R Programming Fundamentals: A Lab-Based Approach

Ezekiel Ogundepo

Table of contents

Preface

Welcome to **"R Programming Fundamentals: 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!

1 Getting Started with R

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 your future explorations in data analysis and statistical computing.

By the end of this lab, you'll have a strong grasp of the basics of R programming, setting you up to dive deeper into more complex topics later on.

Here's what we'll cover:

- **Exploring the RStudio Interface**
  You'll get acquainted with the four main panes of RStudio and see how each one contributes to a smooth and efficient coding experience.
- **Performing Basic Calculations**
  You'll learn how to use R as a calculator, performing arithmetic operations while understanding the order of operations.
- **Understanding Atomic Data Types**
  We'll 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:**
  You'll practice creating variables, assigning values to them, and following proper naming conventions, an essential skill for organizing your code.
- **Using Conditional Statements**
  You'll 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 not only be comfortable with the RStudio environment but also able 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.

1.1 Introduction

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

1.1.1 Why learning R programming?

Learning R opens doors to a vast ecosystem of packages and resources that make data analysis and visualization more accessible and efficient. Its active community continually contributes to its development, ensuring that it stays up-to-date with the latest methodologies in data science.

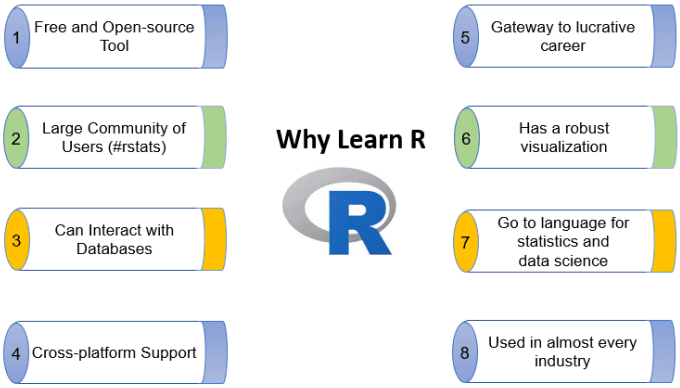


Figure 1.1: Compelling Reasons to Learn R

1.1.2 Companies Using R for Analytics

Many leading companies leverage R for their analytics needs, demonstrating its practical applications in the industry. You can find a list of such companies here.

1.1.3 Learning Curve

While R might seem challenging at first, many users find that it simplifies complex tasks once you get the hang of it. Think of it as making difficult things easy and easy things even easier!

Figure 1.2: Major Companies Using R Programming

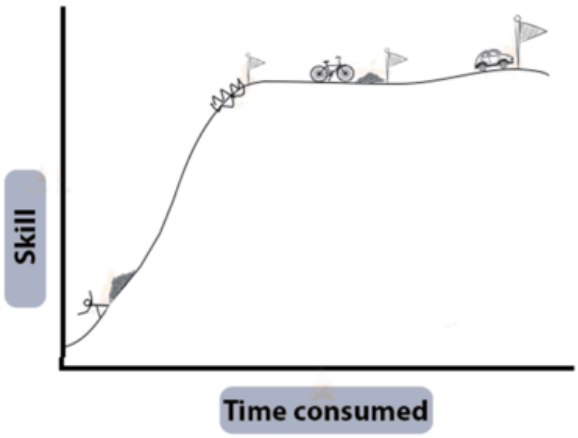


Figure 1.3: The Learning Curve of R Programming

1.1.4 Installing R and RStudio

Before we dive in, you'll need to have both R and RStudio installed on your computer. R is the core programming language, while RStudio provides a user-friendly interface that enhances your coding experience.

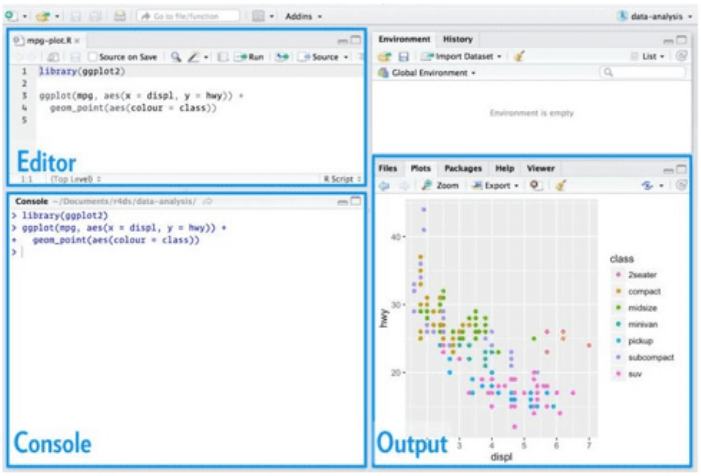


Figure 1.4: Overview of the RStudio Interface

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:**

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

Installing RStudio

Once R is installed, you'll want to install RStudio, which provides an easier interface to interact 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.

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

1.2 Experiment 1.1: RStudio Interface and Basic Calculations

In this experiment, you will begin working with R. You will learn how to navigate the four panes in RStudio, use R as a calculator, assign values to variables, and understand basic data types.

1.2.1 The Four Panes of RStudio

RStudio is divided into four main panes, each serving a specific purpose to enhance your coding workflow[1].

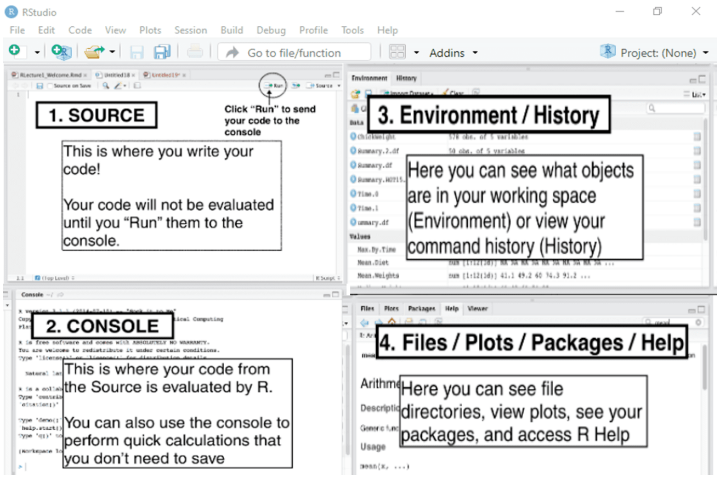


Figure 1.5: Annotated Overview of Key RStudio Panels

Source Pane

- This is where you write your R code. Think of it as your notepad or a place to draft your work.

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

- 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 great for writing scripts that you can save and use later.

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 shows up 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.

Environment/History Pane

- **Environment Tab**: This shows you 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 keeps a record of every command you've entered, allowing you to track what you've done so far.

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.

Figure 1.6: R Programming as a Powerful Calculator

1.2.2 Basic Calculations in R Programming

R can perform all standard arithmetic operations, making it a handy calculator.

The basic operators include:

- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/)
- Exponentiation (^)
- Modulo (%%)
- Parenthesis ()

Arithmetic Operations

```
6 + 12 - 8
```

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

```
2 * 3
```

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

```
100 / 50
```

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

```
3 * 5 / 3
```

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

```
3^2
```

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

Modulus

The modulo (or "modulus" or "mod") is the remainder after division. For example, `9 mod 2 = 1`. Because `9/2 = 4` with a remainder of 1. In mathematics, we write that as `9 mod 2 = 1` and in R we write it as `9 %% 2 = 1`.

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

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

Parenthesis or brackets

Parentheses are used to denote grouping of operation in mathematics. It denotes modifications to normal order of operations. Do you remember **BODMAS** in mathematics? We shall use BEDMAS: Brackets, Exponentiation, Division, Multiplication, Addition, Subtraction in programming.

In an expression like $3 \times (2+3)$, the part of the expression within the parentheses, $(2+3) = 5$, is evaluated first, and then this result is used in the rest of the expression i.e. $3 \times 5 = 15$.

```
3 * (2 + 3) # Returns 15
```

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

```
(3 + 2) * (6 - 4) # Returns 10
```

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

Operations Involving Square Roots

To calculate square roots, use the `sqrt()` function.

$\sqrt{125}$

```
sqrt(125)
```

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

$$\frac{19}{\sqrt{19}}$$

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

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

1.2.3 Comments in R

Comments are lines in your code that R ignores during execution. They are marked by the `#` symbol and are essential for:

1. Understanding your code later.
2. Helping others understand your code.
3. Documentation purposes.

Example:

```
# Multiplying 2 by 8

2 * 8
```

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

It's good practice to add a space after the `#` for readability.

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

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

1.2.4 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
```

1.2.5 Exercise 1.1.1

- **Explore RStudio**: Open RStudio and familiarize yourself with the four panes.
- **Perform Calculations**: In the Source Pane, compute the following, adding comments where appropriate:
  - $2 + 6 - 12$
  - $4 \times 3 - 8$
  - $81 \div 6$
  - $16 \bmod 3$
  - 2^3
  - $(3 + 2) \times (6 - 4) + 2$

1.3 Experiment 1.2: Atomic Data Type and Variable Assignment in R

R works with several atomic data types:

- **Numeric**: Integers or doubles (e.g., `4`, `-2`, `4.7`, `-0.26`)
- **Character**: Text strings enclosed in quotes (e.g., `"Nigeria"`, `"Hello world"`)
- **Logical**: Boolean values (`TRUE`, `FALSE`)

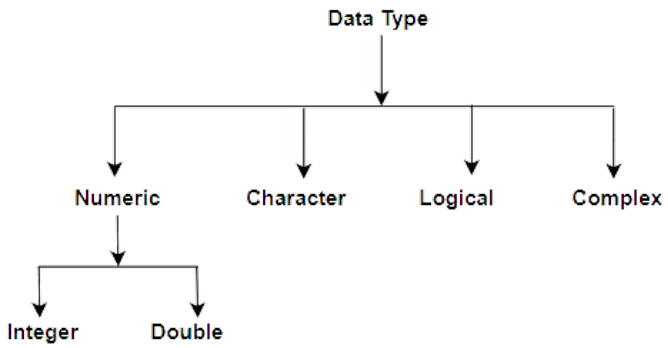


Figure 1.7: Data Types in R Programming

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"
```

1.3.1 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. This preference is largely based on convention and readability, as it helps keep your code clean and consistent[2].

Let's walk through a few examples to see variable assignment 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 that you can reuse or modify later in your code. For instance, if you want to check the value of `number`, just type:

```
number
```

[2]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.

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

… and R will display the stored value.

> **Tip**
>
> If you're using a Windows, a quick way to type the assignment operator `<-` is by pressing `ALT + _`, while on a Mac, you can use `Option + _`. This shortcut can save you 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.

1.3.2 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).

Quick Tips

- **Name Your Variables Clearly**: Choose names that describe the data they hold, 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 you need it, 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 always reassign or update your variables as you go!

1.3.3 Exercise 1.2.1: Acceptable vs. Unacceptable Variable Names

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 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 of Variable Names

Acceptable Variable Names	Unacceptable Variable Names
`health.status`	`health(status)`
`covid_19_cases`	`covid-19-cases`
`budget2024`	`2024budget`
`sales_price_2024`	`sales price 2024`

Discussion Questions

Comparison of Variable Naming Conventions
Acceptable vs. Unacceptable Variable Names

Acceptable Variable Names	Unacceptable Variable Names
health.status	health(status)
covid_19_cases	covid-19-cases
budget2024	2024budget
sales_price_2024	sales price 2024

Data Type Conversion in R
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)

1. **Periods and Underscores**: Why are periods (`.`) and underscores (`_`) commonly used in acceptable variable names instead of symbols like hyphens or spaces?
2. **Special Characters**: What happens if you use special characters like parentheses (`()`) in a variable name? Why does R disallow these?
3. **Starting with Letters**: Why is it important to start a variable name with a letter rather than a number?

Reflect on these questions and write down your answers in a few sentences for each. Use these answers as a guide to create variable names that follow R's rules and make your code easy to understand.

1.3.4 Data Type Conversions

Sometimes you need to convert data from one type to another, known as **typecasting**. Use the `as.` functions. The following table shows examples of those functions:

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)`

Suppose you have:

```
weight <- "64.45"
```



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

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

Convert `weight` to numeric:

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

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

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

Handling `NA` Results

If R can't convert a value, it returns `NA` (Not Available). This often happens when:

- Converting a character string that contains letters or symbols to numeric.
- Converting non-boolean strings to logical.

```r
height <- "161.5 cm"

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

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

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

```r
smiling_face <- "No"

as.logical(`smiling_face`) # Returns NA
```

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

1.3.5 Exercise 1.2.2

Determine the classes of the following variables and convert them if necessary:

```r
age <- 15

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

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

```r
diabetic_status <- "No"

class(diabetic_status) # What is the class?
```

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

```
five_less_than_2 <- FALSE
```

```
class(five_less_than_2) # What is the class?
```

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

```
weight <- "60.4 kg"
```

```
class(weight) # What is the class?
```

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

```
# Can you convert weight to numeric?
```

```
smile_face <- "FALSE"
```

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

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

```
# What happens if you convert smile_face to logical?
```

1.4 Experiment 1.3: Conditional Statements in R

Conditional statements allow your program to make decisions based on certain conditions. The primary constructs are `if`, `else if`, and `else`.

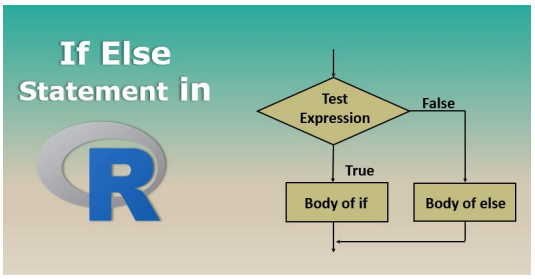


Figure 1.8: If-Else Statement in R Programming

1.4.1 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"
```

1.4.2 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"
```

1.4.3 The `else if` Statement

For situations with multiple conditions to check sequentially, `else if` can be used. 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 (`&`)
- OR (`|`)
- NOT (`!`)

Example using AND (`&`):

```
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 (`|`):

```
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"
```

1.4.4 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:

```r
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.

1.4.4.1 Example: Day of the Week Activities Using Character Switching

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

```
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"`.

1.4.4.2 Example: Mapping Codes to Descriptions Using Character Switching

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

```
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"
)

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.

1.4.4.3 Example: Simple Calculator Using Numeric Switching

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

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

# Use switch() to perform the selected operation
```

```r
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.

1.4.5 Exercise 1.3.1

Task 1

What is the output of the following code?

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

```
#> [1] "Even"
```

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`

1.4.6 Exercise 1.3.2: 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!

1.5 Additional R Learning Resources

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

- YaRrr! The Pirate's Guide to R by N*athaniel D. Phillips*
- R for Data Science by Hadley Wickham, Mine Çetinkaya-Rundel, and Garrett Grolemund.
- R for Data Science: Exercise Solutions by Jeffrey B. Arnold
- Big Book of R by Oscar Baruffa
- Posts you might have missed!

1.6 Summary

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

- **Navigating the RStudio Interface**
  You learned how to use RStudio's four main panes to write, execute, and manage your R code effectively.
- **Performing Basic Calculations**
  You practiced using R for arithmetic operations, understood operator precedence, and learned how to use mathematical functions.
- **Understanding Atomic Data Types**
  You explored numeric, character, and logical data types, and learned how to identify and convert between them.
- **Assigning Variables**
  You mastered variable assignment, followed naming conventions, and performed operations using variables.
- **Constructing Conditional Statements**
  You learned how to control the flow of your programs using `if`, `else if`, and `else` statements, and how to use logical operators.

As you move forward in this book, these foundational skills will be invaluable. In the next lab, we'll delve into R's basic data structures, such as vectors, matrices, and data frames, which are essential for data manipulation and analysis.

Keep practicing, and don't hesitate to revisit this lab if you need a refresher. Happy coding!

2 Understanding Data Structures

In this Lab 2, we'll explore the fundamental data structures that are essential for data analysis in R: `vectors`, `matrices`, `data frames`, and `lists`. Mastering these structures will enable you to handle data efficiently and perform various operations crucial for statistical analysis and data science tasks.

By the end of 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 Lab 2, you'll build a solid foundation in handling data structures in R, which is crucial for more advanced data analysis and programming tasks.

2.1 Introduction

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

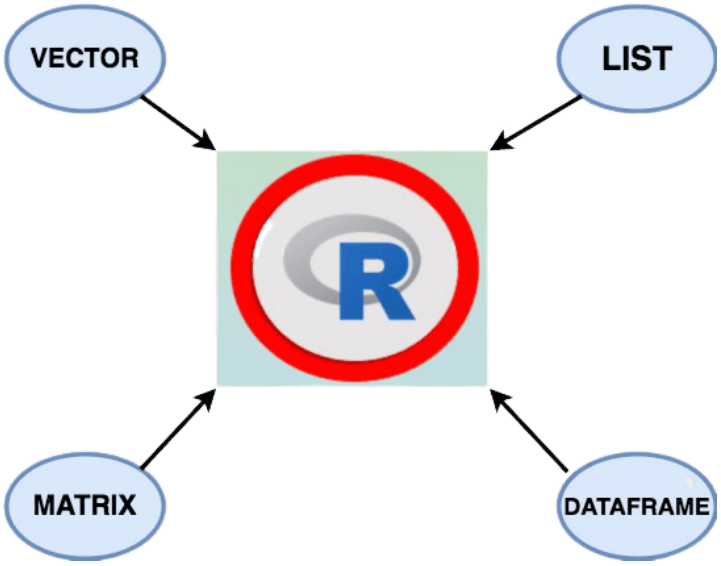


Figure 2.1: Data Structures in R Programming

2.2 Experiment 2.1: Vectors

A vector is a one-dimensional array that holds elements of the same data type. This is the most basic and frequently used data structure in R.

2.2.1 Creating a Vector

To create a vector, use the `c()` function:

```
gender <- c("Male", "Female", "Female", "Male", "Female", "Male")
```

Adding a space after every comma in `c()` makes your code more readable:

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

covid_confirmed

#>  [1] 31 30 37 25 33 34 26 32 23 45
```

You can check the class of the vector using the class() function:

```
class(covid_confirmed)

#> [1] "numeric"
```

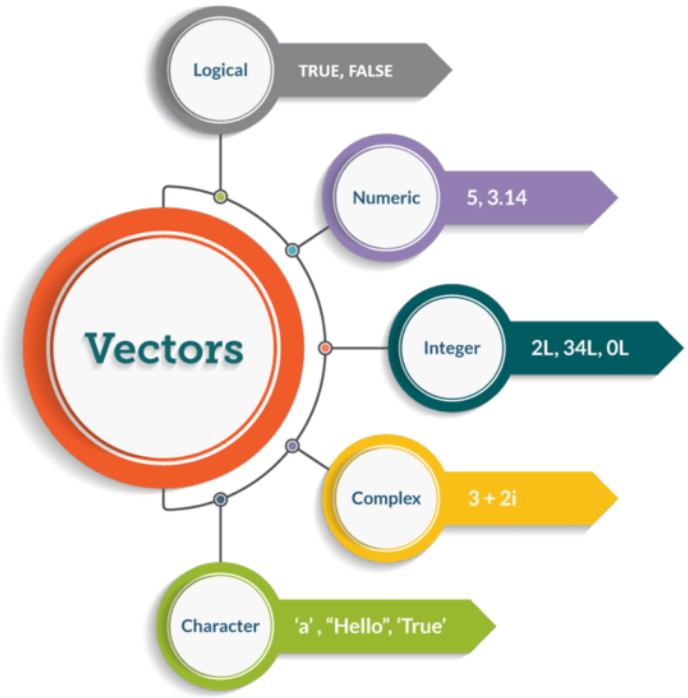


Figure 2.2: Types of Vectors in R Programming

2.2.2 Factor vectors

A categorical variable where each level is a category will be of type `factor`. For example, gender is a categorical variable with two levels: "Male" or "Female".

You can create a factor vector directly using the `factor()` function:

```
gender_factor <- factor(c("Male", "Female", "Female", "Male", "Female"))

gender_factor
```

```
#> [1] Male   Female Female Male   Female
#> Levels: Female Male
```

Check the class and levels:

```
class(gender_factor) # Returns "factor"
```

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

```
levels(gender_factor) # Returns "Female" "Male"
```

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

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

```
gender <- c("Male", "Female", "Female", "Male", "Female")

gender_factor <- as.factor(gender)

gender_factor
```

```
#> [1] Male   Female Female Male   Female
#> Levels: Female Male
```

```
class(gender_factor)
```

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

```
levels(gender_factor)
```

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

2.2.3 Length of a vector

Find the length of a vector using the `length()` function:

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

length(covid_confirmed)

#> [1] 10
```

2.2.4 Arithmetic Operations with Vectors

Operations with vector are performed element-wise.

```
egg_weight1 <- c(59, 56, 61, 68, 52, 53, 69, 54, 57, 51)

egg_weight2 <- c(56, 51, 69, 52, 57, 68, 61, 54, 59, 53)

total_weight <- egg_weight1 + egg_weight2

total_weight

#>  [1] 115 107 130 120 109 121 130 108 116 104
```

2.2.5 Vector selection

To select elements of a vector, use square brackets `[ ]` and indicate the index of elements to select. R indexing starts at 1.

For example:

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

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

Access the first element:

```
weekday[1]

#> [1] "Monday"
```

Access the second element:

```
weekday[2]
```

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

2.2.6 Exercise 2.1.1: Vector Selection

Given the quiz scores of 13 students:

`10, 15, 10, 9, 18, 16, 14, 12, 16, 13, 15, 20, 17.`

- Create a vector named `score` containing the data.
- Access individual scores of the 1st, 5th, and 10th students.
- Access them all together.

2.3 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. Figure 2.3 illustrates how matrices are typically represented in mathematics.

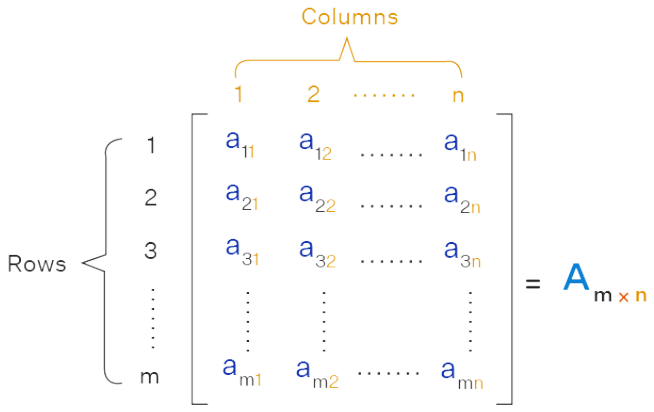


Figure 2.3: Matrix Representation in Linear Algebra

2.3.1 Creating Matrices

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

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

where:

- `data`: Elements to arrange in the matrix.
- `nrow`: Number of rows.
- `ncol`: Number of columns.
- `byrow`: Fill matrix by rows if `TRUE`.

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
```

2.3.2 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
```

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

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

2.3.3 Arithmetic Operation in Matrices

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

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

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

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

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

Addition

```
A + B
```

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

Multiplication

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

```
A %*% B
```

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

2.3.4 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!

2.3.5 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!

2.4 Experiment 2.3: Data frame

A data frame is a versatile table-like structure, allowing columns of different data types. It has variables as columns and observations as rows, similar to a spreadsheet or a SQL table.

2.4.1 Creating a Data Frame

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

```
demographic_data <- data.frame(
  age = c(16, 18, 13, 17, 22),
  gender = c("Female", "Female", "Male", "Female", "Male"),
  bank_account = c(TRUE, FALSE, FALSE, TRUE, FALSE)
)

demographic_data
```

```
#>   age gender bank_account
#> 1  16 Female         TRUE
#> 2  18 Female        FALSE
#> 3  13   Male        FALSE
#> 4  17 Female         TRUE
#> 5  22   Male        FALSE
```

The number `1 2 3 4 5` at the left hand side on your `console` are `row` labels. Also, each column in a data frame is a vector.

Example: COVID 19 Data Frame

Create a data frame with columns `states`, `confirmed cases`, `recovered cases` and `death cases`.

```
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
```

2.4.2 Exploring Data Frames

When working with large datasets, it's useful to show only part of the data.

1. `head()`: Shows the first observations.

2. `tail()`: Shows the last observations.

Both `head()` and `tail()` print a top line called `header`, which contains the names of the different variables in your data set.

Another method that is often used to get a rapid overview of your dataset is the function `str()`.

3. `str()`: shows you the structure of your dataset. The structure of a data frame tells you:

- The total number of observations
- The total number of variables
- A full list of the variables names
- The first observations

Applying the `str()` function will often be the first thing that you do when receiving a new dataset. It is a great way to get more insight in your dataset before diving into the real analysis.

4. `names()`: Prints each column name.

5. `nrow()`: Returns the number of rows.

6. `ncol()`: Returns the number of columns.

7. `dim()`: Returns the number of rows and columns.

8. `View()`: Opens a spreadsheet-style data viewer (in RStudio).

9. `summary()`: Returns summary statistics of all columns.

Consider the following vectors:

```r
set.seed(2021) # For 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
```

Create a data frame:

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

2.4.3 Explore the data

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:

```
names(medical_data)
```

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

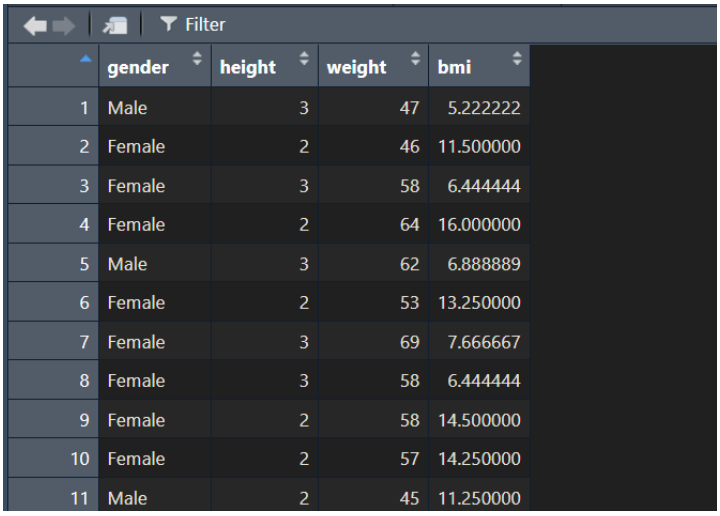
You can also use:

```
colnames(medical_data)
```

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

View the dataset (in RStudio):

```
View(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
7	Female	3	69	7.666667
8	Female	3	58	6.444444
9	Female	2	58	14.500000
10	Female	2	57	14.250000
11	Male	2	45	11.250000

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

Descriptive statistics:

```
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
```

2.4.4 Built-in Datasets

There are several ways to find the included datasets in R. Using `data()` will give you a list of available dataset.

```
data()
```

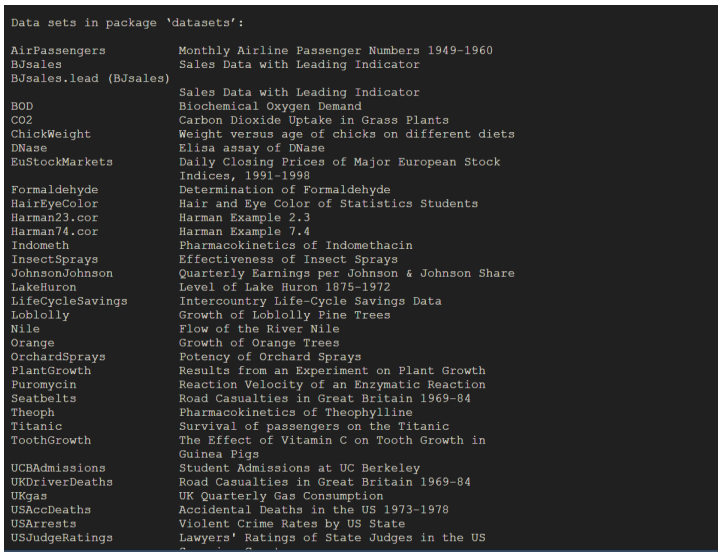


Figure 2.5: Sample Datasets Available in the R 'datasets' Package

For example, to load the built-in dataset `iris`, use:

```
data("iris")

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
```

and to load the built-in dataset `airquality`, use:

```
data("airquality")

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
```

To get help on a built-in dataset, such as `airquality`, use:

```
?airquality
```

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

2.4.5 Subsetting Data Frames

Every column in a data frame has a name and if you can recall, we can print the names attribute of a data frame, `iris`, by using:

```
names(iris)
```

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

and to access a specific column in a data frame by name, you will use the `$` operator in the form of `df$colname`, where `df` is the name of the data frame and `colname` is the name of the column you are interested in. This operation will then return the column you want as a vector.

Access specific columns using the $ operator

Use the `$` operator to get a vector of `Sepal.Length` from the `iris` data frame:

```
iris$Sepal.Length
```

```
#>   [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4 5.1
#>  [19] 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5 4.9 5.0
#>  [37] 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0 6.4 6.9 5.5
#>  [55] 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8 6.2 5.6 5.9 6.1
#>  [73] 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4 6.0 6.7 6.3 5.6 5.5
#>  [91] 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7 6.3 5.8 7.1 6.3 6.5 7.6 4.9 7.3
#> [109] 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7 6.0 6.9 5.6 7.7 6.3 6.7 7.2
#> [127] 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7 6.3 6.4 6.0 6.9 6.7 6.9 5.8 6.8
#> [145] 6.7 6.7 6.3 6.5 6.2 5.9
```

Use the `$` operator to get a vector of `Species` from the `iris` data frame:

```
iris$Species
```

```
#>   [1] setosa     setosa     setosa     setosa     setosa     setosa
#>   [7] setosa     setosa     setosa     setosa     setosa     setosa
#>  [13] setosa     setosa     setosa     setosa     setosa     setosa
#>  [19] setosa     setosa     setosa     setosa     setosa     setosa
#>  [25] setosa     setosa     setosa     setosa     setosa     setosa
#>  [31] setosa     setosa     setosa     setosa     setosa     setosa
#>  [37] setosa     setosa     setosa     setosa     setosa     setosa
#>  [43] setosa     setosa     setosa     setosa     setosa     setosa
#>  [49] setosa     setosa     versicolor versicolor versicolor versicolor
#>  [55] versicolor versicolor versicolor versicolor versicolor versicolor
#>  [61] versicolor versicolor versicolor versicolor versicolor versicolor
#>  [67] versicolor versicolor versicolor versicolor versicolor versicolor
#>  [73] versicolor versicolor versicolor versicolor versicolor versicolor
#>  [79] versicolor versicolor versicolor versicolor versicolor versicolor
#>  [85] versicolor versicolor versicolor versicolor versicolor versicolor
#>  [91] versicolor versicolor versicolor versicolor versicolor versicolor
#>  [97] versicolor versicolor versicolor versicolor virginica  virginica
#> [103] virginica  virginica  virginica  virginica  virginica  virginica
#> [109] virginica  virginica  virginica  virginica  virginica  virginica
#> [115] virginica  virginica  virginica  virginica  virginica  virginica
#> [121] virginica  virginica  virginica  virginica  virginica  virginica
#> [127] virginica  virginica  virginica  virginica  virginica  virginica
```

```
#> [133] virginica  virginica  virginica  virginica  virginica  virginica
#> [139] virginica  virginica  virginica  virginica  virginica  virginica
#> [145] virginica  virginica  virginica  virginica  virginica  virginica
#> Levels: setosa versicolor virginica
```

Because the `$` operator returns a vector, you can easily calculate descriptive statistics on columns of a data frame by applying your favorite vector function (like `mean()`, `sd()`, or `table()`) to a column using `$`.

Let's calculate the mean of `Sepal.Length` with the`mean()` function and the frequency of each `Species` with the `table()`function in the `iris` data frame:

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

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

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

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

Access elements using `[row, column]`

Just like a matrix, you can access specific data in a data frame by using `[row, column]`, where rows and columns are vectors of integers.

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

Data Frame Slicing in R
Examples and Interpretations

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

2.4.6 Exercise 2.3.1: Subsetting a Dataframe

Using the `airquality` dataset:

- Examine the `airquality` dataset.
- Select the first three columns.
- Select rows `1-3` and columns `1` and `3`.
- Select rows `1-5` and column `1`.
- Select the first row.
- Select the first 6 rows .

2.5 Experiment 2.4: Lists

A list in R is like a container that can hold various elements, such as vectors, matrices, data frames, and even other lists.

2.5.1 Creating a List

Use the `list()` function to create a list:

```
my_list <- list(
  age = 19,
  gender = "Male",
  pass = TRUE
)
```

Here, `my_list` consists of three components:

- `age`: Numeric value.
- `gender`: Character string.
- `pass`: Logical value.

2.5.2 Accessing List Elements

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

```
my_list
```

```
#> $age
#> [1] 19
#>
#> $gender
#> [1] "Male"
#>
#> $pass
#> [1] TRUE
```

You can extract individual element in a list by using double square brackets `[[ ]]`. For example,

```
my_list[[1]] # Returns 19
```

```
#> [1] 19
```

```
my_list[["age"]] # Returns 19
```

```
#> [1] 19
```

Using single square brackets [ ] returns a list containing the element.

```
my_list[1]
```

```
#> $age
#> [1] 19
```

2.6 Summary

In this lab 2, you have acquired foundational skills in R's basic data structures:

- **Understanding** the characteristics and differences between vectors, matrices, data frames, and lists.
- **Creating and manipulating** these data structures effectively.
- **Accessing and modifying** data elements within each structure using appropriate indexing and functions.

These skills are essential for any data analysis or data science task in R, and they form the basis for more advanced topics that you will encounter as you continue learning. Congratulations on building this crucial foundation!

In the next lab, you'll explore how to write your own functions in R. Functions are a powerful tool that will help you streamline your code, automate tasks, and make your programs more efficient. Get ready to enhance your programming skills by learning how to create custom functions!

3 Writing Custom Functions

In this Lab 3, we will 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.

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 behavior 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 Lab 3, 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.

3.1 Introduction

Imagine you want to perform a task repeatedly, like squaring numbers or checking for missing values in a dataset. Instead of writing the same code again and again, you create a function—a reusable block of code that does this task for you! R already has many built-in functions such as such as `c()`, `mean()`, `print()`, `class()`, `length()`, but we can also write our own to perform tasks tailored to our needs.

Functions usually take in some form of data structure—like a value, vector, or dataframe—as an argument, process it, and return a result.

Figure 3.1: Core Functions in R Programming

3.1.1 Types of Functions

Function extends the functionality of R. Broadly, we can categorize functions into two types:

- **Built-in Functions**: These are pre-defined in R, such as `print()` and `mean()`.
- **User-defined Functions**: These are functions you create to perform specific tasks.

3.1.2 Why Write Your Own Function?

Creating your own functions has several advantages:

- **Code Reusability**: Functions promote code reuse and help you avoid repetition.

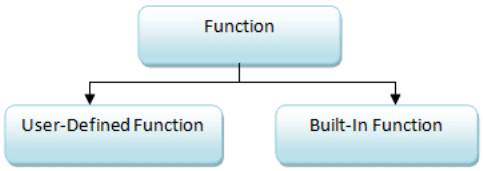


Figure 3.2: Types of Functions in R Programming

- **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.

3.1.3 When Should You Write a Function?

Consider writing a function whenever you find yourself copying and pasting a block of code more than twice. If you're repeating the same code, it's a good indication that a function could simplify your work.

3.2 Experiment 3.1: Creating a Function

There are three key steps to creating a new function:

- **Function Name**: Decide on a descriptive name for your function, such as `square_it`.
- **Function Arguments**: Specify the inputs your function will accept inside the `function()` keyword, for example, `function(x, y)`.
- **Function Body**: Write the R code that uses those arguments, enclosed within curly braces `{}`. This is where you'll define what the function does with the inputs—whether it's creating a plot, calculating a statistic, running a regression analysis, etc.

The general structure of a function is as follows:

```
function_name <- function(argument1, argument2, ...) {
  # Function body
  return(value)
}
```

> **Note**
>
> Note that the arguments can be any type of object—such as a scalar, matrix, dataframe, vector, or logical—and you don't need to define what they are beforehand.

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.

3.2.1 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.

3.2.2 Creating a Function to Square 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:

```
square_it <- function(x) {
  return(x^2)
}
```

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:

- Testing with 12:

```
square_it(12)
```

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

- Testing with 6:

```
square_it(x = 6)
```

```
#> [1] 36
```

This basic function highlights the usefulness of custom functions in R, enabling specific operations with minimal code.

3.2.3 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:

```
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:

```
check_NA(airquality)
```

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

- For the `iris` dataset:

```
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.

3.2.4 Data Frame Manipulation Using `switch()`

Suppose we have a data frame containing information about employees. We want to perform different operations on this data frame 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 Section 1.4.4 for a detailed tutorial and comprehensive understanding of the `switch()` function.

Step 1: Create a Sample Data Frame

```
library(tidyverse)

# Sample data frame
staff_data <- data.frame(
  EmployeeID = 1:6,
  Name = c("Alice", "Ebunlomo", "Festus", "TY Bello", "Fareedah", "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  TY Bello Data Science  82000
#> 5          5  Fareedah    Marketing  73000
#> 6          6 Testimony      Finance  78000
```

Step 2: Define the Function

```r
# Define the function
data_frame_operation <- function(data, operation = "filter" # or any of "summary", "add_colu
) {
  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: Use 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  TY Bello Data Science  82000  8200
#> 5          5  Fareedah    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  TY Bello 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
```

3.2.5 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.

3.2.6 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.

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.

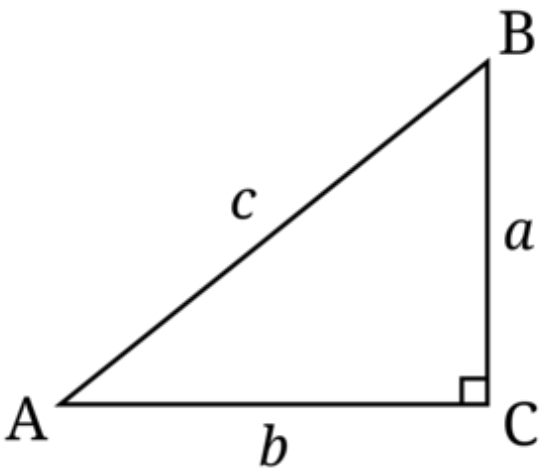


Figure 3.3: Geometric Representation: Right-Angled Triangle

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.

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

Based on the example in Section 3.2.4, 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:

```r
# 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

```r
# Perform the 'raise_salary' operation
data_frame_operation(staff_data, "---")
```

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

3.3 Experiment 3.2: Understanding Variable Scope Within Functions

When writing functions in R, it's crucial to understand how variables behave inside and outside those functions. This concept is known as **variable scope**. Variable scope determines where a variable is accessible in your code and how changes to variables within functions can affect variables outside of them.

3.3.1 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).

3.3.2 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: Local Variable

Let's look at an example to illustrate this:

```
greet <- function() {
  announcement <- "Hello from inside the function!"
  print(announcement)
}

greet() # This will print the announcement defined inside the function
```

```
#> [1] "Hello from inside the function!"
```

```
print(announcement) # This will result in an error because 'announcement' is not defined glo
```

```
#> Error: object 'announcement' not found
```

In this example, `announcement` is a local variable within the `greet` function. Trying to access `announcement` outside the function results in an error because it doesn't exist in the global environment.

Example: Global Variable Access

Functions in R can access global variables unless there is a local variable with the same name:

```
announcement <- "Hello from the global environment!"

greet <- function() {
  print(announcement)
}

greet() # This will print "Hello from the global environment!"
```

```
#> [1] "Hello from the global environment!"
```

Here, the function `greet` accesses the global variable `announcement` because there is no local variable named `announcement` inside the function.

3.3.3 Variable Shadowing

If a local variable inside a function has the same name as a global variable, the local variable will **shadow** the global one within that function:

```
announcement <- "Hello from the global environment!"

greet <- function() {
  announcement <- "Hello from inside the function!"
  print(announcement)
}

greet() # Prints: Hello from inside the function!
```

```
#> [1] "Hello from inside the function!"
```

```
print(announcement) # Prints: Hello from the global environment!
```

```
#> [1] "Hello from the global environment!"
```

In this case, the `announcement` variable inside `greet` is local and doesn't affect the global `announcement` variable.

3.4 Summary

In this lab, you have developed essential skills in creating custom functions in R:

- **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!

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 & Workflows

In this Lab 4, we will explore essential practices that will enhance your efficiency and effectiveness as an R programmer. You'll learn how to extend R's capabilities by installing and loading packages, ensure that your analyses are reproducible using RStudio Projects, proficiently import and export datasets in various formats, and handle missing data responsibly. These skills are crucial 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.

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 to extend the functionality of R for your data analysis tasks.
- **Ensure Reproducibility with R and RStudio Projects**
  Set up and manage RStudio Projects to organize 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 of Various Formats**
  Import data into R from different file types such as CSV, Excel, SPSS, and more, using appropriate packages and functions. Export your data frames and analysis results to various formats for sharing or reporting.
- **Handle and Impute Missing Data**
  Identify missing values in your datasets, understand how they can impact your analyses, and apply appropriate techniques to handle and impute missing data effectively.

By completing Lab 4, you'll enhance your ability to manage and analyze data in R efficiently, ensuring that your work is organized, reproducible, and ready to share with others. These foundational skills will support your growth as a proficient R programmer and data analyst.

4.1 Introduction

In R, a **package** is a collection of functions, data, and compiled code that extends the basic functionality of R. Think of it as a toolkit for specific tasks or topics. For example, packages like `tidyr` and `janitor` are designed for data wrangling.

The place where these packages are stored on your computer is called a **library**. When you install a package, it gets saved in your library, making it easily accessible when needed.

4.2 Compiling R Packages from Source

To work effectively with packages in R, especially when compiling from source, certain tools are necessary depending on your operating system.

- **Windows:** Rtools is a collection of software needed to build R packages from source, including a compiler and essential libraries. Since Windows does not natively support code compilation, Rtools provides these capabilities. You can download Rtools from CRAN: https://cran.rstudio.com/bin/windows/Rtools/. After downloading and installing the version of Rtools that matches your R version, R will automatically detect it.

> **Note**
>
> To check your R version, run the following code in the console:
>
> `R.version`

- **Mac OS:** Unlike Windows, Mac OS users need the Xcode Command Line Tools, which provide similar compiling capabilities as Rtools. These tools include necessary libraries and compilers. 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 entering:

```
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
> ```

4.3 Experiment 4.1: Installing and Loading Packages

As you work in R, you'll often need additional functions that aren't included in the base installation. These come in the form of packages, which you can easily install and load into your R environment.

4.3.1 Installing Packages from CRAN

The Comprehensive R Archive Network (CRAN) hosts thousands of packages. To install a package from CRAN, use `install.packages()`:

```
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'`).

4.3.2 Installing Packages from External Repositories

Packages not available on CRAN can be installed from external sources like GitHub. First, install a helper package like `devtools` or `remotes`:

```
install.packages("devtools")
# or
install.packages("remotes")
```

Then, to install a package from GitHub, for example `fakir` package, use:

```
devtools::install_github("ThinkR-open/fakir")
# or
remotes::install_github("ThinkR-open/fakir")
```

You can also install development versions of packages using these helper packages. For instance:

```
remotes::install_github("datalorax/equatiomatic")
```

4.3.3 Loading Installed 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
```

Executing this single line of code loads the core `tidyverse` packages, which are essential tools for almost every data analysis project[1].

Other installed packages can also be loaded in such manner:

```
library(janitor)
```

```
#>
#> Attaching package: 'janitor'
```

```
#> The following objects are masked from 'package:stats':
#>
#>     chisq.test, fisher.test
```

```
library(bulkreadr)
```

```
#> Welcome to bulkreadr package! To learn more, please run:
#> browseURL("https://gbganalyst.github.io/bulkreadr")
#> to visit the package website.
```

```
#>
#> Attaching package: 'bulkreadr'
```

```
#> The following object is masked from 'package:janitor':
#>
#>     convert_to_date
```

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.

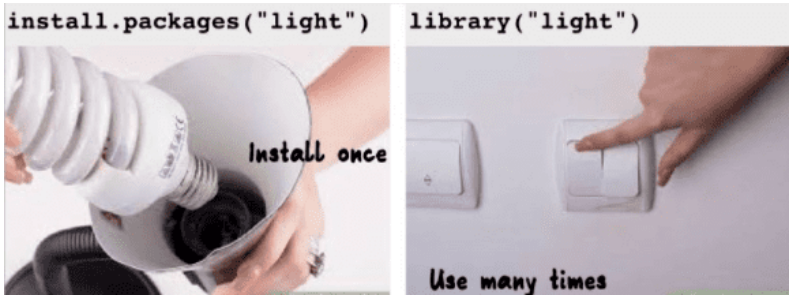


Figure 4.1: Installing vs. Loading Packages in R

4.3.4 Using Functions from a Package

There are two main ways to call a function from a package:

1. Load the package and call the function directly:

```
library(janitor)

clean_names(data_frame)
```

2. Use the `::` operator to call a function without loading the package:

```
janitor::clean_names(data_frame)
```

Tip

Using `package::function()` is helpful because it makes your code clearer about where the function comes from, especially if multiple packages have functions with the same name.

Example:

Let's see how to load a package and use one of its functions. We'll use the `clean_names()` function from the `janitor` package to clean column names in a data frame.

```
library(janitor)
```

Now, let's clean the column names of iris data frame:

```
clean_names(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
#> 7            4.6         3.4          1.4         0.3     setosa
#> 8            5.0         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
#> 11           5.4         3.7          1.5         0.2     setosa
#> 12           4.8         3.4          1.6         0.2     setosa
#> 13           4.8         3.0          1.4         0.1     setosa
#> 14           4.3         3.0          1.1         0.1     setosa
#> 15           5.8         4.0          1.2         0.2     setosa
#> 16           5.7         4.4          1.5         0.4     setosa
#> 17           5.4         3.9          1.3         0.4     setosa
#> 18           5.1         3.5          1.4         0.3     setosa
#> 19           5.7         3.8          1.7         0.3     setosa
#> 20           5.1         3.8          1.5         0.3     setosa
#> 21           5.4         3.4          1.7         0.2     setosa
#> 22           5.1         3.7          1.5         0.4     setosa
#> 23           4.6         3.6          1.0         0.2     setosa
#> 24           5.1         3.3          1.7         0.5     setosa
#> 25           4.8         3.4          1.9         0.2     setosa
#> 26           5.0         3.0          1.6         0.2     setosa
#> 27           5.0         3.4          1.6         0.4     setosa
#> 28           5.2         3.5          1.5         0.2     setosa
#> 29           5.2         3.4          1.4         0.2     setosa
#> 30           4.7         3.2          1.6         0.2     setosa
#> 31           4.8         3.1          1.6         0.2     setosa
#> 32           5.4         3.4          1.5         0.4     setosa
#> 33           5.2         4.1          1.5         0.1     setosa
#> 34           5.5         4.2          1.4         0.2     setosa
#> 35           4.9         3.1          1.5         0.2     setosa
#> 36           5.0         3.2          1.2         0.2     setosa
#> 37           5.5         3.5          1.3         0.2     setosa
#> 38           4.9         3.6          1.4         0.1     setosa
```

```
#> 39          4.4         3.0          1.3         0.2     setosa
#> 40          5.1         3.4          1.5         0.2     setosa
#> 41          5.0         3.5          1.3         0.3     setosa
#> 42          4.5         2.3          1.3         0.3     setosa
#> 43          4.4         3.2          1.3         0.2     setosa
#> 44          5.0         3.5          1.6         0.6     setosa
#> 45          5.1         3.8          1.9         0.4     setosa
#> 46          4.8         3.0          1.4         0.3     setosa
#> 47          5.1         3.8          1.6         0.2     setosa
#> 48          4.6         3.2          1.4         0.2     setosa
#> 49          5.3         3.7          1.5         0.2     setosa
#> 50          5.0         3.3          1.4         0.2     setosa
#> 51          7.0         3.2          4.7         1.4 versicolor
#> 52          6.4         3.2          4.5         1.5 versicolor
#> 53          6.9         3.1          4.9         1.5 versicolor
#> 54          5.5         2.3          4.0         1.3 versicolor
#> 55          6.5         2.8          4.6         1.5 versicolor
#> 56          5.7         2.8          4.5         1.3 versicolor
#> 57          6.3         3.3          4.7         1.6 versicolor
#> 58          4.9         2.4          3.3         1.0 versicolor
#> 59          6.6         2.9          4.6         1.3 versicolor
#> 60          5.2         2.7          3.9         1.4 versicolor
#> 61          5.0         2.0          3.5         1.0 versicolor
#> 62          5.9         3.0          4.2         1.5 versicolor
#> 63          6.0         2.2          4.0         1.0 versicolor
#> 64          6.1         2.9          4.7         1.4 versicolor
#> 65          5.6         2.9          3.6         1.3 versicolor
#> 66          6.7         3.1          4.4         1.4 versicolor
#> 67          5.6         3.0          4.5         1.5 versicolor
#> 68          5.8         2.7          4.1         1.0 versicolor
#> 69          6.2         2.2          4.5         1.5 versicolor
#> 70          5.6         2.5          3.9         1.1 versicolor
#> 71          5.9         3.2          4.8         1.8 versicolor
#> 72          6.1         2.8          4.0         1.3 versicolor
#> 73          6.3         2.5          4.9         1.5 versicolor
#> 74          6.1         2.8          4.7         1.2 versicolor
#> 75          6.4         2.9          4.3         1.3 versicolor
#> 76          6.6         3.0          4.4         1.4 versicolor
#> 77          6.8         2.8          4.8         1.4 versicolor
#> 78          6.7         3.0          5.0         1.7 versicolor
#> 79          6.0         2.9          4.5         1.5 versicolor
#> 80          5.7         2.6          3.5         1.0 versicolor
#> 81          5.5         2.4          3.8         1.1 versicolor
```

```
#> 82          5.5         2.4          3.7         1.0 versicolor
#> 83          5.8         2.7          3.9         1.2 versicolor
#> 84          6.0         2.7          5.1         1.6 versicolor
#> 85          5.4         3.0          4.5         1.5 versicolor
#> 86          6.0         3.4          4.5         1.6 versicolor
#> 87          6.7         3.1          4.7         1.5 versicolor
#> 88          6.3         2.3          4.4         1.3 versicolor
#> 89          5.6         3.0          4.1         1.3 versicolor
#> 90          5.5         2.5          4.0         1.3 versicolor
#> 91          5.5         2.6          4.4         1.2 versicolor
#> 92          6.1         3.0          4.6         1.4 versicolor
#> 93          5.8         2.6          4.0         1.2 versicolor
#> 94          5.0         2.3          3.3         1.0 versicolor
#> 95          5.6         2.7          4.2         1.3 versicolor
#> 96          5.7         3.0          4.2         1.2 versicolor
#> 97          5.7         2.9          4.2         1.3 versicolor
#> 98          6.2         2.9          4.3         1.3 versicolor
#> 99          5.1         2.5          3.0         1.1 versicolor
#> 100         5.7         2.8          4.1         1.3 versicolor
#> 101         6.3         3.3          6.0         2.5  virginica
#> 102         5.8         2.7          5.1         1.9  virginica
#> 103         7.1         3.0          5.9         2.1  virginica
#> 104         6.3         2.9          5.6         1.8  virginica
#> 105         6.5         3.0          5.8         2.2  virginica
#> 106         7.6         3.0          6.6         2.1  virginica
#> 107         4.9         2.5          4.5         1.7  virginica
#> 108         7.3         2.9          6.3         1.8  virginica
#> 109         6.7         2.5          5.8         1.8  virginica
#> 110         7.2         3.6          6.1         2.5  virginica
#> 111         6.5         3.2          5.1         2.0  virginica
#> 112         6.4         2.7          5.3         1.9  virginica
#> 113         6.8         3.0          5.5         2.1  virginica
#> 114         5.7         2.5          5.0         2.0  virginica
#> 115         5.8         2.8          5.1         2.4  virginica
#> 116         6.4         3.2          5.3         2.3  virginica
#> 117         6.5         3.0          5.5         1.8  virginica
#> 118         7.7         3.8          6.7         2.2  virginica
#> 119         7.7         2.6          6.9         2.3  virginica
#> 120         6.0         2.2          5.0         1.5  virginica
#> 121         6.9         3.2          5.7         2.3  virginica
#> 122         5.6         2.8          4.9         2.0  virginica
#> 123         7.7         2.8          6.7         2.0  virginica
#> 124         6.3         2.7          4.9         1.8  virginica
```

```
#> 125          6.7         3.3          5.7         2.1  virginica
#> 126          7.2         3.2          6.0         1.8  virginica
#> 127          6.2         2.8          4.8         1.8  virginica
#> 128          6.1         3.0          4.9         1.8  virginica
#> 129          6.4         2.8          5.6         2.1  virginica
#> 130          7.2         3.0          5.8         1.6  virginica
#> 131          7.4         2.8          6.1         1.9  virginica
#> 132          7.9         3.8          6.4         2.0  virginica
#> 133          6.4         2.8          5.6         2.2  virginica
#> 134          6.3         2.8          5.1         1.5  virginica
#> 135          6.1         2.6          5.6         1.4  virginica
#> 136          7.7         3.0          6.1         2.3  virginica
#> 137          6.3         3.4          5.6         2.4  virginica
#> 138          6.4         3.1          5.5         1.8  virginica
#> 139          6.0         3.0          4.8         1.8  virginica
#> 140          6.9         3.1          5.4         2.1  virginica
#> 141          6.7         3.1          5.6         2.4  virginica
#> 142          6.9         3.1          5.1         2.3  virginica
#> 143          5.8         2.7          5.1         1.9  virginica
#> 144          6.8         3.2          5.9         2.3  virginica
#> 145          6.7         3.3          5.7         2.5  virginica
#> 146          6.7         3.0          5.2         2.3  virginica
#> 147          6.3         2.5          5.0         1.9  virginica
#> 148          6.5         3.0          5.2         2.0  virginica
#> 149          6.2         3.4          5.4         2.3  virginica
#> 150          5.9         3.0          5.1         1.8  virginica
```

Example:

Using a function without loading the package:

```
janitor::clean_names(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
#> 7            4.6         3.4          1.4         0.3     setosa
#> 8            5.0         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
#> 11          5.4         3.7          1.5         0.2     setosa
#> 12          4.8         3.4          1.6         0.2     setosa
#> 13          4.8         3.0          1.4         0.1     setosa
#> 14          4.3         3.0          1.1         0.1     setosa
#> 15          5.8         4.0          1.2         0.2     setosa
#> 16          5.7         4.4          1.5         0.4     setosa
#> 17          5.4         3.9          1.3         0.4     setosa
#> 18          5.1         3.5          1.4         0.3     setosa
#> 19          5.7         3.8          1.7         0.3     setosa
#> 20          5.1         3.8          1.5         0.3     setosa
#> 21          5.4         3.4          1.7         0.2     setosa
#> 22          5.1         3.7          1.5         0.4     setosa
#> 23          4.6         3.6          1.0         0.2     setosa
#> 24          5.1         3.3          1.7         0.5     setosa
#> 25          4.8         3.4          1.9         0.2     setosa
#> 26          5.0         3.0          1.6         0.2     setosa
#> 27          5.0         3.4          1.6         0.4     setosa
#> 28          5.2         3.5          1.5         0.2     setosa
#> 29          5.2         3.4          1.4         0.2     setosa
#> 30          4.7         3.2          1.6         0.2     setosa
#> 31          4.8         3.1          1.6         0.2     setosa
#> 32          5.4         3.4          1.5         0.4     setosa
#> 33          5.2         4.1          1.5         0.1     setosa
#> 34          5.5         4.2          1.4         0.2     setosa
#> 35          4.9         3.1          1.5         0.2     setosa
#> 36          5.0         3.2          1.2         0.2     setosa
#> 37          5.5         3.5          1.3         0.2     setosa
#> 38          4.9         3.6          1.4         0.1     setosa
#> 39          4.4         3.0          1.3         0.2     setosa
#> 40          5.1         3.4          1.5         0.2     setosa
#> 41          5.0         3.5          1.3         0.3     setosa
#> 42          4.5         2.3          1.3         0.3     setosa
#> 43          4.4         3.2          1.3         0.2     setosa
#> 44          5.0         3.5          1.6         0.6     setosa
#> 45          5.1         3.8          1.9         0.4     setosa
#> 46          4.8         3.0          1.4         0.3     setosa
#> 47          5.1         3.8          1.6         0.2     setosa
#> 48          4.6         3.2          1.4         0.2     setosa
#> 49          5.3         3.7          1.5         0.2     setosa
#> 50          5.0         3.3          1.4         0.2     setosa
#> 51          7.0         3.2          4.7         1.4 versicolor
```

```
#> 52          6.4         3.2          4.5         1.5 versicolor
#> 53          6.9         3.1          4.9         1.5 versicolor
#> 54          5.5         2.3          4.0         1.3 versicolor
#> 55          6.5         2.8          4.6         1.5 versicolor
#> 56          5.7         2.8          4.5         1.3 versicolor
#> 57          6.3         3.3          4.7         1.6 versicolor
#> 58          4.9         2.4          3.3         1.0 versicolor
#> 59          6.6         2.9          4.6         1.3 versicolor
#> 60          5.2         2.7          3.9         1.4 versicolor
#> 61          5.0         2.0          3.5         1.0 versicolor
#> 62          5.9         3.0          4.2         1.5 versicolor
#> 63          6.0         2.2          4.0         1.0 versicolor
#> 64          6.1         2.9          4.7         1.4 versicolor
#> 65          5.6         2.9          3.6         1.3 versicolor
#> 66          6.7         3.1          4.4         1.4 versicolor
#> 67          5.6         3.0          4.5         1.5 versicolor
#> 68          5.8         2.7          4.1         1.0 versicolor
#> 69          6.2         2.2          4.5         1.5 versicolor
#> 70          5.6         2.5          3.9         1.1 versicolor
#> 71          5.9         3.2          4.8         1.8 versicolor
#> 72          6.1         2.8          4.0         1.3 versicolor
#> 73          6.3         2.5          4.9         1.5 versicolor
#> 74          6.1         2.8          4.7         1.2 versicolor
#> 75          6.4         2.9          4.3         1.3 versicolor
#> 76          6.6         3.0          4.4         1.4 versicolor
#> 77          6.8         2.8          4.8         1.4 versicolor
#> 78          6.7         3.0          5.0         1.7 versicolor
#> 79          6.0         2.9          4.5         1.5 versicolor
#> 80          5.7         2.6          3.5         1.0 versicolor
#> 81          5.5         2.4          3.8         1.1 versicolor
#> 82          5.5         2.4          3.7         1.0 versicolor
#> 83          5.8         2.7          3.9         1.2 versicolor
#> 84          6.0         2.7          5.1         1.6 versicolor
#> 85          5.4         3.0          4.5         1.5 versicolor
#> 86          6.0         3.4          4.5         1.6 versicolor
#> 87          6.7         3.1          4.7         1.5 versicolor
#> 88          6.3         2.3          4.4         1.3 versicolor
#> 89          5.6         3.0          4.1         1.3 versicolor
#> 90          5.5         2.5          4.0         1.3 versicolor
#> 91          5.5         2.6          4.4         1.2 versicolor
#> 92          6.1         3.0          4.6         1.4 versicolor
#> 93          5.8         2.6          4.0         1.2 versicolor
#> 94          5.0         2.3          3.3         1.0 versicolor
```

```
#> 95          5.6         2.7          4.2         1.3 versicolor
#> 96          5.7         3.0          4.2         1.2 versicolor
#> 97          5.7         2.9          4.2         1.3 versicolor
#> 98          6.2         2.9          4.3         1.3 versicolor
#> 99          5.1         2.5          3.0         1.1 versicolor
#> 100         5.7         2.8          4.1         1.3 versicolor
#> 101         6.3         3.3          6.0         2.5  virginica
#> 102         5.8         2.7          5.1         1.9  virginica
#> 103         7.1         3.0          5.9         2.1  virginica
#> 104         6.3         2.9          5.6         1.8  virginica
#> 105         6.5         3.0          5.8         2.2  virginica
#> 106         7.6         3.0          6.6         2.1  virginica
#> 107         4.9         2.5          4.5         1.7  virginica
#> 108         7.3         2.9          6.3         1.8  virginica
#> 109         6.7         2.5          5.8         1.8  virginica
#> 110         7.2         3.6          6.1         2.5  virginica
#> 111         6.5         3.2          5.1         2.0  virginica
#> 112         6.4         2.7          5.3         1.9  virginica
#> 113         6.8         3.0          5.5         2.1  virginica
#> 114         5.7         2.5          5.0         2.0  virginica
#> 115         5.8         2.8          5.1         2.4  virginica
#> 116         6.4         3.2          5.3         2.3  virginica
#> 117         6.5         3.0          5.5         1.8  virginica
#> 118         7.7         3.8          6.7         2.2  virginica
#> 119         7.7         2.6          6.9         2.3  virginica
#> 120         6.0         2.2          5.0         1.5  virginica
#> 121         6.9         3.2          5.7         2.3  virginica
#> 122         5.6         2.8          4.9         2.0  virginica
#> 123         7.7         2.8          6.7         2.0  virginica
#> 124         6.3         2.7          4.9         1.8  virginica
#> 125         6.7         3.3          5.7         2.1  virginica
#> 126         7.2         3.2          6.0         1.8  virginica
#> 127         6.2         2.8          4.8         1.8  virginica
#> 128         6.1         3.0          4.9         1.8  virginica
#> 129         6.4         2.8          5.6         2.1  virginica
#> 130         7.2         3.0          5.8         1.6  virginica
#> 131         7.4         2.8          6.1         1.9  virginica
#> 132         7.9         3.8          6.4         2.0  virginica
#> 133         6.4         2.8          5.6         2.2  virginica
#> 134         6.3         2.8          5.1         1.5  virginica
#> 135         6.1         2.6          5.6         1.4  virginica
#> 136         7.7         3.0          6.1         2.3  virginica
#> 137         6.3         3.4          5.6         2.4  virginica
```

```
#> 138          6.4         3.1          5.5         1.8  virginica
#> 139          6.0         3.0          4.8         1.8  virginica
#> 140          6.9         3.1          5.4         2.1  virginica
#> 141          6.7         3.1          5.6         2.4  virginica
#> 142          6.9         3.1          5.1         2.3  virginica
#> 143          5.8         2.7          5.1         1.9  virginica
#> 144          6.8         3.2          5.9         2.3  virginica
#> 145          6.7         3.3          5.7         2.5  virginica
#> 146          6.7         3.0          5.2         2.3  virginica
#> 147          6.3         2.5          5.0         1.9  virginica
#> 148          6.5         3.0          5.2         2.0  virginica
#> 149          6.2         3.4          5.4         2.3  virginica
#> 150          5.9         3.0          5.1         1.8  virginica
```

4.4 Experiment 4.2: Data Analysis Reproducibility with R and RStudio Projects

Reproducibility is vital in data analysis. Using RStudio Projects helps you organize your work and ensures that your analyses can be easily replicated.

Figure 4.2: Key Aspects of Reproducibility in Research

4.4.1 Where Does Your Analysis Live?

The working directory is where R looks for files to load and where it saves output files. You can see your current working directory at the top of the console:

Figure 4.3: Console

Or by running:

```
getwd()
```

```
[1] "C:/Users/Ezekiel Adebayo/Desktop/stock-market"
```

If you need to change your working directory, you can use:

```
setwd("/path/to/your/data_analysis")
```

Alternatively, you can use the keyboard shortcut `Ctrl + Shift + H` in RStudio to choose your specific directory.

4.4.2 Paths and Directories

- **Absolute Paths**: These start from the root of your file system (e.g., `C:/Users/YourName/Documents/data`). Avoid using absolute paths in your scripts because they're specific to your computer.
- **Relative Paths**: These are relative to your working directory (e.g., `data/data.csv`). Using relative paths makes your scripts portable and easier to share.

4.4.3 RStudio Projects

An RStudio Project is an excellent way to keep everything related to your analysis—scripts, data files, figures—all in one organized place. When you set up a project, RStudio automatically sets the working directory to your project folder. This feature is incredibly helpful because it keeps file paths consistent and makes your work reproducible, no matter where it's opened. For example, here's a look at how an RStudio project might be organized, as shown in Figure 4.4.

4.4.3.1 Creating a New RStudio Project

Let's create a new RStudio project. You can do this by following these simple steps:

1. **Go to:** `File → New Project`
2. **Choose:** `Existing Directory`

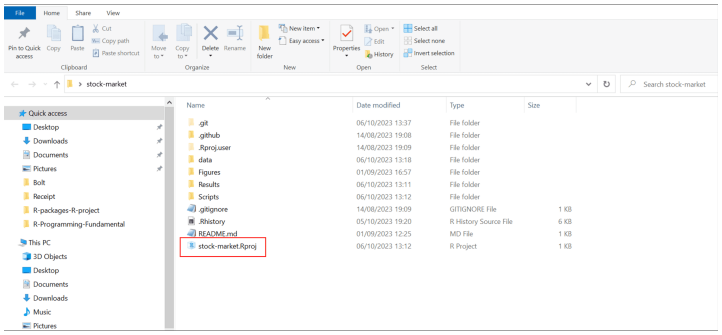


Figure 4.4: Organization of an R Project Directory

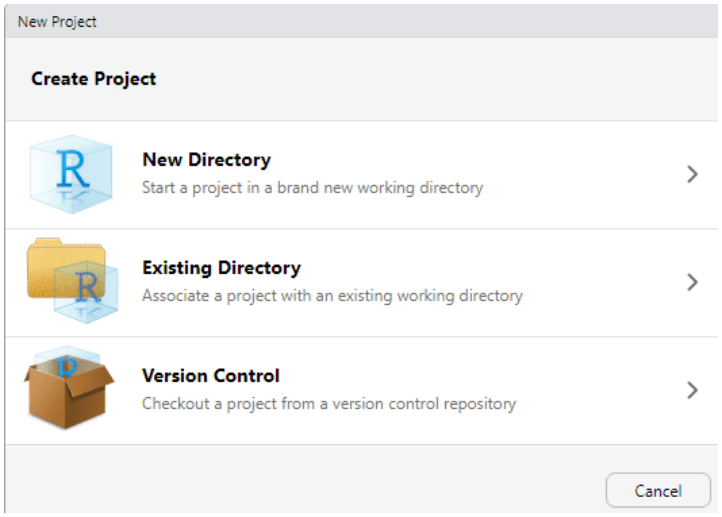


Figure 4.5: Creating a New Project in RStudio

3. **Select** the folder you want as your project's working directory.
4. **Click:** `Create Project`

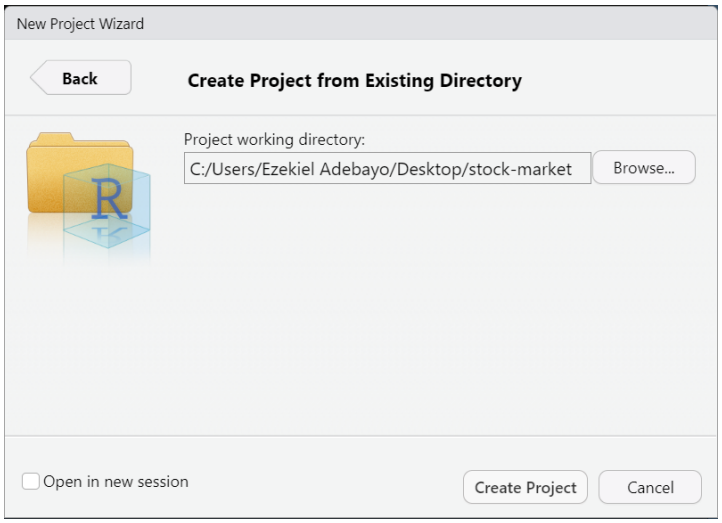


Figure 4.6: Creating a New R Project from an Existing Directory

Once you click "Create Project", you're all set! You'll be inside your new RStudio project.

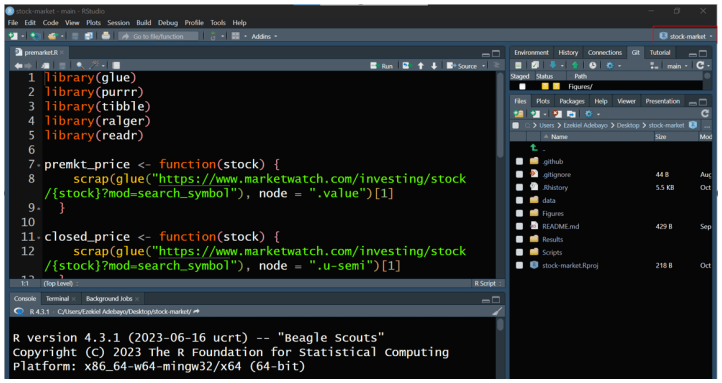


Figure 4.7: RStudio Project: Stock Market Price Scraper Using R

To open this project later, just click the `.Rproj` file in your project folder, and you'll be right back in the organized workspace you set up.

4.5 Experiment 4.3: Importing and exporting data in R

Data import and export are essential steps in data science. With R, you can bring in data from spreadsheets, databases, and many other formats, then save your processed results. Some of

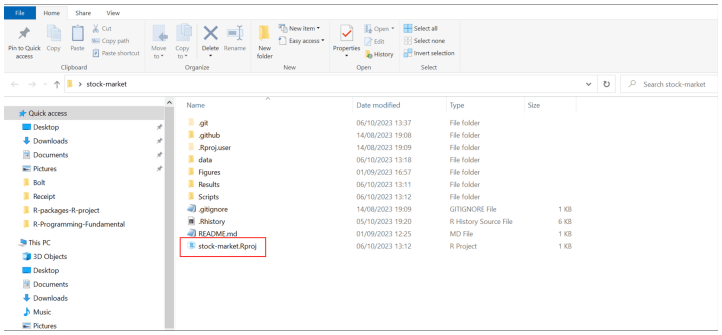


Figure 4.8: Organization of an R Project Directory

the popular R packages for data import are shown in Figure 4.9:

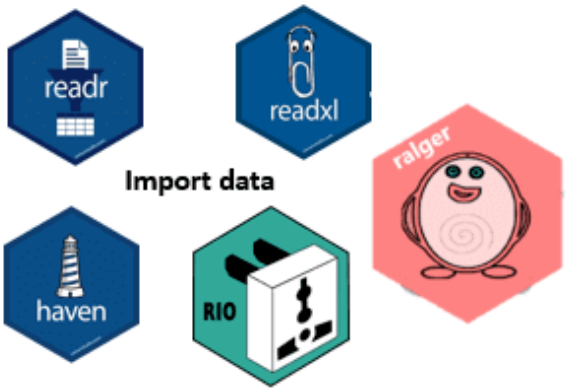


Figure 4.9: Data Import Packages in R

> **Note**
>
> Packages like `readr`, `readxl`, and `haven` are part of the tidyverse, so they come pre-installed with it—no need for separate installations. Here's a full list of tidyverse packages:

```
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"
```

> **Note**
>
> You don't need to install any of these packages individually since they're all included with the tidyverse installation.

4.5.1 Packages for Reading and Writing Data in R

R programming has some fantastic packages that make importing and exporting data simple and straightforward. Let's go through a few of the most commonly used packages and functions that you'll need to know when working with data in R.

`readr` Package

The `readr` package is your go-to for handling CSV files, which are very common in data analysis. Here are the two main functions you'll use:

- `read_csv()`: This function lets you import data from a CSV file into R as a data frame. Think of it as loading your data from a spreadsheet directly into R for analysis.
- `write_csv()`: Once you're done with your data analysis and want to save your results, this function exports your data frame to a CSV file. It's great for sharing your data or saving a backup!

`readxl` Package

The `readxl` package is specifically designed for Excel files. This is super useful if you're working with `.xlsx` files.

- `read_xlsx()`: Use this function to import an Excel file directly into R. It's similar to `read_csv()` but for Excel formats.

`writexl` Package

When you need to export your data to an Excel file, `writexl` is a handy package to have.

- `write_xlsx()`: This function allows you to export your data frame to an Excel file. Perfect for sharing your work with colleagues who prefer Excel!

`haven` Package

The `haven` package is here to help when you're working with data from statistical software like SPSS and Stata.

- `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.
- `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.

`rio` Package

The `rio` package is like the Swiss Army knife of data import and export. It can handle multiple file types, so you don't need to remember specific functions for each format.

- `import()`: Use this function to import data from nearly any file type—CSV, Excel, SPSS, Stata, you name it.
- `export()`: Just like `import()`, this function can export your data frame to a wide variety of formats.

> Tip
>
> For more details on all the options available with `rio` package, check out the rio documentation.

4.5.2 Working with Projects in RStudio

When we're working on projects in R, especially those that involve reading and writing data, it's best to set up an **RStudio project**. An RStudio project automatically manages your working directory, making sure you're always in the right folder, which is essential when importing and exporting data. By using relative paths (like `data/my_file.csv`), your code becomes more portable and less dependent on your specific folder structure.

In practice, this setup will save you from having to manually set or adjust your working directory every time you start R. So when you're working in a project, loading and saving files becomes as easy as referencing their relative file paths.

Example: Importing CSV Data

Let's practice importing data using the `gapminder.csv` file.

Instructions:

1. **Create a Directory**: Make a new folder on your desktop called `Experiment 4.2`.
2. **Download Data**: Go to to Google Drive to download the `r-data` folder (see Section A.1 for more information). Once downloaded, unzip the folder and move it into your `Experiment 4.2` folder.
3. **Create an RStudio Project**: Now, open RStudio and set up a new project:
   - Go to **File > New Project**.
   - Select **Existing Directory** and browse to your `Experiment 4.2` folder.
   - This project setup will organize your work and keep everything related to this experiment in one place.

Your project structure should resemble what's shown in Figure 4.10:

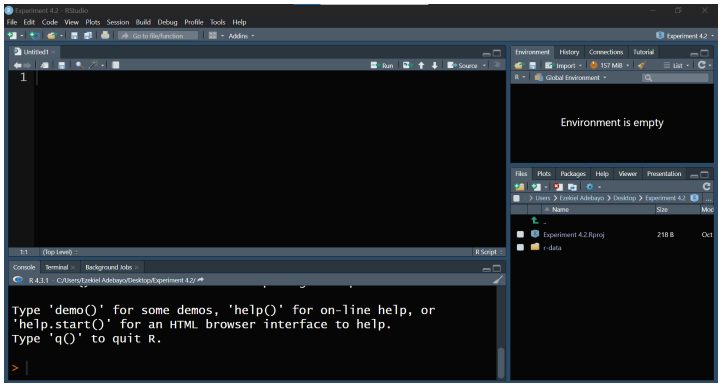


Figure 4.10: Starting a New R Project in RStudio

Now, let's import the `gapminder.csv` file from the `r-data` folder into R. We'll use the `tidyverse` package for easy data manipulation and visualization.

```
library(tidyverse)
library(readxl) # For reading Excel files
library(haven) # For reading SPSS/Stata/SAS files

# 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.
```

```
# Exploring the data
names(gapminder)
```

```
#> [1] "country"   "continent" "year"      "lifeExp"   "pop"       "gdpPercap"
```

```
dim(gapminder)
```

```
#> [1] 1704    6
```

```
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.
```

```
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, you should see your dataset loaded into RStudio, ready for exploration. Your script should look like the one in Figure 4.11:

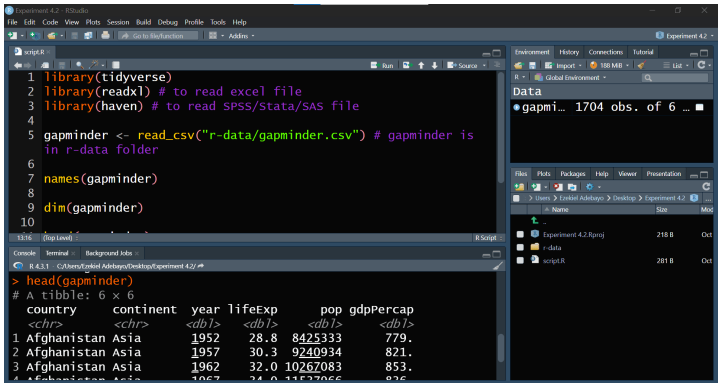


Figure 4.11: Loading and Exploring Data in RStudio

With this setup, you're ready to dive into analyzing the `gapminder` data in R! If you're working with other file formats like Excel or SPSS, check out Section 4.5.1 for detailed instructions on how to import these file types into R.

Example: Exporting Data Frames

Exporting data from R is straightforward. For example, to export the `gapminder` data frame as an Excel file:

```
library(writexl)

write_xlsx(gapminder, "gapminder_nigeria.xlsx")
```

This will create an Excel file in your project directory as shown in Figure 4.12:

4.6 Experiment 4.4: Dealing with Missing Data in R

Missing data is a common issue in data analysis. Recognizing and handling missing values appropriately is crucial for accurate analyses. R provides several functions to help you deal with missing data.

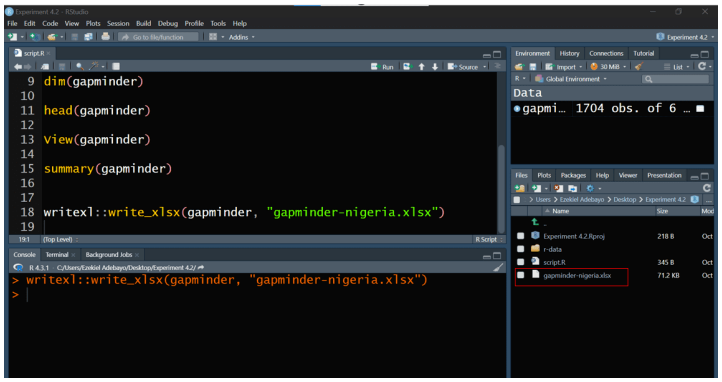


Figure 4.12: Exporting Data to Excel in RStudio using `write_xlsx()`function

4.6.1 Recognizing 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 `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, we have missing values for Francisca's age and Fatima's salary. We can use `anyNA()` to check for any missing values:

```
anyNA(salary_data)
```

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

This indicates that there are missing values in the data frame. You can also check for missing values in specific columns:

```
anyNA(salary_data$Age)
```

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

And for the `Name` column:

```
anyNA(salary_data$Name)
```

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

- **`complete.cases()`**: Returns a logical vector indicating which rows (cases) are complete, meaning they have no missing values. 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
```

```
complete.cases(salary_data)
```

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

This indicates that rows 2 and 3 contain missing values, meaning rows 2 and 3 are not complete.

If you want to check if there are any missing values across all rows (i.e., if any case is incomplete):

```
anyNA(complete.cases(salary_data))
```

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

4.6.2 Summarizing 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:

```
airquality_data <- tibble::tribble(~Ozone, ~Soar.R, ~Wind, ~Temp, ~Month, ~Day, 41, 190, 7.4
```

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 Soar.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. First, install and load the package:

```
install.packages("inspectdf")
```

```
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 Soar.R       2    20
#> 4 Wind         0     0
#> 5 Temp         0     0
#> 6 Day          0     0
```

4.6.3 Handling Missing Values

There are several strategies to handle missing data:

a. Removing Missing Values:

You can remove rows with missing values using `na.omit()`:

```
cleaned_data <- na.omit(airquality_data)
cleaned_data
```

```
#> # A tibble: 6 x 6
#>   Ozone Soar.R  Wind  Temp Month   Day
#>   <dbl>  <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1    41    190   7.4    67     5     1
#> 2    36    118   8      72     5     2
#> 3    12    149  12.6    74     5     3
```

```
#> 4    23    299   8.6    65     5     7
#> 5    19     99  13.8    59     5     8
#> 6     8     19  20.1    61     5     9
```

b. Imputation:

Alternatively, you can fill in missing values with estimates like the `mean`, `median`, or `mode`.

For the `Ozone` column:

```
airquality_data$Ozone[is.na(airquality_data$Ozone)] <- mean(airquality_data$Ozone, na.rm = TR
```

For the `Month` column:

```
airquality_data$Month[is.na(airquality_data$Month)] <- median(airquality_data$Month, na.rm =
```

To see the result of the imputation, we can call out the `airquality_data`:

```
airquality_data
```

```
#> # A tibble: 10 x 6
#>    Ozone Soar.R  Wind  Temp Month   Day
#>    <dbl>  <dbl> <dbl> <dbl> <dbl> <dbl>
#>  1  41      190   7.4    67     5     1
#>  2  36      118   8      72     5     2
#>  3  12      149  12.6    74     5     3
#>  4  18      313  11.5    62     5     4
#>  5  23.1     NA  14.3    56     5     5
#>  6  28       NA  14.9    66     5     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  23.1    194   8.6    69     5    10
```

c. Using packages:

For more advanced imputation methods, you can use packages like `mice` or `Hmisc`. Additionally, the bulkreadr package simplifies the process:

```
library(bulkreadr)
fill_missing_values(airquality_data, selected_variables = c("Ozone", "Soar.R"), method = "mea
```

```
#> # A tibble: 10 x 6
#>    Ozone Soar.R  Wind  Temp Month   Day
#>    <dbl>  <dbl> <dbl> <dbl> <dbl> <dbl>
#>  1  41     190    7.4    67     5     1
#>  2  36     118    8      72     5     2
#>  3  12     149   12.6    74     5     3
#>  4  18     313   11.5    62     5     4
#>  5  23.1   173.  14.3    56     5     5
#>  6  28     173.  14.9    66     5     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  23.1   194    8.6    69     5    10
```

You can also use `fill_missing_values()` to impute all the missing values in the data frame:

```
fill_missing_values(airquality_data, method = "median")
```

```
#> # A tibble: 10 x 6
#>    Ozone Soar.R  Wind  Temp Month   Day
#>    <dbl>  <dbl> <dbl> <dbl> <dbl> <dbl>
#>  1  41     190    7.4    67     5     1
#>  2  36     118    8      72     5     2
#>  3  12     149   12.6    74     5     3
#>  4  18     313   11.5    62     5     4
#>  5  23.1   170.  14.3    56     5     5
#>  6  28     170.  14.9    66     5     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  23.1   194    8.6    69     5    10
```

4.6.4 Exercise 4.1: Medical Insurance Data

In this exercise, you'll explore the `Medical_insurance_dataset.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?
   - How would you handle missing values if there are any?
3. **Basic Descriptive Statistics**:
   - What is the average age of the individuals in the dataset?
   - What's the range of the estimated salaries?

4.7 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.
- **Handling and Imputing Missing Data**: You explored methods to identify missing values using functions like `is.na()`, `anyNA()`, and `complete.cases()`.
- **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 foundational skills are crucial for efficient data analysis in R, enabling you to work with diverse data sources, maintain analysis integrity, and collaborate effectively. Congratulations on advancing your R programming proficiency!

5 Data Analysis and Visualization

In Lab 5, we will explore the fundamental aspects of data analysis and visualization using R. This lab is designed to enhance your proficiency in handling real-world datasets by importing them into R, performing insightful analyses, and creating compelling visualizations. We'll start by introducing the pipe operator `|>`, a powerful tool that streamlines your code and makes data manipulation more intuitive. Next, we'll dive into the **dplyr** package, learning how to efficiently manipulate data using functions like `select()`, `filter()`, `mutate()`, `arrange()`, and `summarise()`. Finally, we'll harness the capabilities of **ggplot2** to visualize data, enabling you to uncover patterns and communicate findings effectively. These skills are essential for any data analyst or scientist, as they form the backbone of data-driven decision-making and storytelling.

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

- **Apply the Pipe Operator `|>` for Streamlined Coding**
  Use the pipe operator to chain multiple functions together, enhancing code readability and efficiency.
- **Manipulate Data Using `dplyr` Functions**
  Employ `dplyr` verbs such as `select()`, `filter()`, `arrange()`, `mutate()`, and `summarise()` to perform common data manipulation tasks effectively.
- **Analyze Datasets to Extract Insights**
  Conduct exploratory data analysis to understand data distributions, identify patterns, and detect anomalies.
- **Create Visualizations with `ggplot2`**
  Develop a variety of plots—including scatter plots, histograms, boxplots, and bar charts—to visualize data and reveal underlying trends.
- **Communicate Findings Effectively**
  Present analysis results in a clear and insightful manner, using visualizations to enhance understanding.

By completing Lab 5, you will be well-equipped to tackle more complex data analysis challenges, making significant strides in your journey to becoming a proficient data professional.

5.1 Introduction

Welcome to the fifth chapter of R Programming Fundamentals: A Lab-Based Approach. In this Lab, we delve into the powerful capabilities of R for data analysis and visualization. R is not just a programming language; it's a comprehensive environment designed for statistical analysis, data modeling, and creating stunning visualizations. Its extensive package ecosystem and active community make it a top choice for data professionals worldwide.

5.2 Experiment 5.1: The Pipe Operator <%>

One of the most effective tools in R for simplifying your 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 `|>`. This operator allows you to express a sequence of operations in a more intuitive and readable manner by chaining functions together in a linear and logical flow, rather than nesting functions within functions. In this lab, we will be using the base pipe operator `|>`, which functions similarly to the `magrittr |>` operator.

Imagine you have data frame, `data` and you want to perform multiple operations on it, such as applying functions `foo` and `bar` in sequence. With the pipe operator, you can write:

```
data |>
  foo() |>
  bar()
```

Instead of the nested approach:

```
bar(foo(data))
```

> 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 labeled Use native pipe operator, |> (requires R 4.1+).

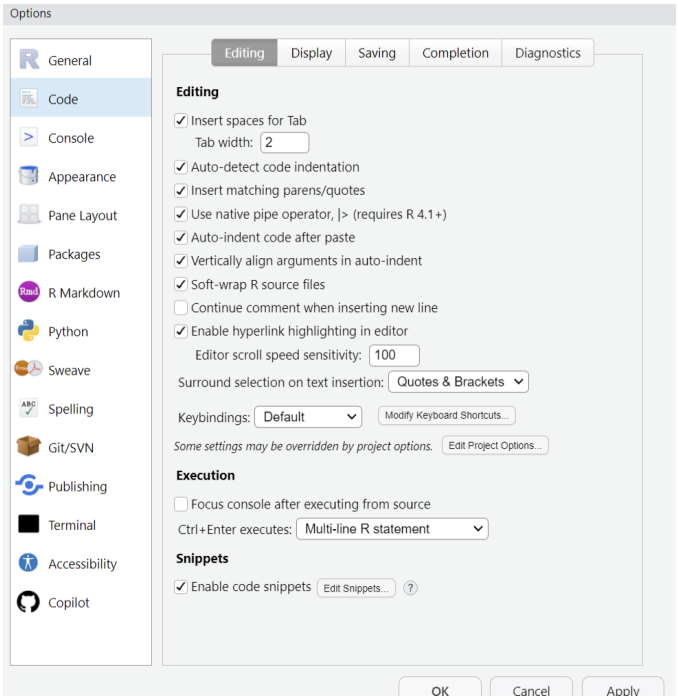


Figure 5.1: To insert |>, make sure the "Use native pipe operator" option is checked

How Does the Pipe Operator Work?

The pipe operator automatically passes the output of the previous function as the first argument to the next function. If a function takes multiple arguments, the piped data is placed as the first argument:

```
# Without pipe

function(argument1, argument2)

# With pipe

argument1 |> function(argument2)
```

Example 1:

Let's see the pipe operator in action:

```
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 equivalent to:

```
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
```

Using the pipe operator focuses on the flow of data transformations, making your code more readable and maintainable.

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:

```
x <- 4.234

round(log(sqrt(x)), 2)
```

```
#> [1] 0.72
```

By piping, you avoid deeply nested functions and enhance code clarity.

5.3 Experiment 5.2: Data Manipulation with dplyr

Data comes in all shapes and sizes, and often, it's not in the ideal format for analysis. This is where data manipulation comes into play. Data manipulation is a fundamental skill in data analysis, and dplyr provides a powerful set of tools to transform and summarize your data efficiently. The dplyr package, part of the tidyverse, is designed to make data manipulation in R more intuitive and efficient.

5.3.1 Why Use dplyr?

- Simplicity: Provides straightforward functions that are easy to learn and remember.
- Efficiency: Optimized for performance, handling large datasets swiftly.
- Readability: Code written with dplyr is often more readable and easier to maintain.
- Integration: Works seamlessly with other tidyverse packages like ggplot2 and tidyr.

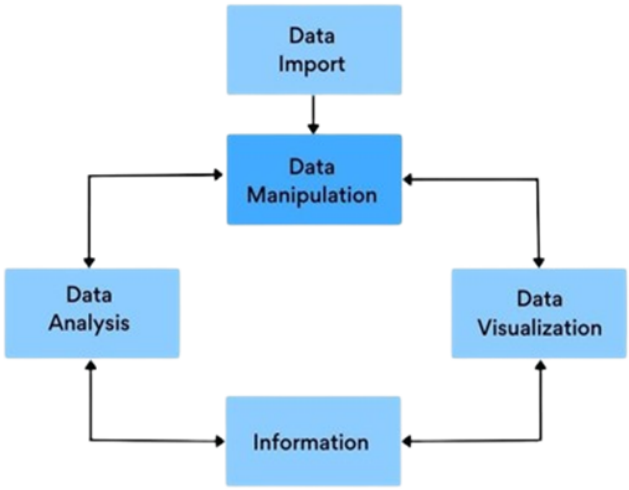


Figure 5.2: Data Exploration and Analysis Workflow

5.3.2 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)
```

5.3.3 Core `dplyr` Verbs

The core functions in `dplyr` are often referred to as "verbs" because they correspond to actions you can perform on your data. These verbs are:

- **`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.

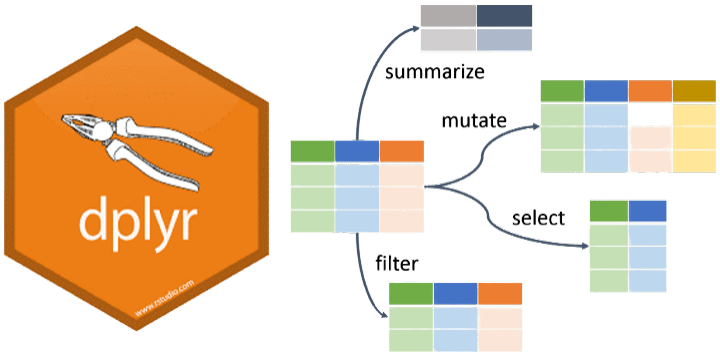


Figure 5.3: Key Data Manipulation Functions in dplyr

Additional useful functions include:

- **`rename()`**: Rename columns.
- **`distinct()`**: Find unique rows.
- **`count()`**: Count unique values of a variable.

- `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.

5.3.4 Using Pipes with `dplyr` functions

When combined with the pipe operator, `dplyr` functions create a seamless workflow as shown in Figure 5.4:

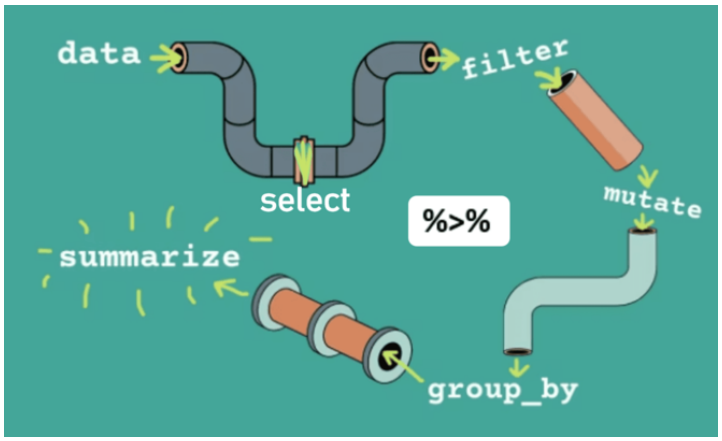


Figure 5.4: Data Transformation Pipeline in dplyr

5.3.5 Example Datasets

We'll start our exploration by working with two fascinating datasets: the **`penguins`** dataset from the `palmerpenguins` package[1] 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.

The `penguins` Dataset

The `penguins` dataset[2] 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:

[1] If you haven't installed it yet, you can do so with `install.packages("palmerpenguins")` and load it using `library(palmerpenguins)`.

[2] 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.

- **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.

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.
- **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.
- **Brain Weight**: The brain weight of the animal.
- **Body Weight**: The body weight of the animal.
- **Conservation Status**: The conservation status of the species.

Example: using `select()`

- Purpose: Select specific columns from a data frame.
- Example: select `Petal.Length` and `Petal.Width` from the `iris` data frame.

```
petal_data <- iris |> select(Petal.Length, Petal.Width)

petal_data |> head()
```

```
#>   Petal.Length Petal.Width
#> 1          1.4         0.2
#> 2          1.4         0.2
#> 3          1.3         0.2
#> 4          1.5         0.2
#> 5          1.4         0.2
#> 6          1.7         0.4
```

As you can see, we have selected `Petal.Length` and `Petal.Width` from the `iris` dataframe

Example: Using `mutate()`

- Purpose: Add a new variable or modifies an existing one
- Example: Add a new column `Petal.Ratio` that is the ratio of `Petal.Width` to `Petal.Length`.

```
modified_iris <- iris |> mutate(Petal.Ratio = Petal.Width / Petal.Length)

modified_iris |> head()
```

```
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width Species Petal.Ratio
#> 1          5.1         3.5          1.4         0.2  setosa   0.1428571
#> 2          4.9         3.0          1.4         0.2  setosa   0.1428571
#> 3          4.7         3.2          1.3         0.2  setosa   0.1538462
#> 4          4.6         3.1          1.5         0.2  setosa   0.1333333
#> 5          5.0         3.6          1.4         0.2  setosa   0.1428571
#> 6          5.4         3.9          1.7         0.4  setosa   0.2352941
```

In this second example, we will be using ggplot2's built-in dataset `msleep`, and we are changing the sleep data from being measured in hours to minutes.

```
msleep %>%
  select(name, sleep_total) %>%
  mutate(sleep_total_min = sleep_total * 60)
```

```
#> # A tibble: 83 x 3
#>    name                        sleep_total sleep_total_min
#>    <chr>                             <dbl>           <dbl>
#>  1 Cheetah                            12.1             726
#>  2 Owl monkey                         17              1020
#>  3 Mountain beaver                    14.4             864
#>  4 Greater short-tailed shrew         14.9             894
#>  5 Cow                                 4               240
#>  6 Three-toed sloth                   14.4             864
#>  7 Northern fur seal                   8.7             522
#>  8 Vesper mouse                        7               420
#>  9 Dog                                10.1             606
#> 10 Roe deer                           3                180
#> # i 73 more rows
```

Example: Using `filter()`

- Purpose: Filters rows based on their values.
- Example: Filter `iris` dataframe where species is "setosa".

```
setosa_flowers <- iris |> filter(Species == "setosa")

setosa_flowers |> 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 returned only the rows where the `Species` column is "setosa".

Filter with multiple conditions:

filter `iris` dataframe where species is `setosa` and `Petal.Length` is greater 1.5.

```
setosa_petal_length <- iris |> filter(Species == "setosa", Petal.Length > 1.5)

setosa_petal_length |> head()
```

```
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#> 1          5.4         3.9          1.7         0.4  setosa
#> 2          4.8         3.4          1.6         0.2  setosa
#> 3          5.7         3.8          1.7         0.3  setosa
#> 4          5.4         3.4          1.7         0.2  setosa
#> 5          5.1         3.3          1.7         0.5  setosa
#> 6          4.8         3.4          1.9         0.2  setosa
```

Example: Using `arrange()`

- Purpose: Arranges rows by values of a column (default is ascending order).

Ascending order:

```r
ordered_by_length <- iris |> arrange(Petal.Length)

ordered_by_length |> head()
```

```
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#> 1          4.6         3.6          1.0         0.2  setosa
#> 2          4.3         3.0          1.1         0.1  setosa
#> 3          5.8         4.0          1.2         0.2  setosa
#> 4          5.0         3.2          1.2         0.2  setosa
#> 5          4.7         3.2          1.3         0.2  setosa
#> 6          5.4         3.9          1.3         0.4  setosa
```

Descending order:

```r
ordered_by_length_desc <- iris |> arrange(desc(Petal.Length))

ordered_by_length_desc |> head()
```

```
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
#> 1          7.7         2.6          6.9         2.3 virginica
#> 2          7.7         3.8          6.7         2.2 virginica
#> 3          7.7         2.8          6.7         2.0 virginica
#> 4          7.6         3.0          6.6         2.1 virginica
#> 5          7.9         3.8          6.4         2.0 virginica
#> 6          7.3         2.9          6.3         1.8 virginica
```

Example: Using `summarise()`

`summarise()` (or summarize() in American English):

Purpose: Reduces data to a single row by computing summary statistics.

Example: Find the average sepal length for each species, ignoring any missing values.

```r
average_lengths <- iris |> summarise(Avg.Length = mean(Sepal.Length, na.rm = TRUE))

average_lengths
```

```
#>   Avg.Length
#> 1   5.843333
```

The code cell produced a single row with the average Sepal.Length.

Example: Using `group_by()` and `summarise()`

Purpose: Groups the data by one or more columns and is usually used in combination with `summarise()` to compute group-wise summaries. That is, it allows you to split the data frame by one or more variables, apply functions to each group, and then combine the output. You can also use the `group_by()` verb to operate within groups of rows with `mutate()` and `summarize()`.

Example: Compute the average Sepal Length separately for each species.

```
iris |>
  group_by(Species) |>
  summarise(Avg.Length = mean(Sepal.Length, na.rm = TRUE))
```

```
#> # A tibble: 3 x 2
#>   Species    Avg.Length
#>   <fct>           <dbl>
#> 1 setosa           5.01
#> 2 versicolor       5.94
#> 3 virginica        6.59
```

Example: Using `rename()`

- Purpose: Rename columns.
- Example: Change the column name `Sepal.Length` to `Sepal_Length` and `Species` to `Category` in the `iris` dataset.

```
renamed_iris <- iris |> rename(Sepal_Length = Sepal.Length, Category = Species)

renamed_iris |> head()
```

```
#>   Sepal_Length Sepal.Width Petal.Length Petal.Width Category
#> 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
```

> **Note**
>
> `data |> rename(new_name = old_name)` will rename the `old_name` column to `new_name`.

> **Further Resources on dplyr**
>
> For a deeper dive into `dplyr`, I highly recommend exploring the comprehensive series by Suzan. Start with "Data Wrangling Part 1: Basic to Advanced Ways to Select Columns" and continue through to "Part 4" for valuable insights into data manipulation using `dplyr`.

5.3.6 Exercise 5.1: Analyzing 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**
   - Download the penguins dataset.
   - 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.

5.4 Experiment 5.3: Data Visualization

Data visualization is the art and science of representing data through graphical means, such as charts, graphs, and maps. By transforming numerical or textual information into visual formats, data visualization allows us to see patterns, trends, and insights that might be difficult or impossible to detect in raw data. It brings data to life, telling stories that are easily understood and can be quickly communicated to others.

In today's data-driven world, the ability to visualize data effectively is becoming an essential skill across various industries—including data science, finance, education, and healthcare. As we grapple with an ever-growing volume of complex and varied data, visualization provides the tools we need to make sense of it all and to share our findings in a compelling way.

Visual representations are more effective than descriptive statistics or tables when it comes to analyzing data. They enable us to:

[3]For additional insights on data visualization techniques not covered in this book, please refer to the articles by Harvard Business School, DataCamp and Polymer.

Figure 5.5: Data Scientist Analyzing Large-Scale Data[3].

- **Identify Patterns and Trends**: Spot relationships within the data that might not be immediately apparent.
- **Understand Distributions**: See how data is spread out, where concentrations or gaps exist.
- **Detect Outliers**: Quickly identify data points that deviate significantly from the rest of the dataset.
- **Communicate Insights**: Present data in a way that is accessible and engaging to diverse audiences.

By leveraging data visualization, we enhance our ability to analyze complex datasets and to communicate our findings effectively.

5.4.1 Importance of Data Visualization

Data visualization plays a crucial role in the data analysis process for several reasons:

1. **Simplifies Complex Data**: Large datasets can be overwhelming when presented in raw form. Visualization condenses and structures this data, making it understandable at a glance. For example, a line chart can succinctly display trends over time that would be difficult to discern from a table of numbers.
2. **Reveals Patterns and Trends**: Visual tools help us identify relationships within the data, such as correlations between variables or changes over time. This can lead to new insights and hypotheses. For instance, a scatter plot might reveal a positive correlation between hours studied and exam scores.

3. **Supports Decision Making**: Visual evidence provides a compelling basis for conclusions and recommendations. Decision-makers can grasp complex information quickly and make informed choices. A well-designed dashboard can highlight key performance indicators, aiding strategic planning.
4. **Engages the Audience**: Visuals are inherently more engaging than raw numbers or text. They capture attention and can make presentations more persuasive. Using colorful charts and interactive elements can enhance audience understanding and retention of information.
5. **Facilitates Communication**: Visualization transcends language barriers and can communicate complex ideas simply. It enables collaboration across teams and disciplines by providing a common visual language.

5.4.2 Choosing the Right Visualization

Selecting the appropriate type of visualization is essential to effectively communicate your data's story. Here are some considerations to guide your choice:

1. **Define Your Objective**
    - **What Do You Want to Communicate?**
        - Are you aiming to compare values, show the composition of something, understand distribution, or analyze trends over time?
        - Clarify the key message or insight you wish to convey.
2. **Understand Your Data**
    - **Data Relationships**
        - Identify the types of variables you have (categorical, numerical, time-series).
        - Determine if you are exploring relationships between variables, distributions, or looking for outliers.
3. **Know Your Audience**
    - **Audience Understanding**
        - Consider the background and expertise of your audience.
        - Will they understand complex visualizations, or is a simpler chart more appropriate?
        - Tailor your visualization to their needs and expectations.

4. **Consider Practical Constraints**
    - **Medium of Presentation**
        - Will the visualization be presented digitally, in print, or verbally?
        - Interactive visualizations may not translate well to static formats.
    - **Data Quality and Quantity**
        - Large datasets may require aggregation.
        - Poor quality data may limit the types of visualizations you can use.
5. **Aesthetics and Clarity**
    - **Visual Appeal**
        - Use color, shape, and size effectively to enhance comprehension without overwhelming the viewer.
        - Avoid clutter by keeping designs clean and focused.
6. **Ethical Representation**
    - **Accuracy and Honesty**
        - Ensure scales are appropriate and do not mislead.
        - Represent data truthfully to maintain credibility.

By thoughtfully considering these factors, you can choose a visualization that not only presents your data effectively but also resonates with your audience.

5.4.3 Types of Data Visualization Analysis

Data visualization can be categorized based on the number of variables you are analyzing:

1. **Univariate Analysis**
    - **Definition**: Examining one variable at a time.
    - **Purpose**: Understand the distribution, central tendency, and spread of a single variable.
    - **Common Visualizations**:
        - **Histograms**: Show frequency distribution.
        - **Boxplots**: Display median, quartiles, and potential outliers.

- **Bar Charts**: Represent categorical data counts.

Example: Analyzing the distribution of ages in a population using a histogram.

2. **Bivariate Analysis**

   - **Definition**: Studying the relationship between two variables.
   - **Purpose**: Explore associations, correlations, and potential causations.
   - **Common Visualizations**:
     - **Scatter Plots**: Show relationships between two numerical variables.
     - **Line Charts**: Depict trends over time or ordered categories.
     - **Heatmaps**: Represent data in a matrix form with color encoding.

   Example: Investigating the relationship between advertising spend and sales revenue using a scatter plot.

3. **Multivariate Analysis**

   - **Definition**: Analyzing more than two variables simultaneously.
   - **Purpose**: Understand complex interactions and higher-dimensional relationships.
   - **Common Visualizations**:
     - **Bubble Charts**: Add a third variable to a scatter plot through size or color.
     - **Multidimensional Scatter Plots**: Use color, shape, and size to represent additional variables.
     - **Parallel Coordinates Plot**: Visualize high-dimensional data by plotting variables in parallel axes.

   Example: Evaluating factors affecting customer satisfaction by analyzing service quality, price, and brand reputation together.

Understanding the type of analysis you need guides the selection of appropriate visualization techniques, ensuring that you capture the necessary insights from your data.

5.4.4 Common Data Visualization Techniques

While there are hundreds of different graphs and charts available, focusing on the core ones will equip you with the tools needed for most day-to-day analytical tasks.

Let's explore some of the most commonly used data visualization techniques.

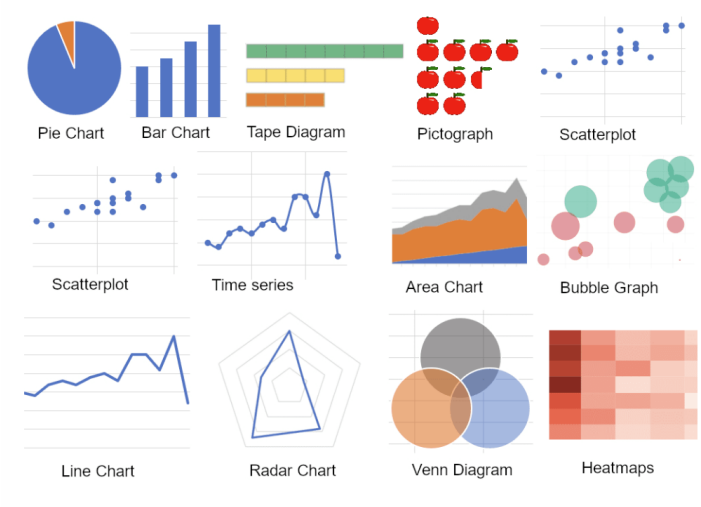


Figure 5.6: Common Data Visualization Techniques.

Bar Chart

A bar chart represents categorical data with rectangular bars, where the length of each bar is proportional to the value it represents. Bars can be plotted vertically or horizontally.

When to Use:

- Comparing quantities across different categories.
- Showing rankings or frequencies.
- Displaying discrete data.

Example Uses:

- Comparing sales figures across different regions.
- Showing the number of students enrolled in various courses.
- Visualizing survey responses by category.

Key Features:

- **Categories on one axis** (usually the x-axis for vertical bars).
- **Values on the other axis** (usually the y-axis for vertical bars).
- Bars are separated by spaces to emphasize that the data is discrete.

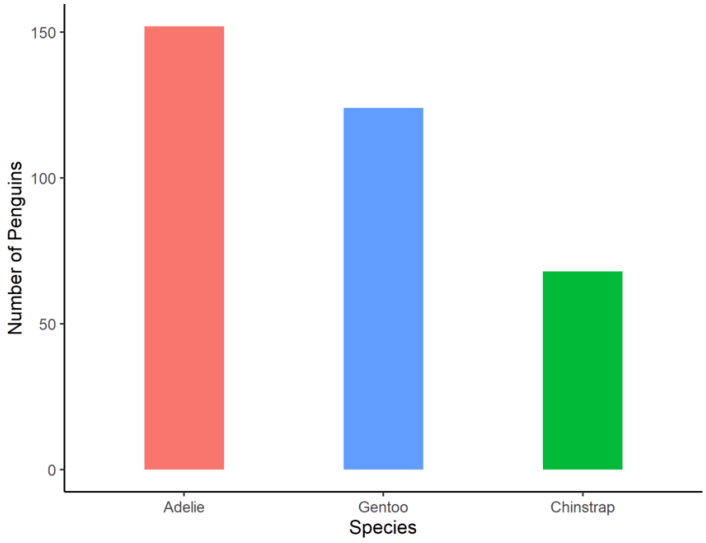


Figure 5.7: Bar Chart Illustration

Histogram

A histogram displays the distribution of a numerical variable by grouping data into continuous intervals, known as bins. It shows the number of data points that fall 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.
- Analyzing the spread of housing prices in a market.

Key Features:

- **Continuous data on the x-axis**, divided into bins.
- **Frequency or count on the y-axis**.
- Bars are adjacent, indicating the continuous nature of the data.

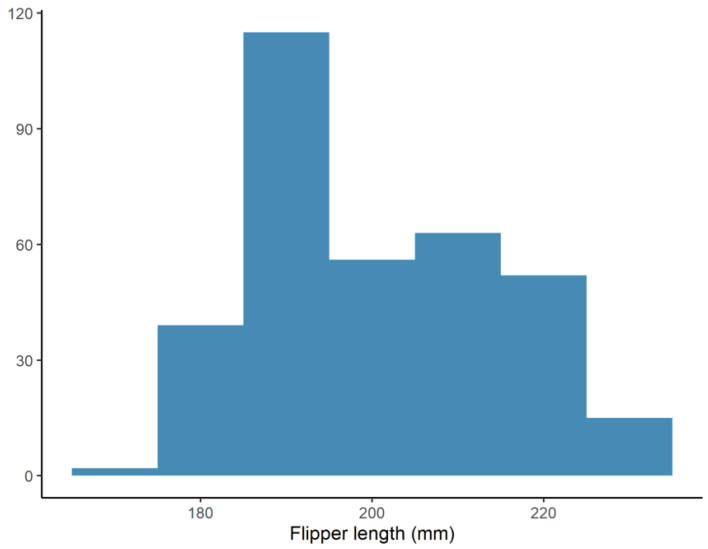


Figure 5.8: Histogram Illustration

Circular charts

A circular chart is a type of statistical graphic represented in a circular format to illustrate numerical proportions. A pie chart and a doughnut chart 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:

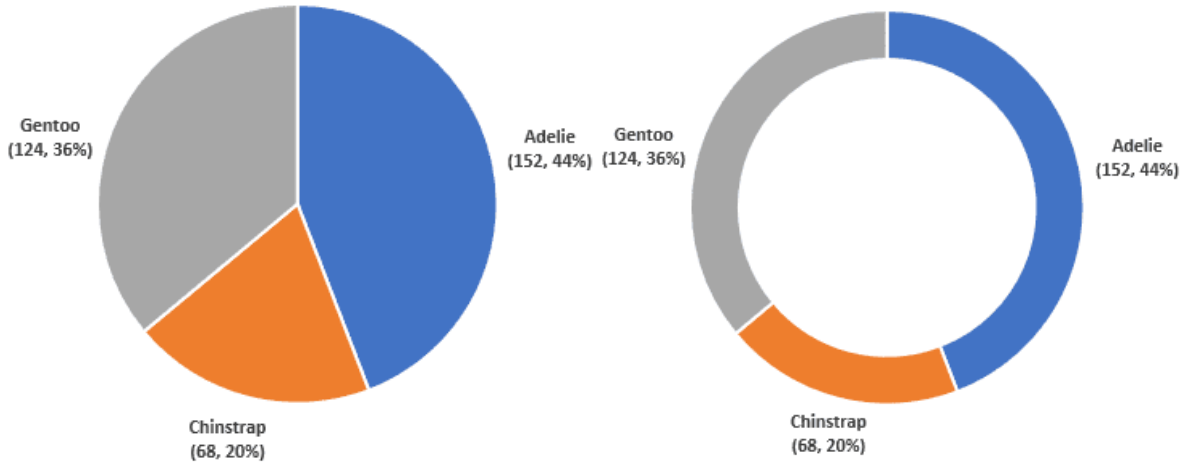
- Showing parts of a whole.
- Representing percentage or proportional data.
- Comparing categories within a dataset where the total represents 100%.
- When there are a limited number of categories (ideally less than six).

Example Uses:

- Displaying market share of different companies.
- Illustrating budget allocations across departments.
- Showing survey results for single-choice questions.
- Comparing population distributions across different regions.

Key Features:

- The circle represents 100% of the data.
- Slices or segments are proportional to each category's percentage.
- Includes both solid circles (pie chart) and circles with a hollow center (doughnut chart).
- Effective for highlighting significant differences between categories.



(a) Pie Chart Illustration

(b) Doughnut Chart Illustration

Figure 5.9: Circular Statistical Charts

> **Note**
>
> Circular charts can become difficult to interpret when there are many small or similarly sized slices. In such cases, alternative visualizations like bar charts or stacked charts might be more effective. Additionally, a doughnut chart allows for extra data or labeling in the center space, offering more flexibility in design.

Scatter Plot

A scatter plot uses Cartesian coordinates to display values for typically two variables for a set of data. Each point represents an observation.

When to Use:

- Exploring relationships or correlations between two continuous numerical variables.
- Detecting patterns, trends, clusters, or outliers.

Example Uses:

- Examining the relationship between hours studied and exam scores.
- Analyzing the correlation between advertising spend and sales revenue.
- Investigating the association between temperature and energy consumption.

Key Features:

- One variable on the x-axis, another on the y-axis.
- Points plotted in two-dimensional space.
- Can include a trend line to highlight the overall relationship.

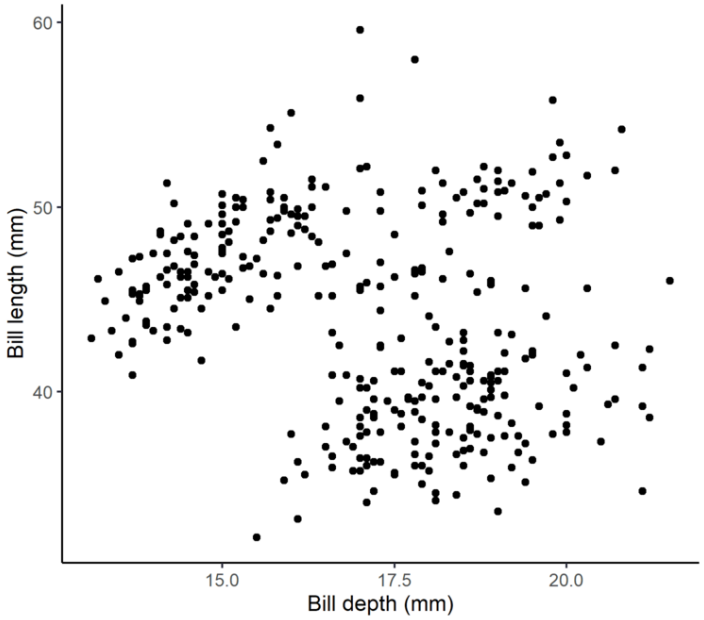


Figure 5.10: Scatter Plot Illustration

Box and Whisker Plot

A box plot summarizes a data set by displaying it along a number line, highlighting the median, quartiles, and potential outliers.

When to Use:

- Comparing distributions across different categories.
- Identifying central tendency, dispersion, and skewness.
- Highlighting outliers in the data.

Example Uses:

- Comparing test scores between different classrooms.
- Analyzing the spread of salaries across industries.
- Visualizing the distribution of delivery times from various suppliers.

Key Features:

- **Box** shows the interquartile range (IQR), from the first quartile (Q1) to the third quartile (Q3).
- **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.

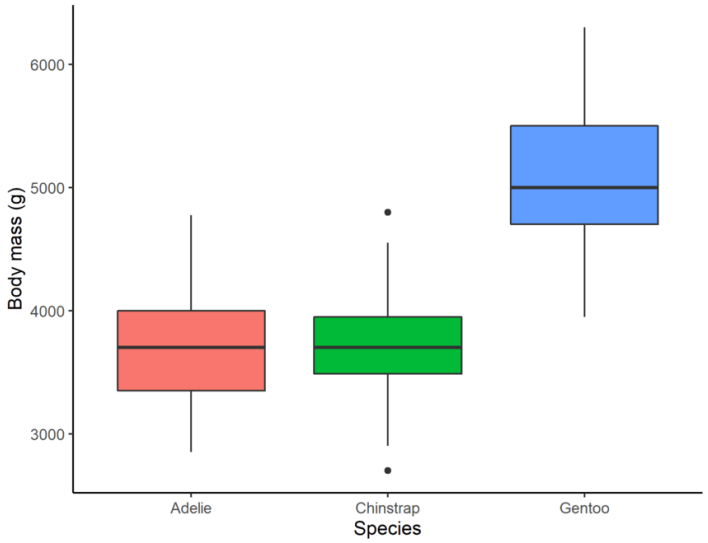


Figure 5.11: Box Plot Illustration

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 visualize 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 over time.
- Showing temperature changes throughout the day.
- Visualizing website traffic trends.

Key Features:

- Time or sequential data on the x-axis.
- Quantitative values on the y-axis.
- Lines can represent different categories or groups.

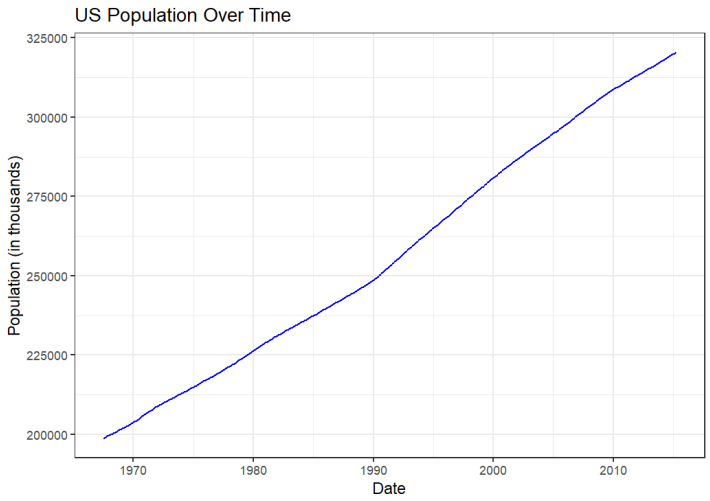


Figure 5.12: Line Chart Illustration

Areas chart

An area chart is similar to a line chart but with the area below the line filled in. It emphasizes 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.
- Visualizing population growth.
- Comparing energy consumption by source over time.

Key Features:

- Time or sequential data on the x-axis.
- Quantitative values on the y-axis.
- Areas can be stacked to show cumulative totals.

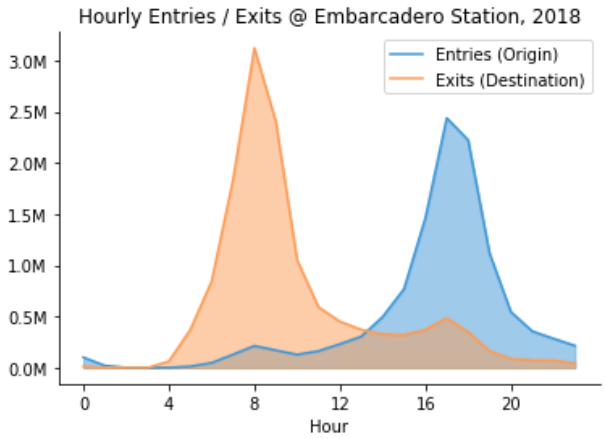


Figure 5.13: Area Chart Illustration

These core visualization techniques form the foundation of data storytelling. By mastering them, you'll be equipped to handle most day-to-day data visualization tasks effectively[4]. Remember that the key to successful data visualization is not just the choice of chart type but also clarity, accuracy, and the ability to convey the intended message to your audience.

[4]For additional insights on data visualization techniques not covered in this book, please refer to the articles by Harvard Business School, DataCamp and Polymer.

5.4.5 Data Visualization with `ggplot2`

R has several systems for making graphs, but `ggplot2` is one of the most elegant and versatile tools for creating high-quality visualizations. The `ggplot2` package, part of the `tidyverse`, is built upon the principles of the Grammar of Graphics, a systematic approach to describing and constructing graphs. This grammar provides a coherent framework for building a wide variety of statistical graphics by mapping data variables to visual properties. The following are the advantage of `ggplot`:

Advantages of Using `ggplot2`

Let's explore the key benefits that make ggplot2 a preferred choice for data visualization in R.

- **Consistency and Grammar**: The structured approach makes it easier to build complex plots by adding layers.
- **Customization**: Nearly every aspect of the plot can be customized to suit your needs.
- **Extension**: `ggplot2` is extensible, allowing for additional packages like `ggthemes`, `ggrepel`, and `plotly` for interactive graphics.
- **Professional Quality**: Produces publication-ready graphics suitable for reports, presentations, and academic papers.

5.4.5.1 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 visualized.
2. **Aesthetics (`aes()`)**: Mappings between data variables and visual properties, such as position (`x`, `y`), color, 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 or boxes.
    - `color`: color of points, lines, or areas.
    - `size`: size of points or lines.
    - `shape`: shape of points.
    - `alpha`: transparency level.

- `group`: identifies series of points with a grouping variable
- `facet`: create small multiples

3. **Geometric Objects (or geoms)**: These are the fundamental visual components 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 particular type of visualization, enabling users to create a wide variety of plots to suit their analytical needs. Examples include `geom_point()` for a scatter plot, `geom_line()` for a line chart, `geom_bar()` for a bar chart, `geom_histogram()` for a histogram, `geom_boxplot()` for a boxplot, and `geom_violin()` for a violin plot.

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 summarizing or smoothing data. For instance, `stat_smooth()` adds a smoothed line to a scatter plot.
5. **Scales**: Control how data values are translated into aesthetic values, such as the range and breaks of axes, or the mapping of data values to colors.
6. **Coordinate Systems**: Define the space in which the data is represented, 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) split by one or more variables to display different subsets of the data by using `facet_wrap()` or `facet_grid()`.
8. **Themes**: Customize the non-data components of the plot, like background, gridlines, text, and overall appearance, using functions like `theme_minimal()`, `theme_bw()`, `theme_classic()`, or by modifying elements with `theme()`.
9. **Labels**: Titles, axis labels, legend titles, and other annotations can be added to a plot by using `labs()`.

5.4.6 Building Plots with `ggplot2`

To create a plot using `ggplot2`, you start with the `ggplot()` function, specifying your data and aesthetic mappings. You then add layers to the plot using the `+` operator. Each layer can add new data, mappings, or geoms, allowing for intricate and customized visualizations. The basic structure of a ggplot2 plot can be represented as:

```
ggplot(data = <DATA>, aes(<MAPPINGS>)) +
  <GEOM_FUNCTION> + <OTHER_LAYERS>
```

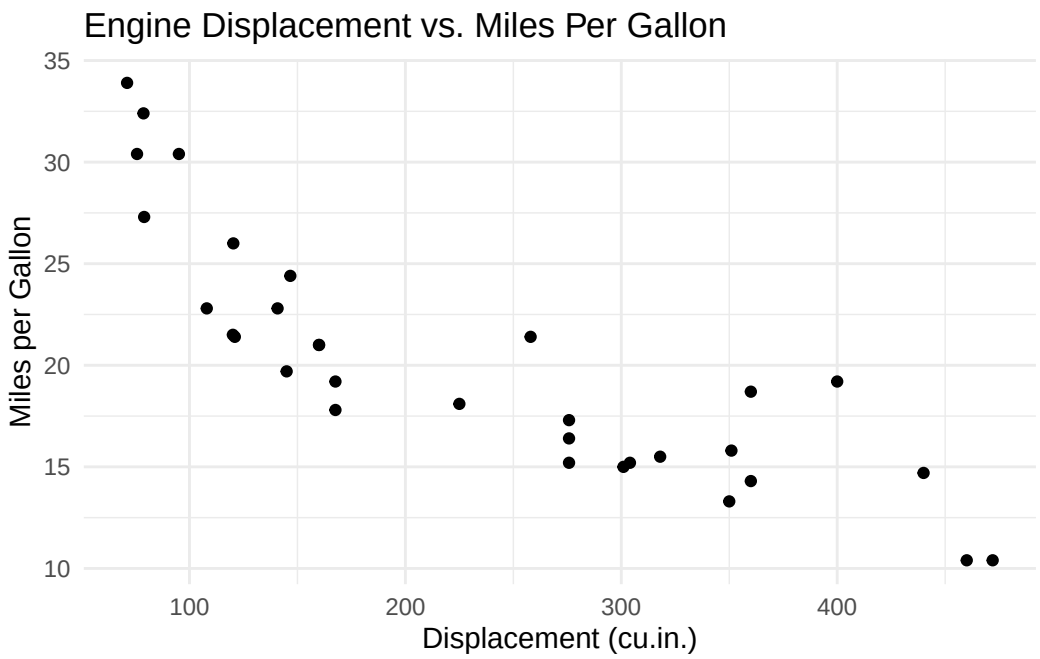
> **Tip**
>
> Please see Section 5.4.5.1 for a breakdown of each component of this structure.

Creating a Scatter Plot with `geom_point()`

Suppose you want to visualize the relationship between engine displacement and miles per gallon in the `mtcars` dataset:

```
library(tidyverse)

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:

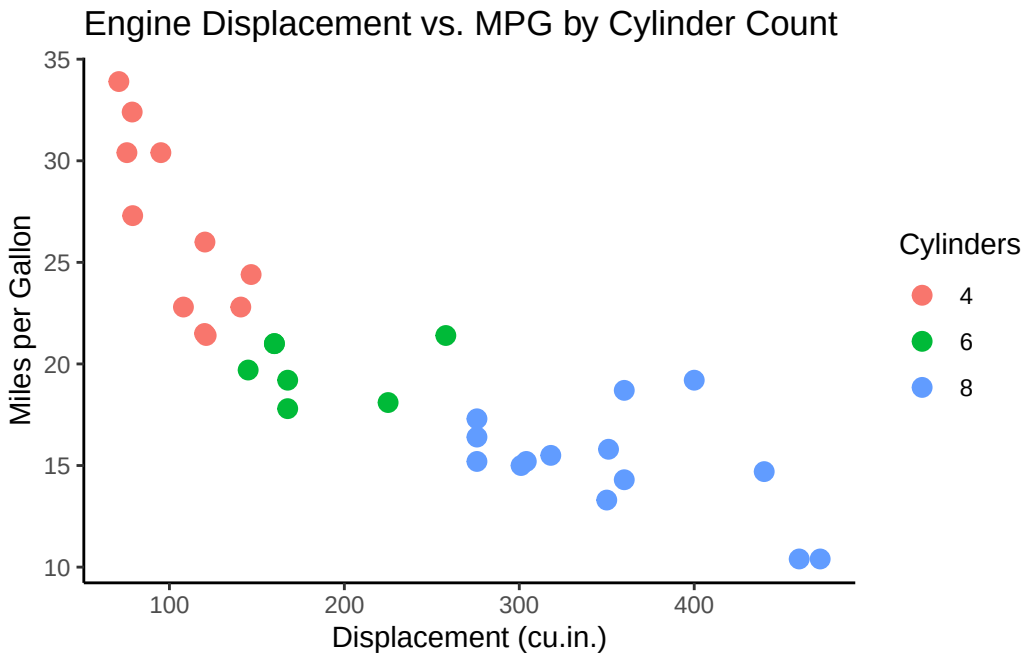
- **Data**: `mtcars`

- **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.

Customizing Aesthetics and Geoms

You can map additional variables to aesthetics to add more dimensions to your plot:

```
ggplot(data = mtcars, aes(x = disp, y = mpg, color = factor(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()
```

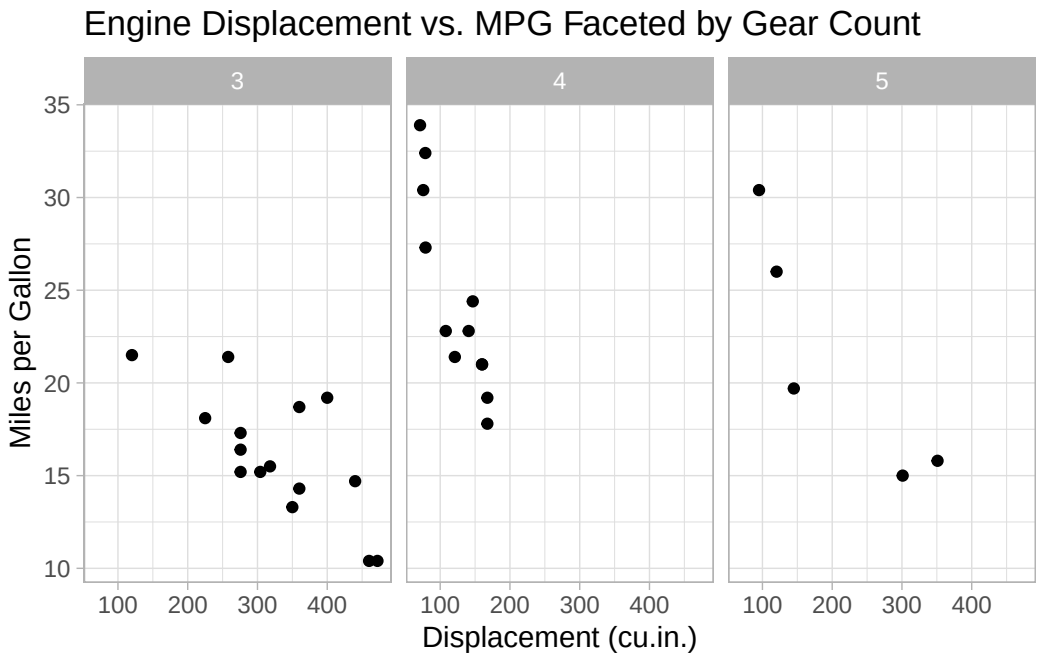


Here, the `color` aesthetic maps the number of cylinders (`cyl`) to different colors, allowing you to distinguish groups within the data.

Faceting for Multi-Panel Plots

Faceting splits your data into subsets and displays them in separate panels:

```
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()
```



This code creates a scatter plot for each unique value of `gear`, allowing for easy comparison across groups.

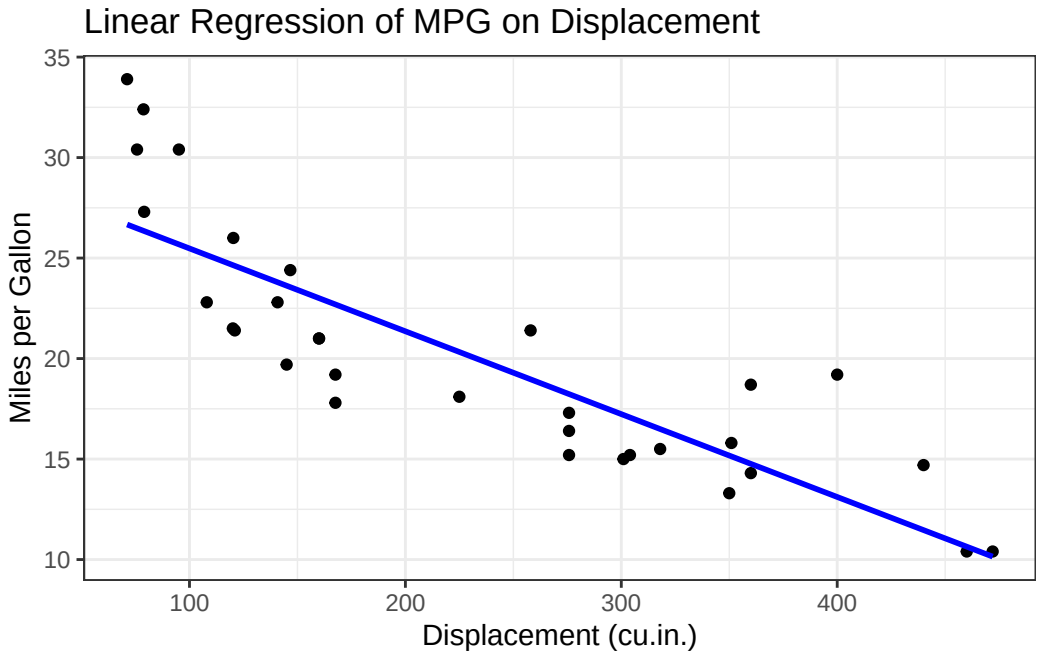
Incorporating Statistical Transformations

You can add statistical summaries or models to your plots:

```
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'
```



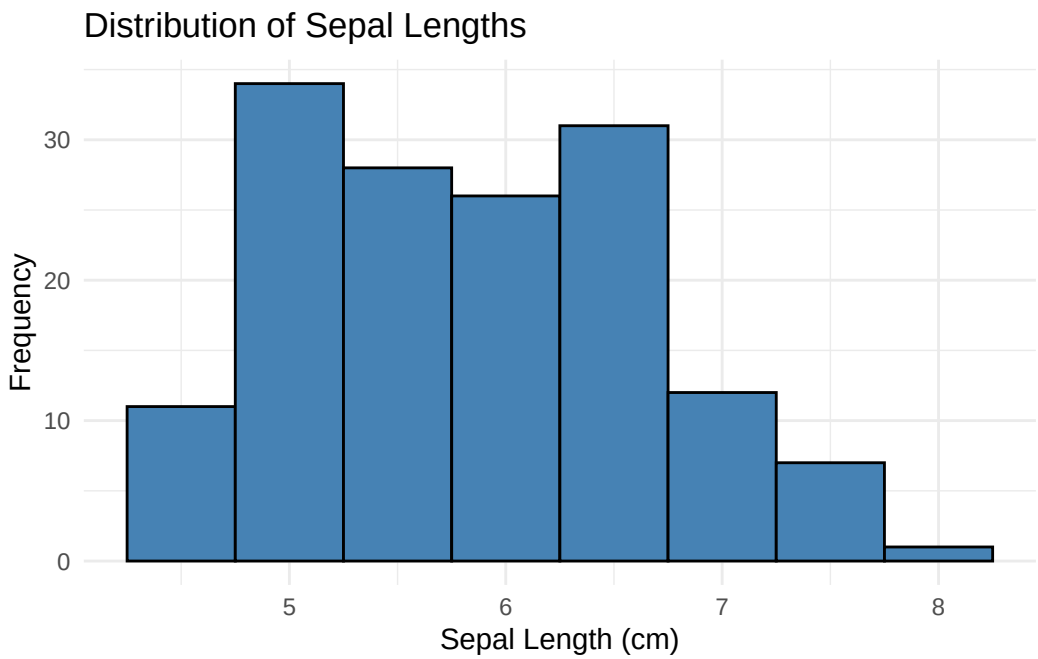
The `geom_smooth()` function adds a linear regression line to the scatter plot, providing insight into the trend.

Creating a Histogram with `geom_histogram()`

Suppose you want to a histogram of the `Sepal.Length` in the `iris` dataset:

```
# You can also start with the data frame and then pipe with ggplot

iris |> ggplot(aes(x = Sepal.Length)) +
  geom_histogram(binwidth = 0.5, fill = "steelblue", color = "black") + # binwidth determine
  labs(
    title = "Distribution of Sepal Lengths",
    x = "Sepal Length (cm)",
    y = "Frequency"
  ) +
  theme_minimal()
```



In this example:

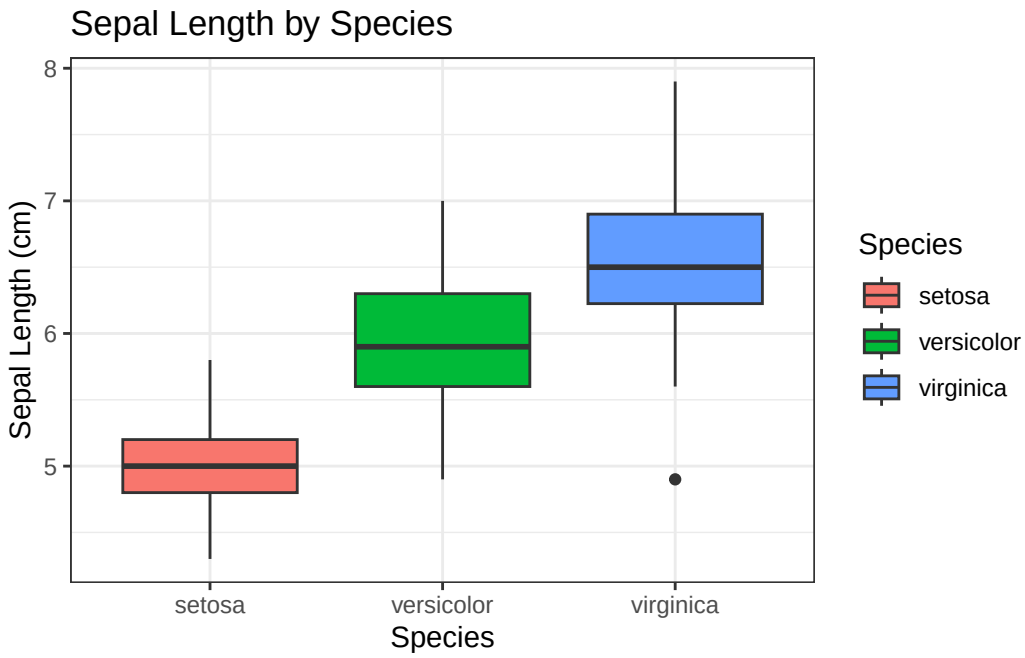
- **Data:** `iris`
- **Aesthetics:** `x = Sepal.Length`
- **Geometric Object:** `geom_histogram()` creates a histogram with `binwidth = 0.5`, where each bar represents the frequency of `Sepal.Length` values. The `fill` parameter sets the bar color to "steelblue" and `color` outlines each bar in black.
- **Labels:** `labs()` adds a title and axis labels.
- **Theme:** `theme_minimal()` provides a clean, minimalist background.

Creating a Boxplot with `geom_boxplot()`

Create a boxplot of sepal length by species:

```
iris |> ggplot(aes(x = Species, y = Sepal.Length, fill = Species)) +
  geom_boxplot() +
  labs(
    title = "Sepal Length by Species",
    x = "Species",
    y = "Sepal Length (cm)"
```

```
) +
theme_bw()
```



In this example:

- **Data:** `iris`
- **Aesthetics:** `x = Species`, `y = Sepal.Length`, `fill = Species`
- **Geometric Object:** `geom_boxplot()` creates a box plot to visualize the distribution of `Sepal.Length` for each `Species`. The `fill` parameter colors the boxes based on the species.
- **Labels:** `labs()` adds a title and axis labels.
- **Theme:** `theme_bw()` applies a theme with a white background and black grid lines, giving the chart a classic look.

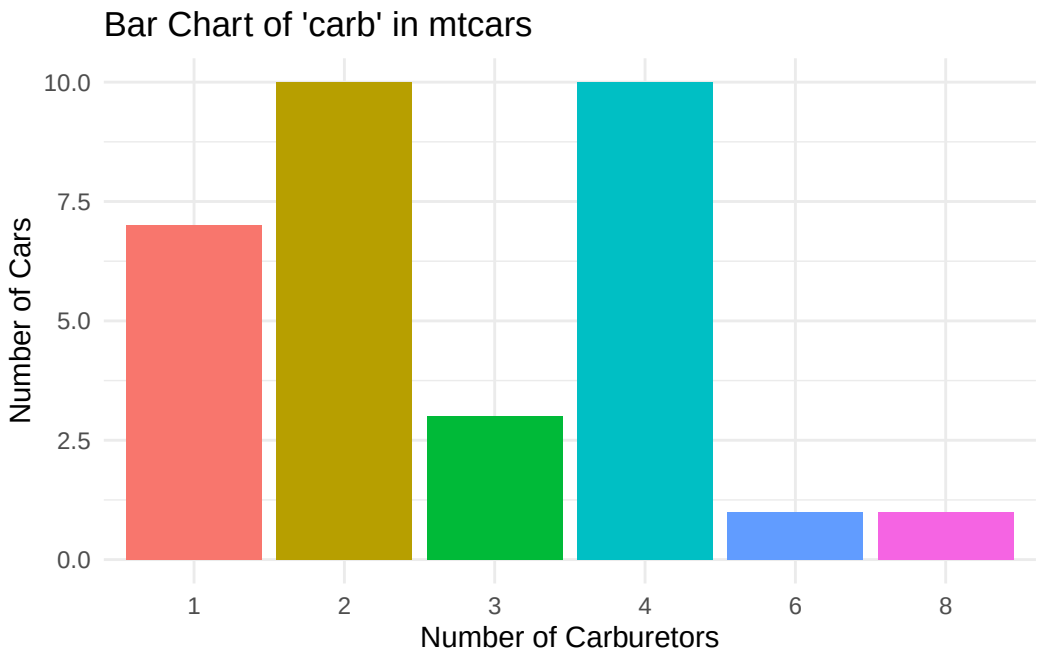
Creating a Bar Chart with `geom_bar()`

Visualize the count of cars by the number of carburetors:

```
mtcars <- mtcars |> mutate(carb = as.factor(carb))

mtcars |> ggplot(aes(x = carb, fill = carb)) +
```

```r
geom_bar(show.legend = FALSE) +
labs(
  title = "Bar Chart of 'carb' in mtcars",
  x = "Number of Carburetors",
  y = "Number of Cars"
) +
theme_minimal()
```



In this example:

- **Data:** `mtcars`, with the `carb` column converted to a factor using `mutate()`.
- **Aesthetics:** `x = carb`, `fill = carb` to color the bars based on the different levels of carb.
- **Geometric Object:** `geom_bar()` creates a bar chart showing the count of cars for each value of `carb`. The `show.legend = FALSE` parameter hides the legend.
- **Labels:** `labs()` adds a title and axis labels.
- **Theme:** `theme_minimal()` provides a clean, minimalist background.

Creating a Line Chart with `geom_line`

Using the `economics` dataset in ggplot2 package, create a line plot to visualize the trend of unemployment over time. Specifically, plot `date` on the x-axis and `unemploy` (the number of unemployed individuals) on the y-axis. Color the line blue and apply the `theme_bw()` theme to give the plot a clean, professional look.

```
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:

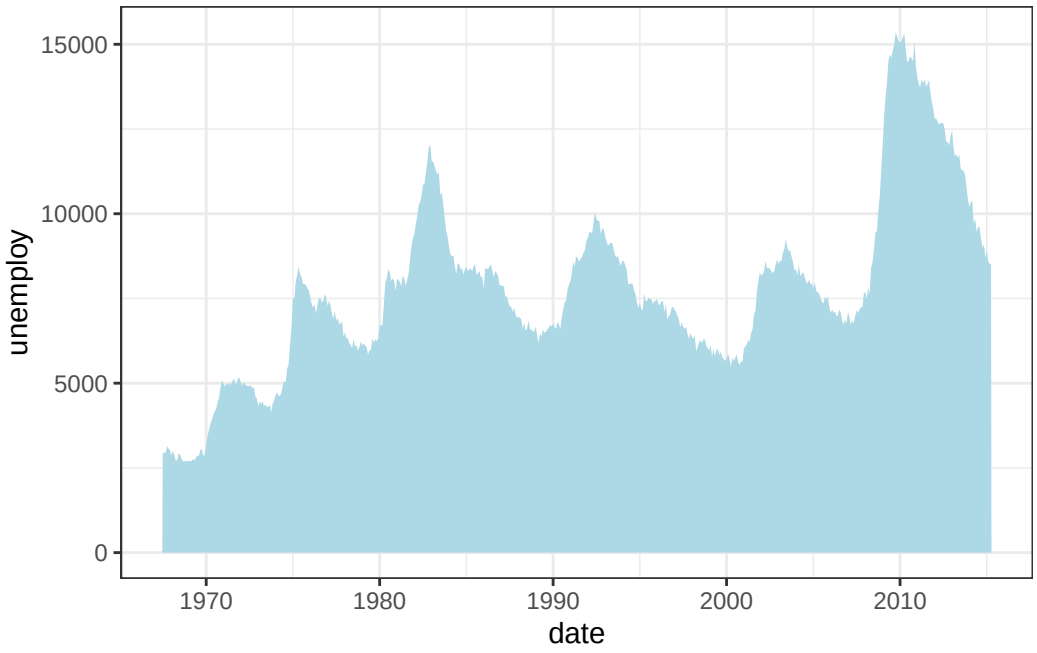
- **Data:** US economic time series data
- **Aesthetics:** `x = date, y = unemploy` plots unemployment over time.
- **Geometric Object:** `geom_line(color = "blue")` creates a line chart to show the trend of unemployment, with the line colored blue.

- **Labels:** `labs()` adds a title and axis labels, specifying "Unemployment Trends Over Time" for the title, "Date" for the x-axis, and "Number of Unemployed Individuals" for the y-axis.
- **Theme:** `theme_bw()` applies a black-and-white theme for a clear and classic look.

Creating an Area Chart with `geom_area`

Using the same economics data, create an area plot that displays the number of unemployed individuals over time.

```
economics |> ggplot(aes(x = date, y = unemploy)) +
  geom_area(fill = "lightblue") +
  theme_bw()
```



In this example:

- **Data:** US economic time series data
- **Aesthetics:** `x = date`, `y = unemploy` to set the x-axis as the date and the y-axis as the unemployment count.
- **Geometric Object:** `geom_area()` creates an area chart showing the trend of unemployment over time. The `fill = "light-blue"` parameter colors the area with a light blue shade.

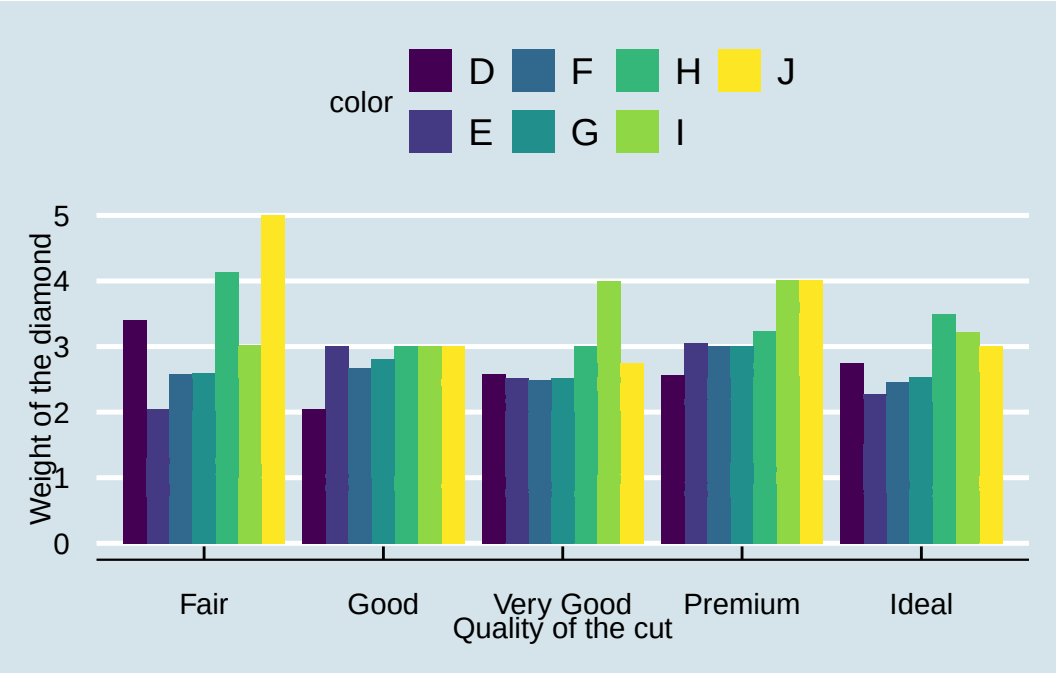
- **Labels:** None are explicitly added here, so the default axis labels (`date` and `unemploy`) will be used.
- **Theme:** `theme_bw()` applies a theme with a white background and black grid lines, giving the chart a classic look.

5.4.7 Saving your plots

Once you've created a meaningful and visually appealing plot in R using ggplot2, you might want to save it as an image file to include in reports, presentations, or share with others. The `ggsave()` function is a convenient tool that allows you to export your plots to a variety of formats such as PNG, PDF, JPEG, and more.

Let's walk through the process of creating a plot and then saving it using `ggsave()`.

```
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")
```

```
#> Saving 5.5 x 3.5 in image
```

Explanation:

- `filename = "diamonds-plot"`: Specifies the name of the output file and the format (in this case, PNG).
- By default, `ggsave()` saves the most recently created plot.
- The plot is saved in your current working directory.

Customizing the Output

For reproducible results and to ensure your plots have consistent dimensions, it's a good practice to specify the size and resolution when saving your plots.

```
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
)
```

Explanation:

- `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.

> **Note**
>
> The `ggsave()` function has many additional arguments that allow for fine-tuned control over the saved image. To explore all the options, you can refer to the official documentation:
>
> `?ggsave`

5.4.8 Exercise 5.2: Data Analysis and Visualization with Medical Insurance Data

For this exercise, you will use Rstudio Project, call it `Experiment 5.2` 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. Download medical insurance data
    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.

5.5 Summary

In **Lab 5**, you have acquired essential skills in data analysis and visualization using R:

- **The Pipe Operator `|>`**: You learned how to use the pipe operator from the `magrittr` package to streamline your code. This operator allows you to chain multiple functions together, making your code more readable and intuitive by focusing on the flow of data through a sequence of operations.
- **Data Manipulation with `dplyr`**: You explored the core functions of the `dplyr` package—such as `select()`, `filter()`, `mutate()`, `arrange()`, and `summarise()`—to efficiently manipulate and transform datasets. These functions enable you to select specific columns, filter rows based on conditions, create new variables, sort data, and compute summary statistics.
- **Data Visualization with `ggplot2`**: You discovered how to create a variety of visualizations using the `ggplot2` package, which is based on the Grammar of Graphics. You practiced generating scatter plots, histograms, boxplots, and bar charts to explore and present data visually, enhancing your ability to identify patterns and insights.
- **Integrating Data Analysis and Visualization**: You learned how to combine data manipulation and visualization techniques to create a seamless analytical workflow. By preparing data with `dplyr` and visualizing it with `ggplot2`, you improved your capacity to tell compelling stories with data.

These advanced skills are crucial for any data analyst or scientist, as they enable you to work effectively with real-world datasets, extract meaningful insights, and communicate your findings through clear and impactful visualizations. Congratulations on elevating your R programming proficiency and advancing your expertise in data analysis and visualization!

6 Mastering R through Use Case Projects

R, like any programming language, is best understood not just through theory but through application. Once learners grasp the foundational elements of R, it's crucial to transition into real-world projects that allow for deeper understanding and retention of the material. This is where use case projects come in.

6.1 Why Use Case Projects?

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**: Projects can be added to a student's portfolio, showcasing their skills to potential employers or collaborators.

6.2 Sample Use Case Project: Televison Client Analysis

6.2.1 Background

A small television company is interested in understanding the factors that impact viewers' ratings of the company. Data has been collected from viewers who rated how highly they regard the television company (`regard`) and provided other related measures.

6.2.2 Data Structure

Variables in the dataset:

- **regard**: Viewer rating of the television company (higher ratings indicate higher regard).
- **gender**: Gender the viewer identifies with.
- **views**: Number of views.
- **online**: Number of times accessed bonus online material.
- **library**: Number of times browsed the online library.
- **Show1** to **Show4**: Scores for four different shows.

Download the dataset here.

6.2.3 Tasks

1. **Data Import and Cleaning**
    - Import the dataset into R.
    - Perform data cleaning: handle missing values, detect outliers, and ensure correct data types.
2. **Exploratory Data Analysis (EDA)**
    - Produce descriptive statistics and exploratory graphics for `regard`, focusing on patterns by `gender`.
    - Analyze scores for each of the four shows, highlighting differences by `gender`.
3. **Derived Variables**
    - Create a new variable `mean_show`, calculated as the mean of `Show1` to `Show4`.
4. **Correlation Analysis**
    - Test for a statistically significant linear correlation between `mean_show` and `regard`.
5. **Recommendation**
    - Write a short report (around 100 words) outlining your findings.
    - Include insights on `regard`, relevance to the client, and any limitations of your analysis.

6.3 Exercise 6.1: Analyzing a Rape Survey for the Federal Government of Nigeria

6.3.1 Project Overview

You have been consulted by the Federal Government of Nigeria to analyze a recent rape survey. As a data analyst with no specific instructions, your task is to use your analytical skills to uncover insights that will be valuable for the government.

6.3.2 Dataset

Download the dataset here.

6.3.3 Your Task

- **Data Analysis**: Perform a comprehensive analysis of the survey data.
- **Visualization**: Create visualizations that effectively communicate your findings.
- **Insights**: Identify key issues, trends, and patterns that are important for the government to understand.
- **Recommendations**: Provide suggestions or action items based on your analysis.

6.3.4 Presentation

Prepare to present your data product, explaining:

- **Functionality**: How you analyzed the data and what tools you used.
- **Design Choices**: Why you chose specific methods or visualizations.
- **Findings**: The key insights from your analysis.
- **Future Improvements**: How the analysis could be expanded or refined.

> **!** Wrap-Up
>
> By engaging in these exercises and projects, you're not just learning R—you're mastering it. The key to proficiency is consistent practice and application. These real-world scenarios will challenge you, enhance your problem-solving skills, and prepare you for future data analysis tasks. Keep exploring, stay curious, and continue to build upon the foundation you've established.

A 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.

A.1 Downloading the Data

1. **Access the Data Folder**

   Visit the following link to access the `r-data` folder on Google Drive:
   Google Drive - r-data Folder

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.

A.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.

A.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.

A.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.

A.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.

A.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!