Assignment 1

Summary of Chapters 8–20

R for Data Science by Hadley Wickham, Rundel and Garrett Grolemund

Gladwel Jelagat Sang

Registration No: SDS6/48759/2025

Date: 19 December 2025

University:	University of Nairobi
Programme:	Master of Science in Data Science
Course:	SDS 6103 – Statistical Computing
Year of Study:	Year 1 – 2025/2026

QUESTION 1

R Programming Language

R is a programming language and software environment specifically designed for statistical computing and data analysis. Created in the early 1990s by Ross Ihaka and Robert Gentleman at the University of Auckland, R has become one of the most widely used tools in statistics, data science, and research. R is open-source and free, with a large community of users who contribute thousands of packages that extend its functionality. These packages cover everything from data manipulation and visualization to machine learning and bioinformatics. One of R's greatest strengths is its powerful graphics capabilities, allowing users to create high-quality, publication-ready visualizations with relative ease.

The language is particularly popular in academic research, healthcare, finance, and any field requiring rigorous statistical analysis. R uses a syntax that can be intuitive for statistical operations, though it has a learning curve for those new to programming. Popular integrated development environments like RStudio make it easier to write and debug R code. R excels at working with structured data, performing complex statistical tests, and creating reproducible research through tools like R Markdown. While it may not be as fast as some compiled languages, R's extensive statistical libraries and visualization tools make it an essential skill for anyone working with data.

The installation of R.

Installing R is a straightforward process that varies slightly depending on your operating system, but the general steps remain accessible for users at all technical levels. The first step is to visit the Comprehensive R Archive Network (CRAN) at https://cran.r-project.org/, which serves as the official repository for R downloads. The website automatically detects your operating system and provides appropriate download links for Windows, macOS, or Linux.

While R can run independently, most users also install RStudio, a powerful integrated development environment (IDE) that makes working with R much more user-friendly. RStudio can be downloaded free from https://posit.co/downloads/. Install R first before installing RStudio, as the IDE requires R to be present on your system.After installation, launch R or RStudio to verify everything is working correctly. You should see the R console, where you can type commands. Try entering a simple command like 2 + 2 to confirm R is functioning properly. You can also check your R version by typing version in the console.

CHAPTER 1: DATA VISUALIZATION

Introduction

Data visualization is a crucial early step in data science that helps us quickly understand data by revealing trends, patterns, and outliers that aren't obvious in raw numbers. It should happen before formal statistical modelling because it builds intuition and helps us ask better questions. In R, ggplot2 is the main tool for creating effective visualizations.

The Three Building Blocks of ggplot2

ggplot2 is based on the grammar of graphics, which provides a logical system for building graphs. Every ggplot visualization requires three fundamental components:

· Data – the dataset being visualized.

· Aesthetics Mapping (aes) – how data variables connect to visual properties like x- axis, y-axis, color, shape, and size.

· Geometric objects (geom) – the visible marks that represent data, such as points, bars, and lines

Creating and Customizing Scatter Plots

Scatter plots display relationships between two numerical variables. Using the penguin dataset example, we can show the relationship between flipper length and body mass:

```
ggplot(
data = penguins,
mapping = aes(x = flipper_length_mm, y = body_mass_g)
) +
geom_point()
```

Enhancements include:

· Mapping additional variables like species to color and shape for visual differentiation between groups (Adelie, Chinstrap, and Gentoo)

· Adding a line of best fit using geom_smooth(method = “lm”) for linear regression

· Improving accessibility with colorblind-safe colors using scale_color_colorblind() from the ggthemes package

· Adding clear labels, including title, subtitle, x-axis, y-axis, and legend labels

Visualizing Distributions

Categorical Variables-A categorical variable can only take one of a small set of values. Use geom_bar() to create bar charts showing frequency. The height of bars shows observation counts for each category. Use fct_infreq() to reorder bars in descending order by frequency.

Numerical Variables-A numerical variable can take a wide range of values where addition, subtraction, or averaging makes sense. These can be continuous or discrete. Histograms visualize continuous variable distributions by dividing the x-axis into equally spaced bins, with bar height showing the count of observations in each bin. The binwidth argument controls interval width (measured in the x variable's units).

Visualizing Relationships

Between Categorical and Numerical Variables-Box plots (geom_boxplot()) allow side-by- side comparison. Each box shows the median (line in the middle)**,** Interquartile range/IQR (the box, from 25th to 75th percentile)**,** Outliers (points falling more than 1.5 times the IQR from box edges)**,** and Whiskers (extending to the farthest non-outlier points)**.** Density plots (geom_density()) show the distribution of numerical data by categories, with options for transparency and line thickness for better readability.

Between Two Categorical Variables-Use stacked bar plots to show category distribution within groups. The position = "fill" argument creates relative frequency plots, which are better for comparing distributions across categories.

Between Two Numerical Variables-Scatter plots (geom_point()) visualize relationships between numerical variables. Additional aesthetics like color and shape can represent extra variables, but overloading with too many aesthetics makes plots cluttered and hard to interpret.

Saving Plots

Use ggsave() to save the most recently created plot to disk. For reproducible code, always specify width and height.

CHAPTER 2: WORKFLOW – BASICS

Introduction

Data science requires more than just knowing tools and functions—it needs a smooth and effective workflow. This chapter focuses on using R and RStudio interactively to make experimentation, iteration, and learning easier. R should be viewed not just as a programming language, but as a tool for thinking with data.

Coding Basics

Basic Arithmetic-R performs basic arithmetic operations directly. 1 / 200 * 30

#> [1] 0.15

Creating Objects-The assignment operator (<-) creates new objects that store values for future use. The standard form is always object_name <- value. To view a stored value, type the object name in the console.

Working with Vectors-The c() function combines multiple elements into a vector. Arithmetic operations on vectors apply to every element.

primes <- c(2, 3, 5, 7, 11, 13)

primes * 2

#> [1] 4 6 10 14 22 26

Comments-R ignores text after the # symbol, allowing you to write comments. They explain why you did something, not what or how. Without comments explaining the reasoning, it's difficult or impossible to understand why decisions were made later. For example, if you change geom_smooth()'s span argument from 0.75 to 0.9, readers will see what changed but won't know why unless you comment your reasoning.

Naming Objects-The rules for naming objects are must start with a letter and can only contain letters, numbers, underscore (_), and period (.). The recommended style to use is snake_case (lowercase words separated by underscores) to make names descriptive. R requires precise instructions; mistakes in assignments will prevent correct computations**.**

Calling Functions-R has a large collection of built-in functions that perform most tasks. The function structure is function_name(argument1 = value1, argument2 = value2, …). Functions take input values called arguments, perform operations, and return outputs. This makes R powerful because the same operation can be applied to many kinds of inputs. For example, the seq() function creates regular sequences of numbers.

The Important syntax rule is that Quotation marks and parentheses must always come in pairs. If mismatched, R shows the continuation character "+" indicating something is missing. This can be fixed by either adding the missing pair or pressing ESCAPE to abort and try again.

CHAPTER 3: DATA TRANSFORMATION

Introduction

This chapter introduces the dplyr package, a powerful tool for data transformation that makes code expressive, concise, and readable. With dplyr, we can filter rows, rearrange data, create new variables, and generate summaries efficiently.

The Core Principles of dplyr

The dplyr package provides a consistent set of “verbs” (functions) for solving data manipulation challenges. These verbs work seamlessly with the pipe operator (|>) and are optimized for speed and readability. The dplyr verbs have common characteristics, which are,

· The data frame is always the first argument.

· Subsequent arguments describe which columns to operate on using variable names without quotes.

· The output is always a new data frame.

· dplyr functions never modify their inputs; instead, they create new data frames.

Dplyr verbs are organized into groups based on what they operate on: rows, columns, groups, and tables.

Working with Rows

filter()-Keeps rows based on values in columns. When you filter, dplyr creates and prints a new data frame without modifying the original dataset.

arrange()-Reorders rows by variable values. Use desc() to sort in descending order. The number of rows remains unchanged.

distinct()-Finds all unique rows in the dataset and removes duplicates.

Working with Columns

mutate()-Creates new variables or transforms existing ones without changing the number of rows. The New columns are added on the right-hand side by default. Use .before argument to add columns to the left and use .after to add columns after a specific variable. The keep argument controls which variables to retain. The "used" argument specifies that only columns involved or created in the mutate() step are kept

select()-It narrows down columns of interest when a dataset has many variables.

rename()-Use instead of select() when you want to keep all existing variables and just rename a few.

relocate()-Moves variables around within the data frame. By default, it moves variables to the front. Like mutate(), it uses .before and .after arguments to specify positioning.

Working with Groups

group_by()-Divides the dataset into meaningful groups for analysis. It doesn't change the data itself, but the output indicates the data is grouped.

summarize()-Calculates summary statistics and reduces the data frame to have a single row for each group.

The slice_ functions-Select rows by position rather than by values. They extract specific rows within each group.They include functions like(slice_max() - extracts rows with maximum values, slice_min() - extracts rows with minimum values, slice_head() - extracts first rows and slice_tail() - extracts last rows).

ungroup()-Removes grouping from a data frame without using summarize().

CHAPTER 4: WORKFLOW-CODE STYLE

Importance of Good Code Style

Good code style enhances readability, facilitates collaboration, ensures reproducibility, and reduces errors. Consistency prevents confusion and makes code easier to maintain for both team members and your future self.

The Key Styling Guidelines

Naming Conventions -Use short but descriptive names in snake_case format (lowercase letters, numbers, and underscores). Names should clearly reveal the object's purpose while avoiding extremes. It should neither be too short to be vague nor excessively long.

S**pacing Rules-**Add spaces around mathematical operators (+, -, ==, <) and assignment operators (<-)**.** Place spaces after commas when separating function arguments**.** There are no spaces inside or outside parentheses in function calls. Align equals signs (=) vertically in mutate() for improved readability.

Indentation with Pipes- Always place a space before the pipe operator (|>). The pipe should be the last element on a line. Indent subsequent piped steps by two spaces. For arguments on separate lines, use an additional two-space indent. Finally, place closing parentheses on their own line, unindented to align with the function name.

ggplot2-Apply the same principles as piping, treating the plus sign (+) like the pipe operator (|>). When arguments don't fit on one line, place each argument on its own separate line.

CHAPTER 5 – DATA TIDYING

Principles and Benefits of Tidy Data

Tidy data follows three fundamental rules: each variable forms a column, each observation forms a row, and each value occupies a single cell. This standardized structure enables seamless integration with all tidyverse tools, including dplyr and ggplot2.

Maintaining tidy data offers two primary advantages:

· It creates consistency that makes learning and using analytical tools easier due to underlying uniformity.

· It leverages R's vectorized nature by placing variables in columns, allowing built-in functions to work efficiently with vectors of values.

The reason why data is often untidy is that real-world data is organized for purposes other than analysis (such as simplified data entry) and because many people lack familiarity with tidy data principles. Consequently, most analyses require some degree of data tidying.

Lengthening Data with pivot_longer()

This function addresses the common problem where variables are stored in column names instead of proper columns. It requires three key arguments:

· cols: specifies which columns to pivot (using select() syntax)

· names_to: names the variable stored in column names

· values_to: names the variable stored in cell values

The function can use values_drop_na = TRUE to remove structural missing values (created by data organization) while preserving genuine missing values (unknown information).

How Pivoting Works

During reshaping, existing variable values repeat for each pivoted column, column names become values in a new variable (defined by names_to) and repeat for each original row, and cell values become values in a new variable (defined by values_to), unwound row by row.

Handling Complex Column Names

Multiple Variables in Column Names-When column names contain multiple pieces of information, use a vector for names_to with names_sep to split names into pieces, or use names_pattern for irregular naming schemes.

Mixed Variable Names and Values- When column names mix variable values and names, use the special ".value" sentinel in names_to, which tells pivot_longer() to use the first component of the pivoted column name as a variable name in the output.

Widening Data with pivot_wider()

This function increases columns while reducing rows, useful when one observation spans multiple rows. It requires:

· values_from: columns that define the values

· names_from: column providing new column names

· id_cols: columns that uniquely identify each row (may be needed to prevent duplication)

How pivot_wider() Operates

The function first determines row and column structure using unique values from the measurement column and identifying rows through non-pivoted variables (id_cols). It then creates an empty data frame and fills missing values using the input data.If multiple input rows correspond to one output cell, pivot_wider() produces list-columns, signaling data

quality issues that require repair through grouping and summarizing to ensure each row- column combination has only a single value.

CHAPTER 6 – WORKFLOW: SCRIPTS AND PROJECTS

Introduction

This chapter covers two essential organizational tools for coding: scripts and projects, which help manage and structure your R code effectively.

Scripts

Scripts are necessary for building complex ggplot2 graphics and longer dplyr pipelines. Create a new script through File > New File > R script or using Cmd/Ctrl+Shift+N. Scripts allow you to edit and re-run code easily, saving working code for future reference. When running a code, there are two essential keyboard shortcuts for efficient workflow:

· Cmd/Ctrl+Enter: Executes the current R expression in the console and moves the cursor to the next statement for easy stepping through your script.

· Cmd/Ctrl+Shift+S: Executes the complete script in one step.

Always start scripts with the packages you need so others know which packages to install when sharing code. Also, practice using keyboard shortcuts to send code from the script editor to the console efficiently.

RStudio Diagnostics - RStudio highlights syntax errors with red squiggly lines and shows a cross in the sidebar, helping you identify and understand problems in your code.

Saving and Naming

RStudio automatically saves script contents when you quit and reloads them when you reopen. Use informative file names following these principles:

- Machine-readable: Don’t rely on case sensitivity to distinguish files
- Human-readable: Use descriptive file names
- Default ordering compatible: Start file names with numbers for alphabetical sorting

Number key scripts in running order, use consistent naming schemes, label figures appropriately, and distinguish reports by dates in file names. For directories with many files, organize different file types into separate folders.

Projects

Handling Real-World Analysis

Real-world analysis requires quitting R and returning later, working on multiple analyses simultaneously, or sharing work with others. This necessitates two key decisions, which are identifying the source of truth and determining where your analysis lives.

The Source of Truth-R scripts, along with data files, should be your source of truth, as they allow you to recreate your environment. The reverse recreating scripts from only your environment is difficult and error-prone. For you to maintain scripts as the source of truth

- Run usethis::use_blank_slate() or manually configure RStudio settings
- Set “Restore .RData into workspace at startup” to off
- Set “Save workspace to .RData on exit” to “Never”

This creates short-term inconvenience (restarting RStudio won’t remember previous code, objects, or datasets) but prevents long-term problems by forcing you to capture all important procedures in code. Press, Cmd/Ctrl + Shift + 0/F10 to restart R and Cmd/Ctrl + Shift + S to re-run the current script.

Where Does Your Analysis Live?-R uses the concept of a working directory for loading and saving files. RStudio displays the current working directory at the top of the console, and you can print it using getwd(). While you can technically set the working directory using setwd(), this is not recommended. Instead, use RStudio projects, which automatically manage working directories.

Creating RStudio Projects

RStudio projects keep all associated files (input data, R scripts, analytical results, and figures) together in one directory.

Steps for creating RStudio Projects

- Click File > New Project
- Select “New Directory”
- Choose “New Project”
- Provide a meaningful name and choose the save location

Once created, the project home becomes the working directory. An .Rproj file is created in the project folder. Double-clicking this file reopens the project with the same working directory and command history, with previously open files accessible, but with a completely fresh environment (clean slate). The advantage is that saving figures to files using R code (rather than the mouse or clipboard) ensures reproducibility. You can always recreate your work and find figures in your project directory.

Relative vs. Absolute Paths

Always use relative paths (relative to the working directory/project home) inside projects, not absolute paths. For example, data/diamonds.csv is a relative path that works on anyone’s computer regardless of their directory structure. Absolute paths point to the same place regardless of the working directory, but look different across operating systems. In Windows, it starts with drive letters (C:) or double backslashes\\ while in Mac/Linux, it starts with a slash (/). Never use absolute paths in scripts, as they hinder sharing.

Operating System Path Differences

Mac and Linux use forward slashes (/) in paths, while Windows uses backslashes (\). R accepts either type, but backslashes have special meaning in R, requiring you to type two backslashes for each actual backslash. To avoid frustration, always use forward slashes (Linux/Mac style).

CHAPTER 7 – DATA IMPORT

Introduction

This chapter focuses on reading plain-text rectangular files into R, particularly comma- separated values (CSV) files, using the readr package.

Reading Data from a File

We can read data from a file using the read_csv() function into R. In a CSV file, the first row (header) provides column names, subsequent rows contain data, and columns are separated (delimited) by commas. When you run read_csv(), it prints a message showing the number of rows and columns delimiter used, column specifications, and information about retrieving full column specifications and quieting the message.

Practical Data Cleaning

After reading data, the first step is transforming it into a workable form for analysis. This addresses common data issues such as non-syntactic column names (containing spaces), missing values represented as "N/A", and inconsistent data entry. Use janitor::clean_names() to convert column names to snake_case and the na argument to specify which values should be treated as missing. After reading data, consider variable types. The if_else() function has three arguments: A logical vector (the test), the value to return when TRUE (yes argument), and the value to return when FALSE (no argument).

Controlling Column Types

Guessing Types -Readr uses a heuristic algorithm to infer column types by sampling 1,000 rows evenly distributed from first to last (ignoring missing values). It applies these rules sequentially:

· Logical: Contains only F, T, FALSE, or TRUE

· Numeric: Contains only numbers (1, -4.5, 5e6, Inf)

· Date/Date-Time: Matches ISO8601 standard format

· String: Default if none of the above apply

This heuristic works well for clean data but can fail with messy real-world datasets.

Missing Values, Column Types, and Problems-Column detection fails when it contains unexpected values, resulting in a character column instead of a more specific type. This is often caused by missing values recorded using something other than the NA that readr expects. The readr provides nine column type functions:

· col_logical() and col_double(): Read logicals and real numbers

· col_integer(): Reads integers

· col_character(): Reads strings; useful for numeric identifiers (phone numbers, IDs, credit cards)

· col_factor(), col_date(), and col_datetime(): Create factors, dates, and datetimes respectively

· col_number(): A permissive numeric parser that ignores non-numeric components

· col_skip(): Skips a column so it's not included in the result; useful for large files

Reading Data from Multiple Files

When data is split across multiple files, read_csv() can import and automatically stack them on top of each other in a single data frame. The id argument creates a new column identifying each observation's source file, which is crucial for maintaining data provenance. Pattern- based file selection with list.files() eliminates the need for manual file listing.

Writing Data to a File

The readr package provides write_csv() and write_tsv() as useful functions for writing data back to disk. Important arguments to note are x (the data frame to save) and file(the location to save it). Also, remember to specify how missing values are written with na. Option to append to an existing file. When you read the CSV file back in, the variable type information you previously set up is lost. This happens because when you save to CSV and read it back, you're starting over with reading from a plain text file again.

Manual Data Entry

Manual data entry involves entering data by hand into an R script using two functions that differ in whether you layout the tibble by columns or by rows. When it's hard to see how rows are related in column layout, use tibble() because it lays out data row by row. It is customized for data entry in code; column headings start with ~, and the entries are separated by commas. Making it possible to lay out small amounts of data in an easy-to-read form

Chapter 8: Getting Help

Getting help is a crucial skill for every R user, from beginner to professional data scientist. R provides an extensive, well-integrated help system that allows you to look up function documentation, explore examples, and understand package behavior. In this book,Wickham emphasizes that knowing how to search for help efficiently makes learning faster and reduces frustration.

Using R's Built-In Help System

1. '?function_name' .This displays a help page for a function.

Example:

"'r

?mean

"'

This opens documentation explaining the usage, arguments, return values, and examples for 'mean()'.

2. 'help("topic")' .Equivalent to using '?'.

Example:

"'r

help("median")

"'

3. Searching by Keyword — '??' .This searches all installed packages for matching help files.It is useful when you know the concept but not the specific function.

Example:

"'r

??regression

"'

Reading a Help File Effectively

Help files follow a consistent structure:

Description — what the function does

* **Usage— the function's arguments

* **Arguments** — definitions of each parameter

* **Details** — deeper explanation

* **Value** — what the function returns

* **Examples** — reproducible R examples

Example snippet:

“‘r

?sd

“‘

Check examples at the bottom:

“‘r

x <- 1:10

sd(x)

“‘

## **8.3 Getting Help About Packages**

### **View package documentation**

“‘r

help(package = “dplyr”)

“‘

### **Show all functions in a package**

“‘r

library(help = dplyr)

“‘

## **8.4 Using Vignettes**

Vignettes help you learn a package through rich tutorials.

### **View all vignettes**

“‘r

vignette()

“‘

### **Open a specific vignette**

“‘r

vignette(“dplyr”)

“‘

### **Search for vignettes**

“‘r

vignette(package = “ggplot2”)

“‘

## **8.5 Using External Help Resources**

### **1. RStudio Help Panel**

Provides documentation and links to examples automatically.

### **2. Stack Overflow**

Search format:

“‘

“R how to reorder factor levels”

“‘

### **3. GitHub Issues**

Helpful when using development versions of packages.

### **4. Posit Community**

Active discussions for R beginners and experts.

Asking For Help the Right Way

When asking for help, Wickham recommends following the Reprex principle a reproducible example.

Create a reproducible example

“‘r

install.packages(“reprex”)

```
library(reprex)

reprex({

  mean(c(1, 2, “a”))

})
```
“‘

This allows others to replicate your error exactly.

Conclusion

Chapter 8 teaches you not just how to look up help, but how to understand documentation and communicate effectively when asking for assistance. Efficient use of help resources accelerates your progress as an R learner and analyst.

Chapter 9: Layers (ggplot2 Layers)

Chapter 9 introduces one of the core concepts of **ggplot2**: the **layered grammar of graphics**. Graphics are built from independent layers that combine data, aesthetics, and geoms.

This chapter teaches you how to stack layers to construct complex visualizations step by step.

## **9.1 The Structure of a Layer**

A ggplot layer includes:

1. **Data** — the dataset used

2. **Mapping** — aesthetic mappings, e.g. ‘aes(x, y)‘

3. **Geom** — geometric object (point, line, bar, etc.)

4. **Stat** — statistical transformation (binning, smoothing)

5. **Position** — how elements are placed (stack, dodge, jitter)

Example:

“‘r

```
ggplot(data = mpg, aes(x = displ, y = hwy)) +

  geom_point()
```
“‘

This creates a scatter plot with one layer.

## **9.2 Adding Multiple Layers**

Layers are added with ‘+‘.

Example:

“‘r

ggplot(mpg, aes(displ, hwy)) +

  geom_point() +

  geom_smooth()

“‘

Here:

* ‘geom_point()‘ → raw data
* ‘geom_smooth()‘ → statistical summary (smoothed trend)

## **9.3 Layer Components in Detail**

### **Aesthetic Mapping ‘aes()‘**

Example:

“‘r

geom_point(aes(color = class, size = cyl))

“‘

Aesthetics can be mapped:

* ‘color‘
* ‘size‘
* ‘shape‘
* ‘fill‘
* ‘alpha‘

### **Choosing Geoms**

Common geoms:

Task	Geom
Scatter plot	‘geom_point()‘
Line plot	‘geom_line()‘
Bar chart	‘geom_bar()‘
Boxplot	‘geom_boxplot()‘
Density plot	‘geom_density()‘

Example:

“‘r

geom_boxplot(aes(x = class, y = hwy))

“‘

### **Stats (Statistical Transformations)**

Each geom has a default stat, but you can override it.

Example: manually compute counts:

“‘r

geom_bar(stat = “identity”)

“‘

Or display a smooth model:

“‘r

geom_smooth(method = “lm”)

“‘

### **Position Adjustments**

Avoid overlapping points:

“‘r

geom_jitter()

“‘

Side-by-side bars:

“‘r

geom_bar(position = “dodge”)

“‘

Stack bars:

“‘r

geom_bar(position = “stack”)

“‘

## **9.4 Layer Inheritance**

A layer can inherit mappings from the plot or override them.

Example overriding:

“‘r

ggplot(mpg, aes(displ, hwy)) +

  geom_point(aes(color = class)) +

  geom_smooth(aes(linetype = drv), se = FALSE)

“‘

## **9.5 Building Plots Step-by-Step**

Start with base plot:

“‘r

p <- ggplot(mpg, aes(displ, hwy))

“‘

Add layers:

“‘r

p + geom_point()

p + geom_point() + geom_smooth()

“‘

## **9.6 Clear, Effective Graphics**

Wickham emphasizes:

* Use color and shape intentionally
* Avoid over-plotting
* Layer data and summaries
* Add labels and titles for clarity

Example:

“‘r

ggplot(mpg, aes(displ, hwy, color = class)) +

  geom_point() +

  geom_smooth(se = FALSE) +

  labs(title = “Fuel Efficiency vs Engine Size”,

       x = “Engine Displacement”,

       y = “Highway MPG”)

“‘

## **9.7 Summary**

Layers allow you to build visualizations piece by piece. Understanding how to combine data, aesthetics, geoms, stats, and position adjustments is essential for effective exploratory and communicative graphics.

Chapter 10: Exploratory Data Analysis (EDA)

Exploratory Data Analysis (EDA) is the process of understanding your data before applying models or statistical tests. In *R for Data Science (2nd Edition)*, Wickham highlights that EDA is about forming questions, visualizing patterns, discovering problems in the data, and guiding further analysis.

EDA uses two major tools:

1. **Visualization** (mainly with ‘ggplot2‘)

2. **Transformation** (mainly with ‘dplyr‘)

Together, these help you summarize the structure, variability, and relationships in your dataset.

## **10.1 Asking Good Questions**

EDA always begins with clear questions:

* What type of variation exists?

* What relationships exist between variables?

* Are there patterns or anomalies?

* Are there missing values?

* Are there outliers?

Example (using ‘mpg‘ dataset):

“‘r

summary(mpg)

“‘

## **10.2 Variation Within a Variable**

### **Numerical Variables**

Histograms, density plots, and boxplots show the spread and distribution.

Example:

“‘r

```
ggplot(mpg, aes(hwy)) +
  geom_histogram(binwidth = 2)
```

“‘

Density plot:

“‘r

```
ggplot(mpg, aes(hwy)) +
  geom_density()
```

“‘

Boxplot:

“‘r

```
ggplot(mpg, aes(y = hwy)) +
  geom_boxplot()
```

“‘

### **Categorical Variables**

Use bar charts to show counts.

Example:

“‘r

```
ggplot(mpg, aes(class)) +
  geom_bar()
```

“‘

## **10.3 Exploring Covariation (Relationships Between Variables)**

### **Numerical vs Numerical**

Scatter plots help uncover trends and clusters.

Example:

“‘r

ggplot(mpg, aes(displ, hwy)) +

  geom_point()

“‘

Adding a smooth line:

“‘r

geom_smooth()

“‘

### **Categorical vs Numerical**

Boxplots or violin plots are ideal.

Example:

“‘r

ggplot(mpg, aes(class, hwy)) +

  geom_boxplot()

“‘

### **Categorical vs Categorical**

Use a heatmap or grouped bar chart.

Example:

“‘r

ggplot(mpg, aes(class, fill = drv)) +

geom_bar(position = “dodge”)

“‘

## **10.4 Handling Outliers**

Outliers may represent errors or genuine rare observations.

Example:

“‘r

ggplot(mpg, aes(hwy)) +

  geom_boxplot()

“‘

To filter extreme values:

“‘r

mpg %>%

  filter(hwy < 40)

“‘

## **10.5 Patterns and Anomalies**

EDA helps reveal:

* Data-entry mistakes

* Unexpected shapes in distributions

* Missing-data patterns

* Nonlinear relationships

Example: Checking missing values

“‘r

colSums(is.na(mpg))

“‘

## **10.6 Transformations in EDA**

### **Using ‘dplyr‘ to summarize patterns**

Example:

“‘r

mpg %>%

  group_by(class) %>%

  summarise(mean_hwy = mean(hwy))

“‘

### **Creating new variables**

“‘r

mpg %>%

  mutate(ratio = hwy/displ)

“‘

## **10.7 EDA Workflow**

1. Generate visualizations
2. Transform data
3. Ask questions
4. Visualize again
5. Refine questions

Example workflow:

“‘r

ggplot(mpg, aes(displ, hwy)) + geom_point()

mpg %>% filter(displ < 7)

ggplot(mpg, aes(displ, hwy, color = class)) + geom_point()

“‘

## **10.8 Summary**

EDA is an iterative, question-driven process. It relies on visualization and transformation tools to uncover insights and prepare data for modeling. Wickham stresses that models confirm patterns; EDA *discovers* them.

# **Chapter 11: Communication**

In data science, communicating results is just as important as performing analysis. Chapter 11 focuses on presenting your findings clearly through visualizations, tables, text, and reproducible reports (e.g., Quarto).

The goal is to make your audience understand:

* **What you did**

* **What you found**

* **Why it matters**

## **11.1 The Components of Effective Communication**

Effective communication involves:

1. **Clarity**

2. **Accuracy**

3. **Purpose**

4. **Audience awareness**

## **11.2 Improving Visualizations for Communication**

Exploratory visuals are messy. Communication graphics must be polished.

### **Add labels**

“‘r

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  labs(
    title = “Engine Size vs Highway MPG”,
    x = “Displacement (L)”,
    y = “MPG on Highway”
  )
```

“‘

### **Add themes**

“‘r

```
+ theme_minimal()
```

“‘

### **Focus attention with color**

Use color only when necessary.

Example:

“‘r

```
aes(color = class)
```

“‘

## **11.3 Telling a Story with Data**

Communicative graphics emphasize:

* The main finding
* Relevant comparisons
* Clear interpretations

Example workflow:

```r
mpg %>%
  group_by(class) %>%
  summarise(avg_hwy = mean(hwy)) %>%
  ggplot(aes(class, avg_hwy, fill = class)) +
  geom_col() +
  labs(title = "Average Highway MPG by Vehicle Class")
```

## **11.4 Tables and Summaries**

Tables are useful when exact values matter.

Example using ‘dplyr‘:

```r
mpg %>%
  group_by(drv) %>%
  summarise(
    mean_hwy = mean(hwy),
    sd_hwy = sd(hwy),
    count = n()
  )
```

For better formatting, use:

“‘r

library(knitr)

kable(...)

“‘

## **11.5 Writing Effective Text Explanations**

Good text:

* Interprets graphics

* Highlights important trends

* Avoids unnecessary technical jargon

* Connects findings back to the question

Example explanation:

> Vehicles with front-wheel drive show the highest average fuel efficiency, particularly in smaller engine sizes. This trend indicates that drivetrain and engine displacement influence highway MPG.

## **11.6 Reproducible Communication with Quarto**

Quarto allows integration of:

* Code

* Visuals

* Narrative text

Example document header:

“‘yaml

---

title: “Fuel Efficiency Analysis”

format: html

“‘

Run R code inside:

““r

“‘{r}

ggplot(mpg, aes(displ, hwy)) + geom_point()

““

““

This ensures your results can be reproduced and updated automatically when data changes.

## **11.7 Ethical Communication**

Wickham emphasizes:

- Avoid misleading scales
- Avoid cherry-picking
- Show uncertainty when possible
- Ensure transparency

Example uncertainty:

“‘r

geom_smooth(se = TRUE)

““

## **11.8 Summary**

Communication transforms analysis into understanding. Clear visuals, structured writing, and reproducible documents ensure that your audience grasps your findings and trusts your conclusions.

Chapter 12: Logical Vectors

Logical vectors are one of the fundamental data types in R. They are central to data filtering, comparisons, missing-value handling, and control structures. Logical values in R include:

* ‘TRUE‘

* ‘FALSE‘

* ‘NA‘ (unknown or missing)

In *R for Data Science (2nd Edition)*, logical vectors appear repeatedly in data transformation, visualization, and modeling because they help you select data, create new conditions, and evaluate expressions.

## **12.1 Creating Logical Vectors**

Logical vectors often come from comparisons:

Example:

“‘r

x <- c(1, 5, 10)

x > 3

“‘

Output:

“‘

FALSE TRUE TRUE

“‘

### **Common comparison operators**

Operator	Meaning
‘==‘	equal to
‘!=‘	not equal
‘>‘	greater than
‘<‘	less than
‘>=‘	greater or equal
‘<=‘	less or equal

Example:

“‘r

c(“a”, “b”, “c”) == “b”

“‘

## **12.2 Logical Operations**

Logical operators combine multiple logical conditions.

### **AND (‘&‘)**

Both conditions must be TRUE.

“‘r

x > 3 & x < 9

“‘

OR (‘|‘)

At least one condition must be TRUE.

“‘r

x < 2 | x > 8

“‘

### **NOT (‘!‘)**

Reverses TRUE FALSE.

“‘r

!TRUE

“‘

## **12.3 Logical Vectors in Data Filtering (‘dplyr‘)**

Logical vectors are essential for filtering rows.

### **Filtering example**

“‘r

mpg %>%

  filter(hwy > 30)

“‘

Example with two conditions:

“‘r

mpg %>%

  filter(hwy > 25 & class == “compact”)

“‘

Using ‘|‘:

“‘r

mpg %>%

  filter(class == “compact” | class == “subcompact”)

“‘

## **12.4 Missing Values and Logic**

‘NA‘ complicates logical comparisons because ‘NA‘ means “unknown.”

Example:

“‘r

NA == 5

“‘

Result:

“‘

NA

“‘

### **Checking for missing values**

Use ‘is.na()‘:

“‘r

x <- c(1, NA, 3)

is.na(x)

“‘

### **Filtering out NAs**

“‘r

x[!is.na(x)]

“‘

## **12.5 Logical Vectors in Subsetting**

Subsetting uses logical vectors to select elements.

Example:

“‘r

x[x > 5]

“‘

With missing values:

“‘r

x[x > 5 & !is.na(x)]

“‘

## **12.6 Counting TRUE values**

TRUE = 1 and FALSE = 0.

So you can count using ‘sum()‘:

Example:

“‘r

sum(mpg$hwy > 30)

“‘

Count proportion:

“‘r

mean(mpg$hwy > 30)

“‘

## **12.7 Vector Recycling**

R recycles shorter vectors when comparing.

Example:

“‘r

c(TRUE, FALSE) & c(TRUE, TRUE, FALSE, FALSE)

“‘

Recycled:

“‘

TRUE TRUE FALSE FALSE

“‘

Use carefully to avoid errors.

## **12.8 Summary**

Logical vectors are essential tools for comparisons, filtering, handling missing values, and subsetting. Understanding how TRUE, FALSE, and NA behave ensures accurate data analysis and prevents subtle errors.

# **Chapter 13: Numbers**

Chapter 13 explains how R stores and processes numerical values. Understanding numeric precision, types, rounding, and arithmetic is essential for accurate computation.

R has two main types of numbers:

1. **Integers**

2. **Doubles (floating point numbers)**

Most numbers in R are stored as doubles.

## **13.1 Types of Numbers**

### **Check type**

“‘r

typeof(10)

typeof(10L)

“‘

* ‘10‘ → double

* ‘10L‘ → integer

### **Testing**

“‘r

is.integer(10L)

is.double(10)

“‘

## **13.2 Floating Point Precision**

Floating point numbers are approximations.

Example:

“‘r

0.1 + 0.2

“‘

Result (not exact):

“‘

0.30000000000000004

“‘

### **Use ‘near()‘ from dplyr**

“‘r

near(0.1 + 0.2, 0.3)

“‘

## **13.3 Arithmetic Operations**

Basic arithmetic:

“‘r

5 + 3

5 - 2

5 * 2

5 / 2

5^2

“‘

Integer division:

“‘r

5 %/% 2

“‘

Modulus:

“‘r

5 %% 2

“‘

## **13.4 Special Numeric Values**

### **1. ‘Inf‘ (Infinity)**

“‘r

1 / 0

“‘

### **2. ‘-Inf‘**

“‘r

-1 / 0

“‘

### **3. ‘NaN‘ (Not a Number)**

Invalid operations:

“‘r

0 / 0

“‘

### **4. ‘NA‘**

Missing numeric value.

## **13.5 Rounding Functions**

### **Round to nearest**

“‘r

round(3.567, 2)

“‘

### **Round up**

“‘r

ceiling(3.1)

“‘

### **Round down**

“‘r

floor(3.9)

“‘

### **Truncate**

“‘r

trunc(3.9)

“‘

## **13.6 Generating Sequences of Numbers**

### **Using ‘:‘**

“‘r

1:10

“‘

### **‘seq()‘**

“‘r

seq(0, 1, by = 0.1)

“‘

### **‘rep()‘**

“‘r

rep(5, 3)

“‘

## **13.7 Statistical Summaries**

### **Mean**

“‘r

mean(mpg$hwy)

“‘

### **Median**

“‘r

median(mpg$hwy)

“‘

### **Standard deviation**

“‘r

sd(mpg$hwy)

“‘

### **Quantiles**

“‘r

quantile(mpg$hwy)

“‘

## **13.8 Converting Types**

Convert to integer:

“‘r

as.integer(5.8)

“‘

Convert to double:

“‘r

as.double(5L)

“‘

## **13.9 Handling Numeric Problems**

### **Overflow**

Numbers too large:

“‘r

exp(1000)

“‘

Produces ‘Inf‘.

### **Underflow**

Very small numbers:

“‘r

exp(-1000)

“‘

Produces ‘0‘.

## **13.10 Summary**

Numbers in R are mostly stored as doubles. Understanding floating point precision, numeric types, special values, rounding, and arithmetic safeguards you against common numerical errors. These skills are essential for statistical modeling, simulation, and data cleaning.

Chapter 14: Strings

Strings represent text data in R and are essential for cleaning, transforming, and analyzing textual information. Chapter 14 from *R for Data Science (2nd Edition)* explains how to work with strings using base R and the **stringr** package (part of the tidyverse). 'stringr' provides consistent, easy-to-use functions.

## **14.1 Creating Strings**

Strings are created using quotes:

"'r

x <- "Hello world"

"'

To include quotes inside a string:

"'r

z <- "He said, \"Hi\" "

"'

Multiple strings create a character vector:

"'r

c("apple", "banana", "pear")

"'

## **14.2 String Length**

Count characters with:

“‘r

stringr::str_length(“data”)

“‘

For a vector:

“‘r

str_length(c(“a”, “ab”, “abc”))

“‘

## **14.3 Combining Strings**

### **Using ‘str_c()‘**

“‘r

str_c(“Data”, “Science”)

“‘

Add separators:

“‘r

str_c(“Data”, “Science”, sep = ” “)

“‘

Collapse multiple values:

“‘r

str_c(c(“a”, “b”, “c”), collapse = “,”)

“‘

## **14.4 Subsetting Strings**

Use ‘str_sub()‘ to extract parts:

“‘r

str_sub(“statistics”, 1, 5)

“‘

Output: “‘stati”‘

Negative indexing counts from the end:

“‘r

str_sub(“statistics”, -3, -1)

“‘

Output: “‘ics”‘

You can also modify strings:

“‘r

x <- “statistics”

str_sub(x, 1, 1) <- “S”

“‘

## **14.5 Changing Letter Case**

### **Lowercase**

“‘r

str_to_lower(“HELLO”)

“‘

### **Uppercase**

“‘r

str_to_upper(“hello”)

“‘

### **Title case**

“‘r

str_to_title(“data science”)

“‘

## **14.6 Detecting Patterns in Strings**

Use ‘str_detect()‘:

“‘r

str_detect(“data science”, “data”)

“‘

Returns TRUE or FALSE.

For vectors:

“‘r

str_detect(c(“apple”, “banana”, “pear”), “a”)

“‘

## **14.7 Extracting Patterns**

Use ‘str_extract()‘:

“‘r

str_extract(“My number is 245”, “\\d+”)

“‘

Extract multiple matches:

“‘r

str_extract_all(“a1 b22 c333”, “\\d+”)

“‘

## **14.8 Replacing Text**

### **Replace first match**

“‘r

str_replace(“I like cats”, “cats”, “dogs”)

“‘

### **Replace all matches**

“‘r

str_replace_all(“1 + 2 = 3”, “\\d”, “X”)

“‘

## **14.9 Splitting Strings**

Use ‘str_split()‘:

“‘r

str_split(“a,b,c”, “,”)

“‘

Split into characters:

“‘r

str_split(“data”, “ “)

“‘

## **14.10 Trimming and Padding Strings**

Trim whitespace:

“‘r

str_trim(“ tidyverse ”)

“‘

Pad a string:

“‘r

str_pad(“12”, width = 4, side = “left”, pad = “0”)

“‘

Result: “‘0012”‘

## **14.11 Matching Entire Words**

Use boundaries:

“‘r

str_detect(“data science”, “\\bdata\\b”)

“‘

## **14.12 Summary**

String manipulation is essential for text cleaning, extraction, and preparation. The **stringr** functions provide a consistent and predictable syntax. Skills in combining, detecting, replacing, and splitting strings are crucial for data cleaning and text analysis.

Chapter 15: Regular Expressions

Regular expressions (regex) are powerful tools for pattern matching in text. They allow you to find, validate, extract, replace, or manipulate specific text patterns. Chapter 15 introduces regex through the stringr package.

Regex is essential in data cleaning, web scraping, text mining, and data preparation.

## **15.1 Literal Matches**

Basic regex matches exact text.

Example:

“‘r

str_detect(“apple”, “app”)

“‘

Match multiple words:

“‘r

str_detect(c(“cat”, “dog”, “car”), “ca”)

“‘

## **15.2 Character Classes**

Character classes match any one character from a set.

Pattern	Meaning
‘[abc]‘	a, b, or c
‘[0-9]‘	any digit
‘[a-z]‘	any lowercase letter

Example:

“‘r

str_detect(“room2”, “[0-9]”)

“‘

## **15.3 Predefined Character Classes**

Class	Meaning
—	————
‘\\d‘	digit
‘\\D‘	non-digit
‘\\s‘	whitespace
‘\\w‘	word character

Example:

“‘r

str_detect(“abc123”, “\\d”)

“‘

## **15.4 Quantifiers**

Quantifiers control how many repetitions to match.

Quantifier	Meaning
———-	—————
‘?‘	0 or 1
‘+‘	1 or more
‘*‘	0 or more
‘{n}‘	exactly n
‘{n,}‘	n or more
‘{n,m}‘	between n and m

Example:

“‘r

str_detect(“aaa”, “a+”)

“‘

Match three digits:

“‘r

str_detect(“245”, “\\d{3}”)

“‘

## **15.5 Anchors**

Anchors match positions rather than characters.

Anchor	Meaning
‘^‘	start of string
‘$‘	end of string
‘\\b‘	word boundary

Examples:

“‘r

str_detect(“data”, “^da”)

str_detect(“analysis”, “sis$”)

“‘

Match whole word:

“‘r

str_detect(“big data analytics”, “\\bdata\\b”)

“‘

## **15.6 Escaping Special Characters**

Characters like ‘.‘, ‘+‘, ‘?‘, ‘|‘, ‘(‘, ‘)‘ have special meanings.

Escape them with ‘\\‘.

Example: match a dot

“‘r

str_detect("1.5", "\\.")

"'

Match parentheses:

"'r

str_detect("(x+1)", "\\(")

"'

## **15.7 Groups and Alternation**

### **Grouping with '()'**

"'r

str_extract("2025-01-10", "(\\d{4})")

"'

### **Alternation using '|'**

Match "cat" OR "dog":

"'r

str_detect("I love cats", "cat|dog")

"'

## **15.8 Extracting with Regex**

### **Extract first match**

"'r

str_extract("Phone: 555-1234", "\\d{3}-\\d{4}")

"'

### **Extract all matches**

"'r

str_extract_all("a1 b22 c333", "\\d+")

"'

## **15.9 Replacing with Regex**

Replace patterns:

“‘r

str_replace(“abc123def”, “\\d+”, “NUMBER”)

“‘

Replace all:

“‘r

str_replace_all(“12 34 56”, “\\d”, “X”)

“‘

## **15.10 Practical Examples**

### **1. Validate email**

“‘r

str_detect(“test@gmail.com”, “^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,}$”)

“‘

### **2. Extract year from date**

“‘r

str_extract(“2025-12-11”, “^\\d{4}”)

“‘

### **3. Remove all non-alphabet characters**

“‘r

str_replace_all(“A1B2C3”, “[^A-Za-z]”, “ “)

“‘

## **15.11 Summary**

Regular expressions allow precise control of text matching and manipulation. You can detect, extract, replace, and validate text using patterns. Mastering regex dramatically enhances your data cleaning and text processing capabilities.

CHAPTER 16 — FACTORS

Factors in R are used for **categorical data**, where the values belong to a finite set of categories. Proper handling of factors is crucial for **modeling, plotting, and summarizing**.

### 1. What is a factor?

A factor stores:

* **Levels** — the unique categories

* **Integer codes** referencing those levels

Example:

“‘r

x <- factor(c(“high”, “low”, “medium”, “low”))

x

levels(x)

“‘

* Output shows levels: “‘high” “low” “medium”‘

* Internally, R stores integers referencing levels.

### 2. Why factors matter

* Ensure categories have defined values

* Support **ordering** for ordinal data

* Improve **plots** and **tables**

* Prevent typos (“male” vs “Male”)

“‘r

gender <- factor(c(“male”, “female”, “male”))

summary(gender)

“‘

### 3. Creating factors

Using ‘factor()‘:

“‘r

sizes <- factor(c(“small”, “medium”, “large”))

“‘

Using **forcats** package (part of tidyverse):

“‘r

library(forcats)

fct_count(sizes) # Frequency table

“‘

### 4. Ordered factors

For ordinal data, specify ‘ordered = TRUE‘:

“‘r

sizes <- factor(c(“small”, “medium”, “large”),

levels = c(“small”, “medium”, “large”),

ordered = TRUE)

“‘

* Useful in **plots** or **modeling** where order matters.

### 5. Reordering factors

Reorder for better plots:

“‘r

sales <- data.frame(product = c(“A”,“B”,“C”), revenue = c(100, 200, 50))

library(ggplot2)

ggplot(sales, aes(fct_reorder(product, revenue), revenue)) +

geom_col()

“‘

* ‘fct_reorder()‘ orders levels by a numeric variable.

Reverse or shift levels:

“‘r

fct_rev(sizes)   # Reverse order

fct_shift(sizes, n=1) # Rotate levels

“‘

### 6. Modifying factor levels

* **Rename:** ‘fct_recode()‘

* **Combine categories:** ‘fct_collapse()‘

* **Group rare categories:** ‘fct_lump()‘

“‘r

fruit <- factor(c(“apple”,“banana”,“kiwi”,“kiwi”))

fct_lump(fruit, n=2) # Keep top 2, group others

“‘

### 7. Handling missing or unused levels

“‘r

fruit <- factor(c(“apple”,“banana”,NA))

fct_explicit_na(fruit, na_level = “Missing”) # NA becomes explicit

fct_drop(fruit) # Remove unused levels

“‘

### 8. Real-world examples

* Likert survey responses (“Agree”, “Neutral”, “Disagree”)

* Education levels (“Primary”, “Secondary”, “Tertiary”)

* Months in calendar order

Proper factor handling ensures **consistent plotting, summarization, and model interpretation**.

CHAPTER 17 — DATES AND TIMES

Dates and times are essential in data analysis. R’s **lubridate** package simplifies parsing, manipulating, and performing arithmetic on date-time data.

### 1. Types of date-time objects

* **Date** — day resolution

* **Time** — second resolution

* **POSIXct / POSIXlt** — date + time

Example:

“‘r

library(lubridate)

ymd(“2024-12-11”)   # Date

hms(“12:30:45”)   # Time

ymd_hms(“2024-12-11 12:30:45”) # Date-time

“‘

### 2. Parsing dates and times

Lubridate provides readable functions:

* ‘ymd()‘, ‘mdy()‘, ‘dmy()‘

* ‘ymd_hms()‘, ‘mdy_hm()‘

“‘r

mdy(“December 11, 2024”)

dmy_hm(“11-12-2024 14:30”)

“‘

### 3. Extracting components

“‘r

today <- today()

year(today)

month(today, label = TRUE) # Jan, Feb…

wday(today, label = TRUE)   # Mon, Tue…

hour(now())

minute(now())

“‘

### 4. Creating dates from components

“‘r

make_date(2024, 12, 11)

make_datetime(2024, 12, 11, 14, 30)

“‘

### 5. Date arithmetic

Add/subtract time:

“‘r

today() + days(10)   # 10 days later

now() + hours(3)   # 3 hours later

ymd(“2024-01-01”) %–% ymd(“2024-12-31”) # Interval

interval(start, end)/days(1) # Interval length in days

“‘

Durations vs. periods:

* **Duration:** exact seconds

* **Period:** human calendar units

“‘r

today() + months(1)   # Period

today() + dmonths(1) # Duration

“‘

### 6. Updating dates

“‘r

update(today(), year = 2025, month = 1)

“‘

### 7. Handling invalid dates

“‘r

ymd(“2024-01-31”) + months(1) # Feb 31 -> invalid rollback(ymd(“2024-01-31”) + months(1)) # Rolls back to Feb 29

“‘

### 8. Time zones

“‘r

with_tz(now(), “UTC”)   # Convert to UTC

force_tz(now(), “Africa/Nairobi”) # Force timezone

“‘

### 9. Rounding dates

“‘r

floor_date(now(), “hour”)

ceiling_date(today(), “month”)

round_date(now(), “15 minutes”)

“‘

### 10. Leap years

“‘r

leap_year(2024) # TRUE

“‘

### 11. Real-world examples

* Analyzing daily sales over a year

* Calculating patient age from birthdate

* Handling timestamps from IoT sensors

Summary: Mastery of dates and times ensures accurate time-based analysis, correct intervals, and robust temporal visualizations.

Chapters 16 & 17 are now written with **examples for every key function**, detailed explanations, and suitable for ~2 Word pages each.

If you want, I can now continue with **Chapters 18 & 19: Missing Values and Joins** next.

Do you want me to continue?

Perfect! Let’s continue with **Chapters 18 & 19** from *R for Data Science (2nd edition)*, keeping the same level of detail with examples for each function/technique.

CHAPTER 18 — MISSING VALUES

Handling missing data is essential because **real-world datasets often contain gaps**. Properly dealing with missing values ensures accurate summaries, models, and visualizations.

### 1. Identifying missing values

In R, missing values are represented as ‘NA‘.

“‘r

x <- c(1, 2, NA, 4)

is.na(x)   # TRUE for missing values

sum(is.na(x))   # Count missing values

“‘

### 2. Removing missing values

Use functions to exclude missing data from analysis:

“‘r

library(dplyr)

df <- tibble(x = c(1, NA, 3), y = c(NA, 2, 3))

# Remove rows with any NA

drop_na(df)

# Remove rows with NA in a specific column

drop_na(df, x)

“‘

### 3. Replacing missing values

“‘r

# Replace with a constant

x[is.na(x)] <- 0

# Using dplyr’s mutate + if_else

df <- df |> mutate(x = if_else(is.na(x), 0, x))

“‘

### 4. Working with missing values in summaries

Most base R summaries skip ‘NA‘ by default:

“‘r

mean(x)   # Returns NA

mean(x, na.rm = TRUE) # Skips NA

sum(df$y, na.rm = TRUE)

“‘

### 5. Missing values in logical operations

“‘r

x <- c(TRUE, FALSE, NA)

x & TRUE   # NA propagates

x | FALSE # NA propagates

!x   # NA stays NA

“‘

### 6. Functions for missing-value manipulation

* ‘complete.cases(df)‘ — returns TRUE for rows with no NA

* ‘na.omit(df)‘ — removes all rows with any NA

* ‘replace_na()‘ from **tidyr**:

“‘r

library(tidyr)

df <- tibble(x = c(1, NA, 3))

df <- df |> replace_na(list(x = 0))

“‘

### 7. Imputing missing values

* Mean/median imputation:

“‘r

x[is.na(x)] <- mean(x, na.rm = TRUE)

“‘

* Forward/backward fill using **tidyr**:

“‘r

df <- tibble(x = c(1, NA, NA, 4))

fill(df, x, .direction = “down”)

fill(df, x, .direction = “up”)

“‘

### 8. Real-world scenarios

* Sensor data gaps

* Missing survey responses

* Incomplete financial records

Summary: Understanding how to detect, remove, replace, and impute missing values allows reliable analyses and reduces bias.

CHAPTER 19 — JOINS

Joins are essential for **combining datasets** based on key variables. This chapter covers **relational data operations**.

### 1. Keys and relationships

* **Primary key**: uniquely identifies a row in one table

* **Foreign key**: links to a primary key in another table

“‘r

students <- tibble(id = 1:3, name = c(“Alice”, “Bob”, “Charlie”))

grades   <- tibble(student_id = c(1,3), grade = c(90, 85))

“‘

### 2. Mutating joins

Add columns from another table:

* **left_join()** — keep all rows from first table:

“‘r

left_join(students, grades, by = c(“id” = “student_id”))

“‘

* **inner_join()** — keep only matching rows:

“‘r

inner_join(students, grades, by = c(“id” = “student_id”))

“‘

* **right_join()** — keep all rows from second table:

“‘r

right_join(students, grades, by = c(“id” = “student_id”))

“‘

* **full_join()** — keep all rows from both tables:

“‘r

full_join(students, grades, by = c(“id” = “student_id”))

“‘

### 3. Filtering joins

* **semi_join()** — keep only rows in first table with a match:

“‘r

semi_join(students, grades, by = c(“id” = “student_id”))

“‘

* **anti_join()** — keep only rows in first table without a match:

“‘r

anti_join(students, grades, by = c(“id” = “student_id”))

“‘

### 4. Handling duplicate keys

If keys are not unique, joins can create **cartesian expansions**:

“‘r

```
grades <- tibble(student_id = c(1,1,3), grade = c(90, 95, 85))
left_join(students, grades, by = c(“id” = “student_id”))
```

“‘

* Duplicates produce multiple rows for each match.

### 5. Nested joins and list-columns

Store matched rows in a **list-column**:

“‘r

```
nested <- nest_join(students, grades, by = c(“id” = “student_id”))
nested$data
```

“‘

### 6. Set operations on data frames

Works on rows:

“‘r

```
df1 <- tibble(x = 1:3)
df2 <- tibble(x = 2:4)
intersect(df1, df2)  # Common rows
union(df1, df2)      # All rows without duplicates
setdiff(df1, df2)    # Rows in df1 not in df2
```

“‘

### 7. Real-world examples

* Combining **customer** and **order** tables
* Merging **experiment results** from multiple trials
* Joining **survey responses** with demographics

Summary: Mastering joins enables flexible and accurate **data integration**, crucial for multi-table datasets.

Perfect! Here’s **Chapter 20 — Spreadsheets** from *R for Data Science (2nd Edition)*, fully detailed with examples for each technique, aimed for ~2 Word pages.

CHAPTER 20 — SPREADSHEETS

Spreadsheets are one of the most common data sources. This chapter explains how to **import, clean, and work with spreadsheet data** in R, mainly using the **readxl** and **tidyxl** packages.

### 1. Reading Excel files

Use **readxl** for ‘.xls‘ and ‘.xlsx‘ files. It’s fast, safe, and does not require external dependencies.

“‘r

library(readxl)

# Read first sheet

data <- read_excel(“data.xlsx”)

# Specify a sheet by name

data <- read_excel(“data.xlsx”, sheet = “Sales”)

# Specify a sheet by number

data <- read_excel(“data.xlsx”, sheet = 2)

“‘

* ‘read_excel()‘ guesses column types automatically.

* To check data types:

“‘r

glimpse(data)

“‘

### 2. Controlling column types

Sometimes automatic guessing fails. Use ‘col_types‘ to enforce correct types.

“‘r

data <- read_excel(“data.xlsx”,

                   col_types = c(“text”, “numeric”, “date”))

“‘

* Types can be “‘text”‘, “‘numeric”‘, “‘date”‘, “‘skip”‘ (to ignore a column).

### 3. Skipping rows and ranges

Spreadsheets often have metadata or titles in top rows.

“‘r

# Skip first 2 rows

data <- read_excel(“data.xlsx”, skip = 2)

# Read a specific range

data <- read_excel(“data.xlsx”, range = “B3:D10”)

“‘

* ‘range‘ can be a cell range or a named range.

### 4. Writing Excel files

Export data for reporting or sharing using **writexl**:

“‘r

library(writexl)

write_xlsx(data, “cleaned_data.xlsx”)

“‘

* Creates a new spreadsheet with the dataframe content.

### 5. Working with multiple sheets

“‘r

sheets <- excel_sheets(“data.xlsx”) # Get sheet names

# Read each sheet into a list of tibbles

all_data <- lapply(sheets, read_excel, path = “data.xlsx”)

“‘

* Each element in ‘all_data‘ is a tibble representing one sheet.

### 6. Tidying messy spreadsheets

Real-world spreadsheets often contain **merged cells, headers in multiple rows, or inconsistent layouts**.

* Use ‘tidyxl‘ to read cell-level information:

“‘r

library(tidyxl)

cells <- xlsx_cells(“messy.xlsx”)

cells

“‘

* Cells include:
  * ‘row‘, ‘col‘
  * ‘data_type‘
  * ‘character‘, ‘numeric‘, ‘date‘
  * ‘formula‘
* Can reconstruct data using **unpivoting**, **pivoting**, or manual cleaning.

### 7. Handling non-tabular data

Sometimes spreadsheets contain **comments, merged headers, or notes**:

“‘r

```
cells <- xlsx_formats(“messy.xlsx”)
cells$formats  # Check formatting information
```

“‘

* Combine ‘xlsx_cells()‘ and ‘xlsx_formats()‘ to reconstruct complex tables.

Dealing with missing values

Spreadsheets often encode missing values as blank cells, “‘NA”‘, or “‘-”‘:

“‘r

```
data <- read_excel(“data.xlsx”, na = c(“ “,”NA”, “-”))
```

“‘

Always check ‘is.na()‘ after importing.

Example workflow: clean sales spreadsheet

“‘r

```
library(readxl)
library(dplyr)
library(tidyr)
# Read spreadsheet
sales <- read_excel(“sales.xlsx”, skip = 1, na = c(“ “,”NA”))
# Rename columns
sales <- sales |> rename(Date = ‘Sale Date‘, Revenue = ‘Total Revenue‘)
# Filter out missing revenue
sales <- sales |> filter(!is.na(Revenue))
# Calculate monthly total
monthly <- sales |>
  mutate(Month = lubridate::floor_date(Date, “month”)) |>
  group_by(Month) |>
  summarise(Total_Revenue = sum(Revenue))
```

“‘

* This shows importing, cleaning, renaming, filtering, and summarizing.

Combining multiple spreadsheets

Often, you have many Excel files or sheets with the same structure:

“‘r

```
files <- list.files(“data_folder”, pattern = “\\.xlsx$”, full.names = TRUE)

all_sales <- lapply(files, read_excel)

all_sales <- bind_rows(all_sales)
```

“‘

* ‘bind_rows()‘ stacks the sheets into one tibble for analysis.

Visualization after import

“‘r

```
library(ggplot2)

ggplot(monthly, aes(x = Month, y = Total_Revenue)) +

  geom_line(color = “blue”) +

  geom_point() +

  labs(title = “Monthly Revenue”, x = “Month”, y = “Revenue”)
```

“‘

* Immediate visualization ensures the spreadsheet data is correct and ready for analysis.

Real-world tips

* Always inspect imported data with ‘glimpse()‘ or ‘head()‘.

* Save clean versions to avoid reprocessing raw files.

* Use consistent naming for sheets and files.

* Document preprocessing steps for reproducibility.

* Be cautious with merged cells, hidden rows, or formulas.

Conclusion

Spreadsheets are ubiquitous but often messy. Using ‘readxl‘, ‘tidyxl‘, and ‘writexl‘ in R allows you to reliably import, clean, and export data while handling missing values, multiple sheets, and non-tabular structures. This ensures your analysis workflow is reproducible, tidy, and ready for visualization or modeling.