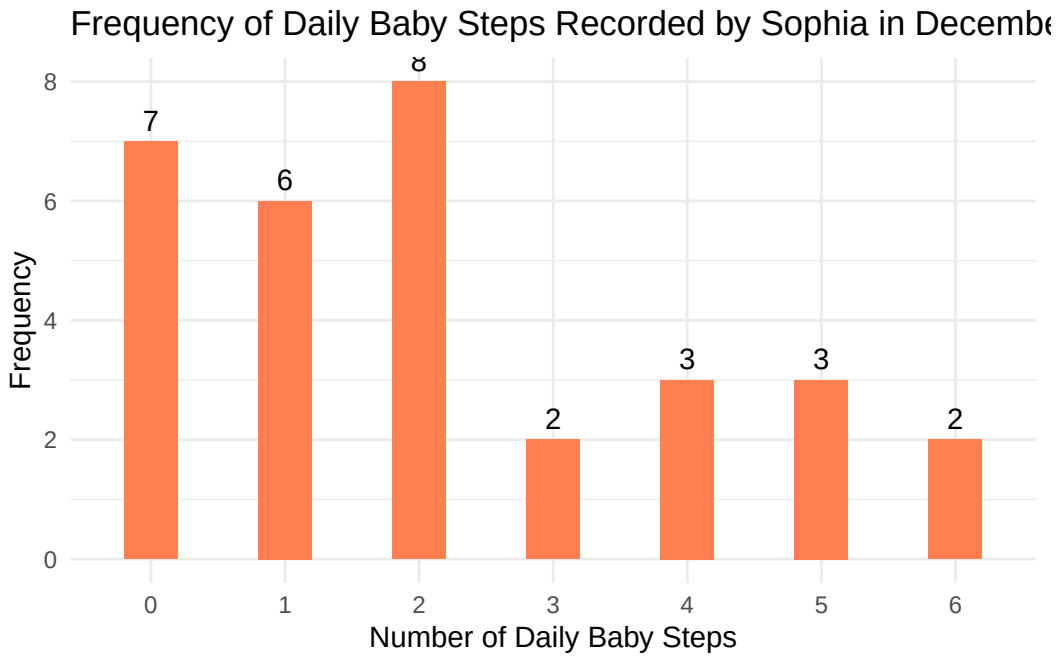
The function returns 2, confirming that the mode is 2.

To illustrate this further, you can visualise the frequency distribution with a bar chart:

```r
# Load necessary library
library(ggplot2)

# Create a frequency table and convert it to a data frame
steps_frequency <- table(baby_steps)
steps_freq_df <- as.data.frame(steps_frequency)

colnames(steps_freq_df) <- c("Number_of_Steps", "Frequency")

# Display the frequency table
steps_freq_df
```

```
#>   Number_of_Steps Frequency
#> 1               0         7
#> 2               1         6
#> 3               2         8
#> 4               3         2
#> 5               4         3
#> 6               5         3
#> 7               6         2
```

```r
# Create a bar chart
steps_freq_df |> ggplot(aes(x = factor(Number_of_Steps), y = Frequency)) +
  geom_bar(stat = "identity", fill = "coral", width = 0.4) +
  theme_minimal() +
  labs(
    title = "Frequency of Daily Baby Steps Recorded by Sophia in December 2024",
    x = "Number of Daily Baby Steps",
    y = "Frequency"
  ) +
  geom_text(aes(label = Frequency), vjust = -0.5)
```



Tip

This bar chart visually represents the distribution of daily baby steps, clearly showing that 2 is the most frequently recorded number of daily steps.

Another example uses data from the United Nations, showing the regional distribution of African countries. Africa has 54 countries grouped into five regions: East, Central, North, Southern, and West Africa. Table 8.4 below shows the number of countries in each region:

Table 8.4: Regional Grouping of African Countries by Number of Nations

Regions	Number of countries
East Africa	18
Central Africa	9
North Africa	6
South Africa	5
West Africa	16

From this table, we can see that East Africa has the largest number of countries (18).

8.4.3.2 Measures of Spread

Measures of spread (or dispersion) describe the variability in the data. Common measures include range, standard deviation, variance, and mean absolute deviation.

- **Range**

The range is the difference between the highest and lowest values in a dataset. It is a simple measure of spread.

For example, in the dataset 14, 25, 17, 21, 11, 17, 22, 25, 16, 13, the maximum value is 25 and the minimum is 11, so the range is $25 - 11 = 14$.

Calculating the Range in R

```
data <- c(14, 25, 17, 21, 11, 17, 22, 25, 16, 13)

max(data) - min(data)
```

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

You can also use the `range()` function to get both the minimum and maximum values:

```
range(data)
```

```
#> [1] 11 25
```

> **Warning**
>
> The range provides only a basic view of variability and can be misleading if there are outliers.

- **Variance and Standard Deviation**

Variance is the average of the squared differences from the mean, indicating how far each value in the dataset is from the mean.

$$\sigma^2 = \frac{\sum(x-\mu)^2}{N} \quad \text{and} \quad s^2 = \frac{\sum(x-\bar{x})^2}{n-1}$$

where:

σ^2 = Population variance

s^2 = Sample variance

$\sum$ = sum of...

μ = population mean

$\bar{x}$ = sample mean

n = sample size

N = population size

The standard deviation is the square root of the variance.

> **Tip**
>
> A small variance indicates that the data points are close to the mean, whereas a large variance suggests that the data points are spread out.

For example, consider the heights (in cm) of six giraffes:

113, 146.5, 132, 70.5, 121, and 55.

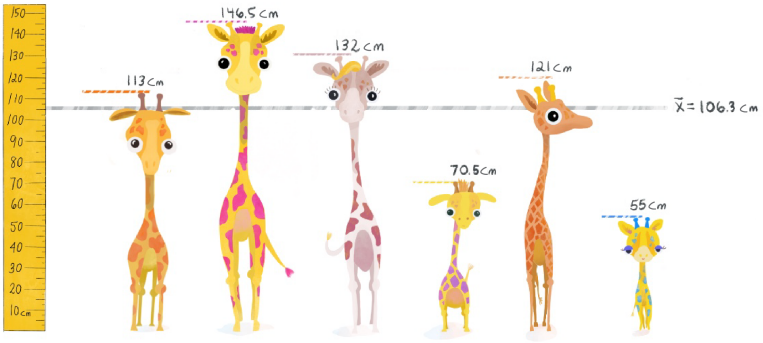


Figure 8.6: Measuring Giraffe Heights

1. **Calculate the Mean**:

   $\bar{x} = \frac{113+146.5+132+70.5+121+55}{6} \approx 106.3$ cm

2. **Compute the Squared Differences from the Mean**:

Table 8.5: Calculation of Squared Deviations from the Mean

x	$x - \bar{x}$	$(x - \bar{x})^2$
113	113 - 106.3 = 6.7	44.89
146.5	146.5 - 106.3 = 40.2	1616.04

x	$x - \bar{x}$	$(x - \bar{x})^2$
132	132 - 106.3 = 25.7	660.49
70.5	70.5 - 106.3 = -35.8	1281.64
121	121 - 106.3 = 14.7	216.09
55	55 - 106.3 = -51.3	2631.69
Total	**0**	**6450.84**

The sum of the squared differences is 6450.84.

3. **Sample Variance**:

$$s^2 = \frac{6450.84}{5} = 1290.17 \, \text{cm}^2$$

4. **Standard Deviation**:

$$s = \sqrt{1290.17} \approx 35.93 \text{ cm}$$

Calculating Variance in R

```
heights <- c(113, 146.5, 132, 70.5, 121, 55)

var(heights)
```

```
#> [1] 1290.167
```

```
sd(heights)
```

```
#> [1] 35.91889
```

> **Note**
>
> Variance is expressed in square units (e.g., cm^2), while the standard deviation is expressed in the original units (e.g., cm), making it easier to interpret.

8.4.3.3 Measures of Partition

Measures of partition divide a distribution into specified equal parts. The most common measures of partition are quartiles and percentiles.

- **Quartile**

Quartiles split the data into four equal parts:

- Q1 (lower quartile)
- Q2 (median)
- Q3 (upper quartile)
- Q4 (which is always the maximum value)

For a dataset with n items arranged in ascending order:

- $Q1$ is the value at the $\frac{n+1}{4}$ position .
- $Q2$ is the value at $\frac{n+1}{2}$ position (the median)
- $Q3$ is the value at $3\left(\frac{n+1}{4}\right)$

Interquartile Range (IQR):

The IQR is the difference between Q3 and Q1: $ IQR = Q3 - Q1 $

For example, thirteen learners took a quiz in *Introduction to Statistical Thinking*, and their marks are:

10, 15, 10, 9, 18, 16, 14, 12, 16, 13, 15, 20, 17.

Sorted in ascending order:

1	2	3	4	5	6	7	**8**	**9**	10	11	12	13
9	10	10	12	13	14	15	15	16	16	17	18	20

- $Q1$ is at the $\left(\frac{13+1}{4}\right) = 3.5^{\text{th}}$ position, so:

  $Q1 = \frac{10+12}{2} = 11$
- $Q3$ is at the $3\left(\frac{13+1}{4}\right) = 10.5^{\text{th}}$ position, so:

  $Q3 = \frac{16+17}{2} = 16.5$
- **Interquartile Range (IQR)**:

  $\text{IQR} = Q3 - Q1 = 16.5 - 11 = 5.5$

Calculating the Quartiles in R

You can compute quartiles using `quantile()` function:

```
quantile(x, na.rm = FALSE)
```

Where:

- **`x`**: The numeric vector
- **`na.rm`**: Logical value indicating whether to remove `NA` values before calculation

```
marks <- c(10, 15, 10, 9, 18, 16, 14, 12, 16, 13, 15, 20, 17)

quantile(marks)
```

```
#>   0%  25%  50%  75% 100%
#>    9   12   15   16   20
```

This returns:

- **0% (Minimum)**: 9
- **25% (Q1)**: 12
- **50% (Median, Q2)**: 15
- **75% (Q3)**: 16
- **100% (Maximum)**: 20

You can also calculate the IQR using::

```
IQR(marks)
```

```
#> [1] 4
```

Which gives $16 - 12 = 4$

- **Percentiles**

Percentiles divide the data into 100 equal parts, indicating the percentage of scores that lie below a particular value. For example, if you are the fourth-tallest person in a group of 20, you are at the 80th percentile because 80% of the group is shorter than you.

Note

Quartiles are specific percentiles:

- $Q1$ = 25th percentile
- $Q2$ = 50th percentile (median)
- $Q3$ = 75th percentile

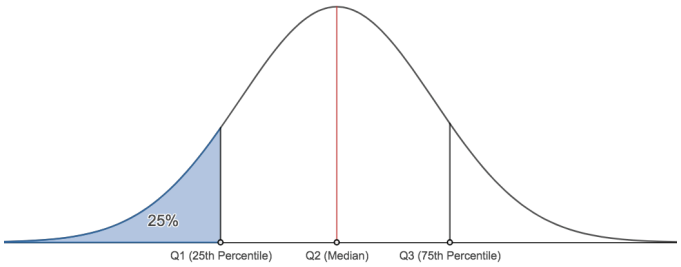


Figure 8.7: Distribution with Quartiles and Percentiles

Calculating Percentiles in R

Use the `quantile()` function with the `probs` argument:

```
quantile(x, probs, na.rm = FALSE)
```

Where:

- **`x`**: The numeric vector
- **`probs`**: Numeric vector of probabilities (between 0 and 1), indicating the desired percentiles (e.g., 0.25 = 25th percentile, 0.5 = 50th percentile, 0.75 = 75th percentile)
- **`na.rm`**: Logical value indicating whether to remove `NA` values before calculation

For example, suppose you have egg weights (in grams):

59, 56, 61, 68, 52, 53, 69, 54, 57, 51.

To find the 25th, 50th, and 75th percentiles:

```
egg_weights <- c(59, 56, 61, 68, 52, 53, 69, 54, 57, 51)

# Calculate the 25th, 50th, and 75th percentiles
quantile(egg_weights, probs = c(0.25, 0.5, 0.75))
```

```
#>   25%   50%   75%
#> 53.25 56.50 60.50
```

This shows:

- **25th Percentile (P25)**: 53.25 grams
  (25% of the egg weights are below 53.25 grams.)
- **50th Percentile (P50 or Median)**: 57.50 grams
  (50% of the egg weights are below 57.50 grams.)
- **75th Percentile (P75)**: 63.25 grams
  (75% of the egg weights are below 63.25 grams.)

8.4.4 Practice Quiz 8.1

Question 1:

A complete set of elements (people, items) that we are interested in studying is called a:

a) Sample

b) Population

c) Parameter

d) Statistic

Question 2:

A subset of a population used to make inferences about the population is called a:

a) Population

b) Sample

c) Statistic

d) Parameter

Question 3:

A value that describes a characteristic of an entire population (e.g., population mean) is known as a:

a) Statistic

b) Parameter

c) Variable

d) Sample estimate

Question 4:

A value computed from sample data (e.g., sample mean) that is used to estimate a population parameter is called a:

a) Parameter

b) Statistic

c) Variable

d) Census

Question 5:

Why do we often rely on samples rather than studying entire populations?

a) It is always more accurate.

b) Populations do not have parameters.

c) Sampling is often more feasible, less costly, and time-efficient

d) Populations are always small and uninteresting.

Question 6:

Statistical thinking involves understanding how to:

a) Manipulate data without purpose

b) Draw meaningful conclusions from data under uncertainty

c) Avoid using data in decision-making

d) Ignore variability in data

Question 7:

If a population parameter is μ, the corresponding sample statistic used to estimate it is typically:

a) s

b) σ

c) $\bar{x}$

d) p

Question 8:

When we attempt to understand the variability in data and the uncertainty in our conclusions, we are engaging in:

a) Statistical thinking

b) Non-statistical reasoning

c) Data neglect

d) Parameter ignorance

Question 9:

If it's too expensive or impractical to study an entire population, we often conduct a:

a) Census

b) Biased survey

c) Sample study

d) Parameter test

Question 10:

The process of using sample data to make conclusions about a larger population is known as:

a) Data summarisation

b) Descriptive statistics

c) Statistical inference

d) Variable classification

See the Solution to Quiz 8.1

8.4.5 Exercise 8.1.2: Professor Francisca - A Generous Giver

Professor Francisca, the Vice-Chancellor of Thomas Adewumi University, Kwara, Nigeria, and a Professor of Computer Science, is known for her generosity. Each week, she awards monetary prizes in dollars to the best student in the weekly Computer Science assignment for the DTS 204 module. The prize amounts are as follows:

495, 503, 503, 498, 503, 505, 503, 500, 501, 489, 498, 488, 499, 497, 508, 507, 507, 509, 508, and 503.

Using R, complete the following tasks to analyse the data:

Task 1: Central Tendency

1. **Calculate the Mean:**
   Determine the average amount of money awarded.
2. **Calculate the Median:**
   Find the median prize amount. Compare the median to the mean and discuss any significant differences.
3. **Determine the Mode:**
   Identify the most frequently occurring amount. Is there more than one mode? What does this indicate about the distribution?

Task 2: Measure of Spread

1. **Calculate the Range**
   Find the range of the amounts. What does this tell you about the variability?
2. **Determine the Standard Deviation**
   Calculate the standard deviation. How does this help in understanding the consistency of the prize amounts?

3. **Find the Variance**
   Compute the variance. Explain how variance relates to standard deviation.

Task 3: Measure of Partition

1. **Calculate the Interquartile Range (IQR)**
   Determine the IQR, which measures the spread of the middle 50% of the amounts. Compare the IQR to the overall range and discuss what this reveals about variability.
2. **Find the Quartiles**
   Identify Q1, the median (Q2), and Q3. Discuss what these quartiles reveal about the distribution.
3. **Calculate Percentile Ranks**
   Determine the percentile ranks for the minimum, maximum, and a selected amount (e.g., $503). Interpret what these percentiles indicate about their position within the distribution.

See the Solution to Exercise 8.1.2

8.5 Experiment 8.2: Five Number Summary and Boxplots

The five-number summary provides a comprehensive overview of a dataset by highlighting five key values:

- Minimum: The smallest observation.
- First Quartile (Q1): The lower quartile.
- Median: The middle value.
- Third Quartile (Q3): The upper quartile.
- Maximum: The largest observation.

These statistics can be visualised using a box plot, which illustrates the distribution of the data and identifies outliers. For an overview of box plots in ggplot2, refer to Chapter 7.8.5; for base plots, see Chapter 7.9.5.

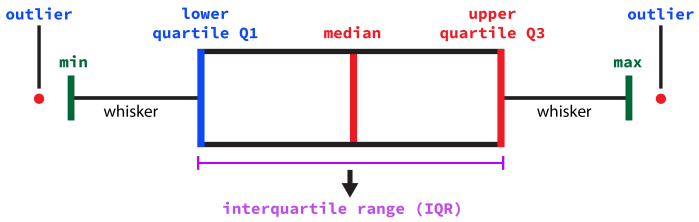
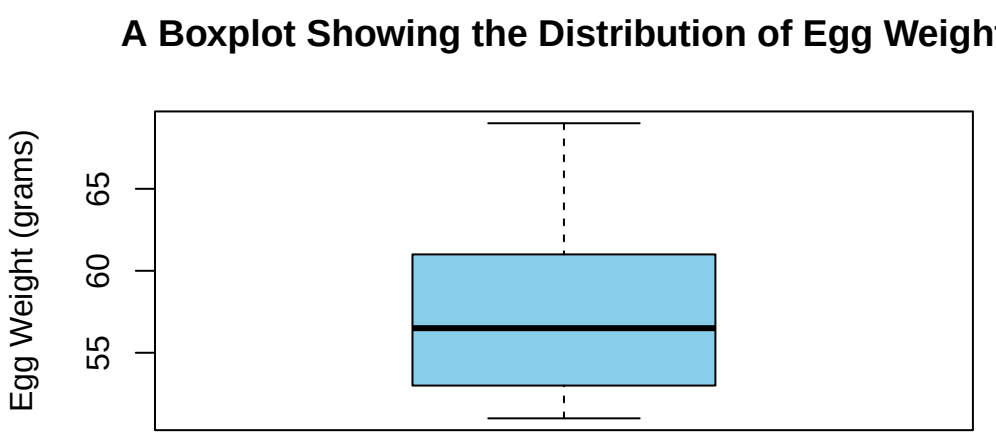


Figure 8.8: Building a Box and Whisker Plot

Consider a dataset of egg weights (in grams): 59, 56, 61, 68, 52, 53, 69, 54, 57, 51. You can visualise this data using the `boxplot()` function in Base R:

```
eggs <- c(59, 56, 61, 68, 52, 53, 69, 54, 57, 51)

boxplot(eggs,
  main = "A Boxplot Showing the Distribution of Egg Weight",
  ylab = "Egg Weight (grams)",
  col = "skyblue"
)
```



> **Note**
>
> This code produces a simple box-and-whisker plot of the egg weights, enabling you to quickly assess the central tendency, spread, and potential outliers.

Another application is analysing life expectancy data across African countries. Data for life expectancy in 54 African countries in 2023, collected by UNICEF, is available in the `africa-life-expectancy.csv` file in the `r-data` directory (or can be downloaded from Google Drive.

To begin, import the necessary packages and data:

```
library(tidyverse)
library(janitor)

# Importing the data
life_expectancy <- read_csv("r-data/africa-life-expectancy.csv")

# Cleaning variable names
life_expectancy <- life_expectancy |> clean_names()

# Viewing the data
life_expectancy
```

```
#> # A tibble: 54 x 2
#>    country                          life_expectancy_in_years_2023
#>    <chr>                                                    <dbl>
#>  1 Angola                                                    64.6
#>  2 Burundi                                                   63.7
#>  3 Benin                                                     60.8
#>  4 Burkina Faso                                              61.1
#>  5 Botswana                                                  69.2
#>  6 Central African Republic                                  57.4
#>  7 Cote d'Ivoire                                             61.9
#>  8 Cameroon                                                  63.7
#>  9 Democratic Republic of the Congo                          61.9
#> 10 Congo                                                     65.8
#> # i 44 more rows
```

If you are from Nigeria, you can filter the data to view Nigeria's life expectancy:

```
life_expectancy |> filter(country == "Nigeria") # Replace "Nigeria" with another country if
```

```
#> # A tibble: 1 x 2
#>   country life_expectancy_in_years_2023
#>   <chr>                           <dbl>
#> 1 Nigeria                          54.5
```

The life expectancy in Nigeria is 54.5.

In this dataset, you can obtain the five-number summary from the variable "Life Expectancy in Years (2023)" using the `summary()` function:

```
life_expectancy |>
  select(life_expectancy_in_years_2023) |>
  summary()
```

```
#>  life_expectancy_in_years_2023
#>  Min.   :54.46
#>  1st Qu.:62.00
#>  Median :65.64
#>  Mean   :65.42
#>  3rd Qu.:68.32
#>  Max.   :76.51
```

Alternatively, you can use the base R `summary()` function directly:

```
summary(life_expectancy$life_expectancy_in_years_2023)
```

```
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>   54.46   62.00   65.64   65.42   68.32   76.51
```

The summary reveals:

- Minimum: 54.46 years
- First Quartile (Q1): 62.00 years
- Median: 65.64 years
- Third Quartile (Q3): 68.32 years
- Maximum: 76.51 years

The Interquartile Range (IQR) is calculated as:

$\text{IQR} = Q3 - Q1 = 68.32 - 62.00 = 6.32$ years

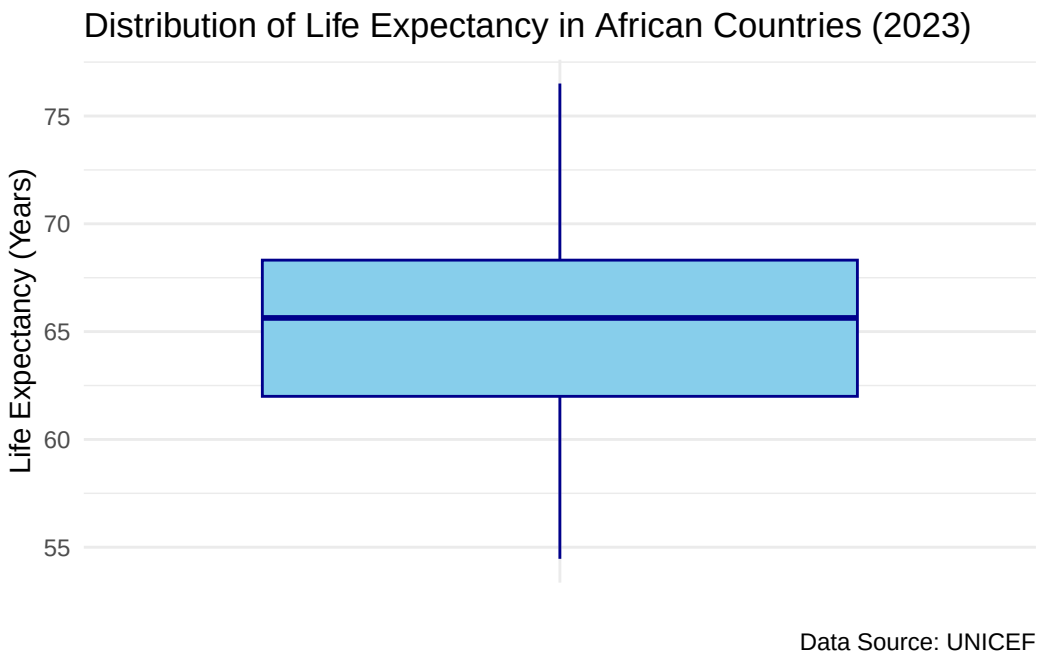
This can be verified in R:

```
IQR(life_expectancy$life_expectancy_in_years_2023)
```

```
#> [1] 6.317
```

To visualise the distribution of life expectancy across African countries, create a box plot using ggplot2:

```
life_expectancy %>%
  ggplot(aes(x = "", y = life_expectancy_in_years_2023)) +
  geom_boxplot(fill = "skyblue", color = "darkblue") +
  theme_minimal() +
  labs(
    title = "Distribution of Life Expectancy in African Countries (2023)",
    x = NULL,
    y = "Life Expectancy (Years)",
    caption = "Data Source: UNICEF"
  )
```



The box plot shows a median life expectancy of roughly 65 years, with the central 50% of observations (from Q1 to Q3) spanning approximately 62 to 68 years. The whiskers extend from around 55 years to nearly 77 years, indicating the minimum and maximum values. Notably, there are no extreme outliers, suggesting that most African countries cluster around a life expectancy in the mid-60s.

8.5.1 Practice Quiz 8.2

Question 1:

Which set of values is included in a five-number summary?

a) Mean, Median, Mode, IQR, Standard Deviation

b) Minimum, Q1, Median, Q3, Maximum

c) Minimum, Mean, Mode, Maximum, Range

d) Q1, Q2, Q3, Q4, Q5

Question 2:

The interquartile range (IQR) is calculated as:

a) Q2 - Q1

b) Q3 - Median

c) Q3 - Q1

d) Median - Minimum

Question 3:

A boxplot is useful for:

a) Displaying frequencies of categorical data

b) Showing the distribution and identifying outliers

c) Calculating correlations between variables

d) Displaying only the mean value

Question 4:

Which value in a five-number summary represents the median of the entire dataset?

a) Q1

b) Q2 (Median)

c) Q3

d) Minimum

Question 5:

If a dataset has many outliers, a boxplot can help by:

a) Ignoring them completely

b) Highlighting them as points beyond the whiskers

c) Removing them automatically

d) Converting them to the mean value

Question 6:

The IQR focuses on the middle 50% of data, making it a good measure of:

a) Central tendency

b) Spread that is not influenced by extreme values

c) Correlation

d) Nominal categories

Question 7:

In R, the `boxplot()` function by default displays:

a) A histogram

b) A correlation matrix

c) A five-number summary depiction

d) A scatter plot

Question 8:

The difference between the maximum and minimum values in a dataset is called the:

a) Standard deviation

b) IQR

c) Range

d) Variance

Question 9:

A box-and-whisker plot typically does NOT show:

a) Median

b) Outliers

c) Mean

d) Interquartile range

Question 10:

When comparing two datasets using boxplots placed side by side, you can quickly assess differences in:

a) Central tendency and spread

b) Exact individual data points

c) Correlation coefficients

d) Detailed frequency distributions

See the Solution to Quiz 8.2

8.5.2 Exercise 8.2.1

Thirty farmers were surveyed about the number of farm workers they employ during a typical harvest season in Igboho, Oyo State, Nigeria. Their responses are:

4, 5, 6, 5, 1, 2, 8, 0, 4, 6, 7, 8, 4, 6, 7, 9, 8, 6, 7, 5, 5, 4, 2, 1, 9, 3, 3, 4, 6, 4.

Task 1:

Calculate the mean, median, and mode of the number of farm workers.

```
farm_workers <- c(
  4, 5, 6, 5, 1, 2, 8, 0, 4, 6, 7, 8, 4, 6, 7, 9,
  8, 6, 7, 5, 5, 4, 2, 1, 9, 3, 3, 4, 6, 4
)

# Code for calculating the mean
...(farm_workers)

# Code for calculating the median
...(farm_workers)

# Code for calculating the mode (using a custom function, e.g., statistical_mode()):
...(farm_workers) # Code for calculating the mode
```

Task 2:

Determine the range and standard deviation of the distribution.

```
# Put your implementation code in Python here

range(...) # Code for calculating the range

...(farm_workers) # Code for calculating the standard deviation
```

Task 3:

Create a box-and-whisker plot of the distribution.

```
...(farm_workers) # Code for creating the boxplot
```

Instructions:

Replace the `...` with the correct R functions and complete the exercise!

See the Solution to Exercise 8.2.1

8.6 Experiment 8.3: Scales of Measurement

Scales of measurement are foundational concepts in statistics and social sciences, as they determine how data is categorised, interpreted, and analysed. Each scale captures a different level of detail and influences the statistical techniques you use. The four scales—nominal, ordinal, interval, and ratio—form a hierarchy, with each subsequent scale offering more detailed information.

8.6.1 Nominal Scale

The nominal scale is the most basic level, categorising data into distinct groups without any inherent order. Essentially, it is used for "naming" data.

Key Features:

- Categories: Data is divided into groups or classes..
- No Order: There is no logical sequence.
- Statistics: Analysis is typically limited to frequency counts and mode.

Examples:

- Gender: Male, Female
- Blood Type: A, B, AB, O

- Ethnicity: Asian, African, European, Latin American
- Marital Status: Single, Married, Divorced, Widowed

> **Note**
>
> In social sciences, nominal data often arises in demographic variables or when categorising types of behaviour.

How to Analyse Nominal Data:

- Use a frequency table to summarise occurrences.
- Visualise with a bar chart or pie chart.
- Perform a Chi-square test to examine relationships between nominal variables.

8.6.2 Ordinal Scale

The ordinal scale introduces a sense of order or rank, although the intervals between ranks are not necessarily equal.

Key Features:

- Order: Data can be ranked or ordered.
- Unequal Intervals: Differences between ranks are not quantifiable.
- Statistics: Median and mode are appropriate, but the mean is not.

Examples:

- Socioeconomic Status: Low, Medium, High
- Customer Satisfaction: Very Unsatisfied, Unsatisfied, Neutral, Satisfied, Very Satisfied
- Political Ideology: Left-wing, Centre-left, Centre, Centre-right, Right-wing
- Therapy Effectiveness Ratings: Effective, Somewhat Effective, Not Effective
- Severity of an Issue: Minor, Moderate, Severe

> **Note**
>
> Ordinal data is common in surveys and questionnaires where responses are ranked, but the intervals between them are not equal.

How to Analyse Ordinal Data:

- Use the median or rank the data.
- Apply non-parametric tests like the Mann-Whitney U test or Kruskal-Wallis test for comparisons.
- Visualise with an ordered bar chart.

8.6.3 Interval Scale

The interval scale applies to numerical data with equal intervals between values, though it lacks a true zero point.

Key Features:

- Equal Intervals: The difference between values is consistent.
- No True Zero: Zero is arbitrary; thus, ratios are not meaningful.
- Statistics: Suitable for calculating the mean, median, and standard deviation.

Examples:

- Temperature (Celsius or Fahrenheit)
- Test Scores (e.g., IQ scores)
- Intelligence Quotient (IQ): A score of 0 does not mean "no intelligence".
- Personality Test Scores: Scales measuring extraversion or agreeableness.
- Psychological Inventories: Depression scale scores, anxiety scale scores.

> **Note**
>
> Interval scales are used in assessments where the distance between scores is meaningful, but ratios (e.g., "twice as much") are not.

How to Analyse Interval Data:

- Calculate the mean, variance, and standard deviation.
- Use parametric tests like the t-test or ANOVA.
- Visualise with a histogram or boxplot.

8.6.4 Ratio Scale

The ratio scale is the highest level of measurement. It incorporates all the properties of the interval scale and includes an absolute zero, which allows for the calculation of meaningful ratios.

Key Features:

- Absolute Zero: Zero indicates the absence of the quantity.
- Equal Intervals: Differences between values are consistent.
- Statistics: All statistical operations (including geometric mean) can be applied.

Examples:

- Weight (kilograms or pounds)
- Height (centimeters or inches)
- Reaction Time (seconds)
- Memory Recall (number of items)
- Physiological Measures (heart rate, cortisol levels)

> Tip
>
> Ratio scales are particularly relevant in the social sciences when dealing with measurable quantities that have a true zero point.

How to Analyse Ratio Data:

- Perform any statistical operation: mean, median, mode, variance, and so on.
- Use parametric tests for advanced analysis.
- Visualise with scatter plots, histogams, or line graphs.

> Recognising Scales in Practice
>
> When working with data, ask yourself:
>
> 1. Is the data categorical or numerical?
> 2. If categorical, does it have an inherent order (ordinal) or not (nominal)?
> 3. If numerical, does it have a true zero (ratio) or is zero arbitrary (interval)?
>
> Understanding these distinctions will guide your choice of statistical methods.

Table 8.7 summarises which measures can be applied to each scale:

Table 8.7: Scale of Measurement and Measures of Central Tendency

Measure	Nominal	Ordinal	Interval	Ratio
Sequence established	–	Yes	Yes	Yes
Mode	Yes	Yes	Yes	Yes
Median	–	Yes	Yes	Yes
Mean	–	–	Yes	Yes
Difference between values	–	–	Yes	Yes
Addition and subtraction	–	–	Yes	Yes
Multiplication and division	–	–	–	Yes
Absolute zero	–	–	–	Yes

8.6.5 Practice Quiz 8.3

Question 1:

A scale that categorises data without any order is known as:

a) Nominal

b) Ordinal

c) Interval

d) Ratio

Question 2:

Which scale provides both order and equal intervals but no true zero point?

a) Nominal

b) Ordinal

c) Interval

d) Ratio

Question 3:

Which scale allows for meaningful ratios and has a true zero?

a) Nominal

b) Ordinal

c) Interval

d) Ratio

Question 4:

Educational levels ranked as “Primary, Secondary, Tertiary” represent which scale?

a) Nominal

b) Ordinal

c) Interval

d) Ratio

Question 5:

Temperatures in Celsius or Fahrenheit are examples of which scale?

a) Nominal

b) Ordinal

c) Interval

d) Ratio

Question 6:

Blood types (A, B, AB, O) are measured on which scale?

a) Nominal

b) Ordinal

c) Interval

d) Ratio

Question 7:

The number of items sold in a store (e.g., 0, 5, 10 units) is best described by which scale?

a) Nominal

b) Ordinal

c) Ratio

d) Interval

Question 8:

Customer satisfaction ratings (e.g., Satisfied, Neutral, Unsatisfied) belong to which scale?

a) Nominal

b) Ordinal

c) Interval

d) Ratio

Question 9:

A key difference between interval and ratio scales is that ratio scales have:

a) Categories only

b) A meaningful zero point

c) No ordering capability

d) Equal intervals that are meaningless

Question 10:

IQ scores are often treated as which type of scale?

a) Nominal

b) Ordinal

c) Interval

d) Ratio

See the Solution to Quiz 8.3

8.6.6 Exercise 8.3.1: Identify the Scale

Now it's your turn to practice. For each example below, identify the correct scale of measurement:

- Blood pressure readings (e.g., 120 mmHg, 130 mmHg)
- The type of car owned (e.g., Sedan, SUV, Truck)
- Rankings in a cooking competition (e.g., 1st, 2nd, 3rd)
- Test scores out of 100 (e.g., 85, 90, 75)
- Age of students in years

See the Solution to Exercise 8.3.1

8.7 Reflective Summary

In this lab, you have gained knowledge and skills in statistical thinking:

- **Concepts of Statistics**: You learned the foundational principles of statistics and how they are applied to extract valuable insights from raw data. Understanding these principles is crucial for interpreting data meaningfully.
- **Types of Data**: You explored the differences between qualitative and quantitative data, recognising their significance in statistical analysis. You also learned how to categorise data for appropriate analysis.
- **Scales of Measurement**: You discovered the four scales of measurement—nominal, ordinal, interval, and ratio—and how to apply them to different data types, ensuring that you use the correct statistical techniques for each.
- **Descriptive Statistical Analysis**: You learned how to calculate and interpret key measures of central tendency (mean, median, mode) and measures of spread (range, variance, standard deviation), essential for summarising and understanding datasets.
- **Visualising Data**: You practised creating boxplots in R and interpreting the five-number summary to visually communicate key insights from your data.

These skills provide a solid base for further exploration in statistical analysis and data science, enabling you to interpret and apply statistical methods to real-world problems with confidence.

What's Next?

In the next lab, we will delve into sampling techniques. You will build on these skills by exploring core sampling methods, distinguishing between probability and non-probability approaches, and applying these techniques in R to design robust studies and draw reliable inferences from your data.

9 Sampling Techniques

9.1 Introduction

Welcome to Lab 9, where we will focus on understanding and applying sampling techniques. In the world of data analysis and statistics, drawing reliable conclusions often depends on how we collect our data. Since it's rarely feasible—financially or logistically—to gather information from every member of a population, we turn to sampling. **Sampling** is the process of selecting a subset of individuals, observations, or objects from a larger population. If done properly, sampling allows us to save time, money, and effort while still gaining valuable insights that can accurately represent the entire group.

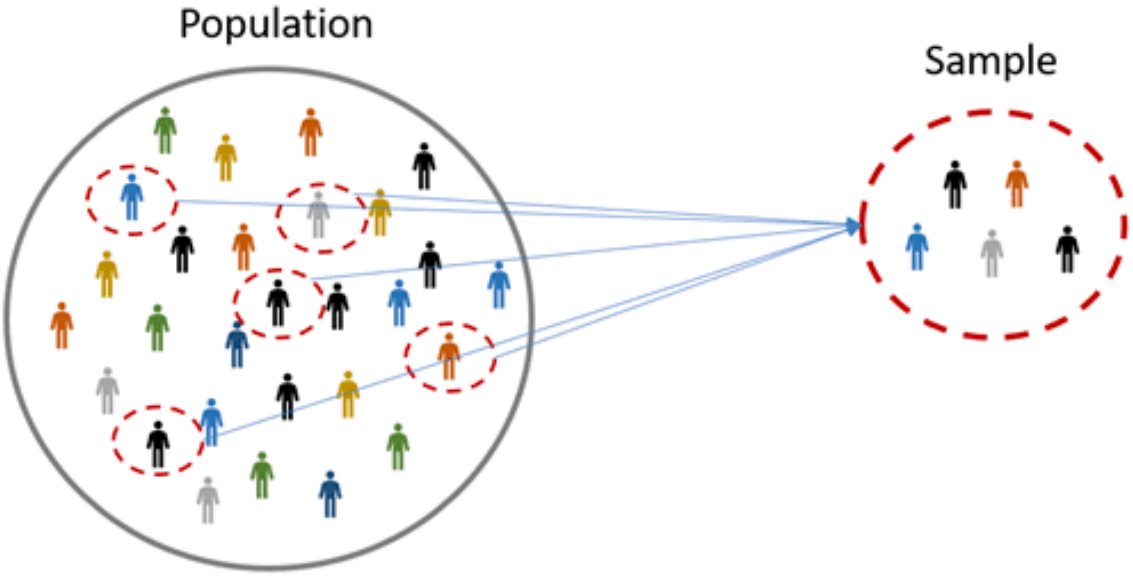


Figure 9.1: Population and Sample Illustration

Sampling is a cornerstone of research, and how you choose your sample can determine the accuracy and credibility of your conclusions. Consider the challenge of predicting election results: it would be impossible to survey every single voter. Instead, we carefully select a

smaller group that reflects the overall population. By choosing the right sampling technique, we can make meaningful predictions and avoid misleading outcomes.

In this lab, we will explore both probability and non-probability sampling methods. You'll learn about a range of techniques, their pros and cons, and the situations in which each method is most appropriate. We will also practice implementing these approaches in R, allowing you to reinforce these concepts with hands-on experience.

9.2 Learning Objectives

By the end of this lab, you should be able to:

- **Understand Probability vs. Non-Probability Sampling**:
  Recognize how probability sampling allows for generalizing results, while non-probability methods are often used for exploratory or hard-to-reach populations.
- **Describe Common Probability Sampling Methods**:
  Learn about simple random, stratified, cluster, and systematic sampling, and understand when to use each method.
- **Describe Common Non-Probability Sampling Methods**:
  Understand convenience, snowball, judgmental (purposive), and quota sampling, and appreciate their limitations.
- **Match Methods to Research Scenarios**:
  Identify which sampling strategy is most suitable given the research goals, data availability, and constraints.
- **Reflect on Trade-Offs**:
  Consider the strengths and weaknesses of different sampling approaches and how these choices affect your ability to generalize findings.

By completing this lab, you'll be better equipped to design robust studies, interpret results confidently, and ensure that your data-driven conclusions stand on a solid methodological foundation.

9.3 Why Do We Sample?

Imagine you want to understand the average height of all adults in your country. Actually measuring every adult's height would be incredibly difficult, time-consuming, and expensive. Instead, you might measure a carefully chosen group (sample) that fairly represents the entire population. From this sample, you can estimate the overall average height. But if your

sample is biased or poorly chosen, your conclusions may be misleading. That's why thoughtful sampling techniques matter.

9.4 Sampling Terminology

- **Population**: The entire group of individuals or items of interest.
- **Sample**: A subset of the population selected for analysis.
- **Sampling Frame**: A list or other resource that identifies all or most members of the population, from which we select the sample.
- **Parameter**: A numerical summary (e.g., mean, proportion) that describes some characteristic of the population.
- **Statistic**: A numerical summary that describes some characteristic of the sample, used to estimate the corresponding population parameter.

9.5 Understanding Probability and Non-Probability Sampling

In research, the strategies we use to select samples can vary greatly depending on the discipline, research area, and specific study. Broadly speaking, there are two main types of sampling methods: probability sampling and non-probability sampling.

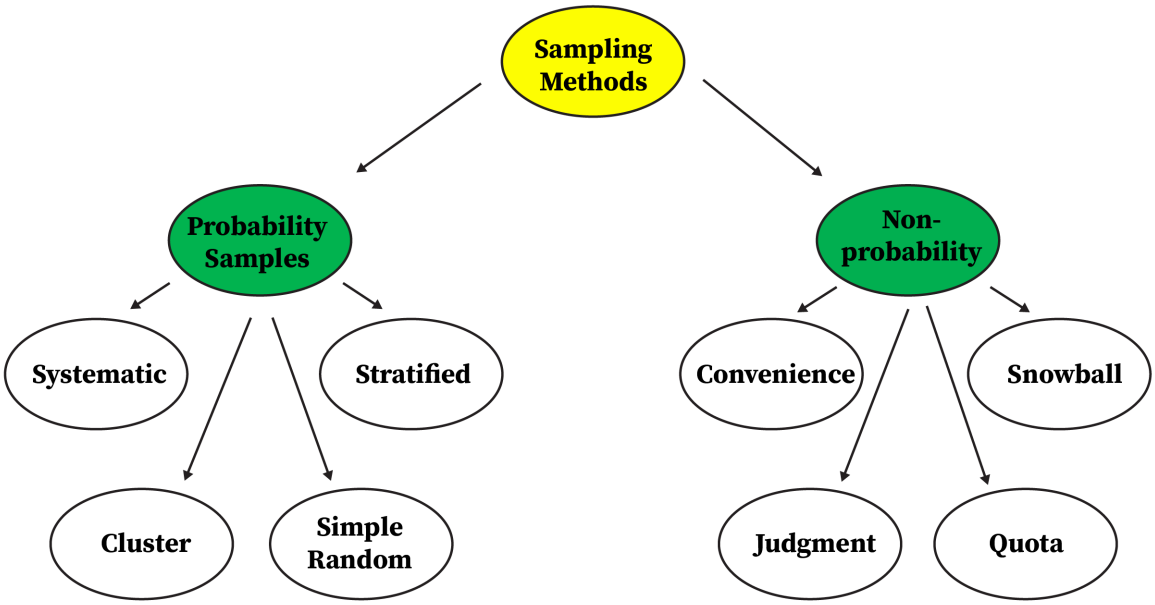


Figure 9.2: Overview of Sampling Methods

Probability Sampling

Probability Sampling methods ensure that every member of the population has a known chance of being included in the sample. This randomness allows you to measure how much uncertainty exists in your estimates. With probability sampling, statistical theory helps you gauge how closely your sample results match the real population values.

Non-Probability Sampling

Non-probability sampling methods do not rely on random selection; instead, they are based on subjective judgement, convenience, referrals, or other non-random criteria. Because not every member of the population necessarily has a known or equal chance of being included, these approaches often lack representativeness and make it harder to generalize results with confidence. Nevertheless, they are commonly used in exploratory research, when dealing with hidden populations, or under severe time and resource constraints, even though their inherent bias can complicate accuracy and limit the applicability of findings.

> Reflection Question 1
>
> Why might you choose a non-probability sampling method if it doesn't allow you to confidently generalize findings to the entire population?

9.6 Experiment 9.1: Probability Sampling Techniques

9.6.1 Simple Random Sampling (SRS)

In a simple random sample, every member of the population has an equal probability of being selected. This is often considered the "gold standard" because it tends to produce unbiased estimates if done correctly. It's like pulling names out of a hat—no individual is favoured over another.

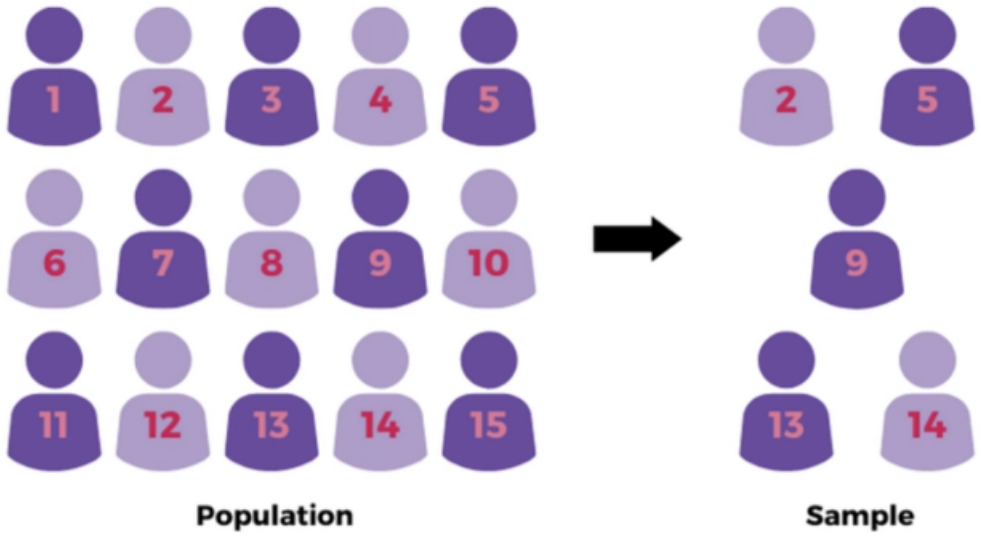


Figure 9.3: Simple Random Sampling Process

When to Use:

- When you have a well-defined population and a good sampling frame.
- When you want each unit in the population to have an equal chance of selection.
- When you do not need to target specific subgroups.

Pros:

Minimizes selection bias and is straightforward to implement if a complete list (sampling frame) exists.

Cons:

Can be difficult when the population is very large or when no complete sampling frame is available.

Example Scenario:

Suppose a university wants to know the average study time of its undergraduate students. Since the university has access to a complete list of all undergraduates, it can take a simple random sample of a few hundred students to estimate the overall average study time. Each student in the population would be equally likely to be selected, ensuring an unbiased estimate if done correctly.

Example Scenario using R:

Before coding, imagine having a large jar with 10,000 student IDs. To find out the average time they spend studying, you wouldn't ask all 10,000 students—too time-consuming. Instead, you mix the IDs thoroughly and draw 500 at random, giving each student an equal chance of selection. In R, we'll recreate this process by using a numeric vector of 10,000 IDs and randomly selecting 500 from it. This smaller group will help us reliably estimate the overall average study time. Here's how:

```r
# Ensures that we get the same random sample each time for reproducibility
set.seed(123)

# Imagine this is our complete list of undergraduates, each assigned a unique ID
student_ids <- 1:10000

# Draw a simple random sample of 500 students from the 10,000
sample_srs <- sample(student_ids, size = 500, replace = FALSE)

# Shows the first few sampled student IDs
head(sample_srs)
```

```
#> [1] 2463 2511 8718 2986 1842 9334
```

By running this code, you'll see a handful of randomly chosen IDs. These represent your simple random sample—your mini version of the entire student body—ready for you to contact and measure their study hours.

9.6.2 Exercise 9.1.1: Simple Random Sampling with the Penguins Dataset

Use the **penguins** dataset from the **palmerpenguins** package to perform a simple random sample of 10 penguins. Compare the mean body mass of this sample to the mean body mass of the entire dataset.

Steps:

1. Load the **palmerpenguins** package and examine the `penguins` dataset.
2. Remove any rows with missing values to ensure you have a complete dataset.
3. Set a seed for reproducibility.
4. Select a simple random sample of 10 penguins from the complete dataset.
5. Calculate the mean body mass of the entire dataset.
6. Calculate the mean body mass of your sample.
7. Compare these two means and reflect on any differences.

See the Solution to Exercise 9.1.1

9.6.3 Stratified Sampling

Stratified sampling involves dividing the population into distinct subgroups (called strata) and then taking a proportional random sample from each subgroup. Strata are often formed based on characteristics you care about—like gender, age range, or region.

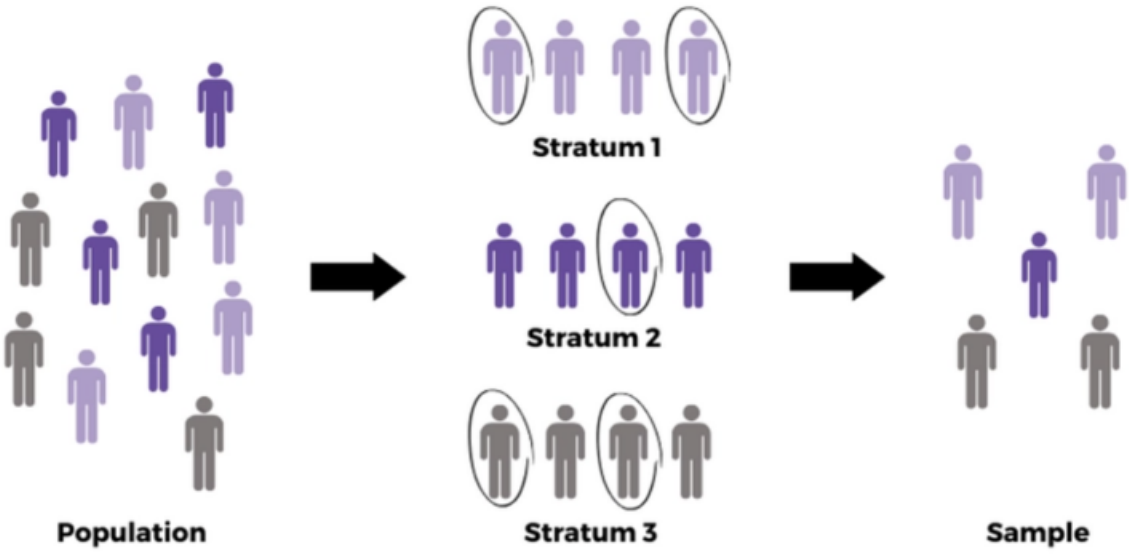


Figure 9.4: Stratified Sampling Process

When to Use:

- When the population is heterogeneous, and you want to ensure representation from all key subgroups.
- When you know something about how the population differs and you want your sample to reflect those differences accurately.

Pros:

Ensures that important subgroups are properly represented, leading to more precise estimates.

Cons:

Requires that you can clearly define and identify meaningful strata.

Example Scenario:

Suppose you want to survey residents of a country about their internet usage. You know age affects usage patterns, so you split the population into age groups—like 18–29, 30–49, and 50+—and then pick a sample from each group in proportion to their presence in the entire population. By doing this, you ensure each age category is fairly represented, resulting in a more balanced and accurate sample.

Example Scenario using R:

Now, let's see how to perform stratified sampling based on age groups in R:

```
# Load dplyr for data manipulation
library(dplyr)

# Ensures reproducibility
set.seed(123)

# Create a hypothetical population dataset with age groups
population_data <- data.frame(
  id = 1:1000,
  age_group = sample(c("18-29", "30-49", "50+"), size = 1000, replace = TRUE)
)

# Check the proportions of each age group in the population
prop.table(table(population_data$age_group))
```

```
#>
#> 18-29 30-49   50+
#> 0.326 0.331 0.343
```

```r
# Perform stratified sampling: take 10% from each age group,
# maintaining the overall proportion of each age group
stratified_sample <- population_data %>%
  group_by(age_group) %>%
  sample_frac(0.1)

# Check the proportions in the stratified sample
prop.table(table(stratified_sample$age_group))
```

```
#>
#> 18-29 30-49   50+
#>  0.33  0.33  0.34
```

> Reflection Question 2
>
> How might stratified sampling improve the accuracy of your results compared to simple random sampling when the population is made up of very different subgroups?

9.6.4 Exercise 9.1.2: Stratified Sampling with the Diamonds Dataset

Use the **diamonds** dataset from the **ggplot2** package to perform stratified sampling based on the `cut` variable. Ensure your sample maintains similar proportions of each cut category as in the full dataset, then compare these distributions.

Steps:

1. Load the **ggplot2** package and review the `diamonds` dataset.
2. Identify the `cut` variable and examine its distribution in the full dataset.
3. Determine the proportion of each `cut` category.
4. Choose a sample size (e.g., 500 diamonds) and perform stratified sampling to maintain these proportions.
5. Compare the distribution of `cut` categories in your stratified sample to that in the full dataset.

See the Solution to Exercise 9.1.2

9.6.5 Cluster Sampling

In cluster sampling, the population is divided into naturally occurring groups (clusters), such as households, schools, or city blocks. Instead of sampling individuals across the entire population, you randomly select a few clusters and then measure all or a sample of the units within those selected clusters.

Figure 9.5: Cluster Sampling Process

When to Use:

- When the population is very large and spread out, making it difficult to select a simple random sample from the entire population.
- When cost or logistical constraints make it easier to collect data from a few whole groups rather than scattering your efforts all over the place.

Pros:

Cost-effective and practical when dealing with large, geographically spread-out populations.

Cons:

The variability within clusters can affect precision. If clusters differ greatly from each other, a few selected clusters may not represent the entire population well.

Example Scenario:

Imagine you want to conduct a national health survey. Your population is huge and spread across hundreds of hospitals around the country. Visiting every hospital or sampling patients one by one from all over would be overwhelming and expensive.

Instead, you randomly pick a small number of hospitals (clusters). Within these selected hospitals, you either survey every patient or take a smaller random sample of patients there.

Example Scenario using R:

Now, let's see how we can simulate this process in R:

```
set.seed(123)

# Suppose we have 100 hospitals (clusters), each with 50 patients
hospital_id <- rep(1:100, each = 50) # 100 hospitals, each with 50 patients

# Create a data frame of patients, assigning hypothetical health measures:
# Blood Pressure and BMI
patient_df <- data.frame(
  patient_id = 1:5000,
  hospital = hospital_id,
  blood_pressure = rnorm(5000, mean = 120, sd = 15),
  bmi = rnorm(5000, mean = 25, sd = 4)
)

# Randomly select 5 hospitals (clusters) for the study
selected_hospitals <- sample(unique(patient_df$hospital),
  size = 5, replace = FALSE
)

# Extract the data for patients in the selected hospitals
cluster_sample <- patient_df |> filter(hospital %in% selected_hospitals)

cluster_sample |> head()
```

```
#>   patient_id hospital blood_pressure      bmi
#> 1        501       11       110.9716 19.57108
#> 2        502       11       105.0945 19.82921
#> 3        503       11       135.4018 18.93117
#> 4        504       11       131.2659 28.43670
#> 5        505       11        97.3625 20.14153
#> 6        506       11       118.5728 27.47622
```

```r
# Check how many patients we have from each selected hospital

cluster_sample |> count(hospital)
```

```
#>   hospital  n
#> 1       11 50
#> 2       24 50
#> 3       50 50
#> 4       59 50
#> 5       95 50
```

In this example, we use cluster sampling by first selecting 5 hospitals (clusters) from the 100 available. After choosing these clusters, we create a `cluster_sample` that includes all patients from the selected hospitals. By surveying every patient within these clusters rather than just a subset, we simplify data collection, reduce costs, and focus our efforts on a few representative groups rather than trying to survey patients from every hospital. This approach still allows us to gather valuable information, such as blood pressure and BMI, to better understand the broader population's health.

9.6.6 Exercise 9.1.3: Cluster Sampling with a Simulated Dataset

In this exercise, you will create a simulated dataset of customers grouped by city. Each city represents a cluster. You will then perform cluster sampling by selecting a few cities and comparing a key characteristic (such as average customer spending) in the sampled clusters versus the entire population.

Steps:

1. Create a simulated dataset of customers, assigning each customer to a city (cluster).
2. Assign some characteristic to each customer (e.g., monthly spending).
3. Randomly select a few cities (clusters).
4. Extract all customers from the selected cities to form your cluster sample.
5. Compare the overall characteristics (e.g., mean monthly spending) of the cluster sample to those of the entire population.

See the Solution to Exercise 9.1.3

9.6.7 Systematic Sampling

Systematic sampling involves selecting every kth individual from a list or sequence after starting at a random point. For example, if you have a list of 10,000 customers and you need a sample of 500, you would pick every (10000/500) = 20th customer after a random start.

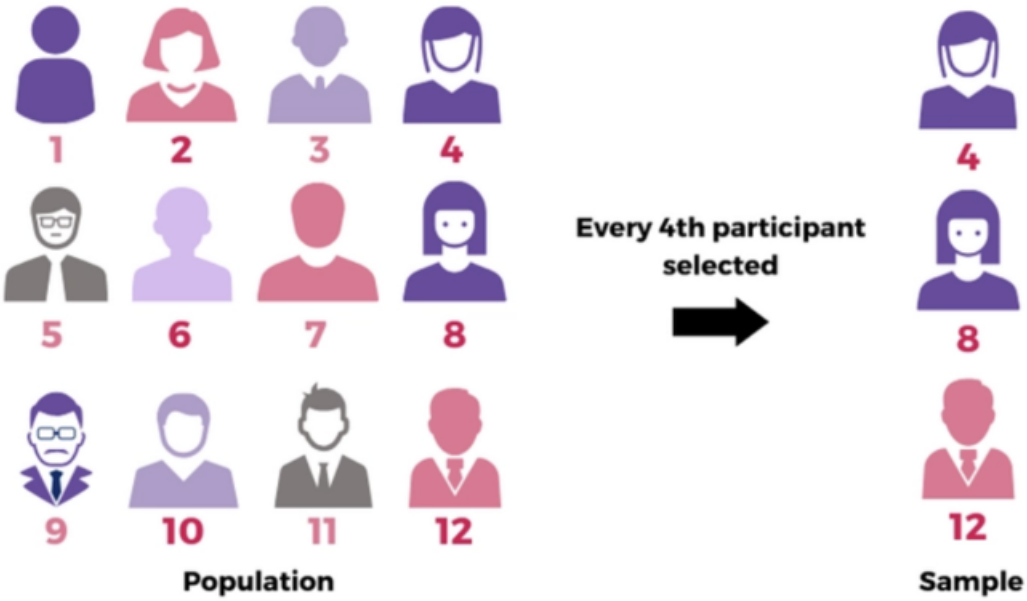


Figure 9.6: Systematic Sampling Process

When to Use:

- When you have a complete list of the population in a random order.
- When it's easier to pick samples at regular intervals rather than random draws.

Pros:

Simple and efficient once you have a list, and ensures a fairly even spread of the sample across the population.

Cons:

Can introduce bias if the list has a hidden pattern that coincides with the selection interval.

Example Scenario:

Suppose your factory produces 10,000 identical parts in a day (that's your total population). You want to inspect 500 of these parts to ensure quality—a big enough sample to spot any

consistent issues. Rather than randomly stopping the production line and risking inefficiency, you decide to pick every 20th item (10,000/500 = 20) for inspection after a randomly chosen start point between 1 and 20.

Example Scenario using R:

Now, let's see how to simulate this process in R.

```r
set.seed(123) # Ensure reproducibility

# Suppose the factory produces 10,000 parts a day
total_parts <- 10000

# We want to inspect 500 parts
sample_size <- 500

# Determine the sampling interval (k)
k <- total_parts / sample_size

# Choose a random start point between 1 and k
start <- sample(1:k, 1)

# Create a sequence of parts to inspect: every k-th part starting from 'start'
inspected_parts <- seq(from = start, to = total_parts, by = k)

# Look at the first few part IDs selected
head(inspected_parts)
```

```
#> [1]  15  35  55  75  95 115
```

This systematic approach ensures a steady, predictable pattern: once you set your start, you simply check every 20th part as it comes off the line. By the end of the day, you'll have a representative sample of 500 parts spaced evenly throughout the entire production run.

9.6.8 Exercise 9.1.4: Systematic Sampling on a Simple List

Use systematic sampling to select individuals from a simple list of IDs (e.g., `1:1000`). Your desired sample size is 100. Choose an appropriate interval k, select every kth individual, and verify that the resulting sample follows the intended pattern. Experiment with different values of k to see how the sample changes.

Steps:

1. Create a vector of individuals (e.g., `1:1000`).
2. Set the desired sample size to 100.
3. Calculate the interval k (for a list of 1000 individuals and a sample of 100, $k = 1000/100 = 10$).
4. Randomly select a starting point between 1 and k.
5. Choose every kth element after that starting point.
6. Check that the sample size is correct and that each selection is spaced by k.
7. Experiment with different k values (e.g., 20 or 50) and observe the differences.

See the Solution to Exercise 9.1.4

9.6.9 Practice Quiz 9.1: Probability Sampling

Question 1:

What is the defining feature of probability sampling methods?

a) They always use large sample sizes

b) Each member of the population has a known, nonzero chance of selection

c) They never require a sampling frame

d) They rely on the researcher's judgment

Question 2:

In simple random sampling (SRS), every member of the population:

a) Has no chance of being selected

b) Is selected to represent different subgroups

c) Has an equal probability of being selected

d) Is chosen based on convenience

Question 3:

Stratified sampling involves:

a) Selecting whole groups at once

b) Sampling every kth individual

c) Ensuring subgroups are represented proportionally

d) Selecting individuals recommended by others

Question 4:

Which method is best if you know certain subgroups (strata) differ and you want each to be represented in proportion to their size?

a) Simple random sampling

b) Stratified sampling

c) Cluster sampling

d) Convenience sampling

Question 5:

Cluster sampling is typically chosen because:

a) It is guaranteed to be perfectly representative

b) It reduces cost and logistical complexity

c) It involves selecting individuals from every subgroup

d) It ensures each individual has the same probability of selection as in SRS

Question 6:

In a national health survey using cluster sampling, which of the following represents a "cluster"?

a) A randomly chosen patient from all over the country

b) A randomly selected set of hospitals

c) A proportionate sample of age groups

d) Every 10th patient in a hospital list

Question 7:

Systematic sampling selects individuals by:

a) Relying on personal judgment

b) Selecting every kth individual after a random start

c) Dividing the population into strata

d) Choosing only those easiest to reach

Question 8:

If the population is 10,000 units and you need a sample of 100, the interval k in systematic sampling is:

a) 10 (10,000 ÷ 1,000)

b) 100 (10,000 ÷ 100)

c) 20 (10,000 ÷ 500)

d) 50 (10,000 ÷ 200)

Question 9:

One advantage of systematic sampling is:

a) It ensures no bias will ever occur

b) It provides a convenient and even spread of the sample

c) It requires no sampling frame

d) It automatically includes all subgroups

Question 10:

Which of the following is NOT a probability sampling method?

a) Simple random sampling

b) Stratified sampling

c) Cluster sampling

d) Convenience sampling

See the Solution to Quiz 9.1

9.7 Experiment 9.2: Non-Probability Sampling Techniques

9.7.1 Convenience Sampling

Convenience sampling involves selecting whichever individuals are easiest to reach. This is a non-probability method, meaning it does not rely on random selection. Because it doesn't ensure every member of the population has a chance to be selected, it can produce biased results.

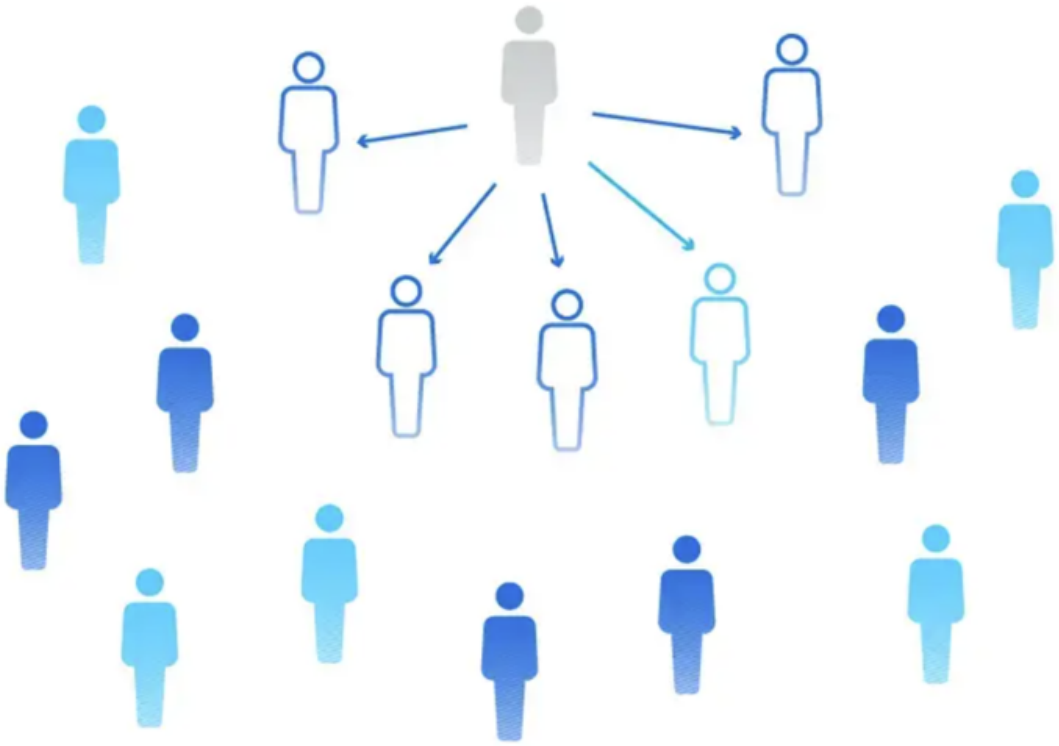


Figure 9.7: Convenience Sampling Illustration

When to Use:

- Often used in preliminary studies or quick polls when resources are limited and accuracy is not the top priority.
- When no sampling frame is available, and some data is better than none (with caution).

Pros:

Quick, inexpensive, and easy when you just need preliminary insights.

Cons:

High potential for bias, as the sample may not represent the broader population.

Example Scenario:

A student researcher stands outside the campus cafeteria and surveys the first 50 people who walk out. This approach is simple and fast, but it may not reflect the broader student population's opinions or habits.

Example Scenario using R:

```
set.seed(123)
# Suppose we have a dataset of 1,000 students, each with a favorite cafeteria meal
students <- data.frame(
  student_id = 1:1000,
  favorite_meal = sample(c("Pizza", "Salad", "Burger"), 1000, replace = TRUE)
)

# A convenience sample might just be the first 50 students in the dataset
convenience_sample <- head(students, 50)

# Check the distribution of favorite meals in the convenience sample
table(convenience_sample$favorite_meal)
```

```
#>
#> Burger  Pizza  Salad
#>     19     16     15
```

Here, the researcher did not randomly select the students. Instead, they simply took whoever was most convenient (the first 50 encountered). While easy, this sample may not accurately represent the entire student body's meal preferences.

9.7.2 Snowball Sampling

Snowball also known as referral or respondent-driven sampling is used when you have difficulty identifying or accessing members of your target population. In this approach, you start by contacting a few known individuals (called "seeds") who fit your criteria. These initial participants then refer you to others who share similar characteristics or experiences, and those people, in turn, refer you to still more. This process continues, "snowballing" into a larger sample.

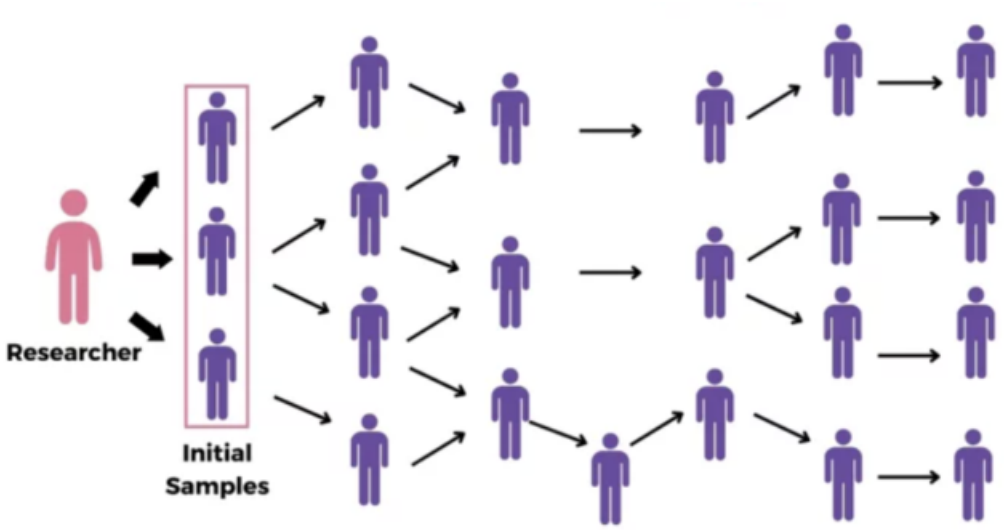


Figure 9.8: Snowball Sampling Process

When to Use:

- When the target population is hidden, rare, or hard to reach (e.g., migrant workers without official registrations, individuals involved in niche subcultures, or certain patient populations).
- When there is no comprehensive sampling frame or list of potential participants.
- Useful in qualitative or exploratory research to gain trust and access through existing social networks.

Pros:

Useful for reaching hidden or hard-to-identify populations (e.g., people in specialized niches or vulnerable communities).

Cons:

Samples may become biased because they rely on social networks and the people your initial contacts know, potentially missing whole segments of the population.

Example Scenario:

You are researching the experiences of freelance data scientists working only on "dark web" analytics—a specialized niche. Identifying such individuals through a public list is almost impossible. You might start with one or two data scientists you know personally, interview them, then ask them to introduce you to colleagues or friends who also fit the criteria.

Example Scenario using R:

In practice, simulating snowball sampling in R might involve having a small "network" or "graph" of individuals and selecting from them based on connections:

```
set.seed(123)

# Simulate a simple network as a data frame of individuals and their connections
individuals <- data.frame(
  id = 1:20,
  # Each individual is connected to 1-3 others at random
  contacts = I(lapply(1:20, function(x) {
    sample(1:20,
      size = sample(1:3, 1), replace = FALSE
    )
  }))
)

# Start with a known "seed"
seed <- 5

# Get the seed's contacts
first_wave <- unlist(individuals$contacts[individuals$id == seed])

# Get contacts of the first wave (second wave)
second_wave <- unique(unlist(individuals$contacts[individuals$id %in% first_wave]))

# Combine all waves (excluding duplicates)
snowball_sample_ids <- unique(c(seed, first_wave, second_wave))
snowball_sample_ids
```

```
#>  [1]  5  9  3  8 20 17 11 12 15 10
```

Here, we've shown a conceptual approach. In reality, you'd rely on your participants to provide referrals, not just a pre-defined network in code.

9.7.3 Judgmental (Purposive) Sampling

Judgmental or purposive sampling involves selecting participants based on the researcher's knowledge, expertise, and judgment. The idea is to choose subjects who are considered to be the most useful or representative to the research study's aims, rather than randomly.

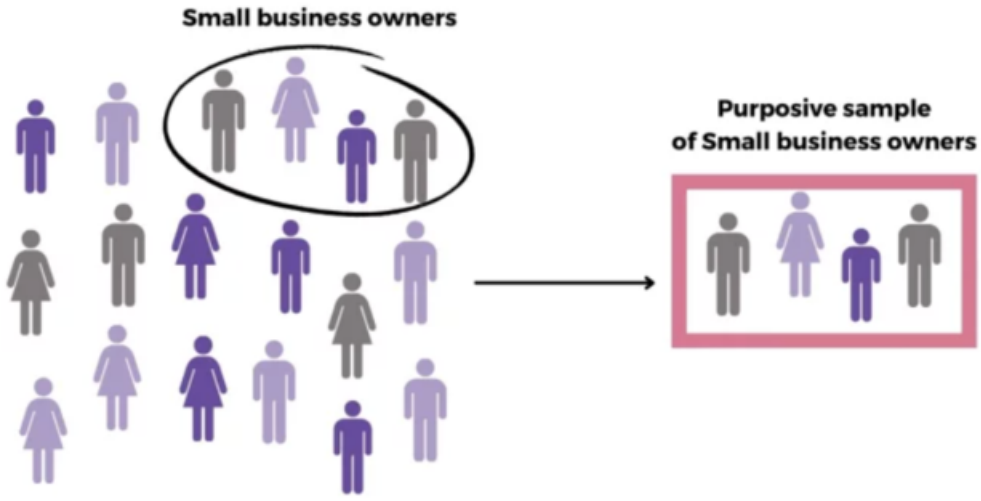


Figure 9.9: Purposive Sampling Process

When to Use:

- When you want to focus on a particular type of participant who can provide the most relevant information for your research question.
- When you have expert knowledge guiding which subjects are most informative.
- Often used in qualitative research or early-stage exploratory studies.

Pros:

Focuses on key individuals who can provide the richest information.

Cons:

Highly subjective and may reflect researcher bias. Not suitable for drawing generalizable statistical conclusions.

Example Scenario:

A researcher wants to understand high-level strategic decision-making processes within a large biotechnology firm. Rather than interviewing employees from all levels, the researcher decides

to focus on those individuals most likely to provide valuable insights into industry trends, company strategy, and organizational challenges—namely the top leadership team.

Example Scenario using R:

Imagine you have a dataset of professionals and their roles:

```
set.seed(123)
# Suppose we have data on 100 professionals in a biotechnology firm
professionals <- data.frame(
  id = 1:100,
  role = sample(c("CEO", "CTO", "Engineer", "Analyst", "Intern"), 100,
    replace = TRUE
  ),
  # Poisson-distributed years of experience
  years_experience = rpois(100, lambda = 10)
)

professionals |> head()
```

```
#>   id     role years_experience
#> 1  1 Engineer                8
#> 2  2 Engineer                8
#> 3  3      CTO                9
#> 4  4      CTO               10
#> 5  5 Engineer               14
#> 6  6   Intern                8
```

A purposive sample might focus on senior leadership roles that the researcher believes offer the most strategic insights into the industry: CEOs and CTOs.

```
purposive_sample <- subset(professionals, role %in% c("CEO", "CTO"))

purposive_sample |> head()
```

```
#>    id role years_experience
#> 3   3  CTO                9
#> 4   4  CTO               10
#> 8   8  CEO                9
#> 9   9  CTO               11
#> 14 14  CEO                8
#> 16 16  CEO                7
```

In this example, the researcher chooses participants not by chance, but based on their roles (CEO and CTO) and, indirectly, on their likely longer tenure. While a CEO or CTO may not always have more experience than an Engineer, it's reasonable to assume that top leadership generally has a wealth of industry insights. This selection method—judgmental or purposive sampling—reflects the researcher's informed decision about who can provide the most valuable information for the study.

9.7.4 Quota Sampling

Quota sampling involves dividing the population into subgroups based on certain characteristics (such as age groups, gender, or education level) and then non-randomly selecting individuals to meet a pre-set quota that matches these characteristics in proportion to their estimated prevalence in the population. Although similar to stratified sampling in concept, the selection within each subgroup is not random, making it a non-probability technique.

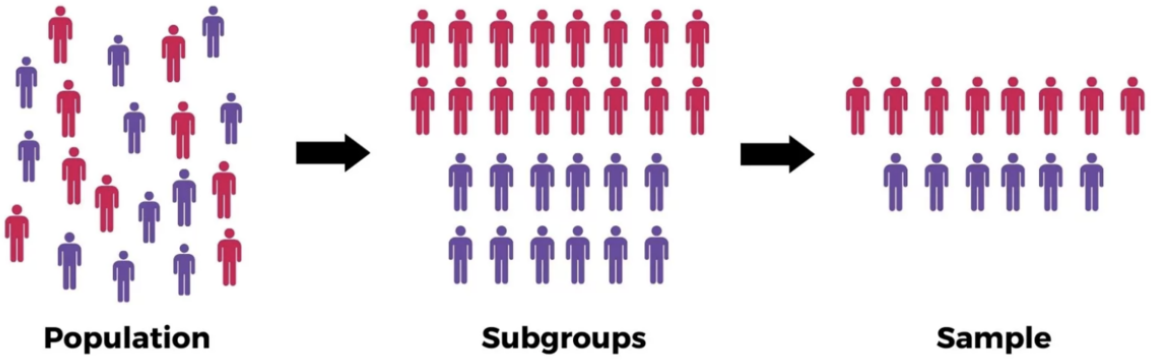


Figure 9.10: Quota Sampling Process

When to Use:

- When you want the sample to reflect certain known proportions of subgroups in the population, but you cannot or do not wish to randomly sample within these strata.
- Often used in market research, opinion polling, or early exploratory surveys where strict probability sampling is not feasible.

Pros:

Ensures representation of certain subgroups, providing more "balanced" samples compared to pure convenience sampling.

Cons:

Still non-random, and the people chosen within each quota are determined by convenience or researcher judgment, potentially introducing bias.

Example Scenario:

A company wants feedback on a new product from a demographic that matches their customer base: 50% women and 50% men. Instead of randomly selecting participants, the researcher ensures that interviews continue until they have reached the quota—e.g., 25 women and 25 men—by selecting participants conveniently until quotas are met.

Example Scenario using R:

Now, let's see how to simulate this process in R.

```
set.seed(123)
customers <- data.frame(
  id = 1:200,
  gender = sample(c("Male", "Female"), 200, replace = TRUE),
  purchase = sample(c("Yes", "No"), 200, replace = TRUE)
)
```

Suppose we want a sample of 40 customers, with a quota: 20 Female and 20 Male, We'll select conveniently (say, the first ones we encounter) until quotas are met.

```
quota_female <- customers |>
  filter(gender == "Female") |>
  slice(1:20)

quota_male <- customers |>
  filter(gender == "Male") |>
  slice(1:20)

quota_sample <- quota_female |> bind_rows(quota_male)

quota_sample |> count(gender)
```

```
#>   gender  n
#> 1 Female 20
#> 2   Male 20
```

In this simplistic example, we chose the first occurrences (mimicking convenience selection) until each quota was filled.

Reflection Question 3

How can non-probability sampling methods still provide value, even if their findings can't be easily generalized to the entire population?

9.7.5 Practice Quiz 9.2: Non-Probability Sampling

Question 1:

Non-probability sampling methods are often chosen because:

a) They guarantee generalizable results

b) They are cheaper, faster, or more practical

c) They eliminate all forms of bias

d) They require a complete list of the population

Question 2:

Which method involves selecting participants who are easiest to reach?

a) Convenience sampling

b) Snowball sampling

c) Purposive sampling

d) Quota sampling

Question 3:

Snowball sampling is most useful for:

a) Large, well-documented populations

b) Populations where every member is easily identified

c) Hidden or hard-to-reach populations

d) Ensuring random selection of subgroups

Question 4:

In snowball sampling, the sample grows by:

a) Randomly picking individuals from a list

b) Selecting every kth individual

c) Asking initial participants to refer others

d) Dividing the population into equal parts

Question 5:

Judgmental (purposive) sampling relies on:

a) Each member of the population having an equal chance

b) The researcher's expertise and judgment

c) Selecting individuals based solely on their availability

d) A systematic interval selection

Question 6:

A researcher who specifically seeks out top experts or key informants in a field is using:

a) Purposive (judgmental) sampling

b) Cluster sampling

c) Systematic sampling

d) Simple random sampling

Question 7:

Quota sampling ensures subgroups are represented by:

a) Randomly selecting from each subgroup

b) Matching known proportions but using non-random selection

c) Following a strict interval for selection

d) Relying on participant referrals

Question 8:

In quota sampling, once you have met the quota for a subgroup:

a) You continue selecting more participants from it anyway

b) You stop selecting participants from that subgroup

c) You switch to random selection

d) You start using a different method

Question 9:

A main drawback of non-probability methods is:

a) They are always expensive

b) They cannot measure uncertainty and generalize results easily

c) They require a complete list of the population

d) They eliminate researcher bias

Question 10:

Which non-probability method would you likely use if you have no sampling frame and need participants quickly, even though it might not be representative?

a) Simple random sampling

b) Stratified sampling

c) Convenience sampling

d) Systematic sampling

See the Solution to Quiz 9.2

9.7.6 Choosing the Right Sampling Technique

Selecting the appropriate sampling technique depends on your research goals, available data, resources, and the need for representativeness.

- Probability-based methods (like simple random, stratified, cluster, and systematic sampling) are generally preferred for statistical inference because they reduce bias and allow for the estimation of sampling error.
- Non-probability methods can be useful for exploratory research, generating hypotheses, or when it's simply impossible to use a probability sample. However, always be aware of their limitations and be cautious when generalizing findings from these samples.

> Reflection Question 4
>
> Think of a scenario where probability sampling would be ideal and another where non-probability sampling would be more realistic. What makes these contexts so different?

9.8 Reproducibility and Ethics

Documenting your sampling decisions, ensuring transparency, and acknowledging limitations are crucial for building trust in your results. Also, consider whether your sampling method might unintentionally exclude or disadvantage certain groups, and think about the ethical implications of doing so.

> Reflection Question 5
>
> How could the sampling method you choose affect the fairness and ethics of a study, especially when dealing with sensitive populations or topics?

Reflective Summary

By exploring these techniques, you've seen how critical the selection process is in shaping the reliability and fairness of your data-driven insights. Probability sampling offers a path to generalizable, statistically sound conclusions, while non-probability methods provide flexibility, convenience, and creative ways to reach challenging populations—albeit with caution.

Understanding the trade-offs among these methods is key. Armed with these insights, you're better prepared to design well-structured studies, interpret results accurately, and acknowledge limitations transparently.

> What's Next?
>
> In the next lab, we dive into the Data Science Concept—a culmination of all the techniques you've learned in this book. Built upon your skills in data wrangling, visualisation, and statistical concept, this lab will show you how to integrate these methods into a complete, real-world data science workflow.

10 Data Science Concept

10.1 Introduction

Welcome to Lab 10! In this lab, we will explore the concept of data science. Data science is both an art and a science, blending statistical thinking, programming skills, computational methods, and domain knowledge to transform raw data into actionable insights. It is the critical bridge between raw numbers and actionable knowledge in a world of information. More than just a field, data science has the potential to revolutionise how we make decisions and understand the world, showcasing its transformative power and impact.

In previous labs, you learned about organising workflows, preparing data, and creating compelling visualisations. Now, you'll contextualise these skills within the broader realm of data science. Understanding its principles and workflows allows you to navigate complex projects with robust, reproducible, and scientifically grounded analyses.

Data science thrives at the intersection of disciplines—mathematics, statistics, computer science, and domain expertise—each contributing essential elements. This interdisciplinary nature of data science not only makes it a challenging and dynamic field but also enriches it with diverse perspectives and approaches. Whether predicting sales, detecting anomalies, recommending products, or automating decisions, data science equips you with the tools and frameworks to tackle intricate challenges. Unsurprisingly, Harvard Business Review dubbed data science the "sexiest job of the 21st century," reflecting its importance and allure[1].

10.2 Learning Objectives

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

- **Define Data Science and Its Interdisciplinary Nature**
  Understand how data science draws upon statistics, programming, domain expertise, and communication to extract meaningful insights.

[1] Davenport, T. H., & Patil, D. J. (2012). Data scientist: The sexiest job of the 21st century. Harvard Business Review, 90(10), 70–76. https://hbr.org/2012/10/data-scientist-the-sexiest-job-of-the-21st-century

- **Recognise Key Use Cases and Applications**
  Identify real-world scenarios—forecasting, pattern detection, recommendation systems—where data science delivers value.
- **Describe the Data Science Lifecycle**
  Understand the phases of data science projects, from importing and tidying data through transformation, modelling, visualisation, and communication.
- **Appreciate the Role of a Data Scientist**
  Recognise the diverse skill set, responsibilities, and evolving role of data scientists in modern organisations.
- **Connect Data Science Concepts to Previous Labs**
  Relate data science principles to earlier labs on data wrangling, visualisation, and reproducibility, reinforcing a cohesive understanding of the end-to-end analytical workflow.

By completing Lab 10, you'll gain a holistic understanding of what data science entails, the roles and skills of data scientists, and the typical lifecycle that guides projects from concept to actionable insights. This foundation sets the stage for integrating all the technical skills you've acquired into meaningful, real-world applications.

10.3 Prerequisites

Before starting this lab, you should have:

- Completed previous labs (especially Labs 4, 5, 6, and 7) to be familiar with setting up reproducible workflows, transforming data, and creating visualisations.
- Basic understanding of statistical concepts and programming fundamentals in R.
- An interest in applying these skills to solve problems and support evidence-based decision-making.

10.4 Real-World Scenario: Data Science in Action

Consider working in a healthcare analytics team that aims to predict patient readmissions. Your data includes patient demographics, medical histories, treatment plans, and outcomes. By applying data science methods—cleaning and transforming raw hospital records, building predictive models, and communicating results to clinicians—you can help target interventions that reduce readmissions, improve patient care, and optimise resource allocation.

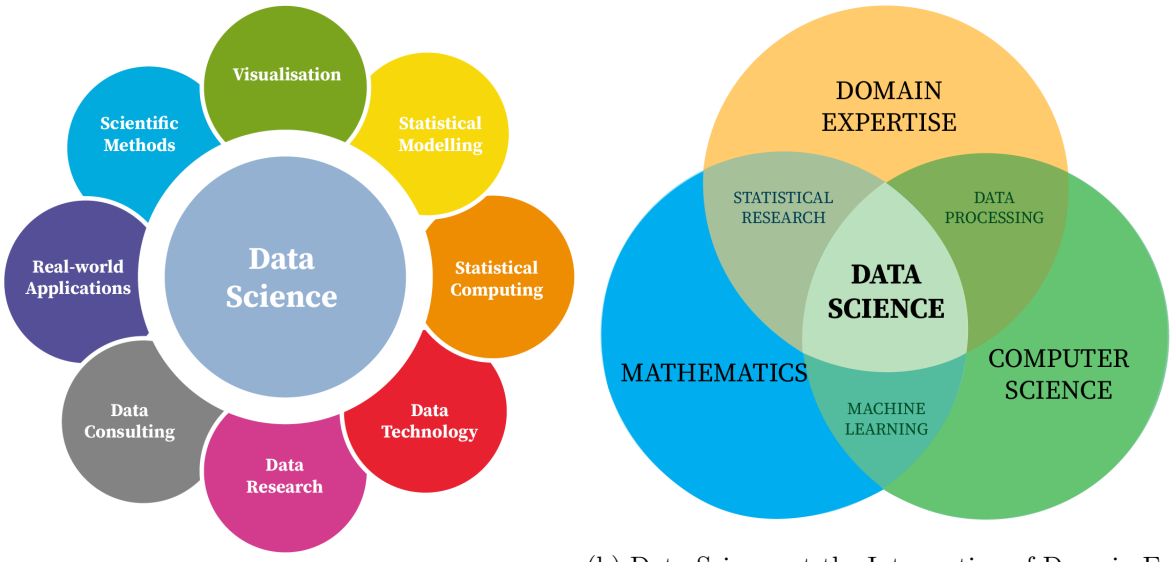
In this scenario, all the components of data science—technical mastery, statistical rigour, domain understanding, and effective communication—come into play. Data science thus not

only solves problems but also supports better-informed strategies that can impact patient lives.

10.5 Understanding Data Science

Imagine data science as a field that draws on a multitude of ideas and skills from various disciplines to solve problems and uncover insights. One way to visualise this interdisciplinary nature is shown in Figure 10.1a. In this diagram, 'data science' sits at the centre, surrounded by petals labelled 'visualisation', 'statistical modelling', 'statistical computing', 'data technology', 'data research', 'data consulting', 'real-world applications', and 'scientific methods'. Each petal represents a key area or responsibility within the data science process, inspiring your interest and engagement in this dynamic field.

Another useful way to understand how these areas connect is through a Venn diagram overlapping three main circles: Mathematics/Statistics, Domain Expertise, and Computer Science/Programming, as shown in Figure 10.1b. At the intersection of these circles, data science draws on their collective strengths. From the mathematical side, it utilises statistical research and machine learning techniques. With domain expertise, it incorporates business- or field-specific knowledge to shape pertinent questions. And from computer science, it relies on programming and data-processing skills to manage large datasets effectively.



(a) Core Disciplines Encompassed by Data Science

(b) Data Science at the Intersection of Domain Expertise, Mathematics, and Computer Science

Figure 10.1: The Scope and Foundations of Data Science

Taken together, these perspectives highlight data science as a highly collaborative and multifaceted discipline. It demands an understanding of essential concepts (such as statistical modelling and coding) and the ability to apply them to real-world challenges across numerous sectors. By mastering the overlapping skills in these areas, you can explore data, generate valuable insights, and ultimately make evidence-based decisions—demonstrating the practical impact of data science.

Reflection Question 10.1

How does integrating multiple disciplines—statistics, programming, domain expertise, communication—enable data scientists to tackle more complex questions than any single field could address alone?

10.6 Data Science Use Cases

Data science has transformed industries by delivering innovative, data-driven solutions to complex problems:

- **Forecasting:** Predict future sales, revenue, or retention.
- **Pattern Detection:** Identify weather trends or detect market shifts.
- **Recommendations:** Suggest products, content, or services.
- **Anomaly Detection:** Spot fraudulent transactions, defects, or suspicious behaviour.
- **Automation and Decision-Making:** Conduct background checks, assess credit, and streamline operations.
- **Classification:** Categorise emails as spam or not, diagnose diseases, and flag phishing sites.
- **Recognition:** Enable facial, voice, or text recognition technologies.

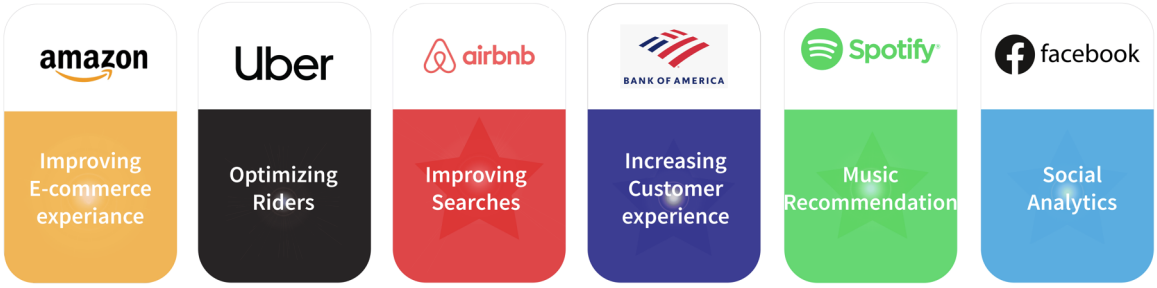


Figure 10.2: Real-World Applications of Data Science Across Industries

Reflection Question 10.2

Which data science application resonates most with your interests or field, and why might it be challenging or rewarding to implement?

10.7 Who is a Data Scientist?

A data scientist:

- **Collects and Organises Data**: Gathers, cleans, and structures data into analysis-ready formats.
- **Analyses Patterns and Trends**: Uses statistical and computational methods to find relationships and identify important features.
- **Communicates Findings**: Translates complex analyses into accessible narratives and insights for decision-makers.

The best data scientists combine technical prowess, creative thinking, and strong communication skills. They must understand how to analyse data, why particular patterns matter, and effectively convey these insights.

Reflection Question 10.3

Beyond technical skills, why is communication and domain knowledge crucial for a data scientist to make meaningful contributions?

10.8 Skills Required for Data Science

A successful data scientist blends multiple competencies:

- **Programming**: Proficiency in R, Python, and SQL.
- **Data Wrangling**: Cleaning, merging, and transforming datasets.
- **Visualisation**: Crafting clear, insightful charts and dashboards.
- **Statistical Analysis**: Applying models, tests, and confidence intervals to draw inferences.
- **Machine Learning**: Building predictive or descriptive models to uncover patterns.
- **Communication**: Presenting results in understandable, persuasive forms.

- **Domain Expertise**: Understanding the context and constraints that shape data interpretation.

Reflection Question 10.4

Which skill area do you feel is your strongest currently, and where do you see room for the most improvement?

10.9 Becoming a Data Scientist

The path to becoming a data scientist involves the following:

1. **Learn Programming Languages**: Start with R for statistical analysis and Python for broader machine learning tasks.
2. **Build Real-World Projects**: Apply your skills to real datasets, refining your approach as you encounter practical challenges.
3. **Master Statistics and Mathematics**: Develop a solid foundation to ensure robust model selection, validation, and interpretation.
4. **Engage in Collaboration**: Join communities, attend meetups, and participate in hackathons. Seek mentorship to broaden your horizons.
5. **Continuous Learning**: Data science evolves rapidly; staying updated with new tools, algorithms, and best practices is essential.

Reflection Question 10.5

In your journey towards becoming a data scientist, how will you balance pursuing technical skills with developing communication and domain-specific expertise?

10.10 Programming Languages for Data Science

R, Python, Julia, and Scala are popular choices in data science, each excelling in different areas. R is renowned for statistical analysis and rich visualisation capabilities—ideal for academic research and analytic prototyping. Python offers versatility and a vast ecosystem of libraries for machine learning and deep learning tasks.

Figure 10.3: Popular Programming Languages for Data Science

Reflection Question 10.6

Considering your goals and project types, which programming language do you find most appealing, and why?

10.11 The Data Science Lifecycle

Data science workflows are iterative, often cycling through exploration, modelling, and refinement phases, usually following six key stages, as shown in Figure 10.4.

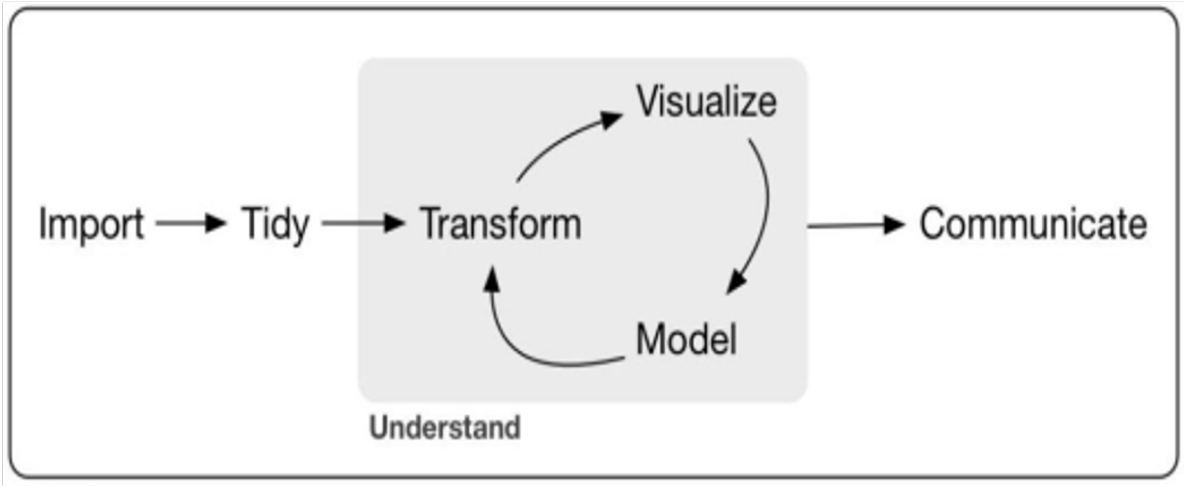


Figure 10.4: Data Science Workflow Phases [2]

10.11.1 Import

Importing data into appropriate tools, such as R, Python, or spreadsheet software like Microsoft Excel, is the starting point of any data science project. This step provides the raw materials for all subsequent stages—without data, there is no data science.

10.11.2 Tidy

Tidying data is one of data scientists' most time-consuming yet essential tasks. Real-world datasets are often messy and unstructured, requiring careful organisation into a format suitable for analysis.

The **principle of tidy data** ensures that:

- Columns represent variables.
- Rows represent observations.
- Cells contain individual values.

This process, often called data wrangling or munging, is a critical foundation for successful analysis.

[2]Wickham, H., Çetinkaya-Rundel, M., & Grolemund, G. (2023). *R for data science* (2nd ed.). O'Reilly Media. ISBN: 978-1-4920-9740-2. Retrieved from https://r4ds.hadley.nz/

10.11.3 Transform

Data transformation is another integral part of data wrangling. This involves creating new variables or modifying existing ones to uncover patterns, relationships, or trends. Transformation helps refine the data into a format that aligns with the project's analytical goals.

10.11.4 Visualise

Visualisation bridges raw data and actionable insights using charts, graphs, and interactive dashboards. Tools like `ggplot2` in R make this process intuitive and effective.

10.11.5 Models

Modelling applies statistical techniques or machine learning algorithms to answer questions or make predictions. For example:

- Linear Regression: Predicting housing prices.
- Classification Models: Identifying spam emails. Proficiency in coding and a solid understanding of statistical methods are critical for interpreting and validating models.

10.11.6 Communicate

Effective communication is the final and most critical stage of the data science workflow. Insights must be presented clearly and persuasively to stakeholders, ensuring they can make informed decisions based on the findings. Tools like Quarto or R Markdown facilitate professional-grade reports.

> Reflection Question 10.7
>
> How does viewing data analysis as a cyclical lifecycle (rather than a linear process) influence how you approach projects, especially when dealing with complex or evolving datasets?

10.12 Reproducibility and Ethical Considerations

As data science projects scale, reproducibility ensures others can verify and extend your work. Following best practices—organising projects (Lab 4), tidying data (Lab 5), and documenting code—is vital. Moreover, ethical considerations (privacy, bias mitigation, fairness) must be integrated into every stage of the data science lifecycle.

Reflection Question 10.8

How can establishing reproducible workflows and considering ethical guidelines early in a project help maintain trust and credibility in your analyses?

10.12.1 Practice Quiz 10.1

Question 1:

Data science is considered interdisciplinary because it involves the integration of:

a) Mathematics, domain expertise, and biological sciences

b) Programming, mathematics/statistics, and domain expertise

c) Philosophy, ethics, and data engineering

d) Chemistry, physics, and computer science

Question 2:

The iterative nature of the data science lifecycle is essential for:

a) Ensuring a one-time solution

b) Continuous refinement and improved insights

c) Avoiding communication and visualisation steps

d) Reducing time spent on data wrangling

Question 3:

In the context of anomaly detection, which of the following scenarios is most relevant?

a) Predicting future sales

b) Identifying fraudulent transactions

c) Recommending products to customers

d) Forecasting weather trends

Question 4:

Why is domain expertise considered critical in data science projects?

a) To eliminate the need for reproducible workflows

b) To ensure analyses are contextually accurate and meaningful

c) To substitute for statistical reasoning

d) To automate the cleaning process

Question 5:

Which of the following ethical considerations is essential in data science?

a) Automating decision-making without human oversight

b) Mitigating bias and ensuring fairness

c) Replacing statistical methods with machine learning

d) Eliminating reproducibility for scalability

Question 6:

In the healthcare analytics example, the role of predictive modelling primarily involves:

a) Replacing clinicians in decision-making

b) Identifying trends in patient demographics

c) Predicting patient readmissions and improving care

d) Tidying and transforming hospital data

Question 7:

During the "Tidy" phase of the data science lifecycle, what is the primary goal?

a) Creating dashboards for analysis

b) Organising data into a structured format for analysis

c) Designing machine learning models

d) Cleaning visualisations for stakeholder presentations

Question 8:

Which stage of the data science lifecycle involves crafting visual narratives to interpret results?

a) Model

b) Transform

c) Visualise

d) Import

Question 9:

Why is the "Communicate" phase considered critical in the data science lifecycle?

a) It automates repetitive data cleaning tasks

b) It presents findings clearly and persuasively to stakeholders

c) It eliminates the need for statistical reasoning

d) It directly replaces the "Model" phase

Question 10:

How does viewing data analysis as a cyclical lifecycle benefit complex projects?

a) Reduces the need for domain expertise

b) Supports iterative refinement and evolving datasets

c) Guarantees fixed solutions for all analyses

d) Simplifies reproducibility without documentation

See the Solution to Quiz 10.1

10.12.2 Exercise 10.1: Identifying Data Science Roles

Task: Suppose your team includes a statistician, a software engineer, a business analyst, and a machine learning researcher. Discuss how each role contributes to a data science project to predict hospital readmissions. Reflect on what gaps remain if one role is absent.

Reflection Question (for Exercise 10.1)

How does the presence (or absence) of specific skill sets within a data science team shape the quality and scope of the project's outcomes?

10.12.3 Exercise 10.2: Mapping Lab Skills onto the Data Science Lifecycle

Task: Revisit Labs 1–6 and identify how each set of skills (e.g., reproducible workflows, data wrangling, visualisation) maps onto the stages of the data science lifecycle. For instance, where does your expertise in `dplyr` fit, and how does your mastery of `ggplot2` support the Communication stage?

Reflection Question (for Exercise 10.2)

Does visualising how your learned skills align with each lifecycle stage help clarify your long-term development path as a data scientist?

10.12.4 Exercise 10.3: Designing a Mini Project

Task: Think of a small data science project you can undertake (e.g., analysing weather data to predict rainfall patterns). Outline which data you will import, how you will tidy and transform it, what models you might apply, and how you will visualise and communicate results. Consider missing data, how you would handle it, and potential ethical implications (e.g., data privacy, sensitive attributes).

Reflection Question (for Exercise 10.3)

How does planning a mini project from start to finish help consolidate your understanding of the data science concept and lifecycle?

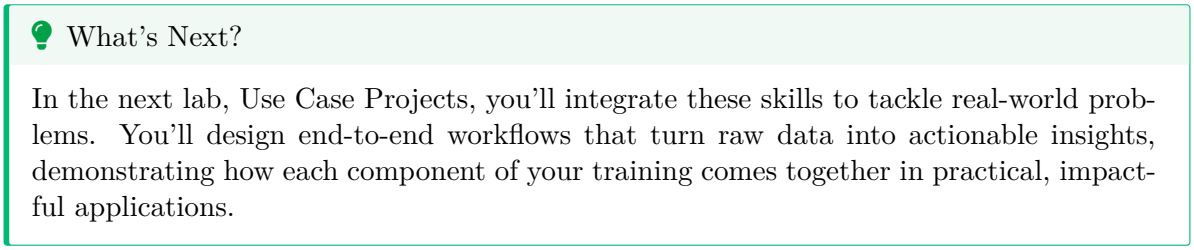
10.13 Reflective Summary

In Lab 10, the techniques learned in earlier labs—such as reproducible workflows, data wrangling, and data visualisation—are contextualised within the comprehensive data science framework.

Key Takeaways:

- **Definition and Scope**: Data science is interdisciplinary, drawing on multiple fields to solve complex, real-world problems.
- **Applications and Use Cases**: Data science drives innovation in forecasting, pattern detection, recommendation systems, anomaly detection, and more.
- **Data Science Roles and Skills**: Successful data scientists blend programming, statistics, domain knowledge, and communication skills.
- **Data Science Lifecycle**: Projects evolve through importing, tidying, transforming, modelling, visualising, and communicating results.
- **Integrating Previous Skills**: The workflows, data wrangling, and visualisations learned in earlier labs form essential components of the data science process.

As you move forward, remembering the holistic nature of data science ensures that each skill—be it project organisation (Lab 4), data wrangling (Lab 5 & Lab 6), or visualisation (Lab 7)—contributes to a cohesive, meaningful analytical journey. Data science is not merely about tools or techniques; it's about asking the right questions, navigating complexities, and conveying insights that drive informed decisions and positive change.



11 Use Case Projects

11.1 Introduction

In previous labs, you learned how to organise workflows, transform data, create visualisations, and contextualise your technical skills within the broader field of data science. **Lab 11** takes this learning further by emphasising the power of applying your skills to realistic, end-to-end projects. Engaging in use case projects bridges the gap between theory and practice, allowing you to internalise concepts, strengthen problem-solving abilities, and gain invaluable experience.

11.2 Learning Objectives

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

- **Integrate Previously Learned Skills into Practical Projects**
  Combine data wrangling, visualisation, reproducibility, and statistical techniques to solve real-world problems.
- **Improve Problem-Solving Through Iterative Practice**
  Gain the ability to troubleshoot, adapt, and refine your approach in response to unexpected challenges in data analysis.
- **Develop a Holistic Understanding of Workflows**
  Recognise how each stage of the data analysis process—importing data, cleaning, transforming, modelling, and communicating—fits into a cohesive project.
- **Build Confidence and Portfolios**
  Complete projects that demonstrate your proficiency, increasing self-assurance and providing tangible evidence of your capabilities.
- **Communicate Findings Effectively**
  Present insights, methods, and recommendations through clear reporting and visualisations, ensuring results can guide informed decisions.

By completing this lab, you'll deepen your understanding of R and data analysis methodologies by tackling real-world scenarios. This hands-on approach cements your learning, enhances creativity, boosts confidence, and helps you build a portfolio of projects that can showcase your capabilities to employers, colleagues, and mentors.

11.3 Prerequisites

Before starting this lab, you should have:

- Completed Labs 1–9, gaining familiarity with setting up R projects, transforming data, visualising information, understanding the data science lifecycle, and contextualising your technical skills.
- Basic understanding of statistical concepts, data wrangling, reproducible workflows, and visualisation techniques.
- An interest in applying your learned skills to realistic scenarios, moving beyond isolated exercises and theoretical discussions.

11.4 Why Use Case Projects?

While earlier labs focused on mastering individual techniques, use case projects show how these techniques fit together in solving a real-world problem. This approach:

1. **Application of Theory**: Practical projects allow learners to apply the theoretical knowledge they've acquired. This transition from theory to application often solidifies understanding.
2. **Problem-Solving Skills**: Real-world projects present unforeseen challenges. By working through these, learners enhance their problem-solving skills and become adept at troubleshooting.
3. **Comprehensive Understanding**: Use case projects often require the integration of various R functions and techniques. This holistic approach ensures a deeper and more comprehensive grasp of R.
4. **Confidence Building**: Successfully completing a use-case project boosts confidence, giving students the assurance that they can tackle real-world data problems using R.
5. **Portfolio Building**: Adds substantial examples of your work for future presentations or job applications.

Reflection Question

How does applying your skills in a realistic project setting differ from learning them in isolation, and why might this approach lead to deeper mastery?

11.5 Use Case 1: Telco Customer Churn Data Analysis and Visualization Assessment

You have been provided with the Telco Customer Churn dataset, which includes detailed information on customer demographics, account details, subscribed services, and churn behaviour. Your task is to leverage your R skills to transform, analyse, and visualise this data, generating actionable insights. Synthesize your findings into a concise report to communicate key patterns, trends, and recommendations.

Dataset Overview

The Telco Customer Churn dataset provides comprehensive details about customers, including their demographics, account information, service usage, and whether they have churned. Key columns include:

- **customerID**: Unique identifier for each customer.
- **gender**: Customer gender ('Male' or 'Female').
- **SeniorCitizen**: Indicator if the customer is a senior (1 for Yes, 0 for No).
- **Partner**: Whether the customer has a partner ('Yes' or 'No').
- **Dependents**: Whether the customer has dependents ('Yes' or 'No').
- **tenure**: Number of months the customer has stayed with the company.
- **PhoneService**: Indicates if the customer has a phone service ('Yes' or 'No').
- **MultipleLines**: Indicates if the customer has multiple phone lines ('Yes', 'No', or 'No phone service').
- **InternetService**: Type of internet service ('DSL', 'Fiber optic', or 'No').
- **OnlineSecurity**, **OnlineBackup**, **DeviceProtection**, **TechSupport**, **StreamingTV**, **StreamingMovies**: Service-specific columns with values ('Yes', 'No', or 'No internet service').
- **Contract**: Customer's contract type ('Month-to-month', 'One year', or 'Two year').

- **PaperlessBilling**: Whether the customer uses paperless billing ('Yes' or 'No').
- **PaymentMethod**: Customer's payment method (e.g., 'Electronic check', 'Mailed check', 'Bank transfer (automatic)', 'Credit card (automatic)').
- **MonthlyCharges**: Amount charged to the customer monthly.
- **TotalCharges**: Total amount charged to the customer.
- **Churn**: Indicates whether the customer has churned ('Yes' or 'No').

For full metadata, see sheet 2 of the `telco-customer-churn.xlsx` file or visit Kaggle.

Tasks

1. **Data Manipulation and Transformation**

   a. **Data Import:**
      - Locate and import the Telco Customer Churn data (`telco-customer-churn.xlsx`) from the `r-data` directory. If you do not already have the file, you can download it from Google Drive.

   b. **Variable Transformation:**
      - Transform the `Churn` column into a binary format (e.g., `1` for churned, `0` for not churned).
      - Recode the `SeniorCitizen` variable into a more descriptive format (e.g., "Yes" for 1 and "No" for 0).

      - Create a new variable, such as `AvgChargePerMonth`, calculated by dividing `TotalCharges` by `tenure` (ensuring that cases where `tenure` is 0 are handled appropriately).
      - Optionally, develop another metric (e.g., a `ServiceCount` that aggregates the number of additional services to which a customer subscribes).

2. **Handling Missing and Inconsistent Values**

   a. **Identify Issues:**
      - Scan the dataset for missing or inconsistent values, and document which columns are affected.

   b. **Data Quality Improvement:**
      - Apply strategies to address any data quality issues (for example, convert data types if necessary, handle missing values, and ensure consistency across columns).

3. **Analysis and Insights**

a. **Overall Churn Patterns:**
   - Determine the overall churn rate in the dataset.

b. **Segmented Analysis:**
   - Calculate the percentage of churned customers across different segments such as:
     - **Gender:** What proportion of male vs. female customers churn?
     - **Contract Type:** How does the churn rate vary across different contract types?
     - **Internet Service:** What are the churn rates for customers with DSL, Fiber optic, or no internet service?
   - For each segment (or combination of segments), compute summary statistics (e.g., counts, averages, medians) for key metrics like `MonthlyCharges` and `tenure`.

c. **Advanced Aggregation:**
   - For each gender, determine summary statistics (mean, median, maximum) for monthly charges.
   - Identify which customer segment or service bundle is associated with the highest churn rate.

4. **Data Visualization**

a. **Exploratory Visualizations:**
   - Create a histogram or density plot to visualize the distribution of customer `tenure`.
   - Develop a bar chart that shows the counts of churned and non-churned customers.

b. **Comparative Visualizations:**
   - Construct a boxplot to compare `MonthlyCharges` across different `Contract` types.
   - Generate a scatter plot displaying the relationship between `tenure` and `MonthlyCharges`, with points colored by churn status. Consider adding a trend line if it enhances interpretation.

c. **Combined Analysis:**
   - Filter the dataset to focus on a specific segment (for example, only customers with 'Fiber optic' service) and create additional visualizations (such as a histogram of their `tenure` or a scatter plot of their charges vs. tenure).
   - For each unique `tenure` value, compute the percentage of customers who churned, and plot these percentages as a line graph.

Deliverables

- **Code:**
  Provide your R script or R Markdown file with clear, commented code showing your data manipulation, analysis, and visualization steps.
- **Report:**
  Write a concise summary that explains:
  - Your approach to data cleaning and transformation.
  - The key findings from your analysis.
  - Insights derived from your visualizations.
  - Any recommendations or follow-up questions that your analysis suggests.

11.6 Use Case 1: The Solution

This document analyses the Telco Customer Churn dataset. It covers data import, transformation, analysis, and visualisation.

Data Manipulation and Transformation

Data Import and Initial Exploration

We begin by loading the required libraries and importing the data.

```
# Load required libraries
library(tidyverse) # For data manipulation and visualisation
library(readxl) # For data import
library(inspectdf) # For inspecting missing values

# Set the file path for the Telco Customer Churn dataset
file_path <- "r-data/telco-customer-churn.xlsx"

# Read the spreadsheet file into a tibble
telco <- read_xlsx(file_path, sheet = 1)

# Explore the dataset structure, summary statistics, and first few     rows
glimpse(telco)
```

```
#> Rows: 2,110
#> Columns: 21
#> $ customer_id       <chr> "1452-KIOVK", "6388-TABGU", "9763-GRSKD", "3655-SNQY~
#> $ gender            <chr> "Male", "Male", "Male", "Female", "Male", "Female", ~
#> $ senior_citizen    <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0~
#> $ partner           <chr> "No", "No", "Yes", "Yes", "No", "Yes", "Yes", "Yes",~
#> $ dependents        <chr> "Yes", "Yes", "Yes", "Yes", "Yes", "Yes", "Yes", "Ye~
#> $ tenure            <dbl> 22, 62, 13, 69, 71, 10, 49, 47, 1, 17, 27, 72, 10, 7~
#> $ phone_service     <chr> "Yes", "Yes", "Yes", "Yes", "Yes", "Yes", "Yes", "Ye~
#> $ multiple_lines    <chr> "Yes", "No", "No", "Yes", "Yes", "No", "No", "Yes", ~
#> $ internet_service  <chr> "Fiber optic", "DSL", "DSL", "Fiber optic", "Fiber o~
#> $ online_security   <chr> "No", "Yes", "Yes", "Yes", "Yes", "No", "Yes", "No",~
#> $ online_backup     <chr> "Yes", "Yes", "No", "Yes", "No", "No", "Yes", "Yes",~
#> $ device_protection <chr> "No", "No", "No", "Yes", "Yes", "Yes", "No", "No", "~
#> $ tech_support      <chr> "No", "No", "No", "Yes", "No", "Yes", "Yes", "No", "~
#> $ streaming_tv      <chr> "Yes", "No", "No", "Yes", "Yes", "No", "No", "Yes", ~
#> $ streaming_movies  <chr> "No", "No", "No", "Yes", "Yes", "No", "No", "Yes", "~
#> $ contract          <chr> "Month-to-month", "One year", "Month-to-month", "Two~
#> $ paperless_billing <chr> "Yes", "No", "Yes", "No", "No", "No", "No", "Yes", "~
#> $ payment_method    <chr> "Credit card (automatic)", "Bank transfer (automatic~
#> $ monthly_charges   <dbl> 89.10, 56.15, 49.95, 113.25, 106.70, 55.20, 59.60, 9~
#> $ total_charges     <chr> "1949.4", "3487.95", "587.45000000000005", "7895.15"~
#> $ churn             <chr> "No", "No", "No", "No", "No", "Yes", "No", "Yes", "Y~
```

```
summary(telco)
```

```
#>  customer_id          gender          senior_citizen      partner
#>  Length:2110        Length:2110        Min.   :0.00000   Length:2110
#>  Class :character   Class :character   1st Qu.:0.00000   Class :character
#>  Mode  :character   Mode  :character   Median :0.00000   Mode  :character
#>                                        Mean   :0.04313
#>                                        3rd Qu.:0.00000
#>                                        Max.   :1.00000
#>   dependents            tenure      phone_service      multiple_lines
#>  Length:2110        Min.   : 0.00   Length:2110        Length:2110
#>  Class :character   1st Qu.:16.00   Class :character   Class :character
#>  Mode  :character   Median :39.00   Mode  :character   Mode  :character
#>                     Mean   :38.37
#>                     3rd Qu.:62.00
#>                     Max.   :72.00
#>  internet_service   online_security    online_backup      device_protection
#>  Length:2110        Length:2110        Length:2110        Length:2110
```

```
#>  Class :character   Class :character   Class :character   Class :character
#>  Mode  :character   Mode  :character   Mode  :character   Mode  :character
#>
#>
#>
#>  tech_support       streaming_tv       streaming_movies     contract
#>  Length:2110        Length:2110        Length:2110        Length:2110
#>  Class :character   Class :character   Class :character   Class :character
#>  Mode  :character   Mode  :character   Mode  :character   Mode  :character
#>
#>
#>
#>  paperless_billing  payment_method     monthly_charges  total_charges
#>  Length:2110        Length:2110        Min.   : 18.70   Length:2110
#>  Class :character   Class :character   1st Qu.: 24.50   Class :character
#>  Mode  :character   Mode  :character   Median : 60.98   Mode  :character
#>                                        Mean   : 59.52
#>                                        3rd Qu.: 85.95
#>                                        Max.   :118.75
#>     churn
#>  Length:2110
#>  Class :character
#>  Mode  :character
#>
#>
#>
```

```
head(telco)
```

```
#> # A tibble: 6 x 21
#>   customer_id gender senior_citizen partner dependents tenure phone_service
#>   <chr>       <chr>           <dbl> <chr>   <chr>       <dbl> <chr>
#> 1 1452-KIOVK  Male                0 No      Yes            22 Yes
#> 2 6388-TABGU  Male                0 No      Yes            62 Yes
#> 3 9763-GRSKD  Male                0 Yes     Yes            13 Yes
#> 4 3655-SNQYZ  Female              0 Yes     Yes            69 Yes
#> 5 9959-WOFKT  Male                0 No      Yes            71 Yes
#> 6 4190-MFLUW  Female              0 Yes     Yes            10 Yes
#> # i 14 more variables: multiple_lines <chr>, internet_service <chr>,
#> #   online_security <chr>, online_backup <chr>, device_protection <chr>,
#> #   tech_support <chr>, streaming_tv <chr>, streaming_movies <chr>,
#> #   contract <chr>, paperless_billing <chr>, payment_method <chr>,
```

```
#> #   monthly_charges <dbl>, total_charges <chr>, churn <chr>
```

```
# Inspect missing values for each column
telco %>%
  inspect_na() %>%
  print(n = 21)
```

```
#> # A tibble: 21 x 3
#>    col_name            cnt  pcnt
#>    <chr>             <int> <dbl>
#>  1 total_charges        11 0.521
#>  2 customer_id           0 0
#>  3 gender                0 0
#>  4 senior_citizen        0 0
#>  5 partner               0 0
#>  6 dependents            0 0
#>  7 tenure                0 0
#>  8 phone_service         0 0
#>  9 multiple_lines        0 0
#> 10 internet_service      0 0
#> 11 online_security       0 0
#> 12 online_backup         0 0
#> 13 device_protection     0 0
#> 14 tech_support          0 0
#> 15 streaming_tv          0 0
#> 16 streaming_movies      0 0
#> 17 contract              0 0
#> 18 paperless_billing     0 0
#> 19 payment_method        0 0
#> 20 monthly_charges       0 0
#> 21 churn                 0 0
```

Data Cleaning and Transformation

We transform the *churn* column into a binary indicator, recode `senior_citizen`, and create a new variable *service_count* to count additional service subscriptions.

```
# Transform Churn to a binary indicator and recode SeniorCitizen for     clarity
telco <- telco %>%
  mutate(
    churn_binary = if_else(churn == "Yes", 1, 0),
```

```
    senior_citizen = if_else(senior_citizen == 1, "Yes", "No")
  )

# Create a ServiceCount variable to sum additional service     subscriptions
telco <- telco %>%
  mutate(
    phone_service_flag = if_else(phone_service == "Yes", 1, 0),
    online_security_flag = if_else(online_security == "Yes", 1, 0),
    online_backup_flag = if_else(online_backup == "Yes", 1, 0),
    device_protection_flag = if_else(device_protection == "Yes", 1, 0),
    tech_support_flag = if_else(tech_support == "Yes", 1, 0),
    streaming_tv_flag = if_else(streaming_tv == "Yes", 1, 0),
    streaming_movies_flag = if_else(streaming_movies == "Yes", 1, 0),
    service_count = phone_service_flag + online_security_flag + online_backup_flag +
      device_protection_flag + tech_support_flag + streaming_tv_flag +
      streaming_movies_flag
  ) %>%
  # Remove temporary flag columns for cleanliness
  select(-ends_with("_flag"))
```

Recoding Additional Demographic and Payment Variables

Recode key demographic variables to enhance interpretability and group similar payment methods.

```
# Recode SeniorCitizen into SeniorStatus
telco <- telco %>%
  mutate(senior_status = if_else(senior_citizen == "Yes", "Senior", "Non-Senior"))

# Recode Partner into PartnerStatus
telco <- telco %>%
  mutate(partner_status = if_else(partner == "Yes", "Partner", "No Partner"))

# Recode Dependents into DependentStatus
telco <- telco %>%
  mutate(dependent_status = if_else(dependents == "Yes", "Dependents", "No Dependents"))

# Group PaymentMethod into broader categories
telco <- telco %>%
  mutate(
    payment_method_group = case_when(
```

```
      payment_method == "Electronic check" ~ "Electronic check",
      payment_method == "Mailed check" ~ "Mailed check",
      str_detect(payment_method, "automatic") ~ "Automatic",
      TRUE ~ payment_method # Catch-all for any unexpected values
    )
  )

# Convert all non-numeric columns (except customerID) to factors for clarity
telco <- telco %>%
  mutate(across(
    .cols = -customer_id, # Exclude customerID column
    .fns = ~ if (!is.numeric(.)) as.factor(.) else .
  ))
```

Analysis and Insights

Summarise Churn Rates by New Variables

The following summaries show how churn rates vary across demographic and payment groups.

```
# Churn rate by SeniorStatus
churn_by_senior <- telco %>%
  group_by(senior_status) %>%
  summarise(
    Count = n(),
    ChurnRate = mean(churn_binary, na.rm = TRUE) * 100
  )

churn_by_senior
```

```
#> # A tibble: 2 x 3
#>   senior_status Count ChurnRate
#>   <fct>         <int>     <dbl>
#> 1 Non-Senior     2019      15.1
#> 2 Senior           91      24.2
```

```
# Churn rate by PartnerStatus
churn_by_partner <- telco %>%
  group_by(partner_status) %>%
```

```
  summarise(
    Count = n(),
    ChurnRate = mean(churn_binary, na.rm = TRUE) * 100
  )

churn_by_partner
```

```
#> # A tibble: 2 x 3
#>   partner_status Count ChurnRate
#>   <fct>          <int>     <dbl>
#> 1 No Partner       361      21.3
#> 2 Partner         1749      14.2
```

```
# Churn rate by DependentStatus
churn_by_dependents <- telco %>%
  group_by(dependent_status) %>%
  summarise(
    Count = n(),
    ChurnRate = mean(churn_binary, na.rm = TRUE) * 100
  )

churn_by_dependents
```

```
#> # A tibble: 1 x 3
#>   dependent_status Count ChurnRate
#>   <fct>            <int>     <dbl>
#> 1 Dependents        2110      15.5
```

```
# Churn rate by PaymentMethodGroup
churn_by_payment <- telco %>%
  group_by(payment_method_group) %>%
  summarise(
    count = n(),
    avg_monthly_charges = mean(monthly_charges, na.rm = TRUE),
    churn_rate = mean(churn_binary, na.rm = TRUE) * 100
  )

churn_by_payment
```

```
#> # A tibble: 3 x 4
```

```
#>   payment_method_group count avg_monthly_charges churn_rate
#>   <fct>                <int>               <dbl>      <dbl>
#> 1 Automatic             1068                63.5       9.93
#> 2 Electronic check       479                73.6      32.2
#> 3 Mailed check           563                40.0      11.7
```

Additional Data Analysis

We further explore churn rates across overall, contract, and internet service segments, as well as summarise monthly charges by gender.

```
# Overall churn rate
overall_churn_rate <- telco %>%
  summarise(
    Total = n(),
    churned = sum(churn_binary, na.rm = TRUE),
    churn_rate = churned / Total * 100
  )

overall_churn_rate
```

```
#> # A tibble: 1 x 3
#>   Total churned churn_rate
#>   <int>   <dbl>      <dbl>
#> 1  2110     326       15.5
```

```
# Churn rate by Contract Type
churn_by_contract <- telco %>%
  group_by(contract) %>%
  summarise(
    count = n(),
    avg_monthly_charges = mean(monthly_charges, na.rm = TRUE),
    avg_tenure = mean(tenure, na.rm = TRUE),
    churn_rate = mean(churn_binary, na.rm = TRUE) * 100
  )

churn_by_contract
```

```
#> # A tibble: 3 x 5
#>   contract       count avg_monthly_charges avg_tenure churn_rate
#>   <fct>          <int>               <dbl>      <dbl>      <dbl>
```

```
#> 1 Month-to-month   789                62.0       20.5      32.8
#> 2 One year         531                60.6       40.9       9.23
#> 3 Two year         790                56.3       54.5       2.28
```

```
# Churn rate by Internet Service Type
churn_by_internet <- telco %>%
  group_by(internet_service) %>%
  summarise(
    count = n(),
    churnRate = mean(churn_binary, na.rm = TRUE) * 100
  )

churn_by_internet
```

```
#> # A tibble: 3 x 3
#>   internet_service count churnRate
#>   <fct>            <int>     <dbl>
#> 1 DSL                805     11.9
#> 2 Fiber optic        662     30.5
#> 3 No                 643      4.35
```

```
# Summary statistics for MonthlyCharges by Gender
charges_by_gender <- telco %>%
  group_by(gender) %>%
  summarise(
    mean_monthly_charges = mean(monthly_charges, na.rm = TRUE),
    median_monthly_charges = median(monthly_charges, na.rm = TRUE),
    max_monthly_charges = max(monthly_charges, na.rm = TRUE)
  )

charges_by_gender
```

```
#> # A tibble: 2 x 4
#>   gender mean_monthly_charges median_monthly_charges max_monthly_charges
#>   <fct>                 <dbl>                  <dbl>                <dbl>
#> 1 Female                 59.4                   61.2                 119.
#> 2 Male                   59.6                   61.0                 117.
```

```
# Identify Contract type with highest churn rate
highest_churn_contract <- churn_by_contract %>%
  arrange(desc(churn_rate))
```

```
highest_churn_contract
```

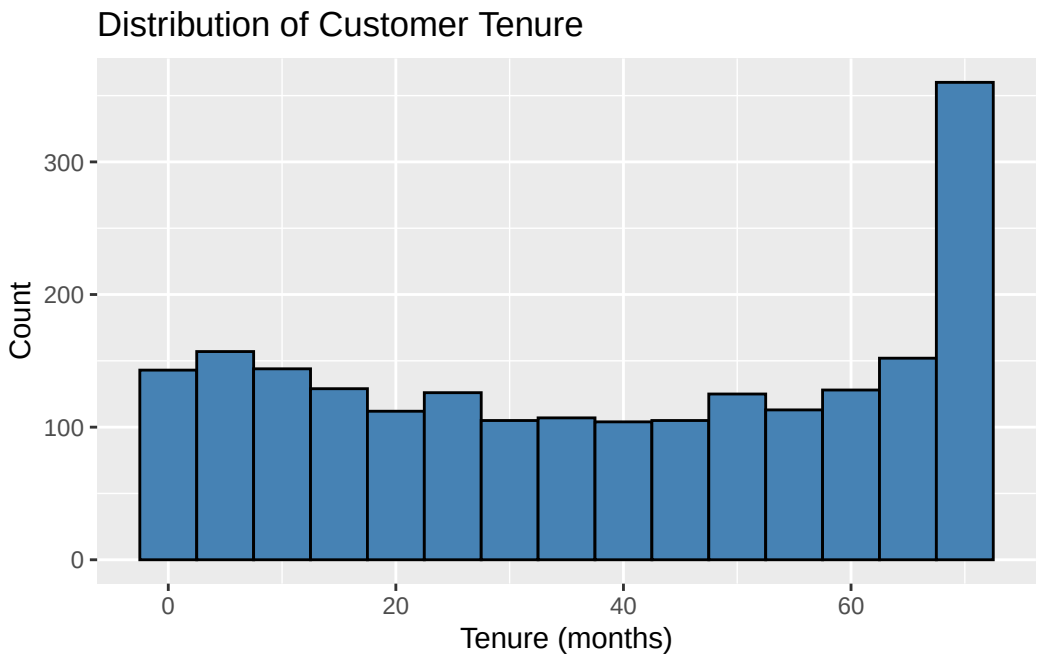
```
#> # A tibble: 3 x 5
#>   contract       count avg_monthly_charges avg_tenure churn_rate
#>   <fct>          <int>               <dbl>      <dbl>      <dbl>
#> 1 Month-to-month   789                62.0       20.5      32.8
#> 2 One year         531                60.6       40.9       9.23
#> 3 Two year         790                56.3       54.5       2.28
```

Data Visualisation

Visualisations help uncover trends and patterns that might not be immediately apparent from summary tables.

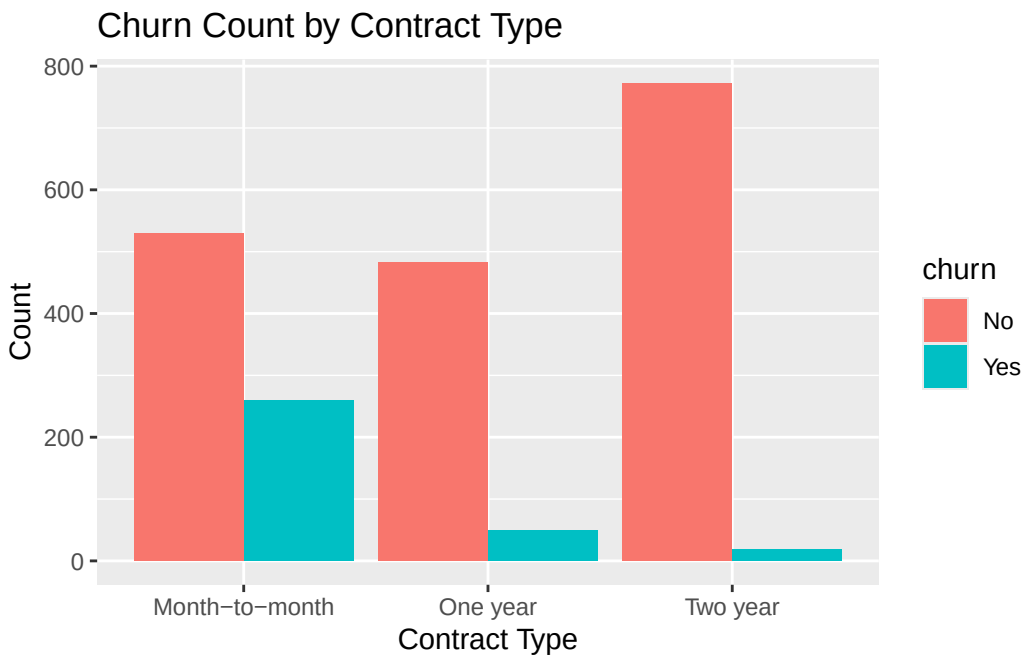
Histogram of Customer Tenure

```
telco |> ggplot(aes(x = tenure)) +
  geom_histogram(binwidth = 5, fill = "steelblue", color = "black") +
  labs(title = "Distribution of Customer Tenure", x = "Tenure (months)", y = "Count")
```



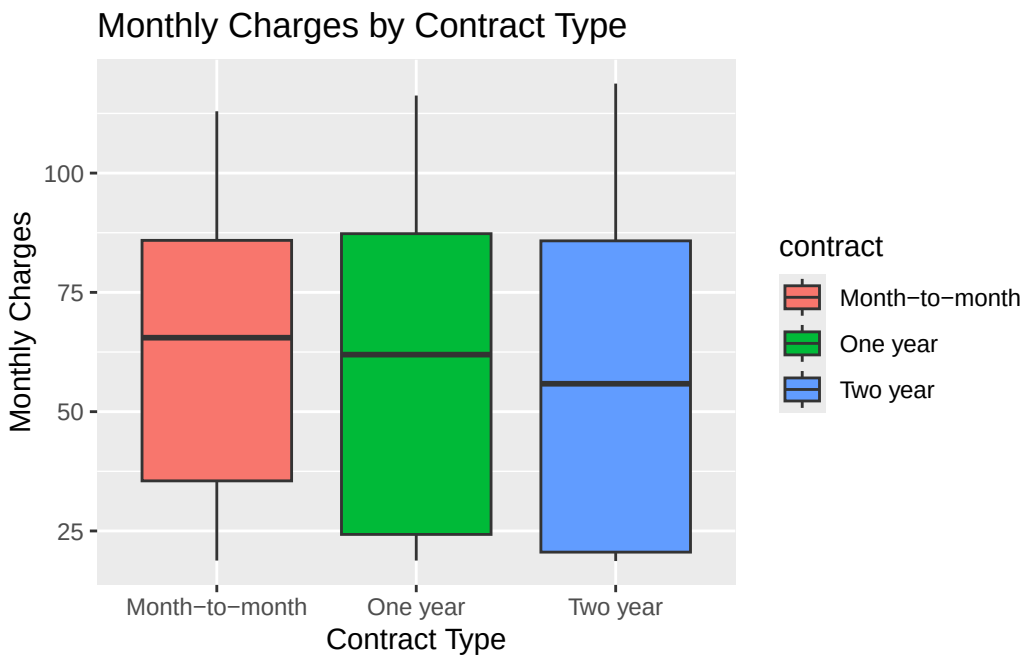
Bar Chart of Churn Count by Contract Type

```
telco |> ggplot(aes(x = contract, fill = churn)) +
  geom_bar(position = position_dodge()) +
  labs(title = "Churn Count by Contract Type", x = "Contract Type", y = "Count")
```



Boxplot: MonthlyCharges across Contract Types

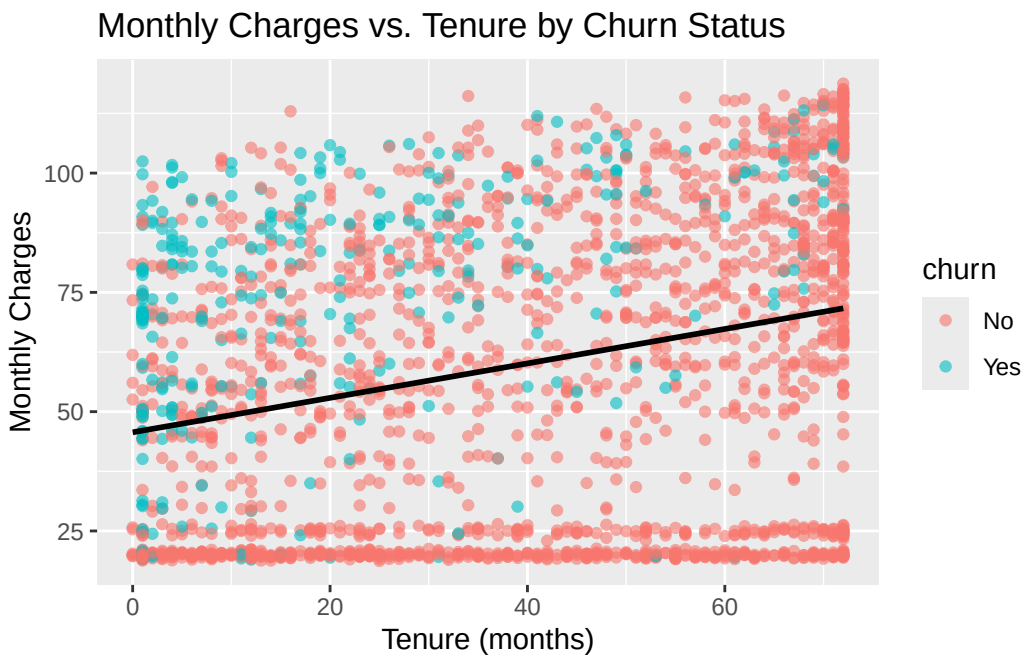
```
telco |> ggplot(aes(x = contract, y = monthly_charges, fill = contract)) +
  geom_boxplot() +
  labs(title = "Monthly Charges by Contract Type", x = "Contract Type", y = "Monthly Charges
```



Scatter Plot: Tenure vs MonthlyCharges coloured by Churn Status

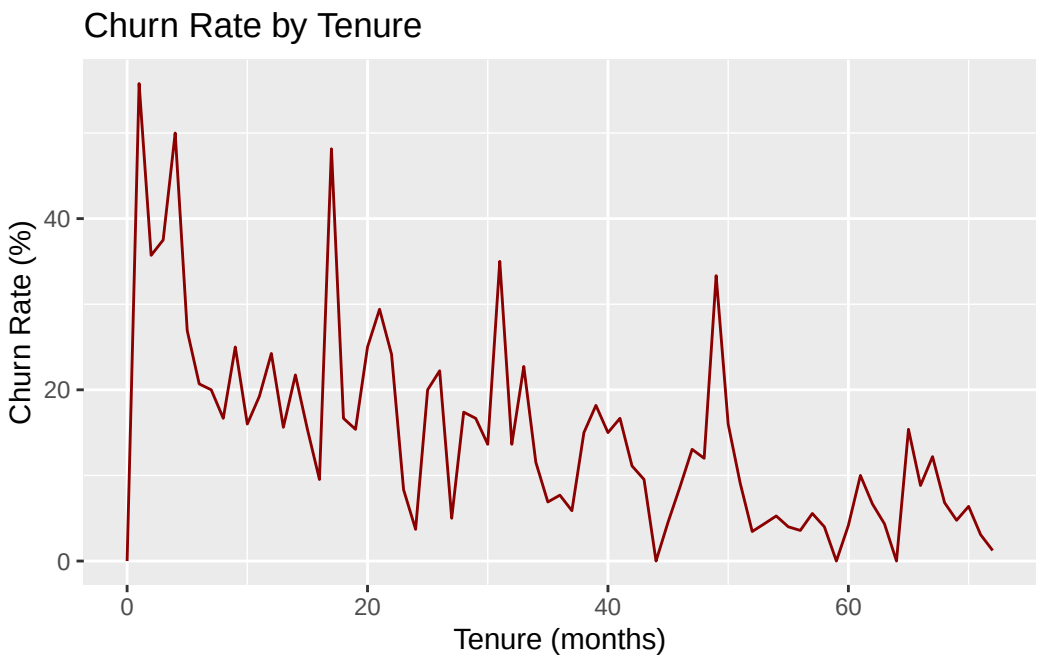
```
telco |> ggplot(aes(x = tenure, y = monthly_charges, color = churn)) +
  geom_point(alpha = 0.6) +
  geom_smooth(method = "lm", se = FALSE, color = "black") +
  labs(title = "Monthly Charges vs. Tenure by Churn Status", x = "Tenure (months)", y = "Mon
```

```
#> `geom_smooth()` using formula = 'y ~ x'
```



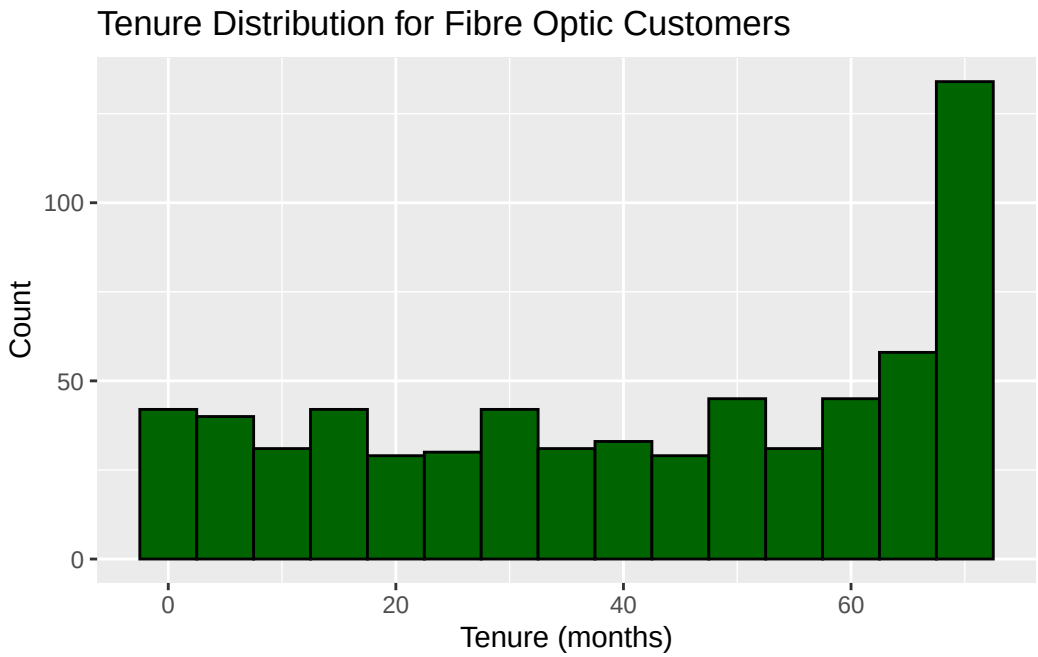
Line Plot: Churn Rate by Tenure

```
churn_by_tenure <- telco %>%
  group_by(tenure) %>%
  summarise(
    Total = n(),
    Churned = sum(churn_binary, na.rm = TRUE),
    ChurnRate = Churned / Total * 100
  )

churn_by_tenure |> ggplot(aes(x = tenure, y = ChurnRate)) +
  geom_line(color = "darkred") +
  labs(title = "Churn Rate by Tenure", x = "Tenure (months)", y = "Churn Rate (%)")
```



Histogram: Tenure Distribution for Fibre Optic Customers

```
fiber_customers <- telco %>% filter(internet_service == "Fiber optic")

fiber_customers |> ggplot(aes(x = tenure)) +
  geom_histogram(binwidth = 5, fill = "darkgreen", color = "black") +
  labs(title = "Tenure Distribution for Fibre Optic Customers", x = "Tenure (months)", y = "
```



Telco Customer Churn Analysis Report

Introduction

This report presents a comprehensive analysis of the Telco Customer Churn dataset, which contains detailed records of 7,043 customers. The dataset includes demographic information, account details, service usage metrics, and churn behaviour. Our primary objective is to identify key drivers of churn and develop actionable recommendations to improve customer retention. The analysis leverages robust data transformation, exploratory statistics, and visualisation techniques to uncover insights that can guide strategic decision-making.

Data Preparation and Transformation

Our initial steps focused on ensuring data quality and creating new variables to facilitate deeper insights. We:

- **Cleaned the data** by converting variables such as *TotalCharges* into a numeric format and recoding *SeniorCitizen* from a binary indicator into a descriptive format.
- **Transformed the churn indicator** into a binary variable (*ChurnBinary*), enabling precise calculation of churn rates.

- **Derived new metrics** like *AvgChargePerMonth* (TotalCharges divided by tenure) and *ServiceCount* (the total number of additional services a customer subscribes to).
- **Re-encoded key demographic variables**:
  - **senior_status** distinguishes between "Senior" and "Non-Senior" customers.
  - **partner_status** categorises customers as having a partner or not.
  - **dependent_status** identifies whether customers have dependents.
- **Grouped payment methods** into a new variable (*payment_method_group*) to compare broad categories: Automatic (including Bank transfer and Credit card), Electronic check, and Mailed check.

These transformations provided us with a richer dataset that supports more granular analyses of churn behaviour.

Exploratory Analysis and Key Findings

Overall Churn

Our analysis reveals an overall churn rate of approximately **26.5%**. This indicates that about one in four customers leaves the service, underscoring the need for targeted retention strategies.

Contract Type and Internet Service

- **Contract Type:**
  - Customers on **month-to-month** contracts experience a markedly high churn rate of **42.7%**.
  - In contrast, those on **one-year** and **two-year** contracts churn at rates of **11.3%** and **2.8%** respectively, suggesting that longer-term commitments are associated with greater customer loyalty.
- **Internet Service:**
  - Churn rates vary considerably by internet service type.
  - **Fibre optic** customers have a high churn rate of **41.9%**, while **DSL** customers churn at **19.0%**, and those with **no internet service** exhibit a relatively low churn rate of **7.4%**.

These differences indicate that the type of service subscription plays a crucial role in customer retention. The high churn among fibre optic subscribers may reflect issues with pricing or service expectations.

Demographic Insights

- **Senior Status:**
  - Among **Non-Seniors** (5,901 customers), the churn rate is **23.6%**.
  - In contrast, **Seniors** (1,142 customers) exhibit a significantly higher churn rate of **41.7%**.
  - *Interpretation:* Seniors may face unique challenges or have different expectations that increase their propensity to leave.
- **Partner Status:**
  - Customers without a partner (3,641 customers) show a churn rate of **33.0%**, while those with a partner (3,402 customers) churn at **19.7%**.
  - *Interpretation:* Having a partner may indicate a more stable personal environment, which could translate into lower churn.
- **Dependent Status:**
  - Customers with dependents (2,110 customers) experience a churn rate of **15.5%**, compared to **31.3%** for those without dependents (4,933 customers).
  - *Interpretation:* The presence of dependents appears to be associated with a lower likelihood of churn, perhaps due to higher commitment levels or differing priorities.

Payment Methods

- **Payment Method Group:**
  - Customers paying via **electronic check** (2,365 customers) have the highest churn rate at **45.3%** and also incur the highest average monthly charges (£76.3).
  - In contrast, those using **automatic** payment methods (3,066 customers) have a churn rate of **16.0%** with an average monthly charge of £66.9, while **mailed check** users (1,612 customers) churn at **19.1%** with lower average charges (£43.9).

– *Interpretation:* The elevated churn rate among electronic check users might reflect higher costs or dissatisfaction with billing, indicating an area for potential intervention.

Visual Insights

Key visualisations further illustrate our findings:

- **Tenure Distribution:**
  A histogram of customer tenure reveals two distinct groups—new customers with very short tenures and a substantial cohort of long-term customers. This bimodal distribution suggests that retention strategies may need to be tailored differently for new versus established customers.
- **Churn by Contract Type:**
  Bar charts clearly show that month-to-month contracts drive the majority of churn, reinforcing the numerical findings.
- **Monthly Charges vs. Tenure:**
  A scatter plot demonstrates that customers with high monthly charges and short tenures are particularly prone to churn, highlighting the need for early intervention among this group.
- **Churn Rate by Tenure:**
  A line graph indicates that the churn rate declines steadily as tenure increases, emphasising the importance of retaining customers during the critical early months of service.

Recommendations

Based on these insights, we propose the following strategic recommendations:

1. **Target Month-to-Month Subscribers:**
   - Implement loyalty programmes or incentivise longer-term contracts to reduce the high churn rate in this segment.
2. **Focus on Fibre Optic Subscribers:**
   - Investigate the underlying causes of high churn among fibre optic users, such as pricing or service quality issues, and consider targeted offers to enhance value perception.
3. **Enhanced Engagement for Senior Customers:**

- Develop tailored engagement initiatives for senior customers, who are significantly more likely to churn. This could include specialised support, personalised communication, or alternative service plans.

4. **Review Payment Options:**
    - Explore why customers using electronic checks are experiencing high churn and consider adjusting billing practices or offering alternative payment methods to improve satisfaction.
5. **Early Intervention Strategies:**
    - For new customers, especially those with high initial monthly charges, introduce proactive onboarding measures and personalised offers to encourage longer-term commitments.

Conclusion

The analysis indicates that while the overall churn rate stands at about 26.5%, particular customer segments—such as those on month-to-month contracts, fibre optic subscribers, senior customers, and users of electronic check payment methods—exhibit significantly higher churn rates. Addressing these vulnerabilities through targeted retention strategies and improved customer engagement will be crucial in reducing churn and enhancing long-term profitability.

11.7 Exercise 11.1: Analyzing a Rape Survey for the Federal Government of Nigeria

11.7.1 Project Overview

You have been engaged by the Federal Government of Nigeria to analyse a sensitive dataset about rape incidents gathered from a national survey in Lagos. Without explicit instructions, your task is to apply your data analysis and visualisation skills to uncover insights that may inform policy decisions, resource allocation, or awareness campaigns.

11.7.2 The Dataset

Locate the `rape-survey.xlsx` file in the `r-data` directory. If you don't already have the file, you can download it from Google Drive.

11.7.3 Your Task

- **Data Preparation**: Import and clean the data, addressing missing values, verifying data formats, and ensuring variables align with analytical goals.
- **Exploratory Analysis**: Understand distributions of key variables, identify patterns or risk factors, and discover regional or demographic differences.
- **Visualisation**: Use bar charts, boxplots, or heatmaps to highlight differences among groups, time periods, or severity levels.
- **Insights and Recommendations**:
  Present findings in a concise, data-driven narrative. Consider ethical sensitivities when interpreting results. Provide recommendations that might guide policy, resource investment, or public education efforts.

11.8 Integrating Lab Skills

In previous labs, you learned how to organise projects (Lab 4), wrangle data (Lab 5 and Lab 6), visualise insights (Lab 7), and understand the broader data science field (Lab 10). Applying these techniques to real projects consolidates your skillset:

- **Organisational Skills**: Maintain reproducible workflows to ensure credibility and traceability.
- **Data Wrangling**: Tidy and transform datasets to facilitate smooth analysis and modelling.
- **Visualisation**: Create plots that reveal patterns, trends, or anomalies, guiding decision-makers effectively.
- **Statistical and Analytical Rigor**: Use tests and models judiciously, validating assumptions and ensuring insights are robust.

> **Note**
>
> **Reflection Question**
> Having seen how individual techniques fit into larger projects, how does this perspective influence the way you approach learning new analytical methods or tools?

11.9 Conclusion and Further Steps

Use case projects are where R's capabilities truly shine. By applying your skills to realistic scenarios, you cultivate problem-solving abilities, enhance creativity, and build confidence. Beyond academic exercises, these projects mirror professional tasks you might encounter in workplaces or research settings.

Next Steps:

- Continue seeking or designing new use case projects to keep skills sharp and current.
- Explore complex datasets, integrate more advanced modelling techniques, or experiment with interactive dashboards.
- Share your results publicly (in blogs, portfolios, or open-source contributions) to receive feedback and build a professional presence.

> **Note**
>
> **Reflection Question:**
> As you move forward, what kind of real-world projects are you most interested in tackling, and how will these projects shape your future growth as a data professional?

11.10 General Practice Quiz 11

Question 1:

What is the main purpose of the pipe operator `|>` in R?

a) To run code in parallel.

b) To nest functions inside one another.

c) To pass the output of one function as the input to the next, improving code readability.

d) To automatically clean missing data.

Question 2:

In a reproducible R workflow (as discussed in early labs), which file type is commonly used to document code, results, and narrative together?

a) CSV files

b) R Markdown (or Quarto) documents

c) PNG images

d) Excel spreadsheets

Question 3:

When creating a new RStudio Project to ensure reproducibility and organisation of your analysis, what is one key advantage?

a) It automatically generates a machine learning model.

b) It sets the working directory to the project folder, simplifying relative paths.

c) It prevents all missing values.

d) It disables package installation from CRAN.

Question 4:

The principle of **tidy data** states that:

a) Each dataset should have no missing values.

b) Each column represents a variable, each row represents an observation, and each cell contains a single value.

c) Each dataset must have at least 10 columns.

d) Each value in the dataset must be numeric.

Question 5:

Which `dplyr` verb is used to filter rows based on logical conditions?

a) `select()`

b) `mutate()`

c) `filter()`

d) `summarise()`

Question 6:

To create new columns or modify existing ones in your dataset using `dplyr`, you would use:

a) `select()`

b) `mutate()`

c) `arrange()`

d) `group_by()`

Question 7:

Which `ggplot2` component maps data variables to visual properties like axes, colour, or size?

a) Theme

b) Facets

c) Aesthetics (`aes()`)

d) Scales

Question 8:

To reorder rows of data based on a variable's value using `dplyr`, which function should be applied?

a) `rename()`

b) `arrange()`

c) `distinct()`

d) `count()`

Question 9:

In the data science lifecycle discussed, which stage primarily involves creating charts, graphs, or other graphical representations of data?

a) Import

b) Tidy

c) Transform

d) Visualise

Question 10:

What is the role of `group_by()` in conjunction with `summarise()`?

a) It imports a dataset from the internet.

b) It filters rows based on conditions.

c) It splits the data into groups, allowing summarised statistics per group.

d) It changes variable names.

Question 11:

When exploring data from a new dataset, which of the following is a best practice?

a) Immediately running complex models without understanding distributions.

b) Creating exploratory visualisations and computing descriptive statistics.

c) Ignoring missing values.

d) Never using `glimpse()` or `head()`.

Question 12:

Which `ggplot2` function would you use to create a boxplot?

a) `geom_bar()`

b) `geom_point()`

c) `geom_boxplot()`

d) `geom_smooth()`

Question 13:

Converting code, analysis, and narrative into a single reproducible document is commonly achieved with:

a) `read_csv()` only.

b) Proprietary binary formats.

c) R Markdown (or Quarto) documents.

d) Manually copying results into Word documents.

Question 14:

Which operator in R is used to chain data operations in a logical sequence, making code more readable?

a) `%>%` (from magrittr) or `|>` (native pipe)

b) `$`

c) `*`

d) `=`

Question 15:

Data science is often described as an intersection of three main areas. Which combination is correct?

a) Domain expertise, mathematics/statistics, and computer science/programming.

b) Chemistry, physics, and biology.

c) Finance, marketing, and sales.

d) Geography, history, and literature.

Question 16:

In a data science project, why is communicating findings effectively so important?

a) It ensures the code runs faster.

b) It guarantees no missing values remain.

c) It enables stakeholders to understand insights and make informed decisions.

d) It replaces the need for data transformations.

Question 17:

When dealing with missing data, which is NOT a recommended strategy?

a) Identifying and quantifying missing values.

b) Imputing values using mean or median if appropriate.

c) Removing all data points and ignoring the missingness context.

d) Documenting how missing data was handled.

Question 18:

Which `dplyr` function extracts unique rows or identifies distinct values?

a) `distinct()`

b) `rename()`

c) `relocate()`

d) `case_when()`

Question 19:

Why are use case projects invaluable for learners transitioning from theory to practice?

a) They allow bypassing basic R syntax rules.

b) They simplify code without testing problem-solving skills.

c) They help integrate various skills, face real-world challenges, and deepen understanding.

d) They remove the need for documentation.

Question 20:

In the data science lifecycle, what is typically the final stage?

a) Model

b) Communicate

c) Tidy

d) Transform

See the Solution to General Quiz 11

11.11 Reflective Summary

In **Lab 11**, you learned the importance of use case projects for mastering R and data analysis techniques. By engaging with real-world scenarios, you:

- **Applied Theoretical Knowledge in Practice**: Transitioned from isolated exercises to comprehensive, problem-focused projects.
- **Enhanced Problem-Solving Skills**: Overcame practical challenges, honed troubleshooting abilities, and adapted solutions.
- **Developed a Holistic Understanding**: Integrated data wrangling, visualisation, reproducibility, and analysis into cohesive workflows.
- **Built Confidence and Portfolios**: Gained assurance in your skills, creating tangible proof of your capabilities.
- **Reinforced Ethical and Contextual Thinking**: Appreciated the responsibilities and sensitivities required when working with real and potentially sensitive datasets.

These experiences lay the groundwork for your continued development as a data analyst or data scientist. As you tackle increasingly complex projects, remember that every dataset presents an opportunity to refine your methods, discover insights, and communicate stories that inform real-world decisions.

Congratulations on completing the last lab in this book! You have now demonstrated your ability to apply R skills and data science principles in practical contexts, setting the stage for more advanced, specialised, and impactful data analyses in the future.

A Solutions

Lab 1: Getting Started with R

Solution Quiz 1.1

Question 1:

What is the primary role of R in the R programming environment?

a) A user interface for writing code

b) A programming language for statistical computing ✓

c) A package manager

d) A data visualization tool

Question 2:

Which of the following best describes RStudio?

a) A standalone programming language

b) A text editor for writing R scripts

c) An Integrated Development Environment (IDE) for R ✓

d) A package repository for R

Question 3:

Which of the following is the correct sequence of steps to install R and RStudio on your computer?

a) Install RStudio first, then install R from the CRAN website.

b) Install R from the CRAN website first, then install RStudio. ✓

c) Download both R and RStudio from the RStudio website and install them simultaneously.

d) Install R from the Microsoft Store, then install RStudio from the CRAN website.

Question 4:

Which keyboard shortcut runs the current line of code in RStudio on Windows?

a) `Ctrl + S`

b) `Ctrl + Enter` ✓

c) `Alt + R`

d) `Shift + Enter`

Question 5:

After successful installation, which pane in RStudio indicates that R is ready to use?

a) Source Pane

b) Console Pane ✓

c) Environment Pane

d) Files Pane

Return to Quiz 1.1

Solution Quiz 1.2

Question 1:

Which pane in RStudio is primarily used for writing and editing R scripts?

a) `Console Pane`

b) `Source Pane` ✓

c) `Environment Pane`

d) `Files Pane`

Question 2:

What does the Environment Tab in RStudio display?

a) Available packages and their statuses

b) Active variables, data frames, and objects in the current session ✓

c) The file directory of your project

d) Graphical plots and visualizations

Question 3:

How can you execute a selected block of code in the Source Pane?

a) Press `Ctrl + S`

b) Press `Ctrl + Enter`

c) Click the "Run" button

d) Both b) and c) ✓

Question 4:

Which pane would you use to install and load R packages?

a) `Source Pane`

b) `Console Pane`

c) `Files Pane`

d) `Packages Tab` within Files/Plots/Packages/Help Pane ✓

Question 5:

Where can you find R's built-in documentation and help files within RStudio?

a) `Source Pane`

b) `Console Pane`

c) `Environment Pane`

d) `Help Tab` within Files/Plots/Packages/Help Pane ✓

Return to Quiz 1.2

Solution 1.2.1: Basic Calculations

```
2 + 6 - 12
```

```
#> [1] -4
```

```
4 * 3 - 8
```

```
#> [1] 4
```

```
81 / 6
```

```
#> [1] 13.5
```

```
16 %% 3
```

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

```
2^3
```

```
#> [1] 8
```

```
(3 + 2) * (6 - 4) + 2
```

```
#> [1] 12
```

Try changing the numbers or operations in the calculations above to see different results. This hands-on experimentation will deepen your understanding of how R processes arithmetic operations.

Return to Exercise 1.2.1

Solution 1.3.1: A Quick Hands-On

Try it yourself! Create a variable called `my_name` and assign your name to it. Then, print a greeting that says "Hello, [Your Name]!".

```
my_name <- "Alice"
print(paste("Hello,", my_name, "!"))
```

```
#> [1] "Hello, Alice !"
```

You can also use the following:

```
my_name <- "Alice"

cat("Hello,", my_name, "!")
```

```
#> Hello, Alice !
```

Return to Exercise 1.3.1

Solution Quiz 1.3

Question 1:

Which function is used to determine the class of an object in R?

a) `vector()`

b) `c()`

c) `class()` ✓

d) `typeof()`

Question 2:

What will the class of the following object be in R?

```
my_var <- TRUE
```

a) `numeric`

b) `character`

c) `logical` ✓

d) `complex`

Question 3:

Which of the following is an acceptable variable name in R?

a) `2nd_place`

b) `total-sales`

c) `average_height` ✓

d) `user name`

Question 4:

How can you convert a character string `"123"` to a numeric type in R?

a) `to.numeric("123")`

b) `as.numeric("123")` ✓

c) `convert("123", "numeric")`

d) `numeric("123")`

Question 5:

What will be the result of the following R code?

```
weight <- "60.4 kg"
weight_numeric <- as.numeric(weight)
```

a) `60.4`

b) `"60.4"`

c) `NA` with a warning ✓

d) `NULL`

Return to Quiz 1.3

Solution 1.3.3: Variable Assignment and Data Types

```
age <- 15

class(age)
```

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

```
weight <- "60.4 kg"

class(weight)
```

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

```
weight_numeric <- as.numeric(gsub(" kg", "", weight))

class(weight_numeric)
```

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

```
smile_face <- "FALSE"

class(smile_face)
```

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

```
smile_face_logical <- as.logical(smile_face)

class(smile_face_logical)
```

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

Return to Exercise 1.3.3

Solution Quiz 1.4

Question 1:

What will be the output of the following R code?

```r
number <- 10
if (number %% 2 == 0) {
  print("Even")
} else {
  print("Odd")
}
```

a) `Odd`

b) `Even` ✓

c) `TRUE`

d) `FALSE`

Question 2:

Which logical operator in R returns `TRUE` only if both conditions are `TRUE`?

a) `|` (OR)

b) `&` (AND) ✓

c) `!` (NOT)

d) `^` (XOR)

Question 3:

In the `switch()` function, what does the following code return when `choice` is 3?

```r
num1 <- 10
num2 <- 5
choice <- 3

result <- switch(choice,
  num1 + num2,
  num1 - num2,
  num1 * num2,
```

```
  "Invalid operation"
)

print(result)
```

a) `15`

b) `5`

c) `50` ✓

d) `"Invalid operation"`

Question 4:

What is the purpose of including a default case in a `switch()` statement?

a) To handle cases where the expression matches multiple conditions

b) To execute a block of code if none of the specified cases match ✓

c) To prioritize certain cases over others

d) To initialize variables within the switch

Question 5:

Which of the following uses the NOT (`!`) operator correctly in an `if` statement?

a)

```
if (!c) {
  print("The condition is false")
}
```

b)

```
if (c!) {
  print("The condition is false")
}
```

c)

```r
if (c != TRUE) {
  print("The condition is false")
}
```

d) Both a) and c) ✓

Return to Quiz 1.4

Solution 1.4.1: Conditional Statements

Task 1

```r
number <- 10

if (number %% 2 == 0) {
  print("Even")
} else {
  print("Odd")
}
```

```
#> [1] "Even"
```

Answer: `"Even"` because `10 %% 2 == 0` evaluates to `TRUE`.

```r
m <- 5

n <- 7

if (m > n) {
  print("m is greater than n")
} else if (m < n) {
  print("m is less than n")
} else {
  print("m and n are equal")
}
```

```
#> [1] "m is less than n"
```

Return to Exercise 1.4.1

Solution 1.4.2: Menu Selection Using `switch()`

Use the `switch()` Function:

```
option <- "exit"

message <- switch(option,
  balance = "Your current balance is $1,000.",
  deposit = "Enter the amount you wish to deposit.",
  withdraw = "Enter the amount you wish to withdraw.",
  exit = "Thank you for using our banking services.",
  "Invalid selection. Please choose a valid option."
)
```

Display the Message:

```
print(message)
```

```
#> [1] "Thank you for using our banking services."
```

Change the value of `option` to test different menu selections and observe the outputs.

Return to Exercise 1.4.2

Solution 1.4.3: Mini-Project - Basic Calculator in R

```
# Get user input

num1 <- as.numeric(readline(prompt = "Enter the first number: ")) # You entered 15

num2 <- as.numeric(readline(prompt = "Enter the second number: ")) # You entered 5

operation <- readline(prompt = "Choose an operation (+, -, *, /): ") # You chose +
```

```
# Perform calculation

result <- switch(operation,
  "+" = num1 + num2,
  "-" = num1 - num2,
  "*" = num1 * num2,
```

```
  "/" = if (num2 != 0) num1 / num2 else "Error: Division by zero",
  "Invalid operation"
)
# Display result
print(paste("The result is:", result))
```

```
#> [1] "The result is: 20"
```

Return to Exercise 1.4.3

Lab 2: Understanding Data Structures

Reflection Solution 2.1.1

Why is it important to know that R uses 1-based indexing?

Answer: Because starting from 1 affects how you access elements; forgetting this can lead to off-by-one errors.

Return to Reflection Question 2.1.1

Solution 2.1.1: Vector Selection

```
# Task 1: Create the vector
monthly_sales <- c(120, 135, 150, 160, 155, 145, 170, 180, 165, 175, 190, 200)

# Task 2: Access sales for March, June, and December
sales_selected_months <- monthly_sales[c(3, 6, 12)]
sales_selected_months
```

```
#> [1] 150 145 200
```

```
# Task 3: Access sales that are less than 60
sales_lessthan_60 <- monthly_sales[monthly_sales < 60]
sales_lessthan_60
```

```
#> numeric(0)
```

```r
# Task 4: Calculate average sales for the first quarter
first_quarter_sales <- monthly_sales[1:3]

average_first_quarter <- mean(first_quarter_sales)
average_first_quarter
```

```
#> [1] 135
```

```r
# Task 5: Extract the sales figures for the last month of each quarter of the year
quarter_last_months <- monthly_sales[c(3, 6, 9, 12)]
quarter_last_months
```

```
#> [1] 150 145 165 200
```

Return to Exercise 2.1.1

Reflection Solution 2.1.2

- How does converting character vectors to factors benefit data analysis in R?

  Answer: Converting character vectors to factors benefits data analysis by:

  - Ensuring data integrity through predefined categories.
  - Improving efficiency in storage and computation.
  - Allowing statistical functions to correctly interpret and handle categorical variables.

- When would you use a factor instead of a character vector in R?

  Answer: Use a factor when working with categorical data that have a fixed set of possible values, especially when you plan to perform statistical analyses or modelling that treat categories differently than continuous data.

Return to Reflection Question 2.1.2

Solution Quiz 2.1

Question 1:

Which function is used to create a vector in R?

a) `vector()`

b) `c()` ✓

c) `list()`

d) `data.frame()`

Question 2:

Given the vector:

```
v <- c(2, 4, 6, 8, 10)
```

What is the result of `v * 3`?

a) `c(6, 12, 18, 24, 30)` ✓

b) `c(2, 4, 6, 8, 10, 3)`

c) `c(6, 12, 18, 24)`

d) An error occurs

Question 3:

In R, is the vector `c(TRUE, FALSE, TRUE)` considered a numeric vector?

a) True
b) False ✓

Question 4:

What will be the output of the following code?

```
numbers <- c(1, 3, 5, 7, 9)
numbers[2:4]
```

a) `1, 3, 5`

b) `3, 5, 7` ✓

c) `5, 7, 9`

d) `2, 4, 6`

Question 5:

Which of the following best describes a factor in R?

a) A numerical vector

b) A categorical variable with predefined levels ✓

c) A two-dimensional data structure

d) A list of vectors

Question 6:

Which function is used to create sequences including those with either integer or non-integer steps?

a) `:`

b) `seq()` ✓

c) `rep()`

d) `sample()`

Question 7:
What does the following code output?

```
seq(10, 1, by = -3)
```

a) `10, 7, 4, 1` ✓

b) `10, 7, 4`

c) `1, 4, 7, 10`

d) An error occurs

Question 8:

Suppose you want to create a vector that repeats the sequence `1, 2, 3` five times. Which code will achieve this?

a) `rep(c(1, 2, 3), each = 5)`

b) `rep(c(1, 2, 3), times = 5)` ✓

c) `rep(1:3, times = 5)`

d) `rep(1:3, each = 5)`

Question 9:

Suppose you are drawing coins from a treasure chest. There are 100 coins in this chest: 20 gold, 30 silver, and 50 bronze. Use R to draw 5 random coins from the chest. Use `set.seed(50)` to ensure reproducibility.

What will be the output of the random draw?

Code:

```r
set.seed(50)
coins <- c(rep("Gold", 20), rep("Silver", 30), rep("Bronze", 50))
draw <- sample(coins, size = 5, replace = TRUE)
draw
```

```
#> [1] "Gold"   "Bronze" "Bronze" "Bronze" "Silver"
```

a) `Silver, Bronze, Bronze, Bronze, Silver`

b) `Gold, Gold, Silver, Bronze, Bronze`

c) `Gold, Bronze, Bronze, Bronze, Silver` ✓

d) `Silver, Bronze, Gold, Bronze, Bronze`

Question 10:

What will the following code produce?

```r
c(1, 2, 3) + c(4, 5)
```

a) `5, 7, 8`

b) `5, 7, 7` ✓

c) An error due to unequal vector lengths

d) `5, 7, 9`

Explanation:

- The shorter vector `c(4, 5)` is **recycled** to match the length of the longer vector `c(1, 2, 3)`.
- After recycling, `c(4, 5)` becomes `c(4, 5, 4)`.
- The addition is performed element-wise:
  - `1 + 4 = 5`
  - `2 + 5 = 7`
  - `3 + 4 = 7`
- The result is `c(5, 7, 7)`.

This question introduces the concept of **vector recycling** in R.

Return to Quiz 2.1

Solution 2.1.2: Vector and Factor Manipulation

```r
# Task 1: Create the vector
feedback_ratings <- c("Good", "Excellent", "Poor", "Fair", "Good", "Excellent", "Fair")

# Task 2: Convert to ordered factor
feedback_factors <- factor(feedback_ratings,
  levels = c("Poor", "Fair", "Good", "Excellent"),
  ordered = TRUE
)

# Task 3: Summarize feedback ratings
summary(feedback_factors)
```

```
#>      Poor      Fair      Good Excellent
#>         1         2         2         2
```

```r
# Task 4: Count of "Excellent" ratings
excellent_count <- sum(feedback_factors == "Excellent")
excellent_count
```

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

Return to Exercise 2.1.2

Solution 2.2.1: Matrix Transpose

```r
# Define matrix A

A <- matrix(c(1, 3, 5, 2, 4, 6), nrow = 2, ncol = 3, byrow = TRUE)

A_transpose <- t(A)
```

Return to Exercise 2.2.1

Solution 2.2.2: Matrix Inverse Multiplication

```r
# Define matrices A and B
A <- matrix(c(4, 7, 2, 6), nrow = 2, ncol = 2, byrow = TRUE)

B <- matrix(c(3, 5, 1, 2), nrow = 2, ncol = 2, byrow = TRUE)

# Find the inverse of A
A_inverse <- solve(A)

# Multiply A_inverse by B
result <- A_inverse %*% B
```

Return to Exercise 2.2.2

Solution Quiz 2.2

Question 1:

Which R function is used to find the transpose of a matrix?

a) `transpose()`

b) `t()` ✓

c) `flip()`

d) `reverse()`

Question 2:

Given the matrix:

```
A <- matrix(1:6, nrow = 2, byrow = TRUE)
```

what is the value of `A[2, 3]`?

a) 3

b) 6 ✓

c) 5

d) 4

Question 3:

True or False: Matrix multiplication in R can be performed using the `*` operator.

a) True

b) False ✓

Matrix multiplication is performed using the `%*%` operator.

Question 4:

What will be the result of adding two matrices of different dimensions in R?

a) R will perform element-wise addition up to the length of the shorter matrix.

b) An error will occur due to dimension mismatch. ✓

c) R will recycle elements of the smaller matrix.

d) The matrices will be concatenated.

Question 5:

Which function can be used to calculate the sum of each column in a matrix `M`?

a) `rowSums(M)`

b) `colSums(M)` ✓

c) `sum(M)`

d) `apply(M, 2, sum)`

Question 6:

Which function is used to create a matrix in R?

a) `matrix()` ✓

b) `data.frame()`

c) `c()`

d) `list()`

Return to Quiz 2.2

Solution 2.3.1: Subsetting a Dataframe

```
# 1. Examine the airquality dataset

head(airquality) # Shows the first 6 rows by default
```

```
#>   Ozone Solar.R Wind Temp Month Day
#> 1    41     190  7.4   67     5   1
#> 2    36     118  8.0   72     5   2
#> 3    12     149 12.6   74     5   3
#> 4    18     313 11.5   62     5   4
#> 5    NA      NA 14.3   56     5   5
#> 6    28      NA 14.9   66     5   6
```

```
#|
View(airquality) # Opens dataset in a spreadsheet-like viewer (if in RStudio)
```

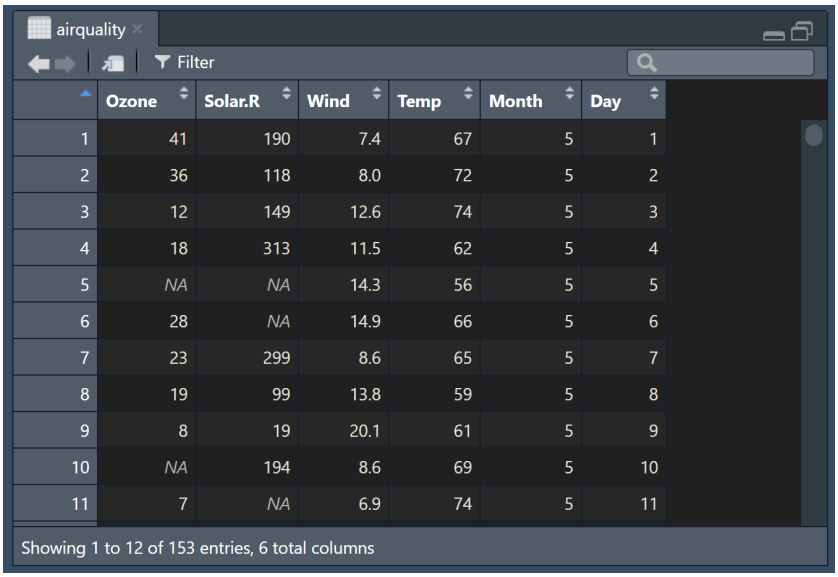


Figure A.1: Airquality Data Frame Preview in RStudio

```
str(airquality) # Display the structure of the dataset
```

```
#> 'data.frame':    153 obs. of  6 variables:
#>  $ Ozone  : int  41 36 12 18 NA 28 23 19 8 NA ...
#>  $ Solar.R: int  190 118 149 313 NA NA 299 99 19 194 ...
#>  $ Wind   : num  7.4 8 12.6 11.5 14.3 14.9 8.6 13.8 20.1 8.6 ...
#>  $ Temp   : int  67 72 74 62 56 66 65 59 61 69 ...
#>  $ Month  : int  5 5 5 5 5 5 5 5 5 5 ...
#>  $ Day    : int  1 2 3 4 5 6 7 8 9 10 ...
```

```
summary(airquality) # Gives summary statistics for each column
```

```
#>      Ozone           Solar.R           Wind             Temp      
#>  Min.   :  1.00   Min.   :  7.0   Min.   : 1.700   Min.   :56.00  
#>  1st Qu.: 18.00   1st Qu.:115.8   1st Qu.: 7.400   1st Qu.:72.00  
#>  Median : 31.50   Median :205.0   Median : 9.700   Median :79.00  
#>  Mean   : 42.13   Mean   :185.9   Mean   : 9.958   Mean   :77.88  
#>  3rd Qu.: 63.25   3rd Qu.:258.8   3rd Qu.:11.500   3rd Qu.:85.00  
```

```
#>  Max.   :168.00   Max.   :334.0   Max.   :20.700   Max.   :97.00
#>  NA's   :37       NA's   :7
#>      Month            Day
#>  Min.   :5.000   Min.   : 1.0
#>  1st Qu.:6.000   1st Qu.: 8.0
#>  Median :7.000   Median :16.0
#>  Mean   :6.993   Mean   :15.8
#>  3rd Qu.:8.000   3rd Qu.:23.0
#>  Max.   :9.000   Max.   :31.0
#>
```

```
#|
# 2. Select the first three columns (columns 1, 2, and 3)
airquality[, 1:3]
```

```
#>   Ozone Solar.R Wind
#> 1    41     190  7.4
#> 2    36     118  8.0
#> 3    12     149 12.6
#> 4    18     313 11.5
#> 5    NA      NA 14.3
#> 6    28      NA 14.9
```

> **Note**
>
> For clarity and conciseness, we have shortened the output to include only six rows.

```
# 3. Select rows 1 to 3, and columns 1 and 3
airquality[1:3, c(1, 3)]
```

```
#>   Ozone Wind
#> 1    41  7.4
#> 2    36  8.0
#> 3    12 12.6
```

```
# 4. Select rows 1 to 5, and column 1
airquality[1:5, 1]
```

```
#> [1] 41 36 12 18 NA
```

```r
# 5. Select the first row
airquality[1, ]
```

```
#>   Ozone Solar.R Wind Temp Month Day
#> 1    41     190  7.4   67     5   1
```

```r
# 6. Select the first 6 rows
airquality[1:6, ]
```

```
#>   Ozone Solar.R Wind Temp Month Day
#> 1    41     190  7.4   67     5   1
#> 2    36     118  8.0   72     5   2
#> 3    12     149 12.6   74     5   3
#> 4    18     313 11.5   62     5   4
#> 5    NA      NA 14.3   56     5   5
#> 6    28      NA 14.9   66     5   6
```

Return to Exercise 2.3.1

Solution 2.2.3: Matrix Operations

```r
# Sales data (units sold)

sales_data <- c(500, 600, 550, 450, 620, 580, 610, 490, 530, 610, 570, 480)

# Create a matrix

sales_matrix <- matrix(sales_data, nrow = 4, ncol = 3, byrow = TRUE)

colnames(sales_matrix) <- c("Product_A", "Product_B", "Product_C")

rownames(sales_matrix) <- c("Region_1", "Region_2", "Region_3", "Region_4")

sales_matrix
```

```
#>          Product_A Product_B Product_C
#> Region_1       500       600       550
#> Region_2       450       620       580
#> Region_3       610       490       530
#> Region_4       610       570       480
```

```r
# Task 1: Total units sold per product
total_units_per_product <- colSums(sales_matrix)

total_units_per_product
```

```
#> Product_A Product_B Product_C
#>      2170      2280      2140
```

```r
# Task 2: Average units sold for Product_A
average_product_a <- mean(sales_matrix[, "Product_A"])
average_product_a
```

```
#> [1] 542.5
```

```r
# Task 3: Region with highest sales for Product_C
max_sales_product_c <- max(sales_matrix[, "Product_C"])
region_highest_sales <- rownames(sales_matrix)[which(sales_matrix[, "Product_C"] == max_sales
region_highest_sales # Returns the region name
```

```
#> [1] "Region_2"
```

Return to Exercise 2.2.3

Solution Quiz 2.3

Question 1:

Which function would you use to view the structure of a data frame, including its data types and a preview of its contents?

a) `head()`

b) `str()` ✓

c) `summary()`

d) `names()`

Question 2:

How do you access the third row and second column of a data frame `df`?

a) `df[3, 2]` ✓

b) `df[[3, 2]]`

c) `df$3$2`

d) `df(3, 2)`

Question 3:

In a data frame, all columns must contain the same type of data.

a) True

b) False ✓

Question 4:

Which of the following commands would open a spreadsheet-style viewer of the data frame `df` in RStudio?

a) `View(df)` ✓

b) `view(df)`

c) `inspect(df)`

d) `display(df)`

Question 5:

What does the `summary()` function provide when applied to a data frame?

a) Only the first few rows of the data frame.

b) Descriptive statistics for each column. ✓

c) The structure of the data frame including data types.

d) A visual plot of the data.

Answer: B

Question 6:

In a data frame, all columns must be of the same data type.

a) True

b) False ✓

Return to Quiz 2.3

Solution 2.3.2: Data Frame Manipulation

```
# Sample sales transactions
transaction_id <- 1:5

product <- c("Product_A", "Product_B", "Product_C", "Product_A", "Product_B")

quantity <- c(2, 5, 1, 3, 4)

price <- c(19.99, 5.49, 12.89, 19.99, 5.49)

total_amount <- quantity * price

sales_transactions <- data.frame(transaction_id, product, quantity, price, total_amount)

sales_transactions
```

```
#>   transaction_id   product quantity price total_amount
#> 1              1 Product_A        2 19.99        39.98
#> 2              2 Product_B        5  5.49        27.45
#> 3              3 Product_C        1 12.89        12.89
#> 4              4 Product_A        3 19.99        59.97
#> 5              5 Product_B        4  5.49        21.96
```

```
# Task 1: Add 'discounted_price' column
sales_transactions$discounted_price <- sales_transactions$price * 0.9
sales_transactions
```

```
#>   transaction_id   product quantity price total_amount discounted_price
#> 1              1 Product_A        2 19.99        39.98           17.991
#> 2              2 Product_B        5  5.49        27.45            4.941
#> 3              3 Product_C        1 12.89        12.89           11.601
#> 4              4 Product_A        3 19.99        59.97           17.991
#> 5              5 Product_B        4  5.49        21.96            4.941
```

```r
# Task 2: Filter transactions with 'total_amount' > $50
high_value_transactions <- sales_transactions[sales_transactions$total_amount > 50, ]
high_value_transactions
```

```
#>   transaction_id   product quantity price total_amount discounted_price
#> 4              4 Product_A        3 19.99        59.97           17.991
```

```r
# Task 3: Average 'total_amount' for 'Product_B'
product_b_transactions <- sales_transactions[sales_transactions$product == "Product_B", ]
average_total_amount_b <- mean(product_b_transactions$total_amount)
average_total_amount_b
```

```
#> [1] 24.705
```

Return to Exercise 2.3.2

Solution Quiz 2.4

Question 1:

Which function is used to create a list in R?

a) `c()`

b) `list()` ✓

c) `data.frame()`

d) `matrix()`

Question 2:

Given the list:

```r
L <- list(a = 1, b = "text", c = TRUE)
```

how would you access the element `"text"`?

a) `L[2]`

b) `L["b"]`

c) `L$b`

d) Both b) and c) ✓

Question 3:

Using single square brackets `[]` to access elements in a list returns the element itself, not a sublist.

a) True

b) False ✓

Using single `[]` returns a sublist, while double `[[ ]]` returns the element itself.

Question 4:

How can you add a new element named `d` with value `3.14` to the list `L`?

a) `L$d <- 3.14`

b) `L["d"] <- 3.14`

c) `L <- c(L, d = 3.14)`

d) All of the above ✓

Question 5:

What will be the result of `length(L)` if

`L <- list(a = 1, b = "text", c = TRUE, d = 3.14)`?

a) 3

b) 4 ✓

c) 1

d) 0

Return to Quiz 2.4

Solution 2.4.1: Working with Lists

```
# Create the list
product_details <- list(
  product_id = 501,
  name = "Wireless Mouse",
  specifications = list(
    color = "Black",
    battery_life = "12 months",
    connectivity = "Bluetooth"
  ),
  in_stock = TRUE
)

# Access elements
product_details$product_id
```

```
#> [1] 501
```

```
product_details$name
```

```
#> [1] "Wireless Mouse"
```

```
product_details$in_stock
```

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

```
# Access nested list
product_details$specifications$color
```

```
#> [1] "Black"
```

```
product_details$specifications$connectivity
```

```
#> [1] "Bluetooth"
```

Return to Exercise 2.4.1

General Solution Quiz 2

Question 1

Which function is used to create a vector in R?

a) `vector()`

b) `c()` ✓

c) `list()`

d) `data.frame()`

Question 2

Which function is used to create a matrix in R?

a) `array()`

b) `list()`

c) `matrix()` ✓

d) `data.frame()`

Question 3

Which function is used to create an array in R?

a) `list()`

b) `matrix()`

c) `c()`

d) `array()` ✓

Question 4

Which function is used to create a list in R?

a) `list()` ✓

b) `c()`

c) `matrix()`

d) `data.frame()`

Question 5

A matrix in R must contain elements of:

a) Multiple data types (e.g., numeric and character mixed)

b) Only character type

c) Only logical type

d) The same type (all numeric, all logical, etc.) ✓

Question 6

An array in R can be:

a) Only two-dimensional

b) Only one-dimensional

c) Two-dimensional or higher ✓

d) Unlimited in one dimension only

Question 7

A list in R is considered:

a) Two-dimensional

b) One-dimensional ✓

c) Multi-dimensional

d) A type of matrix

Question 8

Which of the following is true about a list?

a) It can only contain numeric data

b) It stores data with rows and columns by default

c) It can store multiple data types in different elements ✓

d) It must be strictly two-dimensional

Question 9

What is the most suitable structure for storing heterogeneous data (e.g., numbers, characters, and even another data frame) in a single R object?

a) Vector

b) Matrix

c) Array

d) List ✓

Question 10

How do we typically check the "size" of a list in R?

a) `nrow()`

b) `length()` ✓

c) `dim()`

d) `ncol()`

Question 11

Which function is used to create a data frame in R?

a) `data.frame()` ✓

b) `array()`

c) `c()`

d) `list()`

Question 12

A data frame in R:

a) Must be strictly numeric

b) Can store different data types in each column ✓

c) Is always one-dimensional

d) Is identical to a matrix

Question 13

If you want to assign dimension names to an array, you should use:

a) `rownames()` only

b) `colnames()` only

c) `dimnames()` ✓

d) `names()`

Question 14

When creating a matrix using:

```
matrix(1:6, nrow = 2, ncol = 3, byrow = TRUE)
```

How are the elements placed?

a) Filled by columns first

b) Filled by rows first ✓

c) Randomly placed

d) Not possible to tell

Question 15

In an array with dimensions `c(2, 3, 4)`, how many elements are there in total?

a) 12

b) 18

c) 24 ✓

d) 36

Return to General Quiz 2

Lab 3: Writing Custom Function

Solution 3.1.1: Temperature Conversion

```
celsius_to_fahrenheit <- function(celsius) {
  fahrenheit <- celsius * 1.8 + 32
  return(fahrenheit)
}

# Testing the function
celsius_to_fahrenheit(100)
```

```
#> [1] 212
```

```
celsius_to_fahrenheit(75)
```

```
#> [1] 167
```

```
celsius_to_fahrenheit(120)
```

```
#> [1] 248
```

Return to Exercise 3.1.1

Solution 3.1.2: Pythagoras Theorem

```r
pythagoras <- function(a, b) {
  c <- sqrt(a^2 + b^2)
  return(c)
}

# Testing the function
pythagoras(3, 4)
```

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

```r
pythagoras(4.1, 2.6)
```

```
#> [1] 4.854894
```

Return to Exercise 3.1.2

Solution 3.1.3: Staff Data Manipulation Using `switch()`

```r
library(tidyverse)
```

```
#> -- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
#> v dplyr     1.1.4     v readr     2.1.5
#> v forcats   1.0.0     v stringr   1.5.1
#> v ggplot2   3.5.1     v tibble    3.2.1
#> v lubridate 1.9.3     v tidyr     1.3.1
#> v purrr     1.0.2
#> -- Conflicts ---------------------------------------- tidyverse_conflicts() --
#> x dplyr::filter() masks stats::filter()
#> x dplyr::lag()    masks stats::lag()
#> i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to be
```

```r
# Sample employee data
staff_data <- data.frame(
  EmployeeID = 1:6,
  Name = c("Alice", "Ebunlomo", "Festus", "Othniel", "Bob", "Testimony"),
  Department = c("HR", "IT", "Finance", "Data Science", "Marketing", "Finance"),
  Salary = c(70000, 80000, 75000, 82000, 73000, 78000)
)
```

```r
data_frame_operation <- function(data, operation) {
  result <- switch(operation,

    # Case 1: Summary of the data frame
    summary = {
      print("Summary of Data Frame:")
      summary(data)
    },

    # Case 2: Add a new column 'Bonus' which is 10% of the Salary
    add_column = {
      data$Bonus <- data$Salary * 0.10
      print("Data Frame after adding 'Bonus' column:")
      data
    },

    # Case 3: Filter employees with Salary > 75,000
    filter = {
      filtered_data <- filter(data, Salary > 75000)
      print("Filtered Data Frame (Salary > 75,000):")
      filtered_data
    },

    # Case 4: Group-wise average salary
    group_stats = {
      group_summary <- data %>%
        group_by(Department) %>%
        summarize(Average_Salary = mean(Salary))
      print("Group-wise Average Salary:")
      group_summary
    },

    # Case 5: Add a new column 'raise_salary' which is 5% of the Salary
    raise_salary = {
      data$Salary <- data$Salary * 1.05
      print("Data Frame after 5% salary increase:")
      data
    },

    # Default case
    {
      print("Invalid operation. Please choose a valid option.")
```

```
      NULL
    }
  )

  return(result)
}

# Testing the new operation
data_frame_operation(staff_data, "raise_salary")
```

```
#> [1] "Data Frame after 5% salary increase:"
```

```
#>   EmployeeID      Name   Department Salary
#> 1          1     Alice           HR  73500
#> 2          2  Ebunlomo           IT  84000
#> 3          3    Festus      Finance  78750
#> 4          4   Othniel Data Science  86100
#> 5          5       Bob    Marketing  76650
#> 6          6 Testimony      Finance  81900
```

Return to Exercise 3.1.3

Solution Quiz 3.1

Question 1:

What is the correct way to define a function in R?

a) `function_name <- function { ... }`

b) `function_name <- function(...) { ... }` ✓

c) `function_name <- function[ ... ] { ... }`

d) `function_name <- function(...) [ ... ]`

Question 2:

A variable defined inside a function is accessible outside the function.

a) True

b) False ✓

Question 3:

Which of the following is NOT a benefit of writing functions?

a) Code Reusability

b) Improved Readability

c) Increased Code Complexity ✓

d) Modular Programming

Return to Quiz 3.1

Lab 4: Managing Packages and Workflows

Solution Quiz 4.1

Question 1:

Imagine that you want to install the shiny package from CRAN. Which command should you use?

a) `install.packages("shiny")` ✓

b) `library("shiny")`

c) `install.packages(shiny)`

d) `require("shiny")`

Question 2:

What must you do after installing a package before you can use it in your current session?

a) Restart R

b) Run `install.packages()` again

c) Load it with `library()` ✓

d) Convert the package into a dataset

Question 3:

If you want to install a package that is not on CRAN (e.g., from GitHub), which additional package would be helpful?

a) `installer`

b) `rio`

c) `devtools` ✓

d) `github_install`

Question 4:

Which function would you use to update all outdated packages in your R environment?

a) `update.packages()` ✓

b) `install.packages()`

c) `library()`

d) `require()`

Question 5:

Which function can be used to check the version of an installed package?

a) `version()`

b) `packageVersion()` ✓

c) `libraryVersion()`

d) `install.packages()`

Return to Exercise 4.1

Solution Quiz 4.2

Question 1:

What is a key advantage of using RStudio Projects?

a) They automatically install packages.

b) They allow you to use absolute paths easily.

c) They set the working directory to the project folder, enabling relative paths. ✓

d) They prevent package updates.

Question 2:

Which file extension identifies an RStudio Project file?

a) `.Rdata`

b) `.Rproj` ✓

c) `.Rmd`

d) `.Rscript`

Question 3:

Why are relative paths preferable in a collaborative environment?

a) They are shorter and easier to type.

b) They change automatically when you move files.

c) They ensure that the code works regardless of the user's file system structure. ✓

d) They are required for Git version control.

Return to Quiz 4.2

Solution Quiz 4.3

Question 1:

Which package is commonly used to read CSV files into R as tibbles?

a) `readxl`

b) `haven`

c) `readr` ✓

d) `writexl`

Question 2:

If you need to import an Excel file, which function would you likely use?

a) `read_csv()`

b) `read_xlsx()` ✓

c) `read_sav()`

d) `read_dta()`

Question 3:

Which package would you use to easily handle a wide variety of data formats without memorising specific functions for each?

a) `rio` ✓

b) `haven`

c) `janitor`

d) `readxl`

Question 4:

After cleaning and analysing your data, which function would you use to write the results to a CSV file?

a) `write_xlsx()`

b) `exporter()`

c) `write_csv()` ✓

d) `import()`

Return to Quiz 4.3

Lab 5: Data Transformation

Solution Quiz 5.1

Question 1:

What is the primary purpose of the pipe operator (`|>` or `%>%`) in R?

a) To run code in parallel.

b) To nest functions inside one another.

c) To pass the output of one function as the input to the next, improving code readability. ✓

d) To automatically clean missing data.

Question 2:

Consider the following R code snippets:

```r
numbers <- c(2, 4, 6)

# Nested function version:
result1 <- round(sqrt(sum(numbers)))

# Pipe operator version:
result2 <- numbers |> sum() |> sqrt() |> round()
```

For a new R learner, is the pipe operator version generally more readable than the nested function version?

a) True ✓

b) False

Question 3:

What is the output of the following R code?

```r
result <- c(5, 10, 15)
result |> mean()
```

a) 10 ✓

b) 15

c) 5

d) 30

Question 4:

Which of the following code snippets correctly uses the pipe operator to apply the `sqrt()` function to the sum of numbers from 1 to 4?

a) `sqrt(sum(1:4))`

b) `1:4 |> sum() |> sqrt()` ✓

c) `sum(1:4) |> sqrt`

d) `1:4 |> sqrt() |> sum()`

Question 5:

What will be the output of the following code?

```
result <- letters
result |> head(3)
```

a) `c("a", "b", "c")` ✓

b) `c("x", "y", "z")`

c) `c("A", "B", "C")`

d) An error is thrown.

Return to Quiz 5.1

Solution Quiz 5.2

Question 1:

Which function would you use in `dplyr` to randomly select a specified number of rows from a dataset?

a) `sample(n = 5)`

b) `slice_sample(n = 5)` ✓

c) `filter_sample()`

d) `mutate_sample()`

Question 2:

To calculate the average `sleep_total` for each `vore` category, which combination of functions is most appropriate?

a) `group_by(vore) |> select(sleep_total) |> summarise(mean(sleep_total))`

b) `select(vore, sleep_total) |> summarise(mean(sleep_total)) |> group_by(vore)`

c) `group_by(vore) |> summarise(avg_sleep = mean(sleep_total, na.rm = TRUE))` ✓

d) `filter(vore) |> mutate(avg_sleep = mean(sleep_total))`

Question 3:

To extract rows with the maximum value of a specified variable, which function is appropriate in `dplyr`?

a) `slice_max()` ✓

b) `slice_min()`

c) `mutate()`

d) `select()`

Question 4:

Which `dplyr` function would you use if you want to create a new column called `weight_ratio` by dividing `bodywt` by `mean_bodywt`?

a) `filter()`

b) `select()`

c) `mutate()` ✓

d) `arrange()`

Question 5:

Suppose you need to identify the top 3 penguins with the highest bill aspect ratio from the `penguins` dataset after calculating it in a new column. Which of the following code snippets is the most concise and appropriate?

a)

```
penguins |>
  mutate(bill_aspect_ratio = bill_length_mm / bill_depth_mm) |>
  arrange(desc(bill_aspect_ratio)) |>
  head(3)
```

b)

```
penguins |>
  mutate(bill_aspect_ratio = bill_length_mm / bill_depth_mm) |>
  slice_max(bill_aspect_ratio, n = 3)
```

c) Both a and b are equally concise and valid. ✓

d) Neither a nor b is valid.

Question 6:

Given the following code, which is the correct equivalent using the pipe operator?

```
result <- arrange(filter(select(msleep, name, sleep_total), sleep_total > 8), sleep_total)
```

a) `msleep |> select(name, sleep_total) |> filter(sleep_total > 8) |> arrange(sleep_total)` ✓

b) `msleep |> filter(sleep_total > 8) |> select(name, sleep_total) |> arrange(sleep_total)`

c) `select(msleep, name, sleep_total) |> filter(sleep_total > 8) |> arrange(sleep_total)`

d) `msleep |> arrange(sleep_total) |> filter(sleep_total > 8) |> select(name, sleep_total)`

Question 7:

Which of the following correctly applies a log transformation to numeric columns **only**?

a)

```
mutate_all(log)
```

b)

```
mutate(across(everything(), log))
```

c)

```
mutate(select(where(is.numeric), log))
```

d) ✓

```
mutate(across(where(is.numeric), log))
```

Question 8:

What does `mutate(across(everything(), as.character))` do?

a) Converts all character columns to numeric.

b) Converts all columns in the dataset to character type. ✓

c) Applies a conditional transformation to numeric columns.

d) Filters out non-character values.

Question 9:

To extract the rows with the minimum value of a specified variable, which `dplyr` function should you use?

a) `slice_min()` ✓

b) `slice_max()`

c) `arrange()`

d) `filter()`

Question 10:

If you want to reorder the rows of `msleep` by `sleep_total` in ascending order and then only show the top 5 rows, which code snippet is correct?

a) `msleep |> arrange(sleep_total) |> head(5)` ✓

b) `msleep |> head(5) |> arrange(sleep_total)`

c) `msleep |> summarise(sleep_total) |> head(5)`

d) `msleep |> select(sleep_total) |> arrange(desc(sleep_total)) |> head(5)`

Return to Quiz 5.2

Solution 5.2.1: Top 5 Carnivorous Animals

```
msleep |>
  filter(vore == "carni") |>
  mutate(sleep_to_weight = sleep_total / bodywt) |>
  select(name, sleep_total, sleep_to_weight) |>
  slice_max(sleep_total, n = 5)
```

```
#> # A tibble: 5 x 3
#>   name                        sleep_total sleep_to_weight
#>   <chr>                             <dbl>           <dbl>
#> 1 Thick-tailed opposum               19.4         52.4
#> 2 Long-nosed armadillo               17.4          4.97
#> 3 Tiger                              15.8          0.0972
#> 4 Northern grasshopper mouse         14.5        518.
#> 5 Lion                               13.5          0.0836
```

Return to Exercise 5.2.1

Solution Quiz 5.3

Question 1:

Which function in R checks if there are any missing values in an object?

a) `is.na()`

b) `anyNA()` ✓

c) `complete.cases()`

d) `na.omit()`

Question 2:

Which approach removes any rows containing `NA` values?

a) `na.omit()` ✓

b) `replace_na()`

c) `complete.cases()`

d) `anyNA()`

Question 3:

If you decide to impute missing values in a column using the median, what is one potential advantage of using the median rather than the mean?

a) The median is always easier to compute.

b) The median is more affected by outliers than the mean.

c) The median is less influenced by extreme values and may provide a more robust estimate. ✓

d) The median will always be exactly halfway between the min and max values.

Question 4:

How would you replace all `NA` values in character columns with `"Unknown"`?

a) ✓

```r
mutate(across(where(is.character), ~ replace_na(., "Unknown")))
```

b)

```r
mutate_all(~ replace_na(., "Unknown"))
```

c)

```r
mutate(across(where(is.character), na.omit))
```

d)

```r
mutate(across(where(is.character), replace(. == NA, "Unknown")))
```

Question 5:

What does the `anyNA()` function return?

a) The number of missing values in an object.

b) `TRUE` if there are any missing values in the object; otherwise, `FALSE`. ✓

c) A logical vector of missing values in each row.

d) A subset of the data frame without missing values.

Question 6:

You want to create a new column in a data frame that flags rows with missing values as `TRUE`. Which code achieves this?

a) `df$new_col <- !complete.cases(df)` ✓

b) `df$new_col <- complete.cases(df)`

c) `df$new_col <- anyNA(df)`

d) `df$new_col <- is.na(df)`

Question 7:

Before removing rows with missing values, what is an important consideration?

a) Whether the missing values are randomly distributed across the data. ✓

b) Whether the dataset is stored in a data frame.

c) Whether missing values exist in every column.

d) Whether the missing values are encoded as `NA`.

Question 8:

Why should the proportion of missing data in a row or column be considered before removing it?

a) Removing rows or columns with minimal missing values may lead to excessive data loss. ✓

b) Columns with missing values cannot be visualized.
c) Rows with missing values are always irrelevant.
d) Rows with missing values should never be analysed.

Question 9:

If a dataset has 50% missing values in a column, what is a common approach to handle this situation?

a) Replace missing values with the column mean.

b) Remove the column entirely. ✓

c) Replace missing values with zeros.

d) Leave the missing values as they are.

Question 10:

What does the following Tidyverse-style code do?

```
library(dplyr)

airquality_data <- airquality_data %>%
  mutate(Ozone = if_else(is.na(Ozone), mean(Ozone, na.rm = TRUE), Ozone))
```

a) Removes rows where `Ozone` is missing.

b) Replaces missing values in `Ozone` with the mean of the column. ✓

c) Flags rows where `Ozone` is missing.

d) Deletes the `Ozone` column if it has missing values.

Return to Quiz 5.3

Solution 5.3.1: Missing Data Analysis Report for the Television Company Dataset

In this report, we explore several methods for dealing with missing data in a television company dataset. First, we import the data, then apply four different approaches to address any missing values. After evaluating the results, we conclude with a recommendation on the best method to use.

Data Import & Initial Inspection

We begin by loading the dataset and inspecting its structure, summary statistics, and missing values.

```
library(tidyverse)

# Import the dataset
tv_data <- read_csv("r-data/data-tv-company.csv")

# Inspect the data structure and summary statistics
glimpse(tv_data)
```

```
#> Rows: 462
#> Columns: 9
#> $ regard  <dbl> 8, 5, 5, 4, 6, 6, 4, 5, 7, NA, 6, 5, 5, 3, 4, 5, 5, NA, 5, 7, ~
#> $ gender  <chr> "Male", "Female", "Female", "Female", "Female", "Female", "Fem~
#> $ views   <dbl> 458, 460, 457, 437, 438, 456, NA, 448, 450, 459, 442, 443, 451~
#> $ online  <dbl> 821, 810, 824, 803, 791, 813, 797, 813, 827, 820, 802, 812, 81~
#> $ library <dbl> 104, 99, NA, NA, 84, 104, NA, 94, 100, 103, 101, 90, 99, 94, 9~
#> $ Show1   <dbl> 74, 70, 72, 74, 74, 73, 71, 73, 79, 77, 70, 74, 73, 72, 71, 78~
#> $ Show2   <dbl> 74, 74, 72, 74, 70, 73, 71, 72, 76, 77, 69, 70, 72, 73, 70, 76~
#> $ Show3   <dbl> 64, 58, 59, 58, 57, 61, 58, 58, 62, 60, 62, 59, 59, 58, 58, 60~
#> $ Show4   <dbl> 39, 44, 34, 39, 34, 40, 40, 31, 44, 35, 37, 33, 36, 35, 37, 37~
```

```
summary(tv_data)
```

```
#>      regard         gender              views           online
#>  Min.   :2.000   Length:462         Min.   :430.0   Min.   :787
#>  1st Qu.:5.000   Class :character   1st Qu.:445.0   1st Qu.:809
#>  Median :5.000   Mode  :character   Median :450.0   Median :815
#>  Mean   :5.454                      Mean   :449.9   Mean   :815
#>  3rd Qu.:6.000                      3rd Qu.:456.0   3rd Qu.:821
#>  Max.   :9.000                      Max.   :474.0   Max.   :843
#>  NA's   :30                         NA's   :22
#>     library           Show1           Show2           Show3
#>  Min.   : 84.00   Min.   :66.00   Min.   :64.00   Min.   :55.00
#>  1st Qu.: 95.00   1st Qu.:72.00   1st Qu.:71.00   1st Qu.:59.00
#>  Median : 98.00   Median :73.00   Median :72.00   Median :60.00
#>  Mean   : 98.14   Mean   :73.08   Mean   :72.16   Mean   :59.87
#>  3rd Qu.:101.00   3rd Qu.:75.00   3rd Qu.:74.00   3rd Qu.:61.00
```

```
#>  Max.   :115.00   Max.   :79.00   Max.   :78.00   Max.   :66.00
#>  NA's   :68
#>      Show4
#>  Min.   :21.00
#>  1st Qu.:34.00
#>  Median :37.00
#>  Mean   :37.42
#>  3rd Qu.:41.00
#>  Max.   :50.00
#>
```

```r
# Count missing values per row
count_missing_rows <- function(data) {
  sum(apply(data, MARGIN = 1, function(x) any(is.na(x))))
}

count_missing_rows(tv_data)
```

```
#> [1] 112
```

> **Tip**
>
> `apply(data, MARGIN = 1, function(x) any(is.na(x)))`:
>
> - `MARGIN = 1`: Instructs `apply()` to iterate over rows of the data frame (if set to 2, it would iterate over columns).
> - `function(x) any(is.na(x))`: For each row, checks if any element is missing (i.e., is `NA`), returning `TRUE` if so.
>
> `sum(...)`:
>
> - Sums the logical vector produced by `apply()`, where each `TRUE` is counted as 1, thereby giving the total number of rows with at least one missing value.

```r
# Count missing values per column using inspectdf
tv_data %>%
  inspectdf::inspect_na()
```

```
#> # A tibble: 9 x 3
#>   col_name   cnt  pcnt
```

```
#>   <chr>    <int> <dbl>
#> 1 library     68 14.7
#> 2 regard      30  6.49
#> 3 views       22  4.76
#> 4 gender       0  0
#> 5 online       0  0
#> 6 Show1        0  0
#> 7 Show2        0  0
#> 8 Show3        0  0
#> 9 Show4        0  0
```

Strategies for Dealing with Missing Data

Below, we demonstrate four different methods to handle missing data in the dataset.

Method 1: Complete Case Analysis

Remove all rows with any missing values.

```
tv_data_complete <- tv_data %>% drop_na()
```

> Tip
>
> Alternatively, you can use `na.omit()` to remove rows with missing values:
>
> ```
> tv_data %>% na.omit()
> ```
>
> ```
> #> # A tibble: 350 x 9
> #>    regard gender views online library Show1 Show2 Show3 Show4
> #>     <dbl> <chr>  <dbl>  <dbl>   <dbl> <dbl> <dbl> <dbl> <dbl>
> #>  1      8 Male     458    821     104    74    74    64    39
> #>  2      5 Female   460    810      99    70    74    58    44
> #>  3      6 Female   438    791      84    74    70    57    34
> #>  4      6 Female   456    813     104    73    73    61    40
> #>  5      5 Male     448    813      94    73    72    58    31
> #>  6      7 Female   450    827     100    79    76    62    44
> #>  7      6 Female   442    802     101    70    69    62    37
> #>  8      5 Female   443    812      90    74    70    59    33
> #>  9      5 Female   451    815      99    73    72    59    36
> #> 10      3 Male     440    810      94    72    73    58    35
> #> # i 340 more rows
> ```

Method 2: Numeric Imputation with Column Means

Replace missing values in all numeric columns with the respective column mean.

```
tv_data_mean_imputed <- tv_data %>%
  bulkreadr::fill_missing_values(method = "mean")
```

Method 3: Targeted Replacement using tidyr::replace_na() with Medians

For numeric columns with missing values (`regard`, `views`, and `library`), replace NAs with the column median.

```
tv_data_tidyr <- tv_data %>%
  replace_na(list(
    regard  = median(tv_data$regard, na.rm = TRUE),
    views   = median(tv_data$views, na.rm = TRUE),
    library = median(tv_data$library, na.rm = TRUE)
  ))
```

> **Tip**
>
> Alternatively, you can use the `selected_variables` argument in the `bulkreadr::fill_missing_values()` function with `method = "median"` to impute these missing values:
>
> ```
> tv_data_mean_imputed <- tv_data %>%
>   bulkreadr::fill_missing_values(selected_variables = c("regard", "views", "library"), meth
> ```

Evaluation and Selection

Based on the outputs from the various methods, here is a summary and evaluation of each approach:

Original Data:

- **Dimensions:** 462 rows × 9 columns.
- **Missing Values:**
  - `regard`: 30 missing

- `views`: 22 missing
- `library`: 68 missing

Complete Case Analysis:

- **Dimensions:** 350 rows × 9 columns.
- **Summary:** All rows with any missing values were removed, resulting in a loss of about 24% of the data.
- **Consideration:** Although this method produces a dataset free of missing values, it may discard valuable information and reduce the statistical power of subsequent analyses.

Numeric Imputation (Replacing Missing Numeric Values with Column Means):

- **Dimensions:** 462 rows × 9 columns.
- **Summary:** Missing numeric values in columns such as *regard*, *views*, and *library* have been replaced by their respective column means. The summary no longer shows missing value counts for these columns.
- **Consideration:** This method preserves all observations but may smooth out natural variability, potentially impacting the distribution and variance in the data.

Targeted Replacement using tidyr::replace_na():

- **Dimensions:** 462 rows × 9 columns.
- **Summary:** Missing values in numeric columns (`regard`, `views`, `library`) have been replaced by their respective medians, resulting in a dataset identical in dimensions to the original but with imputed values.
- **Consideration:** This method retains the full dataset and provides robust imputation for numeric data, preserving the distribution better than mean imputation.

Conclusion

In summary, after comparing the different approaches:

- **Complete Case Analysis** yields a clean dataset but reduces the sample size.
- **Numeric Imputation** retains all data by substituting missing values with means, although it may reduce variability.
- **Targeted Replacement (using medians)** preserves the full dataset and is robust to outliers.

Recommendation:

Method 3 (Targeted Replacement using Medians) is preferred because it maintains the full dataset while providing robust imputation for missing numeric values.

Return to Exercise 5.3.1

Lab 6: Tidy Data and Joins

Solution Quiz 6.1

Question 1:

Consider the following data frame:

```
sales_data_wide <- data.frame(
  Month = c("Oct", "Nov", "Dec"),
  North = c(180, 190, 200),
  East  = c(177, 183, 190),
  South = c(150, 140, 160),
  West  = c(200, 220, 210)
)
```

Which function would you use to convert this wide-format dataset into a long-format dataset?

a) `pivot_long()`

b) `pivot_wider()`

c) `separate()`

d) `pivot_longer()` ✓

Question 2:

In the `pivot_longer()` function, if you want the original column names ("North", "East", "South", "West") to appear in a new column called "Region", which argument would you use?

a) `cols`

b) `names_to` ✓

c) `values_to`

d) `names_prefix`

Question 3:

Given the same data frame, which argument in `pivot_longer()` specifies the name of the new column that stores the sales figures?

a) `names_to`

b) `values_to` ✓

c) `cols`

d) `values_drop_na`

Question 4:

What is the primary purpose of using `pivot_wider()`?

a) To convert long-format data into wide format ✓

b) To combine two data frames

c) To split a column into multiple columns

d) To remove missing values

Question 5:

If you apply `pivot_longer()` on `sales_data_wide` without specifying `cols`, what is likely to happen?

a) All columns will be pivoted, including the identifier column "Month", leading to an undesired result. ✓

b) Only numeric columns will be pivoted.

c) The function will automatically ignore non-numeric columns.

d) An error will be thrown immediately.

Question 6:

Which package provides the functions `pivot_longer()` and `pivot_wider()`?

a) dplyr

b) tidyr ✓

c) ggplot2

d) readr

Question 7:

The functions `pivot_longer()` and `pivot_wider()` are inverses of each other, allowing you to switch between wide and long formats easily.

a) True ✓

b) False

Question 8:

In the following code snippet, what is the role of the `cols = c(North, East, South, West)` argument?

```
sales_data_long <- sales_data_wide |>
  pivot_longer(
    cols = c(North, East, South, West),
    names_to = "Region",
    values_to = "Sales"
  )
```

a) It tells `pivot_longer()` which columns to keep as they are.

b) It specifies the columns to be pivoted from wide to long format. ✓

c) It defines the new column names for the output.

d) It removes missing values from these columns.

Question 9:

After reshaping the data to long format, which of the following is a potential advantage?

a) Easier to merge with other datasets

b) Simplified time series analysis and visualisation ✓

c) Increased redundancy in the dataset

d) Reduced number of observations

Question 10:

Which of the following best describes tidy data?

a) Each variable forms a column and each observation a row ✓

b) Data is merged from multiple sources

c) Data is automatically plotted

d) Missing values are always removed

Return to Quiz 6.1

Solution 6.1.1: Tidying the Pew Religion and Income Survey Data

In this solution, we tidy the `religion_income` dataset from the Pew Research Trust's 2014 survey. The dataset includes one column for `religion` and multiple columns for various income ranges (e.g., `<$10k`, `$10-20k`, `$20-30k`, etc.), each indicating the number of respondents who fall within that bracket. Our goals are:

1. **Import and Inspect** the data.
2. **Reshape** the dataset from wide to long format, gathering all income range columns into two new variables: `income_range` and `respondents`.
3. **Create a Summary** that shows the total number of respondents for each income range, sorted in a logical order.
4. **Identify** which religious affiliation has the highest number of respondents in the top income bracket (`>150k`).
5. **Visualise** the distribution of respondents by income range.

Importing and Inspecting the Data

We begin by loading the tidyverse package and importing the dataset from the `r-data` directory. We then use `glimpse()` to verify that the file has loaded correctly and to explore its structure.

```
library(tidyverse)

# Import the dataset
relig_income <- read_csv("r-data/religion-income.csv")
```

```
#> Rows: 18 Columns: 11
#> -- Column specification --------------------------------------------------------
#> Delimiter: ","
#> chr  (1): religion
#> dbl (10): <$10k, $10-20k, $20-30k, $30-40k, $40-50k, $50-75k, $75-100k, $100...
#>
#> i Use `spec()` to retrieve the full column specification for this data.
#> i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
# Inspect the data structure
glimpse(relig_income)
```

```
#> Rows: 18
#> Columns: 11
#> $ religion             <chr> "Agnostic", "Atheist", "Buddhist", "Catholic", "D~
#> $ `<$10k`              <dbl> 27, 12, 27, 418, 15, 575, 1, 228, 20, 19, 289, 29~
#> $ `$10-20k`            <dbl> 34, 27, 21, 617, 14, 869, 9, 244, 27, 19, 495, 40~
#> $ `$20-30k`            <dbl> 60, 37, 30, 732, 15, 1064, 7, 236, 24, 25, 619, 4~
#> $ `$30-40k`            <dbl> 81, 52, 34, 670, 11, 982, 9, 238, 24, 25, 655, 51~
#> $ `$40-50k`            <dbl> 76, 35, 33, 638, 10, 881, 11, 197, 21, 30, 651, 5~
#> $ `$50-75k`            <dbl> 137, 70, 58, 1116, 35, 1486, 34, 223, 30, 95, 110~
#> $ `$75-100k`           <dbl> 122, 73, 62, 949, 21, 949, 47, 131, 15, 69, 939, ~
#> $ `$100-150k`          <dbl> 109, 59, 39, 792, 17, 723, 48, 81, 11, 87, 753, 4~
#> $ `>150k`              <dbl> 84, 74, 53, 633, 18, 414, 54, 78, 6, 151, 634, 42~
#> $ `Don't know/refused` <dbl> 96, 76, 54, 1489, 116, 1529, 37, 339, 37, 162, 13~
```

The religion income data is arranged in a compact or wide format. Each row represents a religious affiliation, and each income range column shows the number of respondents in that bracket.

Tidying the Data

We transform the data from wide to long format using `pivot_longer()`, gathering all columns except `religion` into two new columns: `income_range` (for the bracket names) and `respondents` (for the counts).

```
relig_income_long <- relig_income %>%
  pivot_longer(
    cols = -religion, # All columns except 'religion'
    names_to = "income_range", # New column for the original income range names
    values_to = "respondents" # New column for the corresponding counts
  )

# Inspect the tidied data
glimpse(relig_income_long)
```

```
#> Rows: 180
#> Columns: 3
#> $ religion     <chr> "Agnostic", "Agnostic", "Agnostic", "Agnostic", "Agnostic~
#> $ income_range <chr> "<$10k", "$10-20k", "$20-30k", "$30-40k", "$40-50k", "$50~
#> $ respondents  <dbl> 27, 34, 60, 81, 76, 137, 122, 109, 84, 96, 12, 27, 37, 52~
```

Each row now represents a unique combination of religion and income bracket, along with the corresponding number of respondents.

Creating a Summary Table

We group the tidied data by `income_range` and sum the total respondents. To achieve a logical order (lowest to highest income), we define a custom factor level, then arrange accordingly.

```
income_levels <- c(
  "<$10k",
  "$10-20k",
  "$20-30k",
  "$30-40k",
  "$40-50k",
  "$50-75k",
  "$75-100k",
  "$100-150k",
  ">150k",
  "Don't know/refused"
```

```
)

income_summary <- relig_income_long %>%
  mutate(income_range = factor(income_range, levels = income_levels)) %>%
  group_by(income_range) %>%
  summarise(total_respondents = sum(respondents, na.rm = TRUE)) %>%
  ungroup()

income_summary
```

```
#> # A tibble: 10 x 2
#>    income_range       total_respondents
#>    <fct>                          <dbl>
#>  1 <$10k                           1930
#>  2 $10-20k                         2781
#>  3 $20-30k                         3357
#>  4 $30-40k                         3302
#>  5 $40-50k                         3085
#>  6 $50-75k                         5185
#>  7 $75-100k                        3990
#>  8 $100-150k                       3197
#>  9 >150k                           2608
#> 10 Don't know/refused              6121
```

The data now shows the total number of respondents in each bracket, sorted from `<$10k` to `>150k`, with "Don't know/refused" at the end.

Identifying Which Religion Has the Largest Number of Respondents in the `>150k` Bracket

We can now focus on the `>150k` bracket to see which religion leads in this top income category.

```
top_bracket <- relig_income_long %>%
  filter(income_range == ">150k") %>%
  group_by(religion) %>%
  summarise(total_in_top_bracket = sum(respondents, na.rm = TRUE)) %>%
  arrange(desc(total_in_top_bracket))

top_bracket
```

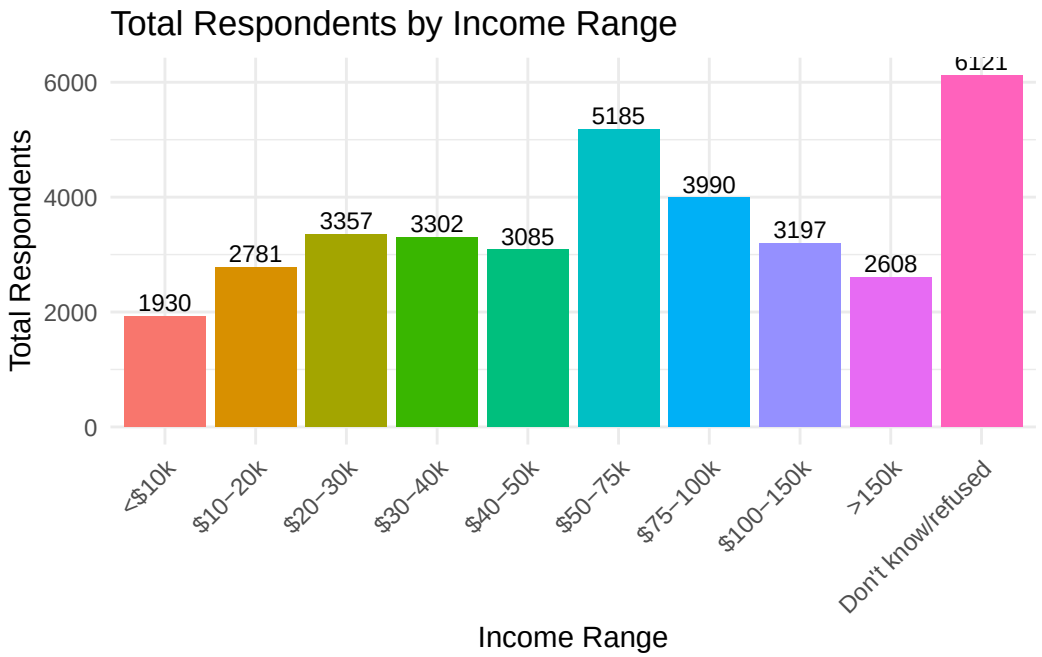
```
#> # A tibble: 18 x 2
#>    religion                total_in_top_bracket
#>    <chr>                                  <dbl>
#>  1 Mainline Prot                            634
#>  2 Catholic                                 633
#>  3 Evangelical Prot                         414
#>  4 Unaffiliated                             258
#>  5 Jewish                                   151
#>  6 Agnostic                                  84
#>  7 Historically Black Prot                   78
#>  8 Atheist                                   74
#>  9 Hindu                                     54
#> 10 Buddhist                                  53
#> 11 Orthodox                                  46
#> 12 Mormon                                    42
#> 13 Other Faiths                              41
#> 14 Don't know/refused                        18
#> 15 Other Christian                           12
#> 16 Jehovah's Witness                          6
#> 17 Muslim                                     6
#> 18 Other World Religions                      4
```

We see that **Mainline Protestant** affiliates have the greatest number of respondents in the `>150k` bracket (634), closely followed by Catholics (633).

Visualising the Distribution of Respondents by Income Range

Finally, we create a bar chart with numeric labels on each bar, making it easy to compare the total respondents across income brackets.

```
income_summary |>
  ggplot(aes(x = income_range, y = total_respondents, fill = income_range)) +
  geom_col(show.legend = FALSE) +
  geom_text(aes(label = total_respondents), vjust = -0.3, size = 3) +
  labs(
    title = "Total Respondents by Income Range",
    x = "Income Range",
    y = "Total Respondents"
  ) +
  theme_minimal() +
  theme(axis.text.x = element_text(angle = 45, hjust = 1))
```



The bar chart shows each income bracket on the x-axis, with its total respondents on the y-axis. Each bar is labelled with the corresponding numeric value, offering a clear comparison. "Don't know/refused" emerges as the largest category (6121), followed by `$50-75k` (5185).

Return to Exercise 6.1.1

Solution Quiz 6.2

Question 1:

Given the following tibble:

```
tb_cases <- tibble(
  country = c("Brazil", "Brazil", "China", "China"),
  year = c(1999, 2000, 1999, 2000),
  rate = c("37737/172006362", "80488/174504898", "212258/1272915272", "213766/1280428583")
)
```

Which function would you use to split the `"rate"` column into two separate columns for cases and population?

a) `separate()` ✓

b) `unite()`

c) `pivot_longer()`

d) `pivot_wider()`

Question 2:

Which argument in `separate()` allows automatic conversion of new columns to appropriate data types?

a) `remove`

b) `auto`

c) `convert` ✓

d) `into`

Question 3:

Which function would you use to merge two columns into one, for example, combining separate "century" and "year" columns?

a) `separate()`

b) `unite()` ✓

c) `pivot_longer()`

d) `pivot_wider()`

Question 4:

In the `separate()` function, what does the `sep` argument define?

a) The new column names

b) The delimiter at which to split the column ✓

c) The data frame to be merged

d) The columns to remove

Question 5:

Consider the following data frame:

```r
tb_cases <- tibble(
  country = c("Afghanistan", "Brazil", "China"),
  century = c("19", "19", "19"),
  year = c("99", "99", "99")
)
```

Which code correctly combines "century" and "year" into a single column "year" without any separator?

a) `tb_cases |> unite(year, century, year, sep = "")` ✓

b) `tb_cases |> separate(year, into = c("century", "year"), sep = "")`

c) `tb_cases |> unite(year, century, year, sep = "_")`

d) `tb_cases |> pivot_longer(cols = c(century, year))`

Question 6:

When using `separate()`, how can you retain the original column after splitting it?

a) Set `remove = FALSE` ✓

b) Set `convert = TRUE`

c) Use `unite()` instead

d) Omit the `sep` argument

Question 7:

Which variant of `separate()` would you use to split a column at fixed character positions?

a) `separate_wider_delim()`

b) `separate_wider_regex()`

c) `separate_wider_position()` ✓

d) `separate()`

Question 8:

By default, the `unite()` function removes the original columns after combining them.

a) True ✓

b) False

Question 9:

What is the main benefit of using `separate()` on a column that combines multiple data points (e.g. "745/19987071")?

a) It facilitates the conversion of string data into numeric data automatically.

b) It simplifies further analysis by splitting combined information into distinct, analysable components. ✓

c) It merges the data with another dataset.

d) It increases data redundancy.

Question 10:

Which argument in `unite()` determines the character inserted between values when combining columns?

a) `separator`

b) `sep` ✓

c) `col`

d) `delimiter`

Return to Quiz 6.2

Solution 6.2.1: Transforming the Television Company Dataset

In this solution, we will demonstrate how to clean and transform the `television-company-data.csv` dataset. Our primary goal is to split the combined `Shows` column into four separate columns (one for each show), calculate the average score across these shows, and then analyse these averages by gender.

Importing and Inspecting the Data

First, we import the dataset using `read_csv()` and inspect its structure using `glimpse()`. This helps us understand the data and verify that the file has been loaded correctly.

```r
library(tidyverse)

# Import the dataset from the r-data directory
tv_data <- read_csv("r-data/television-company-data.csv")

# Inspect the data structure

tv_data
```

```
#> # A tibble: 462 x 6
#>    regard gender views online library Shows
#>     <dbl> <chr>  <dbl>  <dbl>   <dbl> <chr>
#>  1      8 Male     458    821     104 74,  74, 64, 39
#>  2      5 Female   460    810      99 70,  74, 58, 44
#>  3      5 Female   457    824      NA 72,  72, 59, 34
#>  4      4 Female   437    803      NA 74,  74, 58, 39
#>  5      6 Female   438    791      84 74,  70, 57, 34
#>  6      6 Female   456    813     104 73,  73, 61, 40
#>  7      4 Female    NA    797      NA 71,  71, 58, 40
#>  8      5 Male     448    813      94 73,  72, 58, 31
#>  9      7 Female   450    827     100 79,  76, 62, 44
#> 10     NA Female   459    820     103 77,  77, 60, 35
#> # i 452 more rows
```

```r
glimpse(tv_data)
```

```
#> Rows: 462
#> Columns: 6
#> $ regard  <dbl> 8, 5, 5, 4, 6, 6, 4, 5, 7, NA, 6, 5, 5, 3, 4, 5, 5, NA, 5, 7, ~
#> $ gender  <chr> "Male", "Female", "Female", "Female", "Female", "Female", "Fem~
#> $ views   <dbl> 458, 460, 457, 437, 438, 456, NA, 448, 450, 459, 442, 443, 451~
#> $ online  <dbl> 821, 810, 824, 803, 791, 813, 797, 813, 827, 820, 802, 812, 81~
#> $ library <dbl> 104, 99, NA, NA, 84, 104, NA, 94, 100, 103, 101, 90, 99, 94, 9~
#> $ Shows   <chr> "74,  74, 64, 39", "70,  74, 58, 44", "72,  72, 59, 34", "74, ~
```

The television company data contains 462 rows and 10 columns, including variables for viewer regard, gender, number of views, online interactions, library usage, and four show scores.

Splitting the 'Shows' Column

The `Shows` column contains scores for four different shows, separated by commas. We use the `separate()` function to split this column into four new columns named `Show1`, `Show2`, `Show3`, and `Show4`. The argument `convert = TRUE` automatically converts these new columns to numeric values, ensuring they are ready for analysis.

```
tv_data <- tv_data %>%
  separate(Shows,
    into = c("Show1", "Show2", "Show3", "Show4"),
    sep = ",",
    convert = TRUE
  )

# Check the transformed data
glimpse(tv_data)
```

```
#> Rows: 462
#> Columns: 9
#> $ regard  <dbl> 8, 5, 5, 4, 6, 6, 4, 5, 7, NA, 6, 5, 5, 3, 4, 5, 5, NA, 5, 7, ~
#> $ gender  <chr> "Male", "Female", "Female", "Female", "Female", "Female", "Fem~
#> $ views   <dbl> 458, 460, 457, 437, 438, 456, NA, 448, 450, 459, 442, 443, 451~
#> $ online  <dbl> 821, 810, 824, 803, 791, 813, 797, 813, 827, 820, 802, 812, 81~
#> $ library <dbl> 104, 99, NA, NA, 84, 104, NA, 94, 100, 103, 101, 90, 99, 94, 9~
#> $ Show1   <int> 74, 70, 72, 74, 74, 73, 71, 73, 79, 77, 70, 74, 73, 72, 71, 78~
#> $ Show2   <int> 74, 74, 72, 74, 70, 73, 71, 72, 76, 77, 69, 70, 72, 73, 70, 76~
#> $ Show3   <int> 64, 58, 59, 58, 57, 61, 58, 58, 62, 60, 62, 59, 59, 58, 58, 60~
#> $ Show4   <int> 39, 44, 34, 39, 34, 40, 40, 31, 44, 35, 37, 33, 36, 35, 37, 37~
```

We now have separate columns (`Show1`, `Show2`, `Show3`, `Show4`) for each show's score. The dataset still has 462 rows, but now includes 9 columns: the original variables plus the newly created show score columns and excluding variable `Shows`.

Calculating the Mean Show Score

Next, we create a new variable, `mean_show`, which represents the average score across the four shows. In this step, we use `rowwise()` along with `mutate()` to calculate the mean for each observation. The `na.rm = TRUE` argument ensures that any missing values are ignored during the calculation. After computing the mean, we use `ungroup()` to remove the row-wise grouping.

```r
tv_data <- tv_data %>%
  rowwise() %>%
  mutate(mean_show = mean(c(Show1, Show2, Show3, Show4), na.rm = TRUE)) %>%
  ungroup()

# updated dataset with the new variable
tv_data
```

```
#> # A tibble: 462 x 10
#>    regard gender views online library Show1 Show2 Show3 Show4 mean_show
#>     <dbl> <chr>  <dbl>  <dbl>   <dbl> <int> <int> <int> <int>     <dbl>
#>  1      8 Male     458    821     104    74    74    64    39      62.8
#>  2      5 Female   460    810      99    70    74    58    44      61.5
#>  3      5 Female   457    824      NA    72    72    59    34      59.2
#>  4      4 Female   437    803      NA    74    74    58    39      61.2
#>  5      6 Female   438    791      84    74    70    57    34      58.8
#>  6      6 Female   456    813     104    73    73    61    40      61.8
#>  7      4 Female    NA    797      NA    71    71    58    40      60
#>  8      5 Male     448    813      94    73    72    58    31      58.5
#>  9      7 Female   450    827     100    79    76    62    44      65.2
#> 10     NA Female   459    820     103    77    77    60    35      62.2
#> # i 452 more rows
```

The dataset now includes an additional column, `mean_show`, which holds each viewer's average score across the four shows.

Analysing the Data by Gender

To explore how viewer ratings differ by gender, we group the data by the `gender` variable and calculate the average `mean_show` for each group using `group_by()` and `summarise()`. We also count the number of observations per group.

```r
gender_summary <- tv_data %>%
  group_by(gender) %>%
  summarise(
    mean_of_mean_show = mean(mean_show, na.rm = TRUE),
    count = n()
  )

# Display the summary
gender_summary
```

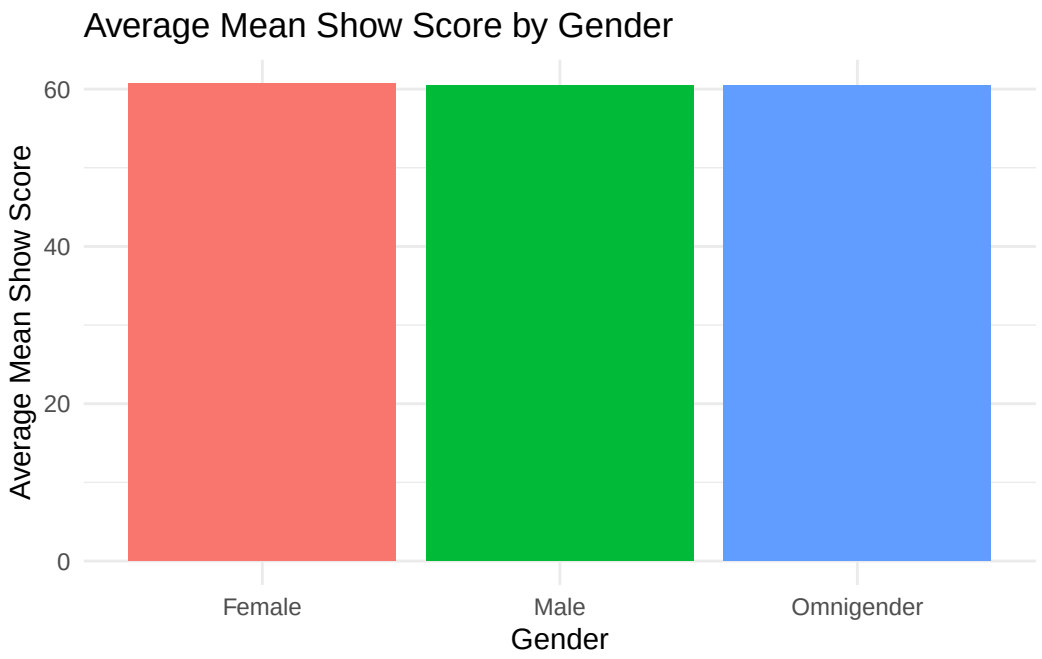
```
#> # A tibble: 3 x 3
#>   gender     mean_of_mean_show count
#>   <chr>                  <dbl> <int>
#> 1 Female                  60.7   304
#> 2 Male                    60.5   154
#> 3 Omnigender              60.5     4
```

We observe that female viewers have a slightly higher average show score (approximately 60.7), while male and omnigender viewers both average about 60.5. The female group is the largest (304 viewers), whereas the omnigender group has only 4 viewers.

Visualising the Results

Finally, we create a bar plot using ggplot2 to visualise the average mean show score by gender. This visualisation helps to clearly compare the scores across different genders.

```
gender_summary |> ggplot(aes(x = gender, y = mean_of_mean_show, fill = gender)) +
  geom_col(show.legend = FALSE) +
  labs(
    title = "Average Mean Show Score by Gender",
    x = "Gender",
    y = "Average Mean Show Score"
  ) +
  theme_minimal()
```



The bar chart confirms that females have a marginally higher average mean show score than males and omnigender viewers, though the difference is small. These findings suggest that overall, viewers' show scores are relatively consistent across genders, with only minor variations.

Return to Exercise 6.2.1

Solution Quiz 6.3

Question 1:

Given the following data frames:

```
df1 <- data.frame(id = 1:4, name = c("Ezekiel", "Bob", "Samuel", "Diana"))

df2 <- data.frame(id = c(2, 3, 5), score = c(85, 90, 88))
```

Which join would return only the rows with matching `id` values in both data frames?

a) `left_join()`

b) `right_join()`

c) `inner_join()` ✓

d) `full_join()`

Question 2:

Using the same data frames, which join function retains all rows from `df1` and fills unmatched rows with `NA`?

a) `left_join()` ✓

b) `inner_join()`

c) `right_join()`

d) `full_join()`

Question 3:

Which join function ensures that all rows from `df2` are preserved, regardless of matches in `df1`?

a) `left_join()`

b) `inner_join()`

c) `full_join()`

d) `right_join()` ✓

Question 4:

What does a full join return when applied to `df1` and `df2`?

a) Only matching rows

b) All rows from both data frames, with `NA` for unmatched entries ✓

c) Only rows from `df1`

d) Only rows from `df2`

Question 5:

In a join operation, what is the purpose of the `by` argument?

a) It specifies the common column(s) used to match rows between the data frames ✓

b) It orders the data frames

c) It selects which rows to retain

d) It converts keys to numeric values

Question 6:

If `df1` contains duplicate values in the key column, what is a likely outcome of an inner join with `df2`?

a) The joined data frame may contain more rows than either original data frame due to duplicate matches. ✓

b) The join will remove all duplicates automatically.

c) The function will return an error.

d) The duplicate rows will be merged into a single row.

Question 7:

An inner join returns all rows from both data frames, regardless of whether there is a match.

a) True

b) False ✓

Question 8:

Consider the following alternative key columns:

```
df1 <- data.frame(studentID = 1:4, name = c("Alice", "Bob", "Charlie", "Diana"))

df2 <- data.frame(id = c(2, 3, 5), score = c(85, 90, 88))
```

How can you join these two data frames when the key column names differ?

a) Rename one column before joining.

b) Use `by = c("studentID" = "id")` in the join function. ✓

c) Use an inner join without specifying keys.

d) Convert the keys to factors.

Question 9:

What is a 'foreign key' in the context of joining datasets?

a) A column in one table that uniquely identifies each row.

b) A column in one table that refers to the primary key in another table. ✓

c) A column that has been split into multiple parts.

d) A column that is combined using `unite()`.

Question 10:

Which join function would be most appropriate if you want a complete union of two datasets, preserving all rows from both?

a) `full_join()` ✓

b) `inner_join()`

c) `left_join()`

d) `right_join()`

Return to Quiz 6.3

Solution 6.3.1: Relational Analysis with the NYC Flights 2013 Dataset

In this solution, we explore relational data analysis using the `nycflights13` dataset. We will:

1. Load and inspect the `flights` and `planes` tables.
2. Perform various join operations (`inner_join`, `left_join`, `right_join`, and `full_join`) to understand their differences.
3. Summarise the number of flights per aircraft manufacturer, handling missing data appropriately.
4. Visualise the top five manufacturers with a bar plot, displaying labels for each bar.

Setup

We install the `nycflights13` package, then load it along with the `tidyverse` package:

```
# install.packages("nycflights13")

library(nycflights13)
library(tidyverse)
```

Inspecting the Data

We begin by inspecting the structure of the `flights` and `planes` tables to identify the available columns and the common key (`tailnum`).

```
glimpse(flights)
```

```
#> Rows: 336,776
#> Columns: 19
#> $ year           <int> 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2~
#> $ month          <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1~
#> $ day            <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1~
#> $ dep_time       <int> 517, 533, 542, 544, 554, 554, 555, 557, 557, 558, 558, ~
#> $ sched_dep_time <int> 515, 529, 540, 545, 600, 558, 600, 600, 600, 600, 600, ~
#> $ dep_delay      <dbl> 2, 4, 2, -1, -6, -4, -5, -3, -3, -2, -2, -2, -2, -2, -1~
#> $ arr_time       <int> 830, 850, 923, 1004, 812, 740, 913, 709, 838, 753, 849,~
#> $ sched_arr_time <int> 819, 830, 850, 1022, 837, 728, 854, 723, 846, 745, 851,~
#> $ arr_delay      <dbl> 11, 20, 33, -18, -25, 12, 19, -14, -8, 8, -2, -3, 7, -1~
#> $ carrier        <chr> "UA", "UA", "AA", "B6", "DL", "UA", "B6", "EV", "B6", "~
#> $ flight         <int> 1545, 1714, 1141, 725, 461, 1696, 507, 5708, 79, 301, 4~
#> $ tailnum        <chr> "N14228", "N24211", "N619AA", "N804JB", "N668DN", "N394~
#> $ origin         <chr> "EWR", "LGA", "JFK", "JFK", "LGA", "EWR", "EWR", "LGA",~
#> $ dest           <chr> "IAH", "IAH", "MIA", "BQN", "ATL", "ORD", "FLL", "IAD",~
#> $ air_time       <dbl> 227, 227, 160, 183, 116, 150, 158, 53, 140, 138, 149, 1~
#> $ distance       <dbl> 1400, 1416, 1089, 1576, 762, 719, 1065, 229, 944, 733, ~
#> $ hour           <dbl> 5, 5, 5, 5, 6, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 5, 6, 6, 6~
#> $ minute         <dbl> 15, 29, 40, 45, 0, 58, 0, 0, 0, 0, 0, 0, 0, 0, 0, 59, 0~
#> $ time_hour      <dttm> 2013-01-01 05:00:00, 2013-01-01 05:00:00, 2013-01-01 0~
```

```
glimpse(planes)
```

```
#> Rows: 3,322
#> Columns: 9
#> $ tailnum      <chr> "N10156", "N102UW", "N103US", "N104UW", "N10575", "N105UW~
#> $ year         <int> 2004, 1998, 1999, 1999, 2002, 1999, 1999, 1999, 1999, 199~
#> $ type         <chr> "Fixed wing multi engine", "Fixed wing multi engine", "Fi~
#> $ manufacturer <chr> "EMBRAER", "AIRBUS INDUSTRIE", "AIRBUS INDUSTRIE", "AIRBU~
#> $ model        <chr> "EMB-145XR", "A320-214", "A320-214", "A320-214", "EMB-145~
#> $ engines      <int> 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, ~
#> $ seats        <int> 55, 182, 182, 182, 55, 182, 182, 182, 182, 182, 55, 55, 5~
#> $ speed        <int> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N~
#> $ engine       <chr> "Turbo-fan", "Turbo-fan", "Turbo-fan", "Turbo-fan", "Turb~
```

The `flights` table contains 336,776 rows and 19 columns, whereas the `planes` table contains 3,322 rows and 9 columns. Both tables share the `tailnum` field, which we will use to link them.

Relational Analysis with Joins

1. Inner Join

An inner join returns only those rows that have matching keys in both tables. In this case, only flights with a corresponding plane record are included.

```
inner_join_result <- inner_join(flights, planes, by = "tailnum")

inner_join_result
```

```
#> # A tibble: 284,170 x 27
#>    year.x month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>     <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
#>  1   2013     1     1      517            515         2      830            819
#>  2   2013     1     1      533            529         4      850            830
#>  3   2013     1     1      542            540         2      923            850
#>  4   2013     1     1      544            545        -1     1004           1022
#>  5   2013     1     1      554            600        -6      812            837
#>  6   2013     1     1      554            558        -4      740            728
#>  7   2013     1     1      555            600        -5      913            854
#>  8   2013     1     1      557            600        -3      709            723
#>  9   2013     1     1      557            600        -3      838            846
#> 10   2013     1     1      558            600        -2      849            851
#> # i 284,160 more rows
```

```
#> # i 19 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#> #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#> #   hour <dbl>, minute <dbl>, time_hour <dttm>, year.y <int>, type <chr>,
#> #   manufacturer <chr>, model <chr>, engines <int>, seats <int>, speed <int>,
#> #   engine <chr>
```

Since `flights` has 336,776 rows, the inner join result of 284,170 rows indicates that some flights lack a matching `tailnum` in the `planes` table (or have missing `tailnum` values).

2. Left Join {.unnumbered}

A left join returns all rows from the left table (`flights`) and any matching rows from the right table (`planes`). Unmatched plane columns are filled with `NA`.

```
left_join_result <- left_join(flights, planes, by = "tailnum")

left_join_result
```

```
#> # A tibble: 336,776 x 27
#>    year.x month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>     <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
#>  1   2013     1     1      517            515         2      830            819
#>  2   2013     1     1      533            529         4      850            830
#>  3   2013     1     1      542            540         2      923            850
#>  4   2013     1     1      544            545        -1     1004           1022
#>  5   2013     1     1      554            600        -6      812            837
#>  6   2013     1     1      554            558        -4      740            728
#>  7   2013     1     1      555            600        -5      913            854
#>  8   2013     1     1      557            600        -3      709            723
#>  9   2013     1     1      557            600        -3      838            846
#> 10   2013     1     1      558            600        -2      753            745
#> # i 336,766 more rows
#> # i 19 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#> #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#> #   hour <dbl>, minute <dbl>, time_hour <dttm>, year.y <int>, type <chr>,
#> #   manufacturer <chr>, model <chr>, engines <int>, seats <int>, speed <int>,
#> #   engine <chr>
```

The result retains all 336,776 flights. Where there is no matching plane information, the plane-related fields will be `NA`.

3. Right Join {.unnumbered}

A right join returns all rows from the right table (`planes`) and any matching rows from the left table (`flights`). This join emphasises the planes, potentially including planes that did not appear in any flight record.

```
right_join_result <- right_join(flights, planes, by = "tailnum")

right_join_result
```

```
#> # A tibble: 284,170 x 27
#>    year.x month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>     <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
#>  1   2013     1     1      517            515         2      830            819
#>  2   2013     1     1      533            529         4      850            830
#>  3   2013     1     1      542            540         2      923            850
#>  4   2013     1     1      544            545        -1     1004           1022
#>  5   2013     1     1      554            600        -6      812            837
#>  6   2013     1     1      554            558        -4      740            728
#>  7   2013     1     1      555            600        -5      913            854
#>  8   2013     1     1      557            600        -3      709            723
#>  9   2013     1     1      557            600        -3      838            846
#> 10   2013     1     1      558            600        -2      849            851
#> # i 284,160 more rows
#> # i 19 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#> #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#> #   hour <dbl>, minute <dbl>, time_hour <dttm>, year.y <int>, type <chr>,
#> #   manufacturer <chr>, model <chr>, engines <int>, seats <int>, speed <int>,
#> #   engine <chr>
```

Because there are fewer planes than flights, and most flights have matching planes, the result (284,170 rows) is similar to the inner join count. Planes never used in any flight appear with `NA` for flight-specific columns.

4. Full Join {.unnumbered}

A full join includes all rows from both tables, matching where possible. Any rows that do not match in either table are shown with `NA` in the missing fields.

```r
full_join_result <- full_join(flights, planes, by = "tailnum")

full_join_result
```

```
#> # A tibble: 336,776 x 27
#>    year.x month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>     <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
#>  1   2013     1     1      517            515         2      830            819
#>  2   2013     1     1      533            529         4      850            830
#>  3   2013     1     1      542            540         2      923            850
#>  4   2013     1     1      544            545        -1     1004           1022
#>  5   2013     1     1      554            600        -6      812            837
#>  6   2013     1     1      554            558        -4      740            728
#>  7   2013     1     1      555            600        -5      913            854
#>  8   2013     1     1      557            600        -3      709            723
#>  9   2013     1     1      557            600        -3      838            846
#> 10   2013     1     1      558            600        -2      753            745
#> # i 336,766 more rows
#> # i 19 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#> #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#> #   hour <dbl>, minute <dbl>, time_hour <dttm>, year.y <int>, type <chr>,
#> #   manufacturer <chr>, model <chr>, engines <int>, seats <int>, speed <int>,
#> #   engine <chr>
```

Here, we see 336,776 rows again. Since flights is larger, the full join remains at 336,776. Additional planes that are not used in any flights appear with `NA` in flight-related columns but do not increase the total row count.

Summary Table: Flights per Aircraft Manufacturer

Next, we create a summary table of flights per aircraft manufacturer. We use the left join result (`left_join_result`) so that all flights remain, even if plane information is missing. We label these missing values as `"Unknown"`.

```r
manufacturer_summary <- left_join_result %>%
  mutate(manufacturer = if_else(is.na(manufacturer), "Unknown", manufacturer)) %>%
  count(manufacturer, sort = TRUE)

manufacturer_summary
```

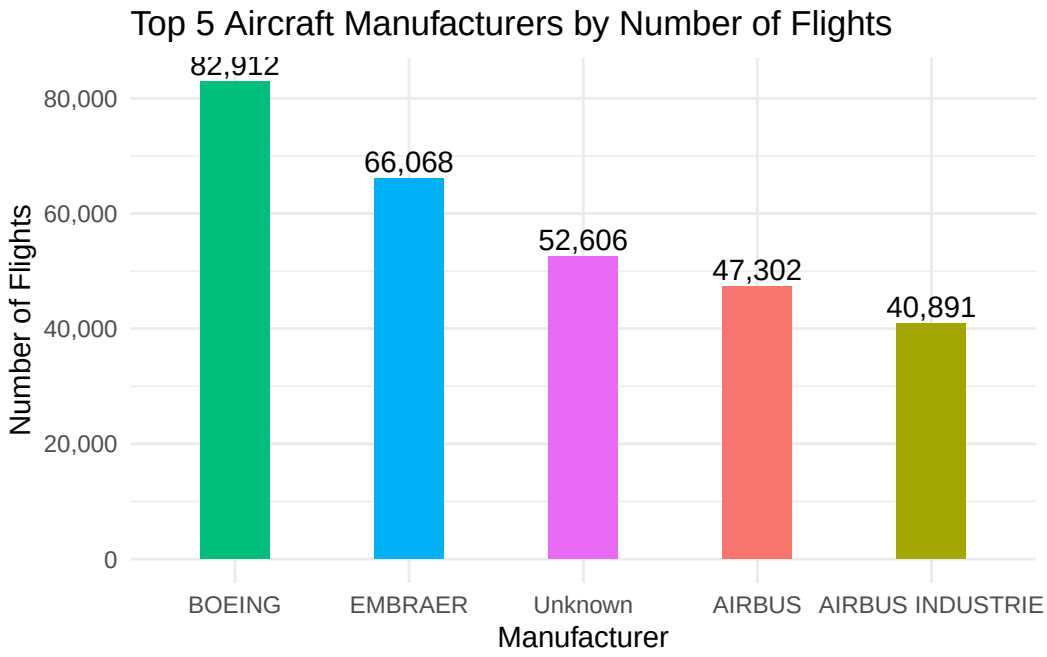
```
#> # A tibble: 36 x 2
#>    manufacturer                      n
#>    <chr>                         <int>
#>  1 BOEING                        82912
#>  2 EMBRAER                       66068
#>  3 Unknown                       52606
#>  4 AIRBUS                        47302
#>  5 AIRBUS INDUSTRIE              40891
#>  6 BOMBARDIER INC                28272
#>  7 MCDONNELL DOUGLAS AIRCRAFT CO  8932
#>  8 MCDONNELL DOUGLAS              3998
#>  9 CANADAIR                       1594
#> 10 MCDONNELL DOUGLAS CORPORATION  1259
#> # i 26 more rows
```

From the output, **BOEING** has the highest number of flights (82,912), followed by **EMBRAER** (66,068). A substantial number of flights (52,606) have no matching manufacturer data, labelled as **Unknown**.

Bar Plot: Top Five Aircraft Manufacturers

Finally, we select the top five manufacturers and visualise them with a bar plot, displaying the exact flight count above each bar.

```
top_manufacturers <- manufacturer_summary %>%
  slice_max(n, n = 5)

top_manufacturers |> ggplot(aes(x = reorder(manufacturer, -n), y = n, fill = manufacturer)) +
  geom_col(show.legend = FALSE, width = 0.4) +
  # Display the exact value above each bar, with thousand separators
  geom_text(aes(label = scales::comma(n)), vjust = -0.3) +
  # Apply thousand separators to the y-axis
  scale_y_continuous(labels = scales::comma) +
  labs(
    x = "Manufacturer",
    y = "Number of Flights",
    title = "Top 5 Aircraft Manufacturers by Number of Flights"
  ) +
  theme_minimal()
```



This chart confirms that Boeing-manufactured planes account for the largest share of flights, followed by Embraer. The "Unknown" category represents flights for which plane data is missing in the `planes` table. You will learn more about data visualisation techniques in Chapter 7 of this book.

Return to Exercise 6.3.1

Lab 7: Data Visualisation

Solution Quiz 7.1

Question 1:

Which principle is the foundation of `ggplot2`'s structured approach to building graphs?

a) The Aesthetic Mapping Principle

b) The Facet Wrapping Technique

c) The Grammar of Graphics ✓

d) The Scaling Transformation Theory

Question 2:

In a `ggplot2` plot, which of the following best describes the role of `aes()`?

a) It specifies the dataset to be plotted.

b) It defines statistical transformations to apply to the data.

c) It maps data variables to visual properties, like colour or size. ✓

d) It sets the coordinate system for the plot.

Question 3:

If you want to display the distribution of a single continuous variable and identify its modality and skewness, which geom is most appropriate?

a) `geom_point()`

b) `geom_bar()`

c) `geom_histogram()` ✓

d) `geom_col()`

Question 4:

When creating a boxplot to show the variation of a continuous variable across multiple categories, what do the "whiskers" typically represent?

a) The median value and the mean value.

b) The full range of the data, excluding outliers.

c) One standard deviation above and below the mean.

d) The maximum and minimum values after applying a 1.5 * IQR rule. ✓

Question 5:

You have a dataset with a categorical variable `Region` and a continuous variable `Sales`. You want to compare total sales across different regions. Which geom and aesthetic mapping would be most appropriate?

a) `geom_bar(aes(x = Region))`, which internally counts the occurrences of each region.

b) `geom_col(aes(x = Region, y = Sales))`, which uses the actual `Sales` values for the bar heights. ✓

c) `geom_line(aes(x = Region, y = Sales))`, connecting points across regions.

d) `geom_area(aes(x = Region, y = Sales))`, to show cumulative totals over regions.

Question 6:

If you want to add a smoothing line (e.g., a regression line) to a scatter plot created with `geom_point()`, which geom should you use and with what parameter to fit a linear model without confidence intervals?

a) `geom_smooth(method = "lm", se = FALSE)` ✓

b) `geom_line(stat = "lm", se = TRUE)`

c) `geom_line(method = "regress", se = FALSE)`

d) `geom_smooth(method = "reg", confint = FALSE)`

Question 7:

Consider you have a factor variable `cyl` representing the number of cylinders in the `mtcars` dataset. If you want to create multiple plots (small multiples) for each value of `cyl`, which ggplot2 function can you use?

a) `facet_wrap(~ cyl)` ✓

b) `facet_side(~ cyl)`

c) `group_by(cyl)` followed by multiple `geom_point()` calls

d) `geom_facet(cyl)`

Question 8:

Which of the following statements about `ggsave()` is true?

a) `ggsave()` must be called before creating any plots for it to work correctly.

b) `ggsave()` saves the last plot displayed, and you can control the output format by specifying the file extension. ✓

c) `ggsave()` cannot control the width, height, or resolution of the output image.

d) `ggsave()` only saves plots as PDF files.

Question 9:

What is the purpose of setting `group` aesthetics in a ggplot, for example in a line plot?

a) To change the colour scale of all elements.

b) To ensure that discrete categories are grouped together for transformations like smoothing.

c) To define which points belong to the same series, enabling lines to connect points within groups instead of mixing data across categories. ✓

d) To modify only the legend titles and labels.

Question 10:

When customizing themes, which of the following options is NOT directly controlled by a `theme()` function in ggplot2?

a) Axis text size, angle, and colour.

b) Background grid lines and panel background.

c) The raw data values in the dataset. ✓

d) The plot title alignment and style.

Return to Quiz 7.1

Solution Quiz 7.2

Question 1:

Which of the following is a key advantage of using Base R graphics for exploratory data analysis?

a) They require additional packages.

b) They offer a quick, function-based approach with no dependencies ✓

c) They utilise a layered grammar for complex plotting.

d) They automatically produce interactive visualisations.

Question 2:

Which function is the generic function in Base R for creating scatterplots, line graphs, and other basic plots?

a) `hist()`

b) `plot()` ✓

c) `boxplot()`

d) `barplot()`

Question 3:

Which function in Base R is specifically used to display data distributions as histograms?

a) `pie()`

b) `plot()`

c) `hist()` ✓

d) `boxplot()`

Question 4:

What is the purpose of the `breaks` argument in the `hist()` function?

a) To set the colour of the bars.

b) To determine the bin width for the histogram ✓

c) To label the axes.

d) To specify the main title.

Question 5:

Which graphical parameter in Base R is used to specify the colour of plot elements?

a) `pch`

b) `lty`

c) `col` ✓

d) `cex`

Question 6:

The `pch` parameter in Base R plots is used to control:

a) The type of point symbol displayed ✓

b) The line thickness.

c) The overall scaling of plot elements.

d) The arrangement of multiple plots.

Question 7:

Which function in Base R is used to adjust global graphical settings, such as margins and layout arrangements?

a) `plot()`

b) `par()` ✓

c) `hist()`

d) `boxplot()`

Question 8:

In a Base R scatter plot, which function is used to add a regression line?

a) `lines()`

b) `abline()` ✓

c) `curve()`

d) `segments()`

Question 9:

What is one of the main reasons Base R graphics are considered advantageous over ggplot2 for certain tasks?

a) They require no additional packages since they are built into R ✓

b) They offer more extensive theme options.

c) They are better suited for interactive visualisations.

d) They automatically manage data transformations.

Question 10:

When saving a Base R plot using the `png()` function, what is the purpose of calling `dev.off()` afterwards?

a) To display the saved plot.

b) To open the saved file in a new window.

c) To close the graphics device and finalise the output file ✓

d) To reset all graphical parameters.

Return to Quiz 7.2

Solution 7.1.2: Reproducing the Smoking, Gender, and Lifespan Chart

In this solution, we will demonstrate how to reproduce the chart that compares the average age at death by smoking status for both males and females. The data comes from the Framingham Heart Study and is contained in the `heart` dataset. Our aim is to filter, summarise, and visualise the data using **dplyr** and **ggplot2**.

Importing and Inspecting the Data

First, we import the `heart.xlsx` file from the `r-data` directory using `read_excel()` and inspect its structure with functions such as `glimpse()`. This step ensures that the dataset has been loaded correctly and familiarises us with its variables.

```
library(tidyverse)
library(readxl)
library(janitor)

# Import the dataset from the r-data directory
heart <- read_excel("r-data/heart.xlsx")
heart <- heart |> clean_names()
```

```r
# Inspect the data structure
glimpse(heart)
```

```
#> Rows: 5,209
#> Columns: 17
#> $ status         <chr> "Dead", "Dead", "Alive", "Alive", "Alive", "Alive", "Al~
#> $ death_cause    <chr> "Other", "Cancer", NA, NA, NA, NA, NA, "Other", NA, "Ce~
#> $ age_ch_ddiag   <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, 57, 55, 79,~
#> $ sex            <chr> "Female", "Female", "Female", "Female", "Male", "Female~
#> $ age_at_start   <dbl> 29, 41, 57, 39, 42, 58, 36, 53, 35, 52, 39, 33, 33, 57,~
#> $ height         <dbl> 62.50, 59.75, 62.25, 65.75, 66.00, 61.75, 64.75, 65.50,~
#> $ weight         <dbl> 140, 194, 132, 158, 156, 131, 136, 130, 194, 129, 179, ~
#> $ diastolic      <dbl> 78, 92, 90, 80, 76, 92, 80, 80, 68, 78, 76, 68, 90, 76,~
#> $ systolic       <dbl> 124, 144, 170, 128, 110, 176, 112, 114, 132, 124, 128, ~
#> $ mrw            <dbl> 121, 183, 114, 123, 116, 117, 110, 99, 124, 106, 133, 1~
#> $ smoking        <dbl> 0, 0, 10, 0, 20, 0, 15, 0, 0, 5, 30, 0, 0, 15, 30, 10, ~
#> $ age_at_death   <dbl> 55, 57, NA, NA, NA, NA, NA, 77, NA, 82, NA, NA, NA, NA,~
#> $ cholesterol    <dbl> NA, 181, 250, 242, 281, 196, 196, 276, 211, 284, 225, 2~
#> $ chol_status    <chr> NA, "Desirable", "High", "High", "High", "Desirable", "~
#> $ bp_status      <chr> "Normal", "High", "High", "Normal", "Optimal", "High", ~
#> $ weight_status  <chr> "Overweight", "Overweight", "Overweight", "Overweight",~
#> $ smoking_status <chr> "Non-smoker", "Non-smoker", "Moderate (6-15)", "Non-smo~
```

The `heart` dataset comprises 5,209 observations and 17 variables, including important fields such as `sex`, `age_at_death`, and `smoking_status`.

We then transform the variable `smoking_status` as ordered factor:

```r
heart <- heart |>
  mutate(smoking_status = factor(smoking_status,
    levels = c(
      "Non-smoker", "Light (1-5)", "Moderate (6-15)", "Heavy (16-25)",
      "Very Heavy (> 25)"
    )
  ))
```

Filtering and Summarising the Data

Next, we filter the data to remove any observations with missing values in the `smoking_status` column. We then group the data by both `smoking_status` and `sex`, calculating the mean of `age_at_death` for each group. This produces a new dataset containing the average age at death per smoking category and gender.

```r
heart_summary <- heart %>%
  filter(!is.na(smoking_status)) %>%
  group_by(smoking_status, sex) %>%
  summarise(
    avg_age_at_death = mean(age_at_death, na.rm = TRUE),
    .groups = "drop"
  )

# Display the summarised data
heart_summary
```

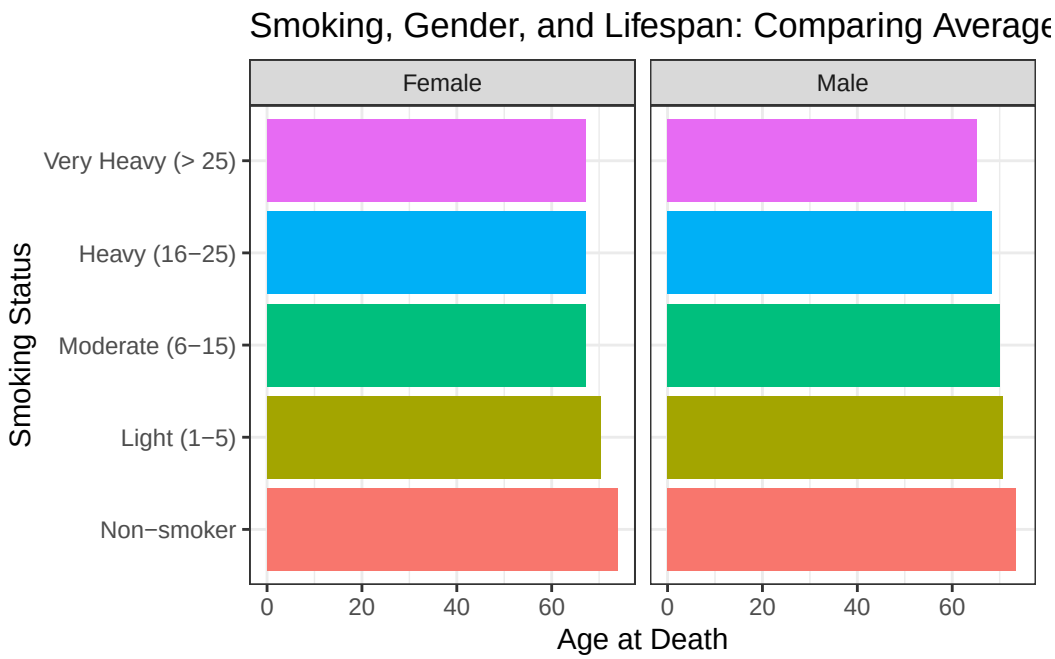
```
#> # A tibble: 10 x 3
#>    smoking_status    sex    avg_age_at_death
#>    <fct>             <chr>             <dbl>
#>  1 Non-smoker        Female             73.9
#>  2 Non-smoker        Male               73.5
#>  3 Light (1-5)       Female             70.4
#>  4 Light (1-5)       Male               70.7
#>  5 Moderate (6-15)   Female             67.1
#>  6 Moderate (6-15)   Male               70.1
#>  7 Heavy (16-25)     Female             67.0
#>  8 Heavy (16-25)     Male               68.3
#>  9 Very Heavy (> 25) Female             67.2
#> 10 Very Heavy (> 25) Male               65.1
```

Visualising the Results

We then use **ggplot2** to create a horizontal bar chart. In this visualisation, the x-axis displays the average age at death, while the y-axis represents the different smoking status categories. The chart is facetted by `sex` to provide separate panels for females and males. The fill colour differentiates the smoking categories, and the plot includes a clear title and axis labels.

```r
heart_summary |>
  ggplot(aes(x = avg_age_at_death, y = smoking_status, fill = smoking_status)) +
  geom_col(show.legend = FALSE) +
  facet_wrap(~sex) +
  labs(
    title = "Smoking, Gender, and Lifespan: Comparing Average Age at Death",
    x = "Age at Death",
    y = "Smoking Status"
```

```
  ) +
  theme_bw()
```



Interpretation

This chart clearly illustrates how the average age at death differs across smoking categories and between genders. Typically, non-smokers appear to have a higher average age at death compared to heavier smokers. Additionally, subtle differences between females and males can be observed, highlighting the significance of considering both smoking status and sex when analysing lifespan.

Return to Exercise 7.1.2

Lab 8: Statistical Concept

Solution Quiz 8.0

Question 1:

Data that focuses on characteristics or qualities rather than numbers is known as:

a) Quantitative data

b) Discrete data

c) Qualitative data ✓

d) Continuous data

Question 2:

Which of the following is an example of discrete data?

a) The height of students in a class

b) The number of cars in a parking lot ✓

c) The amount of rainfall in a day

d) The time taken to complete a task

Question 3:

Quantitative data that can take on any value within a given range is referred to as:

a) Categorical data

b) Nominal data

c) Discrete data

d) Continuous data ✓

Question 4:

Qualitative data differs from quantitative data because qualitative data:
a) Can only be expressed with numbers
b) Has meaningful mathematical operations
c) Describes categories or groups ✓ d) Is always collected from secondary sources

Question 5:

Primary data refers to data that:

a) Has been previously published by others

b) Comes directly from observation or experiment ✓

c) Is always collected online

d) Is obtained only from government agencies

Question 6:

A list of colours observed in a garden (e.g., red, yellow, green) is an example of:

a) Quantitative continuous data

b) Quantitative discrete data

c) Qualitative data ✓

d) Secondary data

Question 7:

Which of the following statements is true?

a) Data is always meaningful without analysis

b) Data, once processed, is known as information ✓

c) Data and information are identical concepts

d) Information is just another term for data collection

Question 8:

A measurement like "23 people attended the seminar" is an example of:

a) Qualitative data

b) Continuous data

c) Discrete data ✓

d) Nominal scale data

Question 9:

Data collected for the first time for a specific research purpose is known as:

a) Secondary data

b) Primary data ✓

c) Nominal data

d) Discrete data

Question 10:

A researcher using census data from a national statistics bureau is working with:

a) Primary data

b) Secondary data ✓

c) Continuous data

d) Nominal data

Return to Quiz 8.0

Solution Quiz 8.1

Question 1:

A complete set of elements (people, items) that we are interested in studying is called a:

a) Sample

b) Population ✓

c) Parameter

d) Statistic

Question 2:

A subset of a population used to make inferences about the population is called a:

a) Population

b) Sample ✓

c) Statistic

d) Parameter

Question 3:

A value that describes a characteristic of an entire population (e.g., population mean) is known as a:

a) Statistic

b) Parameter ✓

c) Variable

d) Sample estimate

Question 4:

A value computed from sample data (e.g., sample mean) that is used to estimate a population parameter is called a:

a) Parameter

b) Statistic ✓

c) Variable

d) Census

Question 5:

Why do we often rely on samples rather than studying entire populations?

a) It is always more accurate.

b) Populations do not have parameters.

c) Sampling is often more feasible, less costly, and time-efficient ✓

d) Populations are always small and uninteresting.

Question 6:

Statistical thinking involves understanding how to:

a) Manipulate data without purpose

b) Draw meaningful conclusions from data under uncertainty ✓

c) Avoid using data in decision-making

d) Ignore variability in data

Question 7:

If a population parameter is μ, the corresponding sample statistic used to estimate it is typically:

a) s

b) σ

c) $\bar{x}$ ✓

d) p

Question 8:

When we attempt to understand the variability in data and the uncertainty in our conclusions, we are engaging in:

a) Statistical thinking ✓

b) Non-statistical reasoning

c) Data neglect

d) Parameter ignorance

Question 9:

If it's too expensive or impractical to study an entire population, we often conduct a:

a) Census

b) Biased survey

c) Sample study ✓

d) Parameter test

Question 10:

The process of using sample data to make conclusions about a larger population is known as:

a) Data summarisation

b) Descriptive statistics

c) Statistical inference ✓

d) Variable classification

Return to Quiz 8.1

Solution- Exercise 8.1.2: Professor Francisca - A Generous Giver

Professor Francisca, the Vice-Chancellor of Thomas Adewumi University, Kwara, Nigeria, and a Professor of Computer Science, is known for her generosity. Each week, she awards monetary prizes (in dollars) to the best student in the weekly Computer Science assignment for the DTS 204 module. The prize amounts are as follows:

495, 503, 503, 498, 503, 505, 503, 500, 501, 489, 498, 488, 499, 497, 508, 507, 507, 509, 508, 503.

Using R, complete the following tasks to analyze the data:

Task 1: Central Tendency

1. **Calculate the Mean**

```
money <- c(495, 503, 503, 498, 503, 505, 503, 500, 501, 489, 498, 488, 499, 497, 508, 50
mean(money)
```

```
#> [1] 501.2
```

2. **Calculate the Median**

```
median(money)
```

```
#> [1] 503
```

3. **Determine the Mode**

```r
statistical_mode <- function(x) {
  uniqx <- unique(x)
  uniqx[which.max(tabulate(match(x, uniqx)))]
}
statistical_mode(money)
```

```
#> [1] 503
```

Task 2: Measure of Spread

1. **Calculate the Range**

```r
range_value <- max(money) - min(money)
range_value
```

```
#> [1] 21
```

2. **Determine the Standard Deviation**

```r
sd(money)
```

```
#> [1] 5.881282
```

The standard deviation helps us understand the consistency of the amounts given out.

3. **Find the Variance**

```r
var(money)
```

```
#> [1] 34.58947
```

Variance is the square of the standard deviation.

Task 3: Measure of Partition

1. **Calculate the Interquartile Range (IQR)**

```r
IQR(money)
```

```
#> [1] 7.5
```

The IQR measures the spread of the middle 50% of the amounts.

2. **Find the Quartiles**

```r
quantile(money)
```

```
#>    0%   25%   50%   75%  100%
#> 488.0 498.0 503.0 505.5 509.0
```

The quartiles reveal the distribution of the amounts.

3. **Calculate Percentile Ranks**

   To determine the percentile ranks for $488 (minimum), $509 (maximum), and $503:

```
ecdf_money <- ecdf(money)
percentile_488 <- ecdf_money(488) * 100 # Percentile rank of $488

percentile_509 <- ecdf_money(509) * 100 # Percentile rank of $509

percentile_503 <- ecdf_money(503) * 100 # Percentile rank of $503
```

   - **Percentile rank of $488**: Indicates the percentage of amounts less than or equal to $488.
   - **Percentile rank of $509**: Indicates the percentage of amounts less than or equal to $509.
   - **Percentile rank of $503**: Indicates the position of $503 within the distribution.

Interpretation

- The mean amount is $501.2, while the median is $503. This slight difference suggests a relatively symmetrical distribution with a slight skew.
- The mode is $503, indicating that this amount was given out most frequently.
- The range of $21 shows the variability between the smallest and largest amounts.
- The standard deviation and variance quantify the overall spread of the amounts.
- The IQR compares the variability of the middle 50% to the overall range, revealing insights about data dispersion.
- The quartiles help understand how the amounts are distributed across the dataset.
- The percentile ranks position specific amounts within the overall distribution, providing context for their relative standing.

Return to Exercise 8.1.2

Solution Quiz 8.2

Question 1:

Which set of values is included in a five-number summary?

a) Mean, Median, Mode, IQR, Standard Deviation

b) Minimum, Q1, Median, Q3, Maximum ✓

c) Minimum, Mean, Mode, Maximum, Range

d) Q1, Q2, Q3, Q4, Q5

Question 2:

The interquartile range (IQR) is calculated as:

a) Q2 - Q1

b) Q3 - Median

c) Q3 - Q1 ✓

d) Median - Minimum

Question 3:

A boxplot is useful for:

a) Displaying frequencies of categorical data

b) Showing the distribution and identifying outliers ✓

c) Calculating correlations between variables

d) Displaying only the mean value

Question 4:

Which value in a five-number summary represents the median of the entire dataset?

a) Q1

b) Q2 (Median) ✓

c) Q3

d) Minimum

Question 5:

If a dataset has many outliers, a boxplot can help by:

a) Ignoring them completely

b) Highlighting them as points beyond the whiskers ✓

c) Removing them automatically

d) Converting them to the mean value

Question 6:

The IQR focuses on the middle 50% of data, making it a good measure of:

a) Central tendency

b) Spread that is not influenced by extreme values ✓

c) Correlation

d) Nominal categories

Question 7:

In R, the `boxplot()` function by default displays:

a) A histogram

b) A correlation matrix

c) A five-number summary depiction ✓

d) A scatter plot

Question 8:

The difference between the maximum and minimum values in a dataset is called the:

a) Standard deviation

b) IQR

c) Range ✓

d) Variance

Question 9:

A box-and-whisker plot typically does NOT show:

a) Median

b) Outliers

c) Mean ✓

d) Interquartile range

Question 10:

When comparing two datasets using boxplots placed side by side, you can quickly assess differences in:

a) Central tendency and spread ✓

b) Exact individual data points

c) Correlation coefficients

d) Detailed frequency distributions

Return to Quiz 8.2

Solution- Exercise 8.2.1

Thirty farmers were surveyed about the number of farm workers they employ during a typical harvest season in Igboho, Oyo State, Nigeria. Their responses are as follows:

4, 5, 6, 5, 1, 2, 8, 0, 4, 6, 7, 8, 4, 6, 7, 9, 8, 6, 7, 5, 5, 4, 2, 1, 9, 3, 3, 4, 6, 4.

Task 1: Calculate the Mean, Median, and Mode

Calculating the Mean:

The **mean** is the sum of all observations divided by the number of observations.

Total number of farm workers employed:

$$\text{Total} = 4 + 5 + 6 + \cdots + 4 = 149$$

Number of observations (n): 30

$$\text{Mean} = \frac{\text{Total}}{n} = \frac{149}{30} \approx 4.97$$

Calculating the Median:

The **median** is the middle value when the data are arranged in ascending order.

First, arrange the data:

0, 1, 1, 2, 2, 3, 3, 4, 4, 4, 4, 4, 5, 5, 5, 5, 6, 6, 6, 6, 6, 7, 7, 7, 7, 8, 8, 8, 9, 9

Since there are 30 observations (an even number), the median is the average of the 15th and 16th values.

15th value: 5

16th value: 5

Median:

$$\text{Median} = \frac{5 + 5}{2} = 5$$

Calculating the Mode:

The **mode** is the value that occurs most frequently.

Frequency of each number:

- **4** occurs 5 times
- **6** occurs 5 times

Both 4 and 6 have the highest frequency.

Therefore, the dataset is **bimodal** with modes at **4** and **6**.

This is how to calculate the mean, median, and mode in R:

```
farm_workers <- c(4, 5, 6, 5, 1, 2, 8, 0, 4, 6, 7, 8, 4, 6, 7, 9, 8, 6, 7, 5, 5, 4, 2, 1, 9,

# Mean
mean(farm_workers)
```

```
#> [1] 4.966667
```

```
# Median
median(farm_workers)
```

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

```
# Mode
statistical_mode <- function(x) {
  uniqx <- unique(x)
  freq <- tabulate(match(x, uniqx))
  uniqx[freq == max(freq)]
}

statistical_mode(farm_workers)
```

```
#> [1] 4
```

Task 2: Determine the Range and Standard Deviation

Calculating the Range:

The **range** is the difference between the maximum and minimum values.

Minimum value:

$$\text{Min} = 0$$

Maximum value:

$$\text{Max} = 9$$

Range:

$$\text{Range} = \text{Max} - \text{Min} = 9 - 0 = 9$$

Calculating the Standard Deviation:

The standard deviation measures the amount of variation or dispersion in the dataset.

This is how to calculate the range and standard deviation in R:

```
# Range
range_values <- range(farm_workers)
range_values # Outputs the minimum and maximum values
```

```
#> [1] 0 9
```

```
range <- diff(range_values)
range # Outputs the range
```

```
#> [1] 9
```

```
# Standard Deviation
sd(farm_workers)
```

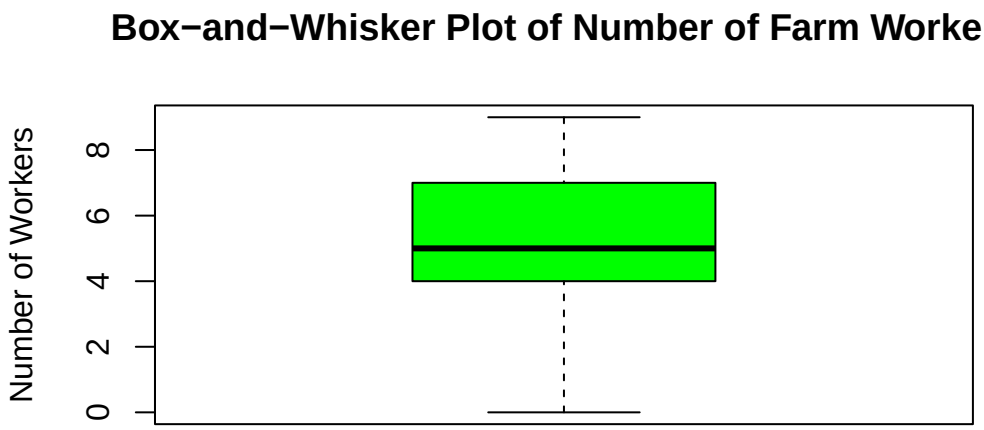
```
#> [1] 2.385059
```

Task 3: Create a Box-and-Whisker Plot of the Distribution

To visualise the distribution of the number of farm workers employed, you can create a box-and-whisker plot.

This is how to create the box plot in R:

```
boxplot(farm_workers,
  main = "Box-and-Whisker Plot of Number of Farm Workers",
  ylab = "Number of Workers",
  col = "green"
)
```



Return to Exercise 8.2.1

Solution Quiz 8.3

Question 1:

A scale that categorises data without any order is known as:

a) Nominal ✓

b) Ordinal

c) Interval

d) Ratio

Question 2:

Which scale provides both order and equal intervals but no true zero point?

a) Nominal

b) Ordinal

c) Interval ✓

d) Ratio

Question 3:

Which scale allows for meaningful ratios and has a true zero?

a) Nominal

b) Ordinal

c) Interval

d) Ratio ✓

Question 4:

Educational levels ranked as "Primary, Secondary, Tertiary" represent which scale?

a) Nominal

b) Ordinal ✓

c) Interval

d) Ratio

Question 5:

Temperatures in Celsius or Fahrenheit are examples of which scale?

a) Nominal

b) Ordinal

c) Interval ✓

d) Ratio

Question 6:

Blood types (A, B, AB, O) are measured on which scale?

a) Nominal ✓

b) Ordinal

c) Interval

d) Ratio

Question 7:

The number of items sold in a store (e.g., 0, 5, 10 units) is best described by which scale?

a) Nominal

b) Ordinal

c) Ratio ✓

d) Interval

Question 8:

Customer satisfaction ratings (e.g., Satisfied, Neutral, Unsatisfied) belong to which scale?

a) Nominal

b) Ordinal ✓

c) Interval

d) Ratio

Question 9:

A key difference between interval and ratio scales is that ratio scales have:

a) Categories only

b) A meaningful zero point ✓

c) No ordering capability

d) Equal intervals that are meaningless

Question 10:

IQ scores are often treated as which type of scale?

a) Nominal

b) Ordinal

c) Interval ✓

d) Ratio

Return to Quiz 8.3

Solution-Exercise 8.3.1: Identify the Scale

1. **Blood pressure readings** (e.g., 120 mmHg, 130 mmHg)

   Answer: Interval Scale

   Explanation:

   - **Numerical Data with Equal Intervals:** Blood pressure readings are numerical values where the difference between measurements is consistent (e.g., the difference between 120 mmHg and 130 mmHg is the same as between 130 mmHg and 140 mmHg).
   - **No True Zero Point:** In the context of blood pressure, a reading of 0 mmHg is not meaningful for living humans; it does not represent the absence of blood pressure but rather an unmeasurable or non-viable state.
   - **Implications:** Because there is no absolute zero, ratios are not meaningful (e.g., we cannot say that 120 mmHg is twice as much as 60 mmHg in a meaningful way).

2. **Type of car owned** (e.g., Sedan, SUV, Truck)

   Answer: Nominal Scale

   Reason:

   - **Categorical Data without Order:** The types of cars are categories used to label different groups. There is no inherent ranking or order among Sedan, SUV, and Truck.
   - **Statistical Analysis:** Only frequency counts and mode are appropriate for nominal data. Calculations like mean or median are not meaningful.

3. **Rankings in a cooking competition** (e.g., 1st, 2nd, 3rd)

   Answer: Ordinal Scale

   Reason:

- **Ordered Categories:** The rankings indicate a clear order of performance, with 1st being better than 2nd, and so on.
- **Unequal Intervals:** The difference in skill or points between 1st and 2nd place may not be the same as between 2nd and 3rd place.
- **Statistical Analysis:** Median and mode are appropriate. Mean is not meaningful due to unequal intervals.

4. **Test scores out of 100** (e.g., 85, 90, 75)

   Answer: Ratio Scale

   Reason:

   - **Numerical Data with Equal Intervals:** The differences between scores are consistent, and a score increase from 75 to 85 is the same increment as from 85 to 95.
   - **Meaningful Zero Point:** A score of 0 represents the absence of correct answers, providing an absolute zero.
   - **Ratios are Meaningful:** It's valid to say that a score of 90 is twice as high as a score of 45.
   - **Statistical Analysis:** All statistical measures are applicable, including mean, median, mode, and coefficient of variation.

5. **Age of students in years**

   Answer: Ratio Scale

   Reason:

   - **Numerical Data with Equal Intervals:** The difference between ages is consistent; the interval between 10 and 15 years is the same as between 20 and 25 years.
   - **Absolute Zero Point:** Age starts at zero (birth), representing a true zero point.
   - **Ratios are Meaningful:** It makes sense to say that a 20-year-old is twice as old as a 10-year-old.
   - **Statistical Analysis:** All statistical operations are valid, allowing for comprehensive analysis.

Return to Exercise 8.3.1

Lab 9: Sampling Techniques

Solution 9.1.1: Simple Random Sampling with the Penguins Dataset

```
# Load the required package
library(palmerpenguins)

# Inspect the penguins dataset
summary(penguins)
```

```
#>       species          island    bill_length_mm  bill_depth_mm
#>  Adelie   :152   Biscoe   :168   Min.   :32.10   Min.   :13.10
#>  Chinstrap: 68   Dream    :124   1st Qu.:39.23   1st Qu.:15.60
#>  Gentoo   :124   Torgersen: 52   Median :44.45   Median :17.30
#>                                  Mean   :43.92   Mean   :17.15
#>                                  3rd Qu.:48.50   3rd Qu.:18.70
#>                                  Max.   :59.60   Max.   :21.50
#>                                  NA's   :2       NA's   :2
#>  flipper_length_mm  body_mass_g       sex           year
#>  Min.   :172.0     Min.   :2700   female:165   Min.   :2007
#>  1st Qu.:190.0     1st Qu.:3550   male  :168   1st Qu.:2007
#>  Median :197.0     Median :4050   NA's  : 11   Median :2008
#>  Mean   :200.9     Mean   :4202                Mean   :2008
#>  3rd Qu.:213.0     3rd Qu.:4750                3rd Qu.:2009
#>  Max.   :231.0     Max.   :6300                Max.   :2009
#>  NA's   :2         NA's   :2
```

```
# Remove rows with missing values
penguins_complete <- na.omit(penguins)

# Check how many rows remain
nrow(penguins_complete)
```

```
#> [1] 333
```

```
# Set a seed for reproducibility
set.seed(123)

# Select a random sample of 10 penguins
sample_indices <- sample(1:nrow(penguins_complete), size = 10, replace = FALSE)
```

```r
penguins_sample <- penguins_complete[sample_indices, ]

# Calculate the mean body mass of the entire dataset
mean_full <- mean(penguins_complete$body_mass_g)

# Calculate the mean body mass of the sampled penguins
mean_sample <- mean(penguins_sample$body_mass_g)

# Print the results
cat("Mean body mass (entire dataset):", mean_full, "grams\n")
```

```
#> Mean body mass (entire dataset): 4207.057 grams
```

```r
cat("Mean body mass (sample of 10):", mean_sample, "grams\n")
```

```
#> Mean body mass (sample of 10): 4550 grams
```

Reflection:

In this exercise, the sample mean (approximately 4550 grams) is higher than the full dataset mean (approximately 4207 grams). This difference can occur because a sample of just 10 penguins may not perfectly represent the entire penguin population. With a small sample size, random chance can lead to a subset of penguins that are, on average, heavier (or lighter) than the overall population. If you were to increase the sample size, you'd generally expect the sample mean to get closer to the true mean of the full dataset.

Return to Exercise 9.1.1

Solution 9.1.2: Stratified Sampling with the Diamonds Dataset

```r
library(dplyr)

# Inspect the diamonds dataset
glimpse(diamonds)
```

```
#> Rows: 53,940
#> Columns: 10
#> $ carat   <dbl> 0.23, 0.21, 0.23, 0.29, 0.31, 0.24, 0.24, 0.26, 0.22, 0.23, 0.~
#> $ cut     <ord> Ideal, Premium, Good, Premium, Good, Very Good, Very Good, Ver~
```

```
#> $ color   <ord> E, E, E, I, J, J, I, H, E, H, J, J, F, J, E, E, I, J, J, J, I,~
#> $ clarity <ord> SI2, SI1, VS1, VS2, SI2, VVS2, VVS1, SI1, VS2, VS1, SI1, VS1, ~
#> $ depth   <dbl> 61.5, 59.8, 56.9, 62.4, 63.3, 62.8, 62.3, 61.9, 65.1, 59.4, 64~
#> $ table   <dbl> 55, 61, 65, 58, 58, 57, 57, 55, 61, 61, 55, 56, 61, 54, 62, 58~
#> $ price   <int> 326, 326, 327, 334, 335, 336, 336, 337, 337, 338, 339, 340, 34~
#> $ x       <dbl> 3.95, 3.89, 4.05, 4.20, 4.34, 3.94, 3.95, 4.07, 3.87, 4.00, 4.~
#> $ y       <dbl> 3.98, 3.84, 4.07, 4.23, 4.35, 3.96, 3.98, 4.11, 3.78, 4.05, 4.~
#> $ z       <dbl> 2.43, 2.31, 2.31, 2.63, 2.75, 2.48, 2.47, 2.53, 2.49, 2.39, 2.~
```

```
diamonds |> count(cut)
```

```
#> # A tibble: 5 x 2
#>   cut           n
#>   <ord>     <int>
#> 1 Fair       1610
#> 2 Good       4906
#> 3 Very Good 12082
#> 4 Premium   13791
#> 5 Ideal     21551
```

```
# Calculate the proportions of each cut in the full dataset
full_props <- prop.table(table(diamonds$cut))
full_props
```

```
#>
#>       Fair       Good  Very Good    Premium      Ideal
#> 0.02984798 0.09095291 0.22398962 0.25567297 0.39953652
```

```
# Set a total sample size
sample_size <- 500

# Perform stratified sampling based on cut
set.seed(123)
stratified_sample <- diamonds %>%
  group_by(cut) %>%
  sample_frac(sample_size / nrow(diamonds))

# Calculate the proportions of each cut in the stratified sample
sample_props <- prop.table(table(stratified_sample$cut))

# Compare the distributions
full_props
```

```
#>
#>       Fair       Good  Very Good    Premium      Ideal
#> 0.02984798 0.09095291 0.22398962 0.25567297 0.39953652
```

```
sample_props
```

```
#>
#>      Fair      Good Very Good   Premium     Ideal
#>     0.030     0.090     0.224     0.256     0.400
```

Reflection:

In this example, the stratified sampling process successfully captured proportions of `cut` categories that closely mirror the full dataset. The original distribution had approximately:

- Fair: 2.98%
- Good: 9.10%
- Very Good: 22.40%
- Premium: 25.57%
- Ideal: 39.95%

Our stratified sample resulted in:

- Fair: 3.0%
- Good: 9.0%
- Very Good: 22.4%
- Premium: 25.6%
- Ideal: 40.0%

These proportions are nearly identical. This demonstrates the strength of stratified sampling: by using known subgroup proportions, we can ensure that even a relatively small sample remains representative of the underlying categories. If we had used simple random sampling, our sample's distribution might have deviated more from the true population proportions. In scenarios where preserving the population structure is important—such as when analysing variations across categories—stratified sampling provides a more reliable and balanced approach.

Return to Exercise 9.1.2

Solution 9.1.3: Cluster Sampling with a Simulated Dataset

```
set.seed(123)

# Suppose we have 10 cities (clusters), each with 200 customers
cities <- rep(paste0("City_", 1:10), each = 200)
n <- length(cities) # total number of customers

# Simulate a dataset of customers
# Monthly spending is drawn from a normal distribution, but let's vary it by city
# For simplicity, we'll say city 1 has a higher spending average, and so forth.
spending <- rnorm(n, mean = 500 + as.numeric(sub("City_", "", cities)) * 10, sd = 50)

customers <- data.frame(
  customer_id = 1:n,
  city = cities,
  monthly_spending = spending
)

# Examine the full population mean monthly spending
mean_full <- mean(customers$monthly_spending)

# Perform cluster sampling:
# Select, for example, 3 cities at random
selected_cities <- sample(unique(customers$city), size = 3, replace = FALSE)

# Extract customers from the selected cities
cluster_sample <- subset(customers, city %in% selected_cities)

# Calculate the mean monthly spending in the cluster sample
mean_cluster_sample <- mean(cluster_sample$monthly_spending)

cat("Mean monthly spending (full population):", round(mean_full, 2), "\n")
```

```
#> Mean monthly spending (full population): 556.46
```

```
cat("Mean monthly spending (cluster sample):", round(mean_cluster_sample, 2), "\n")
```

```
#> Mean monthly spending (cluster sample): 554.82
```

Reflection:

The mean monthly spending for the entire population is about 556.46, while the cluster sample's mean is 554.82, indicating a close match. This suggests that the chosen clusters captured a reasonably representative snapshot of the overall population.

However, if different clusters were selected, the sample mean might have differed more, especially if those clusters had unusual spending patterns. Choosing more clusters typically improves representativeness but comes with additional cost and effort. Ultimately, cluster sampling is a compromise: it's more practical and efficient than sampling individuals spread across all clusters, while still offering a fairly accurate estimate of the population's characteristics.

Return to Exercise 9.1.3

Solution 9.1.4: Systematic Sampling on a Simple List

```
# Create a vector of individuals
individuals <- 1:1000

# Define the desired sample size
sample_size <- 100

# Calculate k (the interval)
k <- length(individuals) / sample_size

# Set a seed for reproducibility
set.seed(123)

# Choose a random starting point between 1 and k
start <- sample(1:k, 1)
start
```

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

```
# Select every k-th individual after the starting point
systematic_sample <- individuals[seq(from = start, to = length(individuals), by = k)]

# Check the length of the sample
length(systematic_sample)
```

```
#> [1] 100
```

```r
# Verify the pattern (difference between consecutive elements should be k)
diff(systematic_sample)
```

```
#>  [1] 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
#> [26] 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
#> [51] 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
#> [76] 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
```

```r
# Print the first few selected IDs
head(systematic_sample)
```

```
#> [1]  3 13 23 33 43 53
```

Now experiment with a different sample size, say 50 (k = 1000/50 = 20)

```r
# Try a sample size of 50 (k = 1000/50 = 20)

sample_size_50 <- 50
k_50 <- length(individuals) / sample_size_50
start_50 <- sample(1:k_50, 1)
systematic_sample_50 <- individuals[seq(from = start_50, to = length(individuals), by = k_50)

length(systematic_sample_50)
```

```
#> [1] 50
```

```r
head(systematic_sample_50)
```

```
#> [1]  14  34  54  74  94 114
```

Now experiment with a different sample size, say 20 (k = 1000/20 = 50)

```r
sample_size_20 <- 20
k_20 <- length(individuals) / sample_size_20
start_20 <- sample(1:k_20, 1)
systematic_sample_20 <- individuals[seq(from = start_20, to = length(individuals), by = k_20)

length(systematic_sample_20)
```

```
#> [1] 20
```

```
head(systematic_sample_20)
```

```
#> [1]   3  53 103 153 203 253
```

Reflection:

With a sample size of 100 (k=10), the selected IDs start at 3 and increment by 10 (e.g., 3, 13, 23…), covering the full range of 1 to 1000 at relatively tight intervals. This provides a fairly even spread across the list.

When the sample size is reduced to 50 (k=20), the sample begins at 14 and then jumps every 20 IDs (e.g., 14, 34, 54, …), resulting in a sparser coverage of the list. While still systematic, these increments skip more numbers between selections.

With a sample size of 20 (k=50), the coverage is even sparser (e.g., 3, 53, 103…), selecting every 50th ID. This leaves large gaps and captures fewer points along the list, potentially missing subtler patterns.

These differences highlight how changing the sample size (and thus k) affects the granularity of coverage. More frequent intervals (small k) give a denser sampling and may better represent variability across the dataset. Larger intervals (large k) might be more efficient but could risk missing important variations if the data has underlying patterns. Systematic sampling is easy to implement and ensures even coverage, but the choice of k and the starting point can significantly influence which individuals get selected.

Return to Exercise 9.1.4

Solution Quiz 9.1: Probability Sampling

Question 1:

What is the defining feature of probability sampling methods?

a) They always use large sample sizes

b) Each member of the population has a known, nonzero chance of selection ✓

c) They never require a sampling frame

d) They rely on the researcher's judgment

Question 2:

In simple random sampling (SRS), every member of the population:

a) Has no chance of being selected

b) Is selected to represent different subgroups

c) Has an equal probability of being selected ✓

d) Is chosen based on convenience

Question 3:

Stratified sampling involves:

a) Selecting whole groups at once

b) Sampling every kth individual

c) Ensuring subgroups are represented proportionally ✓

d) Selecting individuals recommended by others

Question 4:

Which method is best if you know certain subgroups (strata) differ and you want each to be represented in proportion to their size?

a) Simple random sampling

b) Stratified sampling ✓

c) Cluster sampling

d) Convenience sampling

Question 5:

Cluster sampling is typically chosen because:

a) It is guaranteed to be perfectly representative

b) It reduces cost and logistical complexity ✓

c) It involves selecting individuals from every subgroup

d) It ensures each individual has the same probability of selection as in SRS

Question 6:

In a national health survey using cluster sampling, which of the following represents a "cluster"?

a) A randomly chosen patient from all over the country

b) A randomly selected set of hospitals ✓

c) A proportionate sample of age groups

d) Every 10th patient in a hospital list

Question 7:

Systematic sampling selects individuals by:

a) Relying on personal judgment

b) Selecting every kth individual after a random start ✓

c) Dividing the population into strata

d) Choosing only those easiest to reach

Question 8:

If the population is 10,000 units and you need a sample of 100, the interval k in systematic sampling is:

a) 10 (10,000 ÷ 1,000)

b) 100 (10,000 ÷ 100) ✓

c) 20 (10,000 ÷ 500)

d) 50 (10,000 ÷ 200)

Question 9:

One advantage of systematic sampling is:

a) It ensures no bias will ever occur

b) It provides a convenient and even spread of the sample ✓

c) It requires no sampling frame

d) It automatically includes all subgroups

Question 10:

Which of the following is NOT a probability sampling method?

a) Simple random sampling

b) Stratified sampling

c) Cluster sampling

d) Convenience sampling ✓

Return to Quiz 9.1

Solution Quiz 9.2: Non-Probability Sampling

Question 1:

Non-probability sampling methods are often chosen because:

a) They guarantee generalizable results

b) They are cheaper, faster, or more practical ✓

c) They eliminate all forms of bias

d) They require a complete list of the population

Question 2:

Which method involves selecting participants who are easiest to reach?

a) Convenience sampling ✓

b) Snowball sampling

c) Purposive sampling

d) Quota sampling

Question 3:

Snowball sampling is most useful for:

a) Large, well-documented populations

b) Populations where every member is easily identified

c) Hidden or hard-to-reach populations ✓

d) Ensuring random selection of subgroups

Question 4:

In snowball sampling, the sample grows by:

a) Randomly picking individuals from a list

b) Selecting every kth individual

c) Asking initial participants to refer others ✓

d) Dividing the population into equal parts

Question 5:

Judgmental (purposive) sampling relies on:

a) Each member of the population having an equal chance

b) The researcher's expertise and judgment ✓

c) Selecting individuals based solely on their availability

d) A systematic interval selection

Question 6:

A researcher who specifically seeks out top experts or key informants in a field is using:

a) Purposive (judgmental) sampling ✓

b) Cluster sampling

c) Systematic sampling

d) Simple random sampling

Question 7:

Quota sampling ensures subgroups are represented by:

a) Randomly selecting from each subgroup

b) Matching known proportions but using non-random selection ✓

c) Following a strict interval for selection

d) Relying on participant referrals

Return to Quiz 9.2

Lab 10: Data Science Concept

Solution Quiz 10.1

Question 1:

Data science is considered interdisciplinary because it involves the integration of:

a) Mathematics, domain expertise, and biological sciences

b) Programming, mathematics/statistics, and domain expertise ✓

c) Philosophy, ethics, and data engineering

d) Chemistry, physics, and computer science

Question 2:

The iterative nature of the data science lifecycle is essential for:

a) Ensuring a one-time solution

b) Continuous refinement and improved insights ✓

c) Avoiding communication and visualisation steps

d) Reducing time spent on data wrangling

Question 3:

In the context of anomaly detection, which of the following scenarios is most relevant?

a) Predicting future sales

b) Identifying fraudulent transactions ✓

c) Recommending products to customers

d) Forecasting weather trends

Question 4:

Why is domain expertise considered critical in data science projects?

a) To eliminate the need for reproducible workflows

b) To ensure analyses are contextually accurate and meaningful ✓

c) To substitute for statistical reasoning

d) To automate the cleaning process

Question 5:

Which of the following ethical considerations is essential in data science?

a) Automating decision-making without human oversight

b) Mitigating bias and ensuring fairness ✓

c) Replacing statistical methods with machine learning

d) Eliminating reproducibility for scalability

Question 6:

In the healthcare analytics example, the role of predictive modelling primarily involves:

a) Replacing clinicians in decision-making

b) Identifying trends in patient demographics

c) Predicting patient readmissions and improving care ✓

d) Tidying and transforming hospital data

Question 7:

During the "Tidy" phase of the data science lifecycle, what is the primary goal?

a) Creating dashboards for analysis

b) Organising data into a structured format for analysis ✓

c) Designing machine learning models

d) Cleaning visualisations for stakeholder presentations

Question 8:

Which stage of the data science lifecycle involves crafting visual narratives to interpret results?

a) Model

b) Transform

c) Visualise ✓

d) Import

Question 9:

Why is the "Communicate" phase considered critical in the data science lifecycle?

a) It automates repetitive data cleaning tasks

b) It presents findings clearly and persuasively to stakeholders ✓

c) It eliminates the need for statistical reasoning

d) It directly replaces the "Model" phase

Question 10:

How does viewing data analysis as a cyclical lifecycle benefit complex projects?

a) Reduces the need for domain expertise

b) Supports iterative refinement and evolving datasets ✓

c) Guarantees fixed solutions for all analyses

d) Simplifies reproducibility without documentation

Return to Quiz 10.1

Lab 11: Use Case Projects

General Solution Quiz 11

Question 1:

What is the main purpose of the pipe operator (`|>` or `%>%`) in R?

a) To run code in parallel.

b) To nest functions inside one another.

c) To pass the output of one function as the input to the next, improving code readability. ✓

d) To automatically clean missing data.

Question 2:

In a reproducible R workflow (as discussed in early labs), which file type is commonly used to document code, results, and narrative together?

a) CSV files

b) R Markdown (or Quarto) documents ✓

c) PNG images

d) Excel spreadsheets

Question 3:

When creating a new RStudio Project to ensure reproducibility and organisation of your analysis, what is one key advantage?

a) It automatically generates a machine learning model.

b) It sets the working directory to the project folder, simplifying relative paths. ✓

c) It prevents all missing values.

d) It disables package installation from CRAN.

Question 4:

The principle of **tidy data** states that:

a) Each dataset should have no missing values.

b) Each column represents a variable, each row represents an observation, and each cell contains a single value. ✓

c) Each dataset must have at least 10 columns.

d) Each value in the dataset must be numeric.

Question 5:

Which `dplyr` verb is used to filter rows based on logical conditions?

a) `select()`

b) `mutate()`

c) `filter()` ✓

d) `summarise()`

Question 6:

To create new columns or modify existing ones in your dataset using `dplyr`, you would use:

a) `select()`

b) `mutate()` ✓

c) `arrange()`

d) `group_by()`

Question 7:

Which `ggplot2` component maps data variables to visual properties like axes, colour, or size?

a) Theme

b) Facets

c) Aesthetics (`aes()`) ✓

d) Scales

Question 8:

To reorder rows of data based on a variable's value using `dplyr`, which function should be applied?

a) `rename()`

b) `arrange()` ✓

c) `distinct()`

d) `count()`

Question 9:

In the data science lifecycle discussed, which stage primarily involves creating charts, graphs, or other graphical representations of data?

a) Import

b) Tidy

c) Transform

d) Visualise ✓

Question 10:

What is the role of `group_by()` in conjunction with `summarise()`?

a) It imports a dataset from the internet.

b) It filters rows based on conditions.

c) It splits the data into groups, allowing summarised statistics per group. ✓

d) It changes variable names.

Question 11:

When exploring data from a new dataset, which of the following is a best practice?

a) Immediately running complex models without understanding distributions.

b) Creating exploratory visualisations and computing descriptive statistics. ✓

c) Ignoring missing values.

d) Never using `glimpse()` or `head()`.

Question 12:

Which `ggplot2` function would you use to create a boxplot?

a) `geom_bar()`

b) `geom_point()`

c) `geom_boxplot()` ✓

d) `geom_smooth()`

Question 13:

Converting code, analysis, and narrative into a single reproducible document is commonly achieved with:

a) `read_csv()` only.

b) Proprietary binary formats.

c) R Markdown (or Quarto) documents. ✓

d) Manually copying results into Word documents.

Question 14:

Which operator in R is used to chain data operations in a logical sequence, making code more readable?

a) `%>%` (from magrittr) or `|>` (native pipe) ✓

b) `$`

c) `*`

d) `=`

Question 15:

Data science is often described as an intersection of three main areas. Which combination is correct?

a) Domain expertise, mathematics/statistics, and computer science/programming. ✓

b) Chemistry, physics, and biology.

c) Finance, marketing, and sales.

d) Geography, history, and literature.

Question 16:

In a data science project, why is communicating findings effectively so important?

a) It ensures the code runs faster.

b) It guarantees no missing values remain.

c) It enables stakeholders to understand insights and make informed decisions. ✓

d) It replaces the need for data transformations.

Question 17:

When dealing with missing data, which is NOT a recommended strategy?

a) Identifying and quantifying missing values.

b) Imputing values using mean or median if appropriate.

c) Removing all data points and ignoring the missingness context. ✓

d) Documenting how missing data was handled.

Question 18:

Which `dplyr` function extracts unique rows or identifies distinct values?

a) `distinct()` ✓

b) `rename()`

c) `relocate()`

d) `case_when()`

Question 19:

Why are use case projects invaluable for learners transitioning from theory to practice?

a) They allow bypassing basic R syntax rules.

b) They simplify code without testing problem-solving skills.

c) They help integrate various skills, face real-world challenges, and deepen understanding. ✓

d) They remove the need for documentation.

Question 20:

In the data science lifecycle, what is typically the final stage?

a) Model

b) Communicate ✓

c) Tidy

d) Transform

Return to General Practice Quiz 11

B Downloading and Preparing the Data

To fully engage with the exercises and examples in this book, you'll need to download the datasets provided. The data is organized in a folder named `r-data`, which contains all the files we'll use throughout the chapters.

B.1 Downloading the Data

1. **Access the Data Folder**

   Visit the following link to access the `r-data` folder on Google Drive:
   https://bit.ly/r-data-directory or https://drive.google.com/drive/folders/1ZhI-t94uZa82KD8hEN0f1WALfCiRFWCP

2. **Download the `r-data` Folder**

   - Once you're on the Google Drive page, you should see the `r-data` folder listed.
   - Right-click on the `r-data` folder and select **Download**.
   - Google Drive will compress the folder into a ZIP file before downloading it to your computer.

3. **Unzip the Folder**

   - After the download is complete, locate the ZIP file on your computer (usually in your **Downloads** folder).
   - Extract the contents of the ZIP file:
     - **Windows**: Right-click the ZIP file and select **Extract All**, then follow the prompts.
     - **macOS**: Double-click the ZIP file to extract it.
     - **Linux**: Right-click and select **Extract Here**, or use the command line `unzip filename.zip`.

4. **Verify the Contents**

   - Open the extracted `r-data` folder to ensure all files are present.
   - You should see various datasets in formats like CSV, Excel, and others, which we'll use in different labs.

B.2 Setting Up Your Working Directory

To keep your work organized and ensure consistency across exercises, we'll create a dedicated RStudio Project for each lab or exercise that uses data from the `r-data` folder. This approach helps manage your files efficiently and ensures that your working directory is correctly set for each task.

B.2.1 Creating a New RStudio Project for Each Exercise

1. **Identify the Lab or Exercise**
   - Determine which lab or exercise you're working on (e.g., **Lab 2**, **Exercise 4.1**).
2. **Create a Directory for the Project**
   - On your computer, create a new folder with a meaningful name for the lab or exercise, such as `Lab2_Project` or `Exercise4_1_Project`.
3. **Copy Necessary Data Files**
   - From the extracted `r-data` folder, copy the specific data files needed for the exercise into your new project folder.
   - Alternatively, you can copy the entire `r-data` folder into your project directory if multiple datasets are required.
4. **Create a New RStudio Project**
   - Open RStudio.
   - Go to **File** > **New Project**.
   - Choose **Existing Directory**.
   - Browse to the directory you just created for the lab or exercise.
   - Select the folder and click **Create Project**.
5. **Organize Your Project Files**
   - Within your project directory, consider creating subfolders such as `data`, `scripts`, and `output` to further organize your work.
     - Place your data files in the `data` folder.
     - Save your R scripts in the `scripts` folder.
     - Direct any output files (like graphs or reports) to the `output` folder.

6. **Working Within the Project**

    - When you open the RStudio Project, your working directory is automatically set to the project's root directory.
    - When reading or writing files, use relative paths starting from the project directory to ensure your code works on any system where the project folder is set as the working directory.

```r
# Example of reading a CSV file from the data folder
data <- read_csv("r-data/your-dataset.csv")
```

> **Note**
>
> Make sure to use forward slashes `/` in the file path, even on Windows.

B.2.2 Benefits of Using Separate Projects for Each Exercise

- **Organization**: Keeps your work for each lab or exercise neatly contained, preventing files from different tasks from mixing.
- **Reproducibility**: By maintaining all necessary files within each project, you make it easier to revisit or share your work without missing dependencies.
- **Clarity**: Helps you focus on the specific objectives of each exercise without distractions from other projects.

B.3 Data Usage and Ethics

The datasets and link provided are safe and intended for educational use in conjunction with this book to help you practice and apply the concepts covered. Please use the data responsibly and refrain from using it for any unauthorized purposes.

- **Privacy**: Be mindful that while the datasets are fictional or anonymized, they may represent sensitive topics. Handle all data with respect and confidentiality.
- **Attribution**: If you use the datasets in any presentations or projects outside of this book's exercises, please acknowledge the source appropriately.

B.4 Getting Help

If you encounter any issues downloading or accessing the data:

- **Check Your Internet Connection**: Ensure you have a stable connection when downloading the data.
- **Try a Different Browser**: Sometimes switching browsers can resolve download issues.

By setting up the data as described, you'll be ready to dive into the hands-on labs and fully engage with the practical exercises. Having the data organized and accessible will streamline your workflow and enhance your learning experience.

Happy analyzing!