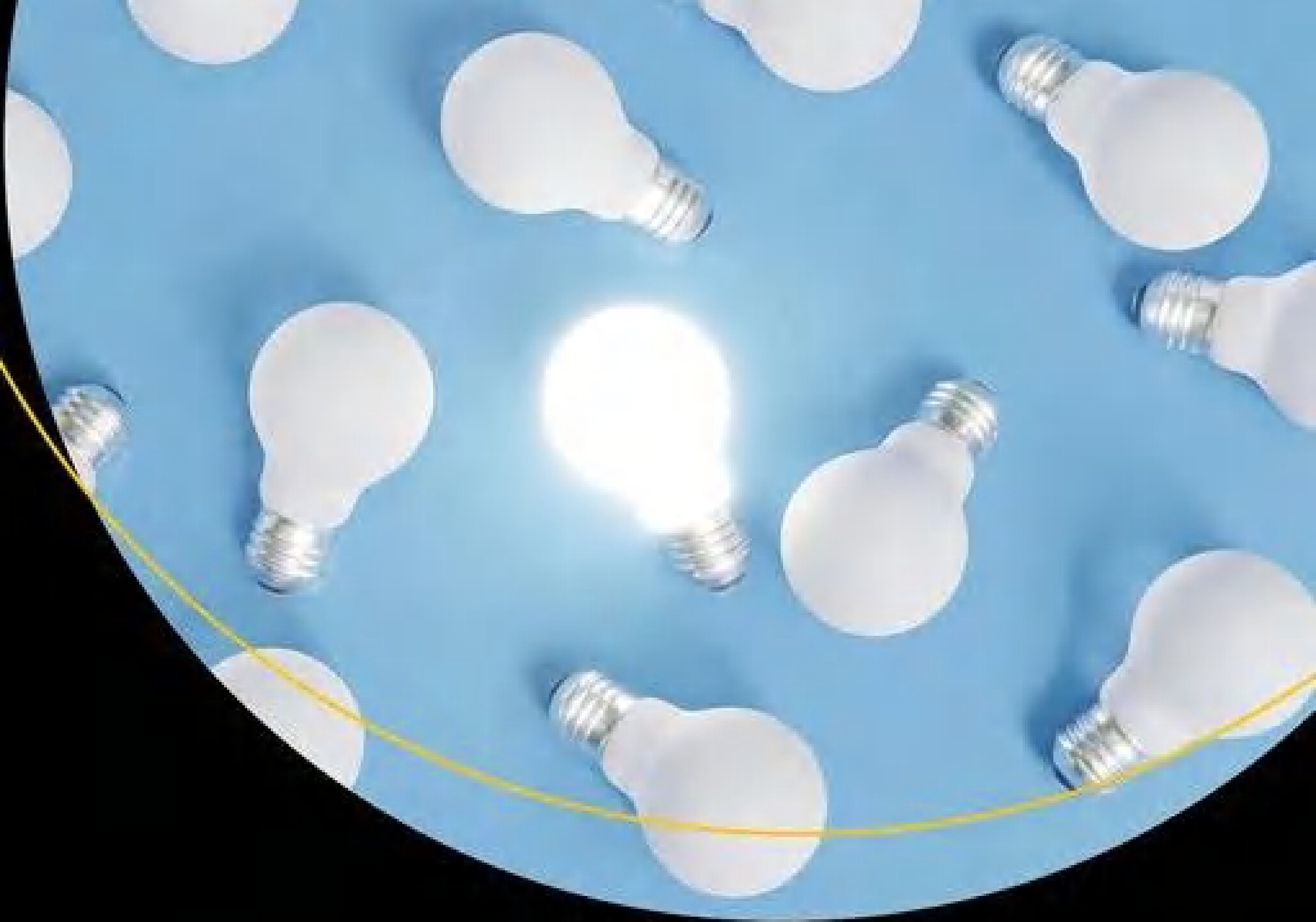


Data Insight Foundations

Step-by-Step Data Analysis with R

—

Nikita Tkachenko

Apress®

Data Insight Foundations

Nikita Tkachenko

Data Insight Foundations

Step-by-Step Data Analysis with R

Apress®

Nikita Tkachenko
San Francisco, CA, USA

ISBN-13 (pbk): 979-8-8688-0579-0 ISBN-13 (electronic): 979-8-8688-0580-6
https://doi.org/10.1007/979-8-8688-0580-6

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Shaul Elson
Development Editor: Laura Berendson
Editorial Assistant: Gryffin Winkler

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

I dedicate this book to my late grandfather, Vladimir Zamashikov.

IT AIN'T MUCH BUT IT'S HONEST WORK
nsight
ndations

Contents

About the Author

Nikita Tkachenko leads Evalyn, a consulting agency specializing in AI-driven customer service audits and data analytics solutions. He helps organizations of all sizes harness AI and data to optimize decision-making, streamline operations, and enhance customer experiences. With a strong foundation in research and analytics, Nikita also teaches courses on research tools, mentors students, and conducts academic research at the University of San Francisco.

About the Technical Reviewer

Sijo Valayakkad Manikandan is an accomplished artificial intelligence and data science leader. He has extensive experience managing large-scale data science projects for Fortune 500 corporations. Sijo holds a Master of Science degree in Business Analytics from the renowned University of Texas at Austin and a Bachelor of Engineering degree from the Birla Institute of Technology and Science in Pilani, India. He combines his academic excellence and practical expertise to deliver exceptional results.

Sijo has made significant contributions to the field of data science and research, not only through his professional and academic achievements but also through his dedication to the community. He is a member of several distinguished organizations, such as the American Statistical Association, and has conducted independent research. Sijo has also actively reviewed academic and professional books, mentored junior data scientists, and guided early-stage startups. Moreover, his expertise has led him to serve on the jury of several prestigious awards, including The Webby Awards.

As a dedicated researcher and thought leader, Sijo continues to profoundly impact the field of data science, inspiring a new generation of data scientists. His contributions have advanced the field of data science and research and helped shape the future of this rapidly evolving industry, making him an invaluable asset to the community and a visionary in his field.

Acknowledgements

I dedicate this book to my late grandfather, Vladimir Zamashikov.

I extend my deepest gratitude to Thomas Weinandy, who convinced me to transform my rough draft into a published book and guided me through the process. Without him, this book would not have been possible.

I am deeply indebted to Parsa Rahimi for his unwavering support, meticulous review of all my chapters, edits, and ideas, and for providing artworks featuring SF.

My thanks also go to Alessandra Cassar and Michael Jonas for their support throughout my academic journey. Special thanks go to Peter Lorentzen, Ker Gibbs, and Jesse Anttila-Hughes for their invaluable guidance and support.

I appreciate Shivani Shukla, Bruce Wydick, Misha Gipsman, Mario Lim, Steve Trettel, Mehmet Emre, Andrew Hobbs, Arman Khachiyan, Robizon Khubulashvili, and Konrad Posch for reviewing chapters and providing their support. Additionally, I am grateful to the faculty, librarians, writing center staff, and the Center for Business Studies and Innovation (CBSI) at the University of San Francisco for their guidance.

Gordon Getty deserves thanks for inspiring me to write more confidently about my skills.

The team at APress, especially Shaul Elson, Commissioning Editor, Laura Berendson, Development Editor, and Gryffin Winkler, Editorial Assistant, guided me through the publishing process. My technical reviewer, Sijo Manikandan, provided comments that significantly enriched the book with additional examples and explanations. Special thanks go to everyone at APress whose efforts made this book a reality.

I gratefully acknowledge John Chetwynd, Jake Consgrove, Timofei Lopukhov, and Alexander Rom, whose support, curiosity, and confidence in my work are deeply appreciated.

I must acknowledge my family, including my mother Ekaterina Tkachenko, whose belief in me inspired me to launch this project, my father Anton Tkachenko, stepmother Daria Tkachenko, and grandparents Valentina and Alexander Trachenko and Raisa Zamashikova.

A special note of thanks goes to Anastasia Ternovskaya for her unwavering belief in me.

I very much appreciate Maria Aksutenko for making the artwork of the “researcher-cat.”

Special thanks also go to the Posit team for their invaluable contributions and support of the community.

Finally, my heartfelt appreciation goes to the open-source community and everyone who has generously shared their knowledge online. You made this possible, and I am thrilled to contribute to the conversation. I hope you find this book useful in your journey toward mastering research.

Introduction

This book was born from my frustrations and experiences in higher education and professional work. It originated from notes and materials from a Spring 2023 course in survey design, inspired by the enthusiastic response and insightful questions from my students.

Young data professionals typically learn about models, experiments, and theories during their classes, frequently returning to that knowledge. However, executing high-quality research and analysis requires a deeper understanding of the tools and the "how" rather than just the "what" and "why." This knowledge often transcends what is taught within the constraints of a standard curriculum. In this book, I aim to bridge that gap, helping you move from knowing what you want to do to understanding how to do it. I have distilled hundreds of hours of frustration into these chapters, so you won't have to traverse that path yourself.

This book is not a comprehensive guide; if that's what you're seeking, you may want to look elsewhere. Instead, the book can serve as a map, outlining the necessary tools and topics for your research journey. The goal is to build your intuition and provide pointers for where to find more detailed information. The chapters are deliberately concise and to the point, aiming to reveal and enlighten rather than bore. You'll learn about efficient data management, reproducible research, literature review and writing practices, and effective data visualization.

Initially inspired by my journey through graduate school in economics, this book offers value across disciplines. It contains essential insights for anyone engaged in data-related work, from the humanities to data analytics and the sciences. Whether you are refining your expertise or new to data analytics, this book promises to offer something of value.

Examples provided are primarily in R, making a basic understanding of the language advantageous but not essential. Several chapters, especially those focusing on theory, require no programming knowledge at all. A diverse audience, including web developers, mathematicians, data analysts, and economists, has found the material beneficial. The book is designed to be inclusive, offering insights irrespective of your programming proficiency or professional background.

Its structure allows for flexible reading paths; you may explore the chapters in sequence for a systematic learning experience or navigate directly to the topics most relevant to you.

Setting Up R and RStudio

Welcome to the exciting world of data analysis with R, a language crafted specifically for statistical analysis and data visualization. R's user-friendly syntax and reproducibility make it an ideal choice for both novices and professionals. However, before diving into data exploration and modeling, it's essential to differentiate between R, the programming language, and RStudio, the integrated development environment (IDE) that enhances R's functionality.

Download and Install R

R is maintained and distributed through the Comprehensive R Archive Network (CRAN), ensuring your access to the latest version and resources.

For macOS Users:

1. Navigate to the CRAN website.
2. Click on "Download R for macOS."
3. Select the appropriate version:
   - For Apple Silicon (e.g., M1, M2), download the version with "-arm64" (e.g., R-4.2.2-arm64.pkg) in its name.
   - For Intel-based Macs, select the version without "-arm64" (e.g., R-4.2.2.pkg).
4. Follow the installation wizard. The default settings are typically sufficient.

For Windows Users:

1. Visit the CRAN website.
2. Choose "Download R for Windows."
3. Select "base" and then the first link at the top of the page (e.g., Download R-4.2.2 for Windows).
4. The installer will guide you through the process. Stick with the default settings for a smooth installation.

5. Additionally, Windows users should download Rtools, which are crucial for compiling packages from sources. Visit Rtools, match the Rtools version with your R version, and follow the installer instructions.

Download and Install RStudio

RStudio provides a user-friendly interface for R, akin to what Microsoft Word offers for text but tailored to R scripting.

- To download RStudio, head to the RStudio download page.
- Click on "DOWNLOAD RSTUDIO DESKTOP FOR WINDOWS" or select your operating system for detailed instructions.

Configure RStudio

Enhance your RStudio experience with these initial setup tips:

- **Change Theme**: Shift from the default theme to a dark theme for improved readability. Go to `Tools` > `Global Options` > `Appearance`, and select "Dracula." Click "Apply."
- **Install Fira Code Font**: For a modern coding aesthetic, install the "Fira Code" font, which supports programming ligatures. Instructions can be found at Fira Code on GitHub. After installation, apply this font in RStudio under `Appearance`.

Install Packages

Packages extend R's functionality. Install them easily with commands in the RStudio console:

```r
# Install a single package
install.packages("tidyverse")

# Install multiple packages
install.packages(c("tidyverse", "gapminder"))
```

To use installed packages, load them into your session:

```r
library(tidyverse)
```

Now that we have R and RStudio up and running, let's dive into some fundamental data manipulation techniques in R.

Chapter 1
Data Manipulation

In data analysis, visualization and manipulation are essential for understanding and communicating complex information. R, a powerful programming language for data analysis, offers a variety of packages that enable the creation of visually compelling plots and the streamlining of data manipulation. One of the most user-friendly and widely used collections of R packages[1] is tidyverse, developed by Hadley Wickham, chief scientist at Posit (RStudio). Tidyverse includes packages that cover all common tasks and can be installed with `install.packages("tidyverse")` and activated using `library("tidyverse")`. In this introduction, we will cover the basics of tidyverse using `readr` for data reading, `dplyr` for data manipulation, `tidyr` for data tidying, and, later in the book, `ggplot2` for data visualization. For more information about tidyverse, visit their website https://www.tidyverse.org/.

1.1 Basics

Let's kick off with some fundamental concepts! R can be employed as a simple calculator.

```r
# A "#" is used to annotate comments!
2 + 2
```

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

```r
2 * 4
```

```
#>   [1] 8
```

[1] A package is a collection of prewritten functions, data, and documentation that enhances the capabilities of the R programming language for specific tasks.

N. Tkachenko, *Data Insight Foundations*, https://doi.org/10.1007/979-8-8688-0580-6_1

```r
2^8
```

```
#>  [1] 256
```

```r
(1 + 3) / (3 + 5)
```

```
#>  [1] 0.5
```

```r
log(10) # Calculates the natural log of 10!
```

```
#>  [1] 2.302585
```

R allows for defining variables and performing operations on them. Both = and <- can be used for assigning values to a variable name, though <- is preferred to avoid confusion and certain errors.

```r
x <- 2 # Equivalent to x = 2
x * 4
```

```
#>  [1] 8
```

The command `x <- 2` assigns the value `2` to `x`. Thus, when we subsequently type `x * 4`, R replaces `x` with `2` to evaluate `2 * 4` and obtain `8`. The value of `x` can be updated as needed using = or <-. Bear in mind that R is case sensitive, so `X` and `x` are recognized as different variables.

```r
x
```

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

```r
(x <- x * 5) # Wrapping with (...) prints the variable
```

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

To further explore operations in R, the following table presents a comprehensive overview of basic arithmetic, comparison, and logical operators you might need.

Operator	Description	Example	Result
+	Addition	`3 + 2`	5
-	Subtraction	`5 - 2`	3
*	Multiplication	`3 * 2`	6
/	Division	`6 / 2`	3
^	Exponentiation	`2 ^ 3`	8
%%	Modulus (remainder)	`5 %% 2`	1
%/%	Integer division	`5 %/% 2`	2
==	Equal to	`2 == 2`	`TRUE`
!=	Not equal to	`2 != 3`	`TRUE`
<	Less than	`2 < 3`	`TRUE`
>	Greater than	`3 > 2`	`TRUE`
<=	Less than or equal to	`2 <= 2`	`TRUE`
>=	Greater than or equal to	`2 >= 2`	`TRUE`
&	Logical AND	`TRUE & FALSE`	`FALSE`
|	Logical OR	`TRUE | FALSE`	`TRUE`
!	Logical NOT	`!TRUE`	`FALSE`

1.1.1 Data Types

R possesses a multitude of data types and classes, including `data.frames`, which are akin to Excel spreadsheets with columns and rows. Initially, we'll examine vectors. Vectors can store multiple values of the same type, with the most basic ones being numeric, character, and logical.

```
x
```

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

```
class(x)
```

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

```
(true_or_false <- TRUE)
```

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

```
class(true_or_false)
```

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

```
(name <- "Parsa Rahimi")
```

```
#>  [1] "Parsa Rahimi"
```

```r
class(name)
```

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

Note that `name` is stored as a single character string. If we want to store the name and surname separately in the same object, we can employ `c()` to concatenate objects of similar classes into a vector.

```r
(name_surname <- c("Parsa", "Rahimi"))
```

```
#>  [1] "Parsa"  "Rahimi"
```

```r
length(name)
```

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

```r
length(name_surname)
```

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

Observe that the length of `name` is 1 and that of `name_surname` is 2! Let's create a numeric vector and perform some operations on it!

```r
(i <- c(1, 2, 3, 4))
```

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

```r
i + 10 # Adds 10 to each element
```

```
#>  [1] 11 12 13 14
```

```r
i * 10 # Multiplies each element by 10
```

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

```r
# Adds together elements in corresponding positions
i + c(2, 4, 6, 8)
```

```
#>  [1]  3  6  9 12
```

The foregoing operations don't modify `i`. The results are merely printed, not stored. If we wish to save the results, we must assign them to a variable.

```r
name
```

```
#>  [1] "Parsa Rahimi"
```

```r
name <- i + c(5, 4, 2, 1)
name
```

```
#>  [1] 6 6 5 5
```

Notice that `name` is no longer "Parsa Rahimi." It was overwritten by a numeric vector rather than a character string. Make sure to perform numeric operations only on numeric objects to avoid errors. The `str()` function can be used to obtain the structure of the object, such as type and length, for example.

```r
name_surname + 2
```

```
#>  Error in name_surname + 2: nonnumeric argument to binary operator
```

```r
str(name_surname)
```

```
#>   chr [1:2] "Parsa" "Rahimi"
```

1.2 Downloading Data

If you're accustomed to Base R (i.e., functions that come with R upon installation), you may be aware of `read.csv()`. However, `readr`, a part of the tidyverse package, offers functions that address common issues associated with Base R's reading functions. `read_csv()` not only loads data ten times faster than `read.csv()`, it also produces a tibble instead of a data frame and avoids inconsistencies of the base version. You might be asking, "What exactly is a tibble?" A tibble is a special type of data frame with several advantages, such as faster loading times, maintaining input types, permitting columns as lists, allowing nonstandard variable names, and never creating row names.

To load your data, you first need to know the path to your data. You can find your file, check its location, and then copy and paste it. If you are a Windows user, your path might contain "\", which is an escape character. To rectify this, replace "\" with "/". Copying the path gives you the absolute path (e.g., `"/Users/User/Documents/your_project/data/file.csv"`), but you can also use a relative path from the folder of the project (e.g., `"/data/file.csv"`). Let's read the data! Using `readr::` specifies which package to use.

1.2.1 Example Data

In this example, we'll be using a real sample from the Climate and Cooperation Experiment conducted in Mexico. During the experiment, subjects were asked to complete three series of Raven's Matrices, four dictator games, and a single lottery game in rooms with varying temperatures. We will primarily be using results from Raven's Matrices games, which consist of 3 sets of 12 matrices. The first set, `pr_`, is the piece-rate round, where participants received points for each correctly solved matrix. The second set, `tr_`, is the tournament round, where participants competed against a random opponent. The winner received double points, and the loser received nothing. The third set, `ch_`, is the choice round, where participants chose whether they wanted to play the piece-rate or tournament round against a different opponent's score from the tournament round.

The data are located in the `data` directory of the GitHub repository. To access and manipulate it, we will require the `tidyverse` package. This package includes several subpackages, including `reader`.

```r
library(tidyverse)
```

```r
# Download data from GitHub
data <- readr::read_csv(
"https://raw.githubusercontent.com/nikitoshina/quarto_book/main/⌋
↪  mexico_sample_data.csv"
)
```

To get a glimpse of the data, we can use the `glimpse()` function, which will provide us with a sample and the type of column. Another common method is to use `head()` to obtain a slice of the top rows or `tail()` to obtain a slice of the bottom rows. To view the entire data set, use `View()`.

```r
data %>% glimpse()
```

```
#>  Rows: 114
#>  Columns: 9
#>  $ id                <chr> "001018001", "001018002", "001018005", "001018009", ~
#>  $ mean_temp_celsius <dbl> 28.58772, 28.58772, 28.58772, 28.58772, 28.58772, 28~
#>  $ gender            <chr> "Female", "Male", "Male", "Male", "Male", "Female", ~
#>  $ pr_correct        <dbl> 7, 5, 6, 1, 7, 2, 6, 7, 5, 6, 8, 2, 2, 5, 6, 5, 2, 6~
#>  $ tr_correct        <dbl> 3, 6, 7, 5, 9, 7, 6, 7, 4, 7, 6, 3, 6, 7, 8, 6, 6, 8~
#>  $ ch_tournament     <dbl> 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0~
#>  $ ch_correct        <dbl> 9, 4, 7, 7, 10, 5, 7, 6, 4, 4, 8, 6, 5, 6, 6, 6, 5, ~
#> $ version            <chr> "A", "A", "A", "A", "A", "A", "A", "A", "A", "A", "A~
#> $ treatment          <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1~
```

The dataset contains 114 rows and 9 columns, along with data types, detailing subjects' performances and choices in an experiment:

- `id` is a unique subject identification number, site_id.session_n.subject_n (001.001.001).
- `mean_temp_celsius` is mean temperature through the session.
- `gender` is the gender of the subject.
- `pr_correct` is the number of correct answers in the *piece-rate* round.
- `tr_correct` is the number of correct answers in the *tournament* round.
- `ch_correct` is the number of correct answers in the *choice* round.
- `ch_tournament` is 1 if participant decided to play *tournament* and 0 if *choice*.
- `version` is either "A" or "B"; we are not concerned with this variable.
- `treatment` is 1 if temperature was higher than 30 Celsius and 0 if below.

1.3 Basic Data Management with dplyr

`dplyr` uses a collection of verbs to manipulate data that are piped (chained) into each other with a piping operator `%>%` from the `magrittr` package. The way you use functions in base R is to wrap a new function over the previous one, such as `k(g(f(x)))`, where `x` is processed by `f()`, then `g()`, and finally `k()`. This will become impossible to read very quickly as you stack up functions and their arguments. To resolve this issue, we will use pipes `x %>% f() %>% g() %>% k()`! Now you can clearly see that we take `x` and apply `f()`, then `g()`, then `k()`.

Note 1.1 Base R now also has its own pipe `|>`, but we will stick to `%>%`.

1.3.1 *select()*

`select()` selects only the columns that you want, removing all others. You can use the column position (with numbers) or name. The columns will be displayed in the order in which you list them. We will select `subject_id`, `mean_temp_celsius`, `gender`, and the results of Raven's Matrices games, excluding version and treatment.

```
data_raven <- data %>% select(
  id, mean_temp_celsius, gender, pr_correct,
  tr_correct, ch_tournament, ch_correct
)
tail(data_raven)
```

```
#> # A tibble: 6 x 7
#>   id     mean_temp_celsius gender pr_correct tr_correct ch_tournament ch_correct
#>   <chr>              <dbl> <chr>       <dbl>      <dbl>         <dbl>      <dbl>
#> 1 00102~              30.5 Female          6          8             1          6
#> 2 00102~              30.5 Male            6          5             0          3
#> 3 00102~              30.5 Male            6          6             1          4
#> 4 00102~              30.5 Female          6          7             0          6
#> 5 00102~              30.5 Female          6          7             0          5
#> 6 00102~              30.5 Male            4          7             0          7
```

You can also exclude columns or select everything else with `select()` using `-`.

```
data_raven %>%
  select(-gender) %>% # Selects all columns except "gender"
  head() # Displays the first few rows
```

```
#> # A tibble: 6 x 6
#>   id        mean_temp_celsius pr_correct tr_correct ch_tournament ch_correct
#>   <chr>                 <dbl>      <dbl>      <dbl>         <dbl>      <dbl>
#> 1 001018001              28.6          7          3             1          9
#> 2 001018002              28.6          5          6             0          4
#> 3 001018005              28.6          6          7             0          7
#> 4 001018009              28.6          1          5             1          7
#> 5 001018010              28.6          7          9             1         10
#> 6 001018013              28.6          2          7             0          5
```

1.3.2 `filter()`

The `filter()` function helps us keep only the rows we need based on certain rules. For example, we use it here to make two separate groups of data: one for Males, another for Females. We use == to check one-to-one equality. Also, you can use symbols like <, <=, >, >=, and `%in%`. The `%in%` symbol is special—it checks whether a value is part of a list.

```
data_male <- data_raven %>% filter(gender == "Male")
data_female <- data_raven %>% filter(gender == "Female")
head(data_male)
```

```
#> # A tibble: 6 x 7
#>   id     mean_temp_celsius gender pr_correct tr_correct ch_tournament ch_correct
#>   <chr>              <dbl> <chr>       <dbl>      <dbl>         <dbl>      <dbl>
#> 1 00101~              28.6 Male            5          6             0          4
#> 2 00101~              28.6 Male            6          7             0          7
#> 3 00101~              28.6 Male            1          5             1          7
#> 4 00101~              28.6 Male            7          9             1         10
#> 5 00101~              30.7 Male            6          7             1          4
#> 6 00101~              30.7 Male            5          6             1          6
```

```
head(data_female)
```

```
#> # A tibble: 6 x 7
#>   id     mean_temp_celsius gender pr_correct tr_correct ch_tournament ch_correct
#>   <chr>              <dbl> <chr>       <dbl>      <dbl>         <dbl>      <dbl>
#> 1 00101~              28.6 Female          7          3             1          9
#> 2 00101~              28.6 Female          2          7             0          5
#> 3 00101~              28.6 Female          6          6             0          7
#> 4 00101~              28.6 Female          7          7             0          6
#> 5 00101~              30.7 Female          5          4             0          4
#> 6 00101~              30.7 Female          8          6             1          8
```

You can chain multiple criteria. In the following example, we filter for Males with temperatures above 30 degrees Celsius and Females with temperatures below 30 degrees Celsius. We use `&` for "and" and `|` for "or" and enclose the conditions in parentheses to avoid confusion.

Note 1.2 I have set `pillar.print_min = 5`, so printing tibbles will only show five rows (instead of the default ten). This way, we don't have to use `head` after each call.

```
data_raven %>%
  filter(
    (gender == "Male" & mean_temp_celsius > 30) |
    (gender == "Female" & mean_temp_celsius < 30)
  )
```

```
#>  # A tibble: 61 x 7
#>    id     mean_temp_celsius gender pr_correct tr_correct ch_tournament ch_correct
#>    <chr>              <dbl> <chr>       <dbl>      <dbl>         <dbl>      <dbl>
#>  1 00101~              28.6 Female          7          3             1          9
#>  2 00101~              28.6 Female          2          7             0          5
#>  3 00101~              28.6 Female          6          6             0          7
#>  4 00101~              28.6 Female          7          7             0          6
#>  5 00101~              30.7 Male            6          7             1          4
#>  # i 56 more rows
```

1.3.3 `arrange()`

The `arrange()` function orders the table using a variable. For example, to see the subject with the lowest score in `pr_correct`, we can use

```
data_raven %>%
  arrange(pr_correct)
```

```
#>  # A tibble: 114 x 7
#>    id     mean_temp_celsius gender pr_correct tr_correct ch_tournament ch_correct
#>    <chr>              <dbl> <chr>       <dbl>      <dbl>         <dbl>      <dbl>
#>  1 00101~              28.6 Male            1          5             1          7
#>  2 00101~              28.6 Female          2          7             0          5
#>  3 00101~              30.7 Female          2          3             0          6
#>  4 00101~              30.7 Female          2          6             0          5
#>  5 00101~              30.7 Male            2          6             1          5
#>  # i 109 more rows
```

To sort in descending order, use the `desc()` modifier. For example, to find the subject with the highest score in pr_correct, use

```
data_raven %>%
  arrange(desc(pr_correct))
```

```
#>   # A tibble: 114 x 7
#>     id     mean_temp_celsius gender pr_correct tr_correct ch_tournament ch_correct
#>     <chr>              <dbl> <chr>       <dbl>      <dbl>         <dbl>      <dbl>
#>   1 00101~              30.7 Female          8          6             1          8
#>   2 00102~              31.6 Male            8          6             1          5
#>   3 00102~              31.6 Male            8          7             0          7
#>   4 00102~              30.4 Male            8          4             0          7
#>   5 00102~              26.8 Male            8          9             1          9
#>   # i 109 more rows
```

1.3.4 mutate()

The `mutate()` function adds new columns or modifies existing ones in a dataset. For instance, you can create a dataset with four rows and add three new variable columns:

```
tibble(rows = 1:4) %>% mutate(
  One = 1,
  Comment = "Something",
  Approved = TRUE
)
```

```
#>   # A tibble: 4 x 4
#>      rows   One Comment   Approved
#>     <int> <dbl> <chr>     <lgl>
#>   1     1     1 Something TRUE
#>   2     2     1 Something TRUE
#>   3     3     1 Something TRUE
#>   4     4     1 Something TRUE
```

You can use `mutate()` to create new variables using existing ones. For instance, you can convert Celsius to Fahrenheit, calculate the improvement in tournament scores over piece-rate round scores, and check the deviation from the mean score in the piece-rate round:

```
data_raven %>% mutate(
  mean_temp_fahrenheit = (mean_temp_celsius * 9 / 5) + 32,
  improvement = tr_correct - pr_correct,
  pr_deviation = pr_correct - mean(pr_correct),
  .keep = "none" # Keep only newly created columns
)
```

```
#>  # A tibble: 114 x 3
#>    mean_temp_fahrenheit improvement pr_deviation
#>                   <dbl>       <dbl>        <dbl>
#>  1                 83.5          -4        1.66
#>  2                 83.5           1       -0.342
#>  3                 83.5           1        0.658
#>  4                 83.5           4       -4.34
#>  5                 83.5           2        1.66
#>  # i 109 more rows
```

Notice how we can nest functions within `mutate()` to first calculate the mean of an entire column and then subtract it from `pr_correct`.

1.3.5 `case_match()`

The `case_match()` function is used to recode or replace specific values within a variable based on a given set of matching conditions. For instance, you can use `case_match()` to simplify categorical data by changing “Male” to “M” and “Female” to “F.”

```
data_raven <- data_raven %>%
  mutate(gender = case_match(gender, "Male" ~ "M", "Female" ~ "F"))
```

1.3.6 `summarize()`

The `summarize()` function reduces all rows into a one-row summary. It can be used to calculate the percentage of participants who were male, the median score in the piece-rate round, the maximum score in the tournament, the percentage of people choosing the tournament in the choice round, and the mean score in the choice round.

In dplyr 1.0.0, `reframe()` was introduced. Unlike `summarize()`, `reframe()` can produce multiple row outputs. Use `summarize()` when expecting one row per group and `reframe()` for multiple rows.

```
data_raven %>%
  summarize(
    # n() returns the number of rows
    prop_male = sum(gender == "M", na.rm = TRUE) / n(),
    pr_median = median(pr_correct),
    tr_max = max(tr_correct),
    ch_ratio = sum(ch_tournament) / n(),
    ch_mean = mean(ch_correct)
  )
```

```
#>  # A tibble: 1 x 5
#>    prop_male pr_median tr_max ch_ratio ch_mean
#>        <dbl>     <dbl>  <dbl>    <dbl>   <dbl>
#>  1     0.482         5      9    0.456    6.01
```

1.3.7 group_by()

The `group_by()` function groups data by specific variables for subsequent operations. By combining `group_by()` and `summarize()`, you can calculate different summary statistics for genders!

```
data_raven %>%
  drop_na(gender) %>%
  group_by(gender) %>%
  summarize(
    pr_mean = mean(pr_correct),
    tr_mean = mean(tr_correct),
    ch_mean = mean(ch_correct),
    pr_sd = sd(pr_correct),
    n = n()
  )
```

```
#>  # A tibble: 2 x 6
#>    gender pr_mean tr_mean ch_mean pr_sd     n
#>    <chr>    <dbl>   <dbl>   <dbl> <dbl> <int>
#>  1 F         5.22    6.17    5.93  1.58    58
#>  2 M         5.47    6.4     6.09  1.59    55
```

Now let’s group by gender and choice in the choice round to look at points in the choice round!

```
data_raven %>%
  drop_na(gender) %>%
  group_by(gender, ch_tournament) %>%
  summarize(
    ch_mean = mean(ch_correct),
    pr_sd = sd(ch_correct),
    n = n()
  )
```

```
#>  # A tibble: 4 x 5
#>  # Groups:   gender [2]
#>    gender ch_tournament ch_mean pr_sd     n
#>    <chr>          <dbl>   <dbl> <dbl> <int>
#>  1 F                  0    5.73  1.70    33
#>  2 F                  1    6.2   1.73    25
#>  3 M                  0    6.07  1.69    29
#>  4 M                  1    6.12  2.25    26
```

1.3.8 ungroup()

The ungroup() function removes grouping. Always ungroup your data after performing operations that required grouping to avoid confusion. The following code filters out missing gender data from data_raven, groups by gender, calculates the count per group, then determines the proportion of males within each group, and finally removes the grouping.

Note 1.3 The summarize function removes the last variable from the grouping, so in this case, ungroup is included for purely demonstration purposes.

```
data_raven %>%
  drop_na(gender) %>%
  group_by(gender) %>%
  mutate(n = n()) %>%
  summarize(mean_male = mean(gender == "M")) %>%
  ungroup()
```

```
#>   # A tibble: 2 x 2
#>     gender mean_male
#>     <chr>      <dbl>
#>   1 F              0
#>   2 M              1
```

Notice how mean_male (the ratio of males to the total) is 0 for Female and 1 for Male. That's because the data were grouped, and we performed operations on Males and Females separately.

```
data_raven %>%
  drop_na(gender) %>%
  group_by(gender) %>%
  mutate(n = n()) %>%
  ungroup() %>%
  summarize(mean_male = mean(gender == "M"))
```

```
#>   # A tibble: 1 x 1
#>     mean_male
#>         <dbl>
#>   1     0.487
```

This time we ungrouped the data before calculating the ratio, which gives us the correct result!

1.3.9 .by

Grouping is a commonly performed operation. Having to repeatedly group and ungroup for individual operations can lead to verbosity. To address this, dplyr introduced a convenient feature with version 1.0.0—the .by argument within dplyr functions. This enhancement streamlines the process and reduces the need for excessive grouping and ungrouping.

```
data_raven %>%
  drop_na(gender) %>%
  # Same as ``group_by %>% mutate %>% ungroup"
  mutate(n = n(), .by = gender) %>%
  summarize(mean_male = mean(gender == "M"))
```

1.3.10 rowwise()

The `rowwise()` function allows for row-wise grouping. There may be situations where you want to perform a calculation row-wise instead of column-wise. However, when you try to perform the operation, you get an aggregate result. `rowwise()` comes to your rescue in such situations. Let's create a data frame with a column of lists and try to find the length of each list:

```
(df <- tibble(
  x = list(1, 2:3, 4:6, 7:11)
))
```

```
#>   # A tibble: 4 x 1
#>     x
#>     <list>
#>   1 <dbl [1]>
#>   2 <int [2]>
#>   3 <int [3]>
#>   4 <int [5]>
```

```
df %>% mutate(length = length(x))
```

```
#>   # A tibble: 4 x 2
#>     x         length
#>     <list>     <int>
#>   1 <dbl [1]>      4
#>   2 <int [2]>      4
#>   3 <int [3]>      4
#>   4 <int [5]>      4
```

In the preceding example, instead of obtaining the lengths of the lists, we got the total number of rows in the dataset (the length of column `x`). Now let's use `rowwise()`:

```
df %>%
  rowwise() %>%
  mutate(length = length(x))
```

```
#>   # A tibble: 4 x 2
#>   # Rowwise:
#>     x         length
#>     <list>     <int>
#>   1 <dbl [1]>      1
#>   2 <int [2]>      2
#>   3 <int [3]>      3
#>   4 <int [5]>      5
```

With `rowwise()`, R runs the `length()` function on each list separately, providing the correct lengths.

Note 1.4 Or you can use `lengths()`, which is a vectorized version, meaning that the function will be applied to each element of a vector.

1.3.11 `count()`

The `count()` function in R is used to count the number of rows within each group of values, similar to a combination of the `group_by()` and `summarize()` functions. It can be used with a single column to count rows by gender:

```
data_raven %>% count(gender)
```

```
#>   # A tibble: 3 x 2
#>     gender     n
#>     <chr>  <int>
#>   1 F         58
#>   2 M         55
#>   3 <NA>       1
```

Or it can be used with multiple columns, counting by both `gender` and `ch_tournament`, which resembles the functionality of `group_by(gender, ch_tournament)`:

```
data_raven %>% count(gender, ch_tournament)
```

```
#>   # A tibble: 5 x 3
#>     gender ch_tournament     n
#>     <chr>          <dbl> <int>
#>   1 F                  0    33
#>   2 F                  1    25
#>   3 M                  0    29
#>   4 M                  1    26
#>   5 <NA>               1     1
```

1.3.12 rename()

The rename() function allows you to change column names, with the new name on the left and the old name on the right. Let's replace id with subject_id and gender with sex:

```
data_raven %>%
  rename(subject_id = id, sex = gender)
```

```
#>   # A tibble: 114 x 7
#>     subject_id mean_temp_celsius sex   pr_correct tr_correct ch_tournament
#>     <chr>                  <dbl> <chr>      <dbl>      <dbl>         <dbl>
#>   1 001018001               28.6 F              7          3             1
#>   2 001018002               28.6 M              5          6             0
#>   3 001018005               28.6 M              6          7             0
#>   4 001018009               28.6 M              1          5             1
#>   5 001018010               28.6 M              7          9             1
#>   # i 109 more rows
#>   # i 1 more variable: ch_correct <dbl>
```

1.3.13 row_number()

row_number() generates a column with consecutive numbers, which can be useful for creating a new ID column. The following example first removes the id column, then creates a new one using row_number():

```
data_raven %>%
  select(-id) %>%
  # .before = 1 places the new "id" column as the first column
  mutate(id = row_number(), .before = 1)
```

```
#>   # A tibble: 114 x 7
#>        id mean_temp_celsius gender pr_correct tr_correct ch_tournament ch_correct
#>     <int>             <dbl> <chr>       <dbl>      <dbl>         <dbl>      <dbl>
#>   1     1              28.6 F               7          3             1          9
#>   2     2              28.6 M               5          6             0          4
#>   3     3              28.6 M               6          7             0          7
#>   4     4              28.6 M               1          5             1          7
#>   5     5              28.6 M               7          9             1         10
#>   # i 109 more rows
```

1.3.14 skim()

skim() from the skimr package provides an extensive summary of a data frame. It offers more than summary(), detailing quartiles, missing values, and histograms. Use it during exploratory data analysis to understand your data as it provides a neat, comprehensive view of each variable, useful for further analysis.

```
library(skimr)
skim(data_raven)
```

Table 1-1 Data Summary

Name	data_raven
Number of rows	114
Number of columns	7
Column type frequency:	
character	2
numeric	5
Group variables	None

Variable type: character

skim_variable	n_missing	complete_rate	Min	Max	Empty	n_unique	Whitespace
id	0	1.00	9	9	0	114	0
gender	1	0.99	1	1	0	2	0

Variable type: numeric

skim_variable	n_missing	complete_rate	Mean	Sd	*p*0	*p*25	*p*50	*p*75	*p*100	Hist
mean_temp_celsius	0	1	30.63	1.43	26.84	30.4	30.69	31.59	32.3	▁▁▂▇▇
pr_correct	0	1	5.34	1.57	1.00	5.0	5.00	6.00	8.0	▁▂▇▅▅
tr_correct	0	1	6.28	1.35	3.00	6.0	6.00	7.00	9.0	▃▃▇▇▃
ch_tournament	0	1	0.46	0.50	0.00	0.0	0.00	1.00	1.0	▇▁▁▁▇
ch_correct	0	1	6.01	1.82	2.00	5.0	6.00	7.00	10.0	▃▃▅▇▁

1.4 Exploring Date and Time with `lubridate`

Navigating the complexities of dates, times, and time zones can be a daunting task, but fear not! The `lubridate` package, a recent addition to `tidyverse`, comes to the rescue with its remarkable capabilities for simplifying date and time manipulations. With `lubridate` you can effortlessly convert strings into dates and dates into strings, alter formats, check for overlaps, and perform date arithmetic.

1.4.1 `ymd()`, `md()`, `hms()`, `ymd_hms()`

When working with dates, various notations exist, each represented by a combination of letters. To extract dates from text, all you need is to discern the order of year, month, day, hours, minutes, and seconds and then employ the corresponding function to work its magic.

```r
library(lubridate)
day_year_month <- "31/2001/01"
(date <- dym(day_year_month))
```

```
#>   [1] "2001-01-31"
```

```r
month_day_year_hour_minute <- "Jan 31st 2001 6:05PM"
(date_time <- mdy_hm(month_day_year_hour_minute))
```

```
#>   [1] "2001-01-31 18:05:00 UTC"
```

```r
sentence_with_date <- "Elizabeth II was Queen of the United Kingdom from 6
↪  February 1952."
# lubridate can identify the date within text and parse it for you!
dmy(sentence_with_date)
```

```
#>   [1] "1952-02-06"
```

1.4.2 year(), month(), day()

Similarly, you can extract specific date and time components using functions aptly named after the elements you wish to retrieve.

```r
day(date_time)
```

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

```r
hour(date_time)
```

```
#>   [1] 18
```

```r
# `label = TRUE` returns month's name, `abbr = FALSE` returns full name.
month(date_time, label = TRUE, abbr = FALSE)
```

```
#>   [1] January
#>   12 Levels: January < February < March < April < May < June < ... < December
```

1.5 Summary

We covered basic operations in R and explored the essential features of tidyverse for data manipulation. This included learning how to load, manage, and handle basic date and time data, providing a solid foundation for beginners. Next, we will discuss what makes data “tidy” and how to organize them effectively.

Chapter 2
Tidy Data

Tidy data is a standard way of structuring a dataset to streamline its usability. This standard, as popularized by Hadley Wickham [36], depends on the organization of rows, columns, and tables and how they correspond to observations, variables, and types. In the context of tidy data:

1. Each variable is represented by a column.
2. Each observation is represented by a row.
3. Each type of observational unit is represented by a table.

Note 2.1 What changes should be put in a column.

Data commonly take two formats: wide and long. Wide data, also referred to as unstacked data, are structured so that each row represents an individual unit of observation, and each column represents a variable. This format is often employed when the number of variables is relatively small and they don't share hierarchical relationships. This format is common in reports, where it can help readability. A notable example of wide data is a panel where columns correspond to years, as in Table 2-1a.

In contrast, long data, also known as stacked data, are organized so that each row represents a single observation of a variable, with columns representing the variable (year) and the unit (population in millions) of observation identifier as in Table 2-1b. This format is usually employed when the number of variables is large or they are hierarchically related. You'll often find that working with long data is much more manageable for conducting analyses.

Table 2-1 Panel Data

(a) Wide Format

Country	2020	2021
USA	329.5	331.9
Russia	144.1	143.4
Mexico	126	126.7

(b) Tidy Long Format

Country	Year	Population
USA	2020	329.5
Russia	2020	144.1
Mexico	2020	126
USA	2021	331.9
Russia	2021	143.4
Mexico	2021	126.7

The first example might cause one to think that long and tidy data is synonyms. Let's consider another example where we wish to make our data wider. In Table 2-2a, each observation—which comprises a country, its population, and birth rate—is spread across two rows. In this case, we aim to widen our data as shown in Table 2-2b.

N. Tkachenko, *Data Insight Foundations*, https://doi.org/10.1007/979-8-8688-0580-6_2

Table 2-2 Messy Long

(a) Across Two Rows

Country	Name	Value
USA	Population	329.5
Russia	Population	144.1
Mexico	Population	126
USA	Birth Rate	1.64
Russia	Birth Rate	1.5
Mexico	Birth Rate	1.9

(b) Tidy Wide

Country	Population	Birth Rate
USA	329.5	1.64
Russia	144.1	1.5
Mexico	126	1.9

Another common issue with messy data occurs when two variables are combined into one column, as seen in Table 2-3a, where "Year" and "Population" are in the same column. These need to be separated into two distinct columns: "Year" and "Population" as in Table 2-3b.

Table 2-3 Separate Data

(a) United Year/Population

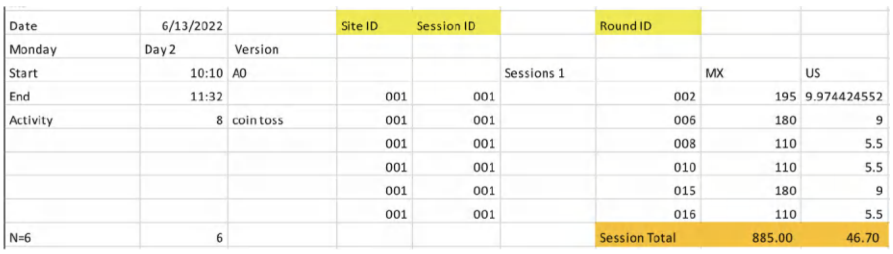
Country	Year/Population
USA	2020/329.5
Russia	2020/144.1
Mexico	2020/126
USA	2021/331.9
Russia	2021/143.4
Mexico	2021/126.7

(b) Tidy Split into Columns

Country	Year	Population
USA	2020	329.5
Russia	2020	144.1
Mexico	2020	126
USA	2021	331.9
Russia	2021	143.4
Mexico	2021	126.7

2.1 Example

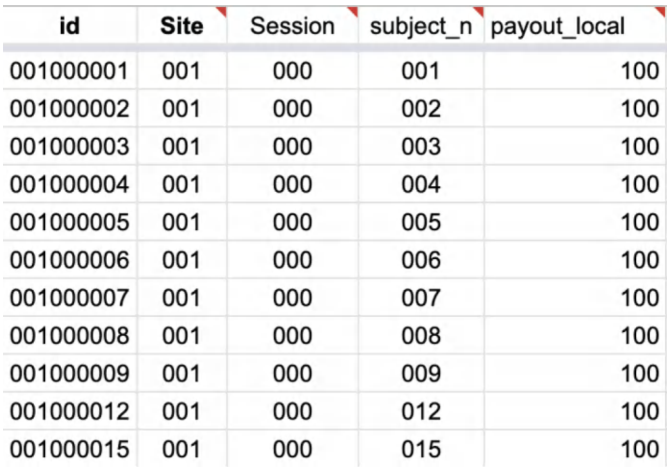
Tidy data is more than a theoretical concept; it has practical implications for data structuring. Consider this real-life example of storing data on payments for experiments in a formlike structure:



Date	6/13/2022		Site ID	Session ID		Round ID		
Monday	Day 2	Version						
Start	10:10	A0			Sessions 1		MX	US
End	11:32		001	001		002	195	9.974424552
Activity	8	coin toss	001	001		006	180	9
			001	001		008	110	5.5
			001	001		010	110	5.5
			001	001		015	180	9
			001	001		016	110	5.5
N=6	6					Session Total	885.00	46.70

Figure 2-1 Form Example

When a computer reads these data, it can't understand our intent, so all columns are read as-is. Adding multiple days necessitates creating similar tables, increasing the chances of error. Want a total for the entire experiment? Then it is time to manually sum all cells. Now compare this to



id	Site	Session	subject_n	payout_local
001000001	001	000	001	100
001000002	001	000	002	100
001000003	001	000	003	100
001000004	001	000	004	100
001000005	001	000	005	100
001000006	001	000	006	100
001000007	001	000	007	100
001000008	001	000	008	100
001000009	001	000	009	100
001000012	001	000	012	100
001000015	001	000	015	100

Figure 2-2 Tidy Form Example

In this format, data input is straightforward: each variable has its own column, and generating summary tables is as simple as creating a pivot table. A tidy data approach from the outset aids in creating robust tables and saves time during analysis.

2.2 `tidyr`

Transforming your data into a tidy format initially demands some effort, but this upfront work pays off by saving time and reducing complications down the line. Tidy data works seamlessly with `tidyverse` packages in R, cutting down on the time needed for manipulating data across various formats. It's important to note that achieving a "tidy data" state might sometimes be ambiguous. The key is to ensure the data format is effective for your specific analysis needs. In the following sections, we'll explore how to effectively achieve this tidy data format with `tidyr`, which is part of `tidyverse`.

2.2.1 *`pivot_longer()`*

A common problem arises in datasets where column names are not variable names but *values* of a variable. This is the case for `pr_correct`, `tr_correct`, and `ch_correct`, where the column names represent the `game` variable's name. Meanwhile, the values in the columns represent the number of correct answers, and each row denotes three observations, not one.

```
(data_raven <- readr::read_csv(
  "https://raw.githubusercontent.com/nikitoshina/quarto_book/main/raven_data.csv"
  ))
```

```
#>  # A tibble: 114 x 7
#>     id    mean_temp_celsius gender pr_correct tr_correct ch_tournament ch_correct
#>     <chr>             <dbl> <chr>       <dbl>      <dbl>         <dbl>      <dbl>
#>    1 0010~             28.6 Female          7          3            1          9
#>    2 0010~             28.6 Male            5          6            0          4
#>    3 0010~             28.6 Male            6          7            0          7
#>    4 0010~             28.6 Male            1          5            1          7
#>    5 0010~             28.6 Male            7          9            1         10
#>  # i 109 more rows
```

To tidy such a dataset, we need to pivot the problematic columns into new pairs of variables. We aim to have a column for the game names and a separate column for game results. This operation requires the following items:

1. Columns whose names are values, not variables—columns we want to pivot. In this case, `pr_correct`, `tr_correct`, `ch_correct` are values, and the variable name is `game`.
2. The name of the variable where we'll move the column names to. Here it's `game`. The default is `name`.
3. The name of the variable where we'll move the column values to. Here it's `n_correct`. The default is `value`.

```
library(tidyr)
library(dplyr)

data_raven %>%
  pivot_longer(
    c(pr_correct, tr_correct, ch_correct),
    names_to = "game",
    values_to = "n_correct"
  ) %>%
  select(id, game, n_correct)
```

```
#>  # A tibble: 5 x 3
#>    id        game       n_correct
#>    <chr>     <chr>          <dbl>
#>  1 001018001 pr_correct         7
#>  2 001018001 tr_correct         3
#>  3 001018001 ch_correct         9
#>  4 001018002 pr_correct         5
#>  5 001018002 tr_correct         6
#>  # i 337 more rows
```

2.2.2 `pivot_wider()`

`pivot_wider()` is the opposite of `pivot_longer()`. It is used when an observation is scattered across multiple rows. For instance, consider the `data_raven_accident` dataset, where `mean_temp_celsius` and `ch_tournament` are stacked. In this case, an observation is spread across two rows.

```
data_raven_accident
```

```
#>  # A tibble: 5 x 3
#>    id        name              value
#>    <chr>     <chr>             <dbl>
#>   1 001018001 mean_temp_celsius  28.6
#>   2 001018001 ch_tournament       1
#>   3 001018002 mean_temp_celsius  28.6
#>   4 001018002 ch_tournament       0
#>   5 001018005 mean_temp_celsius  28.6
#>   # i 223 more rows
```

To tidy this up, we want to create two columns: one for `mean_temp_celsius` and one for `ch_tournament`. We need two parameters:

1. The column to take variable names from. Here it's `name`.
2. The column to take values from. Here it's `value`.

```
data_raven_accident %>%
  pivot_wider(names_from = name, values_from = value)
```

```
#>   # A tibble: 5 x 3
#>     id        mean_temp_celsius ch_tournament
#>     <chr>                 <dbl>         <dbl>
#>   1 001018001              28.6             1
#>   2 001018002              28.6             0
#>   3 001018005              28.6             0
#>   4 001018009              28.6             1
#>   5 001018010              28.6             1
#>   # i 109 more rows
```

It is evident from their names that *`pivot_wider`*`()` and *`pivot_longer`*`()` are inverse functions. *`pivot_longer`*`()` converts wide tables to a longer and narrower format, while *`pivot_wider`*`()` converts long tables to a shorter and wider format.

2.2.3 `separate()` *and* `unite()`

Sometimes, data may come with columns united, requiring that we `separate` them to maintain tidy data. The original function was superseded in favor of *`separate_wider_position`*`()` and *`separate_wider_delim`*`()`, and there is also a `separate_longer_*()` version. These functions differ in how they handle the separation of variables: *`separate_wider_position`*`()` and *`separate_wider_delim`*`()` split variables into wider data frames, creating more columns based on position or delimiter, respectively, while `separate_longer_*()` versions create longer data frames, potentially generating more rows.

```
data_raven_sep
```

```
#>   # A tibble: 5 x 2
#>     id        `gender/pr_correct`
#>     <chr>     <chr>
#>   1 001018001 Female/7
#>   2 001018002 Male/5
#>   3 001018005 Male/6
#>   4 001018009 Male/1
#>   5 001018010 Male/7
#>   # i 109 more rows
```

data_raven_sep contains a column, gender/pr_correct, with values separated by a slash, combining two columns and making analysis difficult. To address this, we need to split the column into two using "/" (forward slash).

```
data_raven_sep %>%
  separate_wider_delim(
    col = "gender/pr_correct",
    delim = "/",
    names = c("gender", "pr_correct")
  )
```

```
#>   # A tibble: 5 x 3
#>     id        gender pr_correct
#>     <chr>     <chr>  <chr>
#>   1 001018001 Female 7
#>   2 001018002 Male   5
#>   3 001018005 Male   6
#>   4 001018009 Male   1
#>   5 001018010 Male   7
#>   # i 109 more rows
```

What if we have one variable that has been split across multiple columns? Consider a situation where our subject ID code, composed of site_id, session_n, and subject_n, has been broken down into three separate columns. In such a scenario, we would need to unite() these columns back into one.

```
data_raven_uni
```

```
#>   # A tibble: 5 x 4
#>     site_id session_n subject_n gender
#>     <chr>   <chr>     <chr>     <chr>
#>   1 001     018       001       Female
#>   2 001     018       002       Male
#>   3 001     018       005       Male
#>   4 001     018       009       Male
#>   5 001     018       010       Male
#>   # i 109 more rows
```

```
data_raven_uni %>%
  unite(c(site_id, session_n, subject_n), col = "id", sep = "")
```

```
#>   # A tibble: 5 x 2
#>     id        gender
#>     <chr>     <chr>
#>   1 001018001 Female
#>   2 001018002 Male
#>   3 001018005 Male
#>   4 001018009 Male
#>   5 001018010 Male
#>   # i 109 more rows
```

2.3 tibble() and tribble()

A tibble is a special kind of data frame in R, as we learned in the last chapter. Tibbles are a modern reimagining of the data frame, designed to be more friendly and consistent than traditional data frames. To create one, we can use `tibble()`, similar to `data.frame()`. Here are some features that make tibbles unique:

- By default, tibbles display only the first ten rows when printed, making them easier to work with large datasets.
- They use a consistent print format, making it easier to work with multiple tibbles in the same session.
- Tibbles have a consistent subsetting behavior, making it easier to select columns by name. When printed, the data type of each column is specified.
- Subsetting a tibble will always return a tibble, so you don't need to use `drop = FALSE`, as you would with traditional data frames.

```
tibble(
  x = c(1, 2, 3),
  y = c("one", "two", "three")
)
```

```
#>  # A tibble: 3 x 2
#>        x y
#>    <dbl> <chr>
#>  1     1 one
#>  2     2 two
#>  3     3 three
```

You can also use `tribble()` to create a row-wise, readable tibble in R. This is especially useful when creating small tables of data. The syntax is as follows: `tribble(~column1, ~column2)`, where you define column names in the first row and values row by row.

```
tribble(
  ~x, ~y,
  1, "one",
  2, "two",
  3, "three"
)
```

```
#>  # A tibble: 3 x 2
#>        x y
#>    <dbl> <chr>
#>  1     1 one
#>  2     2 two
#>  3     3 three
```

2.4 janitor: Clean Your Data

The `janitor` package is designed to make the process of cleaning and tidying data as simple and efficient as possible. To learn more about the functions it provides, check out the package's vignettes.

2.4.1 `clean_names()`

The `clean_names()` function is used to clean variable names, especially those read in from Excel files using `readxl::read_excel()` and `readr::read_csv()`. It parses letter cases, separators, and special characters into a consistent format, converts certain characters like "%" to "percent" and "#" to "number" to retain their meaning and resolves issues of duplicate or empty names. It is recommended to call this function every time data are read.

Note 2.2 When uncertain about the data, it is advisable to increase the number of rows used to infer column types. This approach helps prevent the occurrence of unintentional NAs resulting from incorrectly assumed data types. This practice is particularly crucial when dealing with data from Excel or Google Sheets. For example, when using the `read_csv()` function, you can increase the number of rows used for type inference by specifying the `guess_max` parameter: `read_csv("your_file.csv", guess_max = 10^5)`. Alternatively, you could classify all columns as character types to resolve these issues manually later: `read_csv(..., col_types = cols(.default = "c"))`.

```
# Create a data.frame with dirty names
df_dirty_names <- as.data.frame(matrix(ncol = 6))
names(df_dirty_names) <- c(
  "camelCase", "ábc@!*", "% of respondents (2009)",
  "Duplicate", "Duplicate", ""
)
df_dirty_names %>% colnames()
```

```
#>  [1] "camelCase"               "ábc@!*"
#>  [3] "% of respondents (2009)" "Duplicate"
#>  [5] "Duplicate"               ""
```

```
library(janitor)
df_dirty_names %>%
  clean_names() %>%
  colnames()
```

```
#>  [1] "camel_case"                    "abc"
#>  [3] "percent_of_respondents_2009" "duplicate"
#>  [5] "duplicate_2"                   "x"
```

2.4.2 `remove_empty()`

`remove_empty()` removes empty rows and columns, which is especially useful after reading Excel files.

```
# Create data.frame with NA values
df_na_values <- data.frame(
  numbers = c(1, NA, 3),
  food = c(NA, NA, NA),
  letters = c("a", NA, "c")
)
df_na_values
```

```
#>   numbers food letters
#> 1       1   NA       a
#> 2      NA   NA    <NA>
#> 3       3   NA       c
```

```
df_na_values %>%
  remove_empty(c("rows", "cols"))
```

```
#>   numbers letters
#> 1       1       a
#> 3       3       c
```

2.4.3 *remove_constant()*

remove_constant() drops columns from a data.frame that contain only a single constant value (with an na.rm option to control whether NAs should be considered as different values from the constant).

```
# Create data.frame with a column containing constant value
df_constant <- data.frame(cool_numbers = 1:3, boring = "the same")
df_constant
```

```
#>   cool_numbers   boring
#> 1            1 the same
#> 2            2 the same
#> 3            3 the same
```

```
df_constant %>% remove_constant()
```

```
#>   cool_numbers
#> 1            1
#> 2            2
#> 3            3
```

2.4.4 convert_to_date() and convert_to_datetime()

Do you remember loading data from Excel and seeing `36922.75` instead of dates? Well, `convert_to_date()` and `convert_to_datetime()` will convert this format and other date–time formats to actual dates! If you need more customization, check out `excel_numeric_to_date()`.

```r
convert_to_date(36922.75) # Date
```

```
#> [1] "2001-01-31"
```

```r
convert_to_datetime(36922.75) # Date and time
```

```
#> [1] "2001-01-31 18:00:00 UTC"
```

2.4.5 row_to_names()

`row_to_names()` is a function that takes the names of variables stored in a row of a data frame and makes them the column names of the data frame. It can also remove the row that contained the names, and any rows above it, if desired.

```r
# Create data.frame with column names in a row
df_row_names <- data.frame(
  x_1 = c(NA, "Names", 1:3),
  x_2 = c(NA, "Value", 4:6)
)
df_row_names
```

```
#>     x_1   x_2
#> 1  <NA>  <NA>
#> 2 Names Value
#> 3     1     4
#> 4     2     5
#> 5     3     6
```

```r
# Elevate a row to be the column names
row_to_names(df_row_names, 2)
```

```
#>   Names Value
#> 3     1     4
#> 4     2     5
#> 5     3     6
```

2.5 Summary

This chapter introduced the concept of tidy data, focusing on its structured format where each variable corresponds to a column, each observation to a row, and each observational unit type to a table. We highlighted the distinctions between wide and long data formats and delved into practical R tools for organizing data into this tidy structure, including an overview of tibbles and additional resources for data cleaning. Understanding these principles of tidy data is invaluable and will prove beneficial throughout your career. Next, we will shift our focus to relational data and the concept of joins!

Chapter 3
Relational Data

Data organization involves structuring information in a way that makes it easy to store, manage, update, and retrieve. One effective approach is to use relational structures, which arrange data into tables with rows and columns. In these tables, each row represents a specific entity, while each column details a particular attribute of that entity.

Grasping the basic structure of relational data organization paves the way for understanding how these tables interact and connect. This interaction is crucial to the flexibility and power of relational structures.

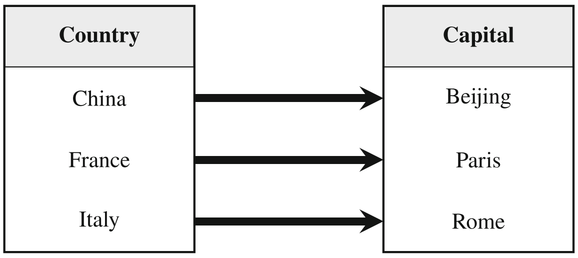
The real strength of this approach lies in the sophisticated connections between tables. These relationships enable complex data structures to be organized and queried efficiently. By understanding these connections, you can fully leverage the potential of relational data organization. Let's delve into these relationships, explore their uses, and learn how to visualize them with diagrams.

3.1 Relationship Types

3.1.1 One to One (1:1)

In a one-to-one cardinality[1] relationship, a single row in the first table corresponds to just one row in the second table, and vice versa. For example, the relationship between countries and their respective capitals:

Figure 3-1
One-to-One Example



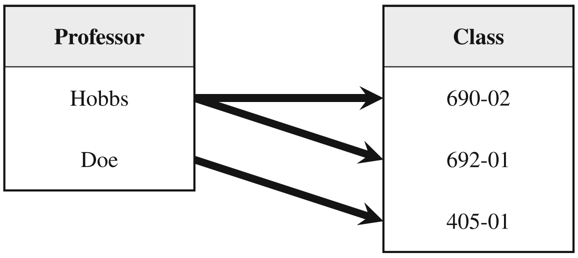
[1] Cardinality in a relationship refers to the number of times an entity from one set is involved in a relationship with entities from another set. It illustrates the connections between entities and how frequently these connections occur.

N. Tkachenko, *Data Insight Foundations*, https://doi.org/10.1007/979-8-8688-0580-6_3

3.1.2 *One to Many (1:M)*

In a one-to-many relationship, a single row in the first table can be linked to multiple rows in the second table, but a row in the second table is related to only one row in the first table. For instance, one professor can teach several classes:

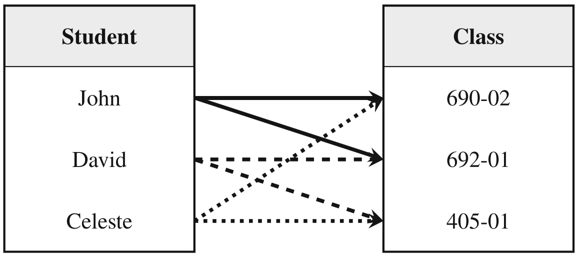
Figure 3-2 One-to-Many Example



3.1.3 *Many to Many (M:N)*

In a many-to-many relationship, a single row in the first table can be associated with many rows in the second table, and similarly, a row in the second table can be associated with many rows in the first table. For example, multiple students can be enrolled in multiple classes:

Figure 3-3 Many-to-Many Example



3.2 The Concept of Keys

A primary key (PK) is a unique identifier for each row within a table; every table should possess a PK. To establish relationships between tables, we integrate PKs from one table into another, where they become foreign keys (FKs). These FKs allow us to make connections between related entities across different tables. For example, in a system with classes and professors, ClassID acts as the PK for the classes, and ProfessorID serves as both a PK for the professors and an FK in the classes table, linking each class to its corresponding professor.

Table 3-1 Keys

ClassID (PK)	ProfessorID (FK)	Credits	Location
620-01	1	4	LM-340
623-01	2	2	UM-102
663-01	2	2	LO-234

ProfessorID (PK)	Professor
1	Arman
2	Alessandra

3.3 Types of Joins

A join operation is used to merge two or more tables based on a related column between them. There are several types of joins. To exemplify how joins operate, we will use two tables: `employees` and `projects`. The `employees` table includes `employee_id` (PK) and `name`, while the `projects` table contains `project_id` (PK) and `employee_id` (FK).

Table 3-2 Employee Table

employee_id	name
1	John
2	Jane
3	Bob
4	Alice
5	Tom

Table 3-3 Project Table

project_id	employee_id
1	1
2	2
3	3
4	1
5	4
6	6

3.3.1 Outer Joins

Outer joins are used to return matched data and unmatched data from one or both tables, effectively creating a more comprehensive table.

3.3.1.1 Left Join

A left join retrieves all rows from the left table, along with matched rows from the right table, adding information to the left table. When performing a left join on employee and project tables, it shows which employees are assigned to which projects. If there's no corresponding match in the right table, the result displays `NA`, indicating, for instance, that Tom has no project assigned.

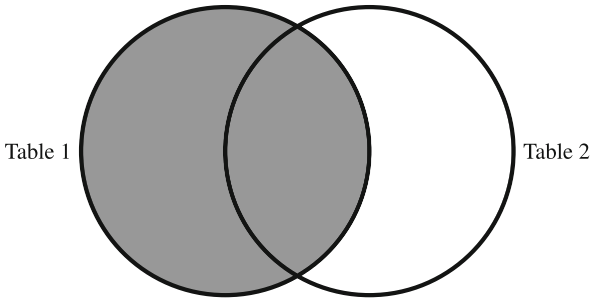


Figure 3-4 Left Join Illustration

```
left_join_result <- employees %>%
  left_join(projects, by = "employee_id")
```

Table 3-4 Result of Left Join

employee_id	name	project_id
1	John	1
1	John	4
2	Jane	2
3	Bob	3
4	Alice	5
5	Tom	NA

3.3.1.2 Right Join

A right join operates similarly to a left join but retrieves all rows from the right table and only the matched rows from the left table. In what follows, we perform a right join on the employee and project tables, showing which projects are assigned to which employees. It is akin to doing a left join between the project and employee tables. It appears that Project 6 has no one assigned to it.

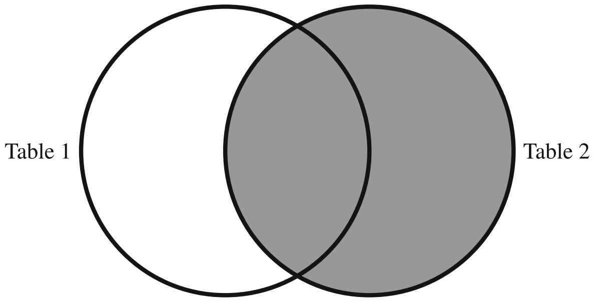


Figure 3-5 Right Join Illustration

```
right_join_result <- employees %>%
  right_join(projects, by = "employee_id")
```

Table 3-5 Result of Right Join

employee_id	name	project_id
1	John	1
1	John	4
2	Jane	2
3	Bob	3
4	Alice	5
6	NA	6

3.3.1.3 Full Join

A full join combines all rows from both the left and right tables, filling in `NA` values for nonmatching rows. This merge offers a complete overview of the combined data. We can observe that Project 6 and Tom are unpaired.

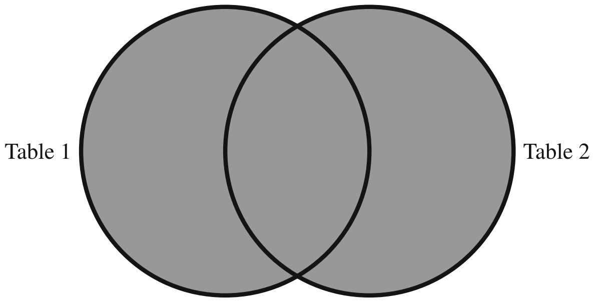


Figure 3-6 Full Join Illustration

```
full_join_result <- employees %>%
  full_join(projects, by = "employee_id")
```

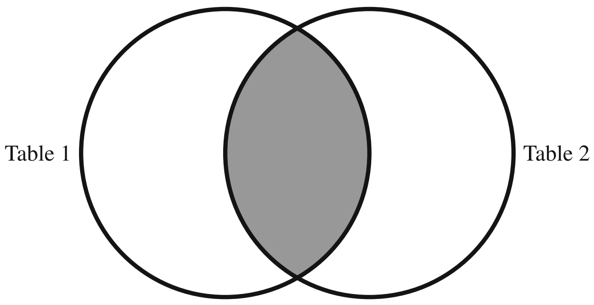
Table 3-6 Result of Full Join

employee_id	name	project_id
1	John	1
1	John	4
2	Jane	2
3	Bob	3
4	Alice	5
5	Tom	NA
6	NA	6

3.3.1.4 Inner Join

An inner join returns only the matched rows between two tables, retaining only those rows that have a match in both tables and creating a complete dataset. As a result, we have no `NA` values because rows without pairs were dropped.

Figure 3-7 Inner Join Illustration



```r
inner_join_result <- employees %>%
  inner_join(projects, by = "employee_id")
```

Table 3-7 Result of Inner Join

employee_id	name	project_id
1	John	1
1	John	4
2	Jane	2
3	Bob	3
4	Alice	5

3.3.2 Filtering Joins

3.3.2.1 Anti-join

An anti-join returns the rows from the left table that do not find a match in the right table, without adding new columns to the output. It is useful for filtering rows based on the absence of matching entries in the other table. For example, instead of performing a left join and looking for missing values, an anti-join can directly identify who doesn't have a project assigned.

```r
anti_join_result <- employees %>%
  anti_join(projects, by = "employee_id")
```

Tom does not have a project assigned! Perhaps he could take on Project 6?

Table 3-8 Result of Anti-Join

employee_id	name
5	Tom

3.3.2.2 Semi-join

A semi-join is similar to an inner join in identifying matching rows between two tables. However, unlike an inner join, it adds no new columns to the output. Instead, it filters the rows from the left table that have a corresponding match in the right table. You'd use a semi-join when you want to filter the left table based on the presence of matching entries in the right table, for example, allowing you to see only the people who have a project assigned.

```r
semi_join_result <- employees %>%
  semi_join(projects, by = "employee_id")
```

Table 3-9 Result of Semi-Join

employee_id	name
1	John
2	Jane
3	Bob
4	Alice

3.4 Visualizing Relationships

We're going to use the Unified Modeling Language (UML) to visualize relationships in data. This might be your first time hearing that there is a language behind diagrams. UML is a standard graphical notation used to describe software designs. It's a powerful tool for planning, visualizing, and documenting projects. Different types of diagrams can be used to depict structures, behaviors, and interactions using a standard set of symbols and notation.

UML coding tools like Mermaid and Graphviz are great options, but I find that drag-and-drop web applications such as LucidChart and Draw.io are more user-friendly.

First, let's introduce an entity, an object (place, person, thing) we want to track. In our case, these will be a customer, an order, and a product. Each entity possesses attributes; for example, a customer has `customer_id`, `name`, `email`, and `address`, for example, and other entities also have a list of attributes. These entities and attributes are represented as rows and columns, respectively, in your tables. Tables can be interconnected, and these relationships are visualized by drawing lines between the tables. Cardinality is used to describe these relationships in numeric terms, akin to our discussion in Section 3.1.

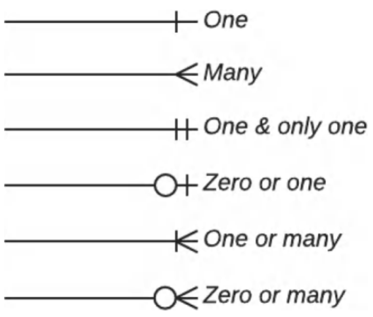


Figure 3-8 Cardinality

For instance, let's examine the relationship between a customer and an order. Using the min–max framework, what is the minimum and maximum number of orders a customer can have? A customer can have zero orders (min = 0) and an indefinite amount of orders (max = many). So the relationship of customer to order is 0 or many. Now let's look in the opposite direction: How many customers can an order have? An order can have only one customer (min = 1, max = 1).

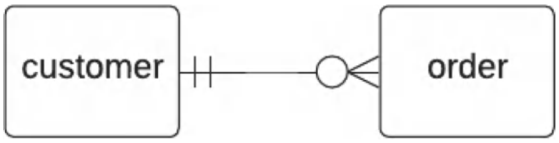


Figure 3-9 Customer–Order Relationship

Next, let's examine the relationship between an order and a product. An order can include one or many products, and each product can be in zero or many orders. The complete diagram would resemble the following:

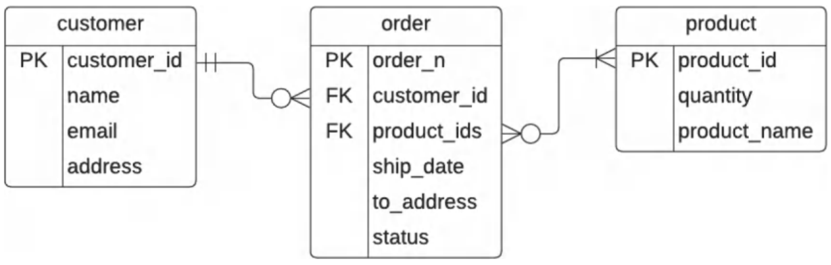


Figure 3-10 Entity–Relationship Diagram

Creating such a diagram is recommended whenever you're planning a project with a complex design. It clarifies the necessary tables and their relationships. You could also sketch a diagram whenever you're unsure about a dataset. If you'd like to delve deeper into this topic, check out the Lucid Software YouTube (https://www.youtube.com/@lucid_software) guides.

3.5 Summary

In this chapter, we delved into the essentials of relational data systems, encompassing their structure, key relationships, and join operations. We examined how tables, rows, and columns constitute the backbone of these systems and explored the critical role of PKs and FKs in linking these tables. Additionally, we discussed various types of relationships and joins. Looking ahead, the next chapter will focus on the practice of data validation.

Chapter 4
Data Validation

Data validation is a crucial part of data analysis, encompassing the maintenance of data integrity, accuracy, and cleanliness for computational tasks. Since the results of analysis are heavily dependent on the quality of the input data, having a robust validation process is essential. A lack of such a process can lead to the effect of “Garbage in, garbage out.”

Imagine dedicating hours to an analysis only to discover a duplicate row or sporadic NAs. The key to avoid such scenarios lies in regular data checks. You can write a function to perform these checks or set up data validation pipelines for multiple checks. If the requirement includes sharing validation results, generate reports accordingly.

While this might seem daunting, there are numerous R packages, along with native functions, specifically designed to significantly simplify this task.

4.1 Manual Inspection

Despite the convenience of automation, remember that you can’t address what you’re not aware of. Sometimes, data may not be ready for immediate analysis and may require cleaning before validation. Therefore, it’s essential to manually open the files, inspect the tables and their values, and conduct preliminary exploratory analysis. From my experience, the most crucial step is to meticulously examine each variable individually, building an understanding of their meanings and implications. This thorough process may consume a few hours, but the payoff is significant: you’ll gain an enhanced familiarity with the dataset, coupled with a comprehensive list of aspects requiring attention and rectification. Lastly, always verify your results by reviewing the output table, an important yet simple step to remain fully engaged with the raw data.

4.2 Handling Data Issues

In the event of data issues, it’s crucial to stop the script execution and alert the user. Base R offers several functions to facilitate this. For instance, the `is_numeric()` function, along with its `is_*()` counterparts, are traditional examples. Control flow functions like `if`, `else`, `stop()`, and logical operators prove to be quite useful.

N. Tkachenko, *Data Insight Foundations*, https://doi.org/10.1007/979-8-8688-0580-6_4

```r
x <- c(1, 2, 3)
# Create a data frame with a duplicate row
df <- data.frame(x = c(x, 1), y = c(x * 2, 2))
# Stops execution if `x` isn't numeric
if (!is.numeric(x)) stop("x is not numeric!")
stopifnot(is.numeric(x)) # Halts if `x` isn't numeric
# Stops the process if `x` contains non-positive values
if (!all(x > 0)) stop("x contains non-positive values!")
# Halts execution if `x` has duplicates
if (any(duplicated(x))) stop("x contains duplicated values!")
# Issues a warning if `df` has duplicate rows
if (any(duplicated(df))) warning("df has duplicated rows!")
```

```
#>  Warning: df has duplicated rows!
```

4.2.1 Assert Your Conditions

`assertr` is an excellent package for tidyverse-compatible data validation. Rather than manually running checks, you can add an assert statement to verify your assumptions about the data. If the assumption holds, the code continues running; however, if it fails, an error is thrown and execution is halted.

`assertr` provides functions which set conditions your data must meet:

- `verify()`: This function checks whether a given logical condition holds true for the entire dataset. If not, it halts the execution and throws an error.
- `assert()`: This function applies a specific predicate function (a function that returns true or false) to selected columns in your data frame. The data passes validation only if all values in those columns satisfy the predicate function's condition.
- `insist()`: Similar to `assert()`, this function allows you to specify a function as its argument. However, it takes a predicate function generator, allowing you to vary the predicate depending on the data you pass. For instance, in the example below, it is used to calculate the standard deviation and use this to find outliers.

`assertr` comes with a collection of "Helper" predicate functions. These are designed to be used in conjunction with the above functions. A few of such functions are `within_bounds()`, `not_na()`, `in_set()`, `within_n_sds()` etc. You can also create and use your own predicate functions if needed.

The syntax of the package integrates smoothly into a typical `tidyverse` pipeline. Here's a brief example using the built-in dataset `mtcars`, which contains fuel consumption and 10 aspects of automobile design and performance for 32 automobiles (1973–74 models):

```r
library(assertr)
library(magrittr) # for %>% pipe

# Load built-in R dataset
data(mtcars)

mtcars %>%
  verify(nrow(.) > 0) %>% # Verify dataset isn't empty
  # Assert that specified engine types (0,1) are present
  assert(in_set(0, 1), vs) %>%
  # Ensure 'mpg' values are within two standard deviations
  insist(within_n_sds(2), mpg)
```

```
#>   The ``mpg'' column violates the assertion ``within_n_sds(2)'' two times:
#>       verb redux_fn       predicate column index value
#>   1 insist       NA within_n_sds(2)    mpg    18  32.4
#>   2 insist       NA within_n_sds(2)    mpg    20  33.9

#>   Error: assertr stopped execution
```

In this example, `verify()` ensures `mtcars` will have more than one row, `assert()` checks for the presence of certain values of engine type (0 = V-shaped, 1 = straight), and `insist()` ensures that there are no outliers—all values are within two standard deviations. If any check fails, the pipeline stops and produces an error message, as it did, indicating that `insist()` failed twice for the `mpg` column.

4.2.2 Precise Validation with Pointblank

The `pointblank` package offers enhanced functionality for constructing validation pipelines and reporting. The package introduces so-called agent objects for continuous validation, allowing for reusable validation checks across different datasets. With its robust functionality, `pointblank` is ideal for significant projects, enabling the creation of validation pipelines, HTML reports, and distribution to stakeholders. In what follows, we'll create an agent and investigate the `mtcars` dataset. It is worth noting that using an agent is not mandatory—you can use the functions similarly to how you use `assertr`.

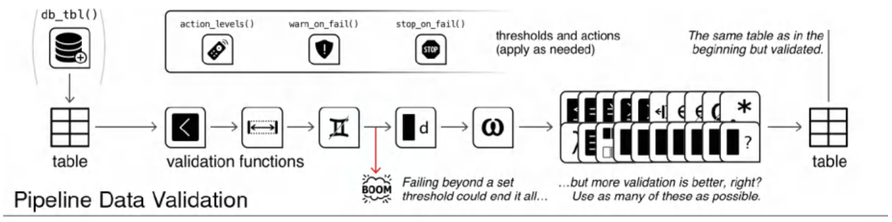

db_tbl()
action_levels()
warn_on_fail()
stop_on_fail()
thresholds and actions (apply as needed)
The same table as in the beginning but validated.
table
validation functions
table
BOOM
Failing beyond a set threshold could end it all...
...but more validation is better, right? Use as many of these as possible.
Pipeline Data Validation

Figure 4-1 Pointblank Workflow Image from Package Documentation

Step 1: Create a Validation Plan (an Agent)

```
library(pointblank)

agent <- create_agent(
  tbl_name = "a simple mtcars data validation",
  label = "an example of using pointblank for data validation",
  tbl = mtcars # attach the data frame to validate
)
```

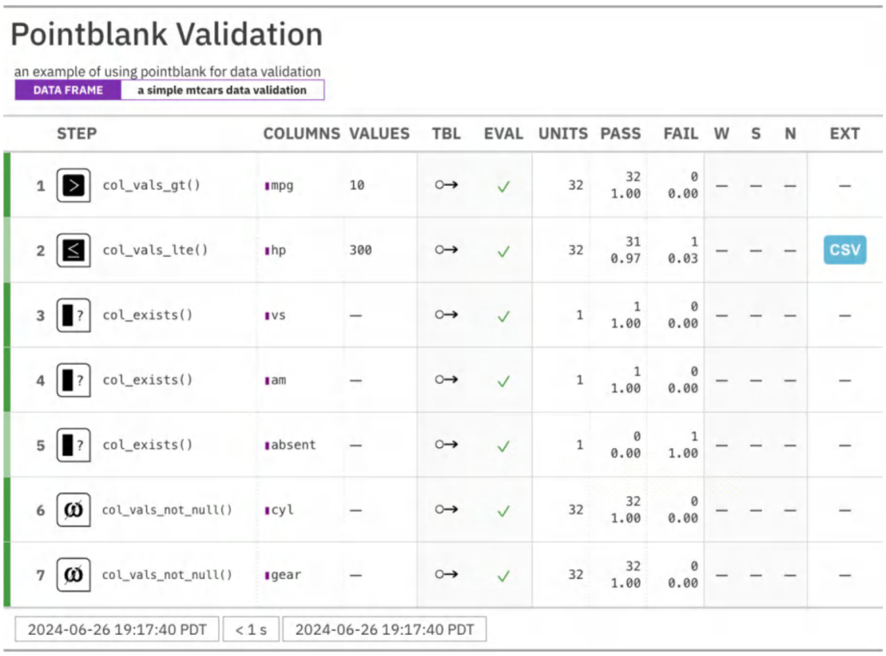
Step 2: Specify Checks

```
agent <- agent %>%
  # Check if 'mpg' values are greater than 10
  col_vals_gt(vars(mpg), value = 10) %>%
  # Check if 'hp' values are less than or equal to 300
  col_vals_lte(vars(hp), value = 300) %>%
  # Check if 'vs', 'am', and absent columns exist
  col_exists(vars(vs, am, absent)) %>%
  # Check if 'cyl' and 'gear' columns have no missing values
  col_vals_not_null(vars(cyl, gear))
```

Step 3: Execute Checks

```
(report <- interrogate(agent, extract_failed = T))
```

The interrogation revealed that all checks were correctly evaluated (EVAL), with almost all checks passing. The exception was a single row that failed because the horsepower exceeded 300. Additionally, there is no column named `absent`. You can view the proportion of PASS/FAIL in the respective columns. The columns with failed checks are highlighted in color on the far left side of the table. The results can be downloaded as a CSV, which is particularly helpful for sharing the report with others, and there's a built-in functionality for sharing via email. To examine the failed rows, use `extract_failed = TRUE` and access them with `$extracts`. Below, we observe the failed row: 335 horsepower, indicating a very fast car (Maserati Bora)!



Pointblank Validation

an example of using pointblank for data validation

DATA FRAME a simple mtcars data validation

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	S	N	EXT
1	col_vals_gt()	mpg	10	○→	✓	32	32 1.00	0 0.00	—	—	—	—
2	col_vals_lte()	hp	300	○→	✓	32	31 0.97	1 0.03	—	—	—	CSV
3	col_exists()	vs	—	○→	✓	1	1 1.00	0 0.00	—	—	—	—
4	col_exists()	am	—	○→	✓	1	1 1.00	0 0.00	—	—	—	—
5	col_exists()	absent	—	○→	✓	1	0 0.00	1 1.00	—	—	—	—
6	col_vals_not_null()	cyl	—	○→	✓	32	32 1.00	0 0.00	—	—	—	—
7	col_vals_not_null()	gear	—	○→	✓	32	32 1.00	0 0.00	—	—	—	—

2024-06-26 19:17:40 PDT | < 1 s | 2024-06-26 19:17:40 PDT

Figure 4-2 Pointblank Output

```r
report$extracts$`2` # `2` refers to the check number
```

```
#>  # A tibble: 1 x 11
#>      mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
#>    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#>  1    15     8   301   335  3.54  3.57  14.6     0     1     5     8
```

4.3 Summary

We learned that data validation is crucial in data analysis, playing a key role in maintaining the integrity, accuracy, and cleanliness of data. This process requires regular checks, which can be automated using control flow functions, as well as the `assertr` and `pointblank` packages in R. Despite the efficiency of automation, manual inspection still holds significant importance and should be the initial step in the validation process. Ensuring the validation of your data is essential for providing high-quality inputs in your reports. In the next chapter, we will explore strategies for dealing with missing values.

Chapter 5
Imputation

Imputation is a statistical technique that fills in missing or incomplete data by estimating values from existing information, but its controversial nature arises from the introduction of artificial data, potentially impacting analysis outcomes as newly completed data is assigned more weight. In this chapter we will cover types of missing data and different ways of dealing with them.

Note 5.1 As in medicine, the best solution to missing data is prevention!

5.1 Types of Missing Data

Understanding the type of missing data is crucial as it can guide the appropriate choice of imputation methods or handling strategies to minimize potential biases in the analysis.

1. **Missing Completely at Random (MCAR)**: The missingness in data is entirely random, and the probability of missing data is the same for all observations, regardless of observed or unobserved values. MCAR is an ideal scenario, as missing data do not depend on any other variables in a dataset, for example, a thermometer battery dying and not recording temperature for a session.
2. **Missing at Random (MAR)**: MAR occurs when the missingness is related to observed variables in the dataset, but not to the unobserved values. We might see MAR if males are less likely to answer survey questions about mental health than women.
3. **Missing Not at Random (MNAR)**: In MNAR, the missingness is related to unobserved values. The probability of missing data depends on the variables that are missing. For instance, avid smokers might skip a section on lung health, which would result in biased survey results.
4. **Missing by Design**: Missing by design occurs when specific data points are intentionally omitted, often as part of the study design or data collection process. In such cases, the missing data are systematic and have a purpose.

Before addressing missing data, it is often essential to distinguish between different types of missingness. Explicitly marking missing data as NA in R helps in identifying the patterns and dealing with the missing values appropriately.

5.2 Dealing with Missing Data

There are several ways to handle missing data:

1. **Keep It**: In some cases, it might be appropriate to retain missing data if the missingness does not significantly affect the analysis.

N. Tkachenko, *Data Insight Foundations*, https://doi.org/10.1007/979-8-8688-0580-6_5

2. **Drop It (Listwise Deletion or Complete Case Analysis)**: This approach involves removing rows with missing data. It's simple and can be suitable if missing data are minimal and random. However, it can lead to information loss, potential bias, and reduced sample size. Despite these drawbacks, it's a common method in quantitative research due to its simplicity.
3. **Impute It**: Imputation involves estimating missing values based on observed data, which allows for a complete dataset and ensures that all cases will be retained for analysis. However, incorrectly done, it will introduce bias.
4. **Set as Dummy or a Factor**: Sometimes, missing values can be treated as a separate category. This can be done by creating a dummy variable or converting the variable into a factor. This approach acknowledges the missing data and incorporates them into the analysis as a distinct group.

Imputation, while a powerful tool, must be applied judiciously as it modifies the original dataset and can substantially influence analysis outcomes. It's essential to comprehend the reasons behind the missing data, the assumptions involved, and the justification for using imputation in the specific context of your analysis.

Note 5.2 Imputation should never be a default choice but a well-considered strategy.

5.2.1 *Explicitly Handling Missing Data with* `complete()`

When working with datasets, it's crucial to understand that missing values aren't always explicitly identifiable. These implicit absences can misleadingly suggest data completeness, emphasizing the necessity of appropriate identification and handling. The `tidyr` package's `complete()` function offers a robust solution.

The `complete()` function generates a new dataframe featuring all potential combinations of specified columns, thereby assuring data comprehensiveness. This process, complemented by filling absent combinations with default values, mitigates the risk of implicit missing data.

Take, for instance, a dataset tracking four students attending various classes over three days. Initially, this dataset might seem comprehensive. However, only the attending students were recorded, leaving a data gap for absentees.

In this case, we employ `complete()` to create `complete_df`, a new dataframe that encapsulates all conceivable combinations of `student_id`, `day`, and `class.` This method ensures accurate recording of each student's attendance for every class on each day, irrespective of the initial data's shortcomings. The "present" column's missing values default to `FALSE`, clearly denoting unrecorded attendance.

```r
df <- tribble(
  ~day, ~class, ~student_id, ~present,
  1, "English", 1, T,
  1, "English", 2, T,
  1, "English", 4, T,
  1, "Science", 2, T,
  2, "Math", 1, T,
  2, "Math", 2, T,
  2, "Math", 4, T,
  2, "English", 4, T,
  2, "English", 1, T,
  3, "Math", 1, T,
  3, "Math", 2, T,
  3, "Math", 1, T
)
```

```
#>   # A tibble: 12 x 4
#>         day class   student_id present
#>       <dbl> <chr>        <dbl> <lgl>
#>    1      1 English          1 TRUE
#>    2      1 English          2 TRUE
#>    3      1 English          4 TRUE
#>    4      1 Science          2 TRUE
#>    5      2 Math             1 TRUE
#>    6      2 Math             2 TRUE
#>    7      2 Math             4 TRUE
#>    8      2 English          4 TRUE
#>    9      2 English          1 TRUE
#>   10      3 Math             1 TRUE
#>   11      3 Math             2 TRUE
#>   12      3 Math             1 TRUE
```

```
# Complete dataframe with all student_id, day, and class combinations
complete_df <- df %>%
  complete(
    # Generate sequence of student IDs, using unique IDs as max value
    student_id = full_seq(student_id, 1),
    # Create combinations that already exist in the data
    nesting(day, class),
    # Fill missing attendance data with FALSE
    fill = list(present = FALSE),
    # Limit fill to newly created (previously implicit) entries
    explicit = FALSE
  ) %>%
  # Order by day, class, student_id, and present
  arrange(day, class, student_id, present)
```

```
#>   # A tibble: 21 x 4
#>     student_id   day class   present
#>          <dbl> <dbl> <chr>   <lgl>
#>    1          1     1 English TRUE
#>    2          2     1 English TRUE
#>    3          3     1 English FALSE
#>    4          4     1 English TRUE
#>    5          1     1 Science FALSE
#>    6          2     1 Science TRUE
#>    7          3     1 Science FALSE
#>    8          4     1 Science FALSE
#>    9          1     2 English TRUE
#>   10          2     2 English FALSE
#>   11          3     2 English FALSE
#>   12          4     2 English TRUE
#>   13          1     2 Math    TRUE
#>   14          2     2 Math    TRUE
#>   15          3     2 Math    FALSE
#>   16          4     2 Math    TRUE
#>   17          1     3 Math    TRUE
#>   18          1     3 Math    TRUE
#>   19          2     3 Math    TRUE
#>   20          3     3 Math    FALSE
#>   21          4     3 Math    FALSE
```

By using `nesting()`, which creates all combinations present data, within `complete()`, we explicitly handle missing data, creating a comprehensive nested dataset suitable for further analysis. This approach guarantees that our analysis will be based on a more complete and reliable dataset, providing accurate attendance data across different classes and days.

5.2.2 Simple Imputations

To explore different imputations, we will use an "airquality" dataset that contains daily measurements of air pollutants and weather conditions in New York City during a five-month period in 1973. It includes data on ozone concentration, solar radiation, temperature, wind speed, and relative humidity. To simplify the example, we will remove all rows with missing `Solar.R` values and focus on `Ozone`. We will also define `head_na` to view the values that were imputed.

```
head_na <- function(data) {
  data %>%
    filter(is.na(Ozone)) %>% # Filter rows with NA in ``Ozone"
    select(-Ozone) %>% # Exclude ``Ozone" column from results
    head(5) # Display the first 5 rows
}
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
```

```
# Remove rows with missing values in ``Solar.R"
airquality <- drop_na(airquality, Solar.R)
```

My favorite way to check for missing data is to use a premade function from the `visdat` package called `vismiss`. It will show a distribution of missing data, you can prearrange data and facets to better identify missing data. The following graph is arranged by day, and each month has its own facet, so we can see a pattern of missing values. It looks like June (6) has many missing values for `Ozone`.

```
airquality %>%
  arrange(Month, Day) %>%
  visdat::vis_miss(facet = Month)
```

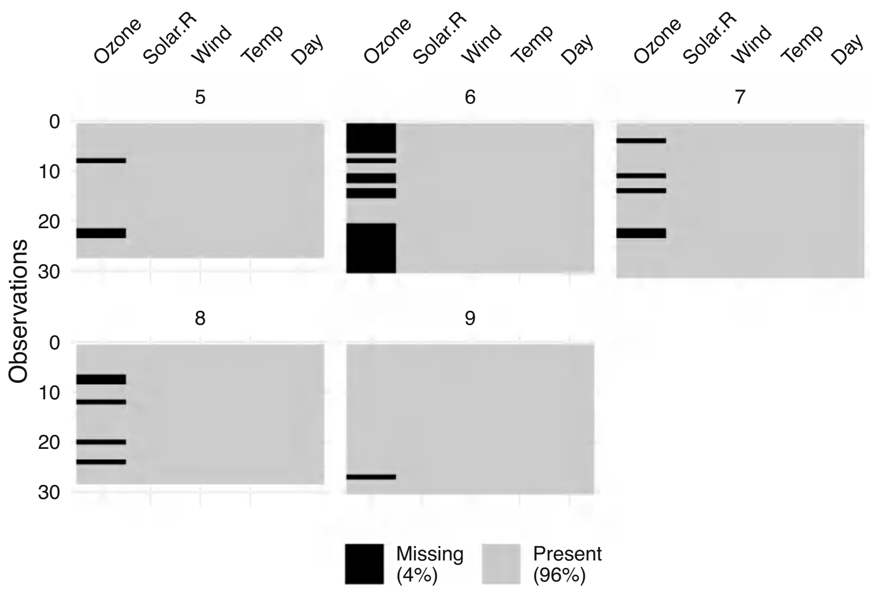


Figure 5-1 vizmiss() Plot

5.2.2.1 Fixed Value Imputations

Missing values are replaced with a predetermined constant value. This method is simple and useful when you believe the fixed value reasonably represents the missing data.

Dummy Variable Adjustment

One simple and generally applicable method of imputation is to replace missing values with a constant, such as 0, and add a binary indicator column to flag missing entries (1 for missing, 0 for not missing). This way you can retain all data points and account for missingness. This approach ensures a complete dataset and allows your model to estimate the impact of missing data through an additional coefficient for the indicator column.

5.2.2.2 Mean and Median Imputation

Mean and median imputation are techniques for handling missing data by replacing missing values with the mean or median of the nonmissing values in the same column. Mean imputation is effective when the data follow a normal distribution but can be skewed by outliers, leading to biased estimates. In contrast, median imputation is more robust to outliers and is suitable for data with skewed or extreme values.

5.2.2.3 Fill

In some scenarios, it’s rational to replace missing data with either preceding or succeeding values. This approach is particularly effective with datasets where the next available value logically replaces missing ones, such as in time series or ordered data.

I recommend familiarizing yourself with `coalesce`, which lets you replace missing values with a predefined value or with values from another column.

```
airquality %>%
  arrange(Month, Day) %>%
  mutate(
    # Impute with a fixed value (0 in this case)
    imp_fixed = coalesce(Ozone, 0),
    Ozone_missing = is.na(Ozone),
    # Impute with the mean
    imp_mean = coalesce(Ozone, round(mean(Ozone, na.rm = TRUE), 2)),
    # Impute with the median
    imp_median = coalesce(Ozone, median(Ozone, na.rm = TRUE)),
    imp_fill = Ozone,
    .keep = "used" # Keep only "used" columns
  ) %>%
  # Fills missing values with the most recent nonmissing value above
  fill(imp_fill, .direction = "down") %>%
  head_na()
```

```
#>   imp_fixed Ozone_missing imp_mean imp_median imp_fill
#> 1         0          TRUE     42.1         31        8
#> 2         0          TRUE     42.1         31       32
#> 3         0          TRUE     42.1         31       32
#> 4         0          TRUE     42.1         31       37
#> 5         0          TRUE     42.1         31       37
```

Remember that the choice of imputation method can significantly impact the results of your analysis. Always consider the nature of your data, the distribution of missingness, and the potential implications of each method before making a decision.

5.2.2.4 K-Nearest Neighbors (KNN)

KNN imputation is effective for datasets with complex structures that simple methods cannot capture. It estimates missing values by leveraging the relationships between variables. Specifically, KNN identifies the “k” nearest data points (neighbors) based on similarity measures, such as Euclidean distance, and uses their values to impute the missing data. This method is particularly useful when the data exhibit intricate patterns or dependencies, allowing for more accurate and context-aware imputations.

```
# KNN imputation using the VIM package
library(VIM)
airquality_knn <- kNN(airquality)

airquality_knn %>%
  slice(which(is.na(airquality$Ozone))) %>%
  select("Ozone") %>%
  head(n=5)
```

```
#>   Ozone
#> 1    16
#> 2    11
#> 3    32
#> 4    40
#> 5    32
```

5.2.2.5 Regression

Regression imputation is useful when there are relationships between variables that a regression model can capture. This method involves predicting missing values based on the observed values of other variables in the dataset. Fitting a regression model, such as linear regression, to the available data, makes it possible to estimate missing values using the established relationships. This approach is particularly effective when the data exhibit clear and predictable associations between variables, allowing for more precise imputations.

```
# Fit a linear regression model
model <- lm(Ozone ~ ., data = airquality)

# Predict missing values
predicted_values <- predict(model, newdata = airquality)

# Replace missing values with predicted values
airquality %>%
  mutate(
    Ozone = Ozone,
    imp_lm = ifelse(is.na(Ozone), predicted_values, Ozone),
    .keep = "used"
  ) %>%
  head_na()
```

```
#>        imp_lm
#>   1  35.446534
#>   2 -16.177404
#>   3   1.688479
#>   4  51.628995
#>   5  40.719713
```

5.2.2.6 Forest

Tree-based methods, such as random forests, are effective for handling complex data structures with nonlinear relationships or interactions between variables. These methods can capture intricate patterns in the data, making them a good choice for imputation when simpler methods fall short. However, tree-based methods may not perform well with small sample sizes or sparse data because they require a certain amount of information to build accurate models. Despite this limitation, their ability to model complex relationships makes them a powerful tool for imputation in many scenarios.

```
# Install and load the missForest package
library(missForest)

airquality %>%
  mutate(
    Ozone = Ozone,
    imp_forest = missForest(.)$ximp$Ozone,
    .keep = "used"
  ) %>%
  head_na()
```

```
#>   imp_forest
#> 1     22.250
#> 2     14.465
#> 3     25.110
#> 4     32.690
#> 5     25.560
```

Remember, each of these methods has its own strengths and weaknesses, and the choice of method should be guided by the nature of your data and the specific requirements of your analysis. Always check the assumptions of the imputation method you're using, and consider the potential impact on your results.

5.2.3 Best Worst-Case and Worst Best-Case Scenarios

A straightforward method to assess the impact of missing data is to replace it with both the worst and best possible outcomes. In clinical trials, it's typical to assume the control group will achieve a beneficial outcome, calculated as the mean plus two standard deviations, while the treatment group will experience a detrimental outcome, calculated as the mean minus two standard deviations. This method creates two new data series, one reflecting an optimistic scenario and the other a pessimistic one, known as the best worst-case scenario. Conversely, the worst best-case scenario reverses these assumptions.

For our purposes, we will apply this method by creating two new columns in our dataset, imputing Ozone values using high (mean plus two standard deviations) and low (mean minus two standard deviations) estimates for each month to analyze how these extremes might affect our results.

```
airquality %>%
  mutate(
    Ozone_best = coalesce(Ozone, mean(Ozone, na.rm = T) +
                                 2 * sd(Ozone, na.rm = T)),
    Ozone_worst = coalesce(Ozone, mean(Ozone, na.rm = T) -
                                  2 * sd(Ozone, na.rm = T)),
    .by = Month
  ) %>%
  summarize(
    Ozone_best = round(mean(Ozone_best), 0),
    Ozone = round(mean(Ozone, na.rm = T), 0),
    Ozone_worst = round(mean(Ozone_worst), 0),
    .by = Month
  )
```

```
#>   Month Ozone_best Ozone Ozone_worst
#> 1     5         29    24          19
#> 2     6         55    29           4
#> 3     7         69    59          49
#> 4     8         75    60          45
#> 5     9         33    31          30
```

5.2.4 Multiple Imputations

Imputing can be better than dropping data because, instead of losing the data, you preserve them! However, there is a method to this [18].

Multiple imputation is a statistical technique that has been increasingly utilized since its inception in the early 1970s. It is a simulation-based method designed to address the issue of missing data in research studies. The process of multiple imputation is carried out in three main steps:

1. **Imputation Step**: In this initial stage, missing values in the dataset are identified and replaced with a set of plausible values, creating multiple completed datasets. These plausible values, or "imputations," are generated based on a chosen imputation model. To reduce sampling variability from the imputation process, it is often preferable to generate a minimum of five datasets. A good rule of thumb is that the number of iterations equals the percentage of missing data.
2. **Complete-Data Analysis (Estimation) Step**: Once the imputed datasets are created, the desired analysis is performed separately on each dataset. For instance, if five datasets were generated during the imputation step, then five separate analyses would be conducted.
3. **Pooling Step**: The results obtained from each complete-data analysis are then combined into a single multiple-imputation result. Each analysis result is considered to have the same statistical weight, so there is no need for a weighted meta-analysis.

It is crucial to ensure compatibility between the imputation model and the analysis model, or that the imputation model is more general than the analysis model. This means that the imputation model should include more independent covariates than the analysis model. For instance, if the analysis model includes significant interactions, then the imputation model should include them as well. Similarly, if the analysis model uses a transformed version of a variable, then the imputation model should use the same transformation.

Note 5.3 It's important to note that the multiple imputations assume that the missing data are missing at random (MAR).

To help you make a decision, answer the following questions. If you respond yes to any of them, use only the observed data. However, discuss and report the extent of the missing data and its limitations. Also, consider including best worst-case and worst best-case analyses in your report.

1. Is it valid to ignore missing data (below 5%)? If yes, the missing data are negligible in this case.
2. Is a large proportion of missing data (above 40%) significant? If yes, the missing data are substantial when the proportion exceeds this threshold.
3. Are data missing only in the dependent variable? If yes, only the dependent variable values are missing.
4. Is the MCAR assumption plausible? If yes, the data are missing completely at random.
5. Is the MNAR assumption plausible? If yes, the data are missing not at random.

If you answer no to all these questions, consider using multiple imputation to handle the missing data.

5.2.4.1 Multivariate Imputation by Chained Equations (MICE)

The `mice` package in R is a powerful tool for handling missing data through multiple imputation. It uses the multivariate imputation by chained equations (MICE) algorithm, which creates multiple imputations (replacement values) for multivariate missing data. The basic idea behind the algorithm is to treat each variable that has missing values as a dependent variable in regression and the others as independent (predictors). You can learn more about MICE in Buuren and Groothuis-Oudshoorn [9]. MICE assumes missing values are MAR. First, install and load the `mice` package:

```
install.packages("mice")
library(mice)
```

The `mice` function supports several imputation methods. The choice of method depends on the nature of variables. Here are a few commonly used methods:

- `"pmm"`: Predictive mean matching, useful for numeric data.
- `"cart"`: Classification and regression trees, suitable for both categorical and continuous data.
- `"lasso.norm"`: Lasso linear regression, useful for regularization in linear models.

Following imputation, each dataset can be analyzed independently. Let's perform some of these imputations! The `mice::complete` function is utilized to create a fully imputed dataset from a `mice` object, with the imputation `method` specified and `printFlag = FALSE` used to prevent output from being displayed in the console. Finally, `$Ozone` extracts the column.

```
# Define a convenience function for imputing Ozone using the mice
mice_impute_ozone <- function(data, method) {
  mice::complete(mice(data, method = method, print = FALSE))$Ozone
}
set.seed(1)
airquality %>%
  mutate(
  Ozone = Ozone,
  imp_pmm = mice_impute_ozone(., "pmm"),
  imp_pmm2 = mice_impute_ozone(., "pmm"),
  imp_cart = mice_impute_ozone(., "cart"),
  imp_cart2 = mice_impute_ozone(., "cart"),
  imp_lasso = mice_impute_ozone(., "lasso.norm"),
  imp_lasso2 = mice_impute_ozone(., "lasso.norm"),
  .keep = "used" # Keep only the variables used in mutate call
) %>%
head_na()
```

```
#>   imp_pmm imp_pmm2 imp_cart imp_cart2  imp_lasso imp_lasso2
#> 1      20       24       19        16  15.557613   39.24990
#> 2       6        8        4         1 -16.446412  -25.03575
#> 3       1       18       34        37   4.801063  -32.22336
#> 4      23       46       20        36  60.099795   59.77815
#> 5      39       20       13        12  58.936793   78.69958
```

Note that running the same specification a second time yields a different result. This variation occurs because the algorithms introduce randomness, leading to uncertainty in imputation (instead of a definitive imputation value of 20, it's more accurate to expect a range, such as between 18 and 22). To accommodate this and incorporate the inherent uncertainty, you can utilize the option to generate multiple datasets (`m = #`) in the `mice()` function to carry out multiple imputations:

```
# Perform multiple imputation
imp <- mice(airquality,
            m = 5, # Number of complete datasets to generate
            method = "pmm", # Specifies method for imputation
            seed = 1337, # Set seed for reproducibility
            print = FALSE) # Don't print history on console
```

Now we can fit a linear regression model and summarize the combined results from the five imputed datasets:

```
# Fit a linear model on each imputed dataset
fit <- with(imp, lm(Ozone ~ Solar.R + Wind + Temp + Month))

# Pool the results of these models
pool <- pool(fit)
summary(pool)
```

```
#>          term     estimate   std.error statistic       df      p.value
#> 1 (Intercept) -46.32721776 23.36571710 -1.982700 51.61341 5.273519e-02
#> 2     Solar.R   0.04150937  0.02376728  1.746492 56.84813 8.612689e-02
#> 3        Wind  -3.64264096  0.67015975 -5.435482 40.01848 2.926986e-06
#> 4        Temp   1.79526496  0.26398251  6.800697 90.87476 1.072586e-09
#> 5       Month  -3.13075952  1.58426819 -1.976155 55.32748 5.313157e-02
```

You can also use `gtsummary` to create a summary table, which we will cover in the "Make Tables" section. Here, `tbl_regression` automatically pools the results.

```
library(gtsummary)
tbl_regression(fit)
```

Characteristic	Beta	95% CI	p-Value
Solar.R	0.04	−0.01, 0.09	0.086
Wind	−3.6	−5.0, −2.3	<0.001
Temp	1.8	1.3, 2.3	<0.001
Month	−3.1	−6.3, 0.04	0.053

Imputation is a valuable tool for handling missing data, but it should be used judiciously and based on a thorough understanding of the data and analysis objectives. Responsible imputation ensures that any assumptions made during the process will align with the data generation method, resulting in an analysis of the data that is more reliable and meaningful.

5.3 Summary

In this chapter, we delved into the use of imputation as a technique for addressing missing data in statistical analysis, explored various types of missing data, and discussed several strategies for managing them. The chapter highlighted essential tools in R for this purpose and examined a range of imputation methods. In particular, we focused on responsible imputation practices for trustworthy data analysis. In the next section, we will shift our attention to the practices of reproducible research.

5.3.1 Table of Imputations

The following table allows you to compare the variation in results produced by each imputation method. Notice how the imputations can vary significantly depending on the chosen method!

```r
mice_impute_ozone <- function(data, method) {
  mice::complete(mice(data, method = method, print = FALSE))$Ozone
}
set.seed(1)
airquality %>%
  mutate(
    # Impute with a fixed value (0 in this case)
    imp_fixed = coalesce(Ozone, 0),
    Ozone_missing = is.na(Ozone),
    # Impute with the mean
    imp_mean = coalesce(Ozone, round(mean(Ozone, na.rm = TRUE), 2)),
    imp_fill = Ozone,
    # Impute with the median
    imp_median = coalesce(Ozone, median(Ozone, na.rm = TRUE)),
    imp_forest = missForest(.)$ximp$Ozone,
    imp_lm = ifelse(is.na(Ozone), predicted_values, Ozone),
    imp_knn = airquality_knn$Ozone,
    imp_pmm = mice_impute_ozone(., "pmm"),
    imp_cart = mice_impute_ozone(., "cart"),
    imp_lasso = mice_impute_ozone(., "lasso.norm"),
    .keep = "used"
  ) %>%
  fill(imp_fill, .direction = "down") %>%
  filter(is.na(Ozone)) %>%
  select(-Ozone) %>%
  mutate(across(everything(), ~ round(., 1))) %>%
  gt::gt()
```

Table 5-1 Different imputation method results, excluding `Ozone_missing` and `imp_fixed` to fit the table

imp_mean	imp_fill	imp_median	imp_forest	imp_lm	imp_knn	imp_pmm	imp_cart	imp_lasso
42.1	8	31	22.8	35.4	16	45	13	48.6
42.1	32	31	12.8	−16.2	11	14	8	−5.0
42.1	32	31	24.8	1.7	32	4	34	2.2
42.1	37	31	36.6	51.6	40	108	20	37.2
42.1	37	31	27.5	40.7	32	63	21	35.8
42.1	37	31	20.0	4.2	18	23	19	25.8
42.1	37	31	47.7	56.8	32	64	49	66.9
42.1	37	31	65.4	62.7	32	82	47	67.2
42.1	37	31	28.7	34.9	39	16	32	44.6
42.1	29	31	63.7	75.6	77	84	73	81.9
42.1	39	31	68.9	73.8	89	84	63	38.2
42.1	39	31	75.0	77.4	71	84	97	70.9
42.1	23	31	33.2	44.0	35	71	36	34.6
42.1	23	31	32.2	49.5	35	37	59	48.3
42.1	13	31	40.4	56.0	21	59	59	72.8
42.1	13	31	59.3	65.1	16	49	48	47.1
42.1	13	31	54.8	57.3	16	63	16	45.6
42.1	13	31	49.1	60.0	64	37	16	42.8
42.1	13	31	23.7	46.9	20	45	37	63.0
42.1	13	31	30.6	52.5	20	20	65	55.7
42.1	13	31	17.5	31.6	20	44	9	3.8
42.1	13	31	33.9	43.7	20	45	65	29.2
42.1	13	31	17.2	23.7	13	9	22	28.3
42.1	13	31	47.2	63.6	23	39	73	30.7
42.1	32	31	41.1	43.9	32	32	39	49.4
42.1	85	31	28.9	51.5	29	40	20	28.2
42.1	27	31	66.7	56.1	35	61	44	63.0
42.1	16	31	41.3	55.0	59	71	44	24.2
42.1	16	31	37.9	53.0	61	49	28	48.2
42.1	110	31	89.0	71.3	89	79	89	81.2
42.1	110	31	45.4	46.3	65	59	44	53.0
42.1	65	31	23.8	30.5	23	21	22	9.9
42.1	9	31	22.2	31.1	31	14	21	40.3
42.1	73	31	76.8	74.6	84	108	61	78.1
42.1	30	31	19.3	25.4	20	18	12	13.3

Chapter 6
Reproducible Research

When you author a research paper, you expand the knowledge frontier. Good research doesn't simply reveal something new; it does so in a systematic and reproducible way. You wouldn't believe a study that is impossible to confirm, right?

Nonreproducible single occurrences are of no significance to science. [25]

A survey of scientists conducted by the journal *Nature* in 2016 [3] discovered that 70% of researchers were unable to replicate the experiments of other scientists, and more than 50% were unable to reproduce their own experiments.

This raises the question of how to present information so that others can confirm the results of a published study. We use the principles of reproducible research.

The Turing Way [10] defines **reproducible** research as a process where work can be independently redone using the original team's data and code. This concept is distinct from **replicability**, which involves conducting the same analysis on different datasets, and **robustness**, which refers to applying different analysis methods on the same dataset; researchers often claim their results are robust by referencing alternative model specifications. Finally, **generalizability** is considered the gold standard, achieved when different data and methods produce similar results.

Figure 6-1 Understanding Reproducible Research as Defined by the Turing Way

		Data	
		Same	Different
Analysis	Same	Reproducible	Replicable
	Different	Robust	Generalizable

In addition, we can distinguish different types of reproducibility [28]:

- **Computational** reproducibility focuses on the details of code, software, implementation, and operating system. For instance, using different language versions and operating systems may lead to different results.
- **Empirical** reproducibility emphasizes information on noncomputational scientific experiments. It encompasses aspects such as survey questions, sampling procedures, experiment execution, and many others.

N. Tkachenko, *Data Insight Foundations*, https://doi.org/10.1007/979-8-8688-0580-6_6

- **Statistical** reproducibility involves the intricacies of statistical tests and model parameters. How you specify your model, the standard errors you use, the packages you employ, and the parameters defined both explicitly and implicitly (package defaults) can play a critical role in determining your outcomes.

Replicability, while important, can be hindered by constraints such as time, resources, and opportunities. Therefore, it is often more feasible to focus on reproducibility—ensuring that analytical data and code will be available for others to confirm findings. Reproducibility is not just a goal but a standard, particularly for studies that are inherently difficult to replicate.

For minimal reproducibility you must provide the analytical data used in your analysis. Although access to both raw data and processing code is desirable, it is not always a prerequisite for reproducing an analysis. Crucially, you should also share the analytic code, including model specifications used to derive the results. It is equally important to provide comprehensive documentation[1] of both data and code to clarify the dataset and explain the code's purpose and functionality. Finally, this information should be made readily accessible through common distribution channels like GitHub, ensuring easy access and use by the broader research community.

6.1 Literate Programming

An important technology for building reproducible research is literate programming. It integrates text and code chunks, creating a seamless blend of human-readable explanations and machine-executable code. The code loads data, generates graphs, and runs models, while the text provides context and interprets the results. This approach enables researchers to produce documents that are both human- and machine-readable.

One of the earliest implementations of literate programming was Sweave, which combined LaTeX and R for documentation and programming. Since then, the field has evolved with the introduction of, for example, RMarkdown, Jupyter Notebooks, and Python Markdown. The latest development in this area is Quarto, which offers researchers a comprehensive solution to creating transparent, reproducible, and well-documented research outputs.

We learned about what reproducible research is; with the help of Figure 6-2, let's explore the different steps you need to take. As an example, we will apply the workflow to a hypothetical customer satisfaction survey analysis project. This project aims to assess customer satisfaction through a survey and produce a report communicating the findings:

1. **Collected Data**: You have gathered survey responses on customer satisfaction, preferences, and feedback. Make sure to retain the raw data for reference.
2. **Processing Code**: Develop separate code to clean the collected data, addressing inconsistencies, missing values, and irrelevant entries. Document assumptions and processes, avoiding opinionated methods that may influence results.
3. **Analytic Data**: The cleaned data serves as the basis for analysis, providing accurate insights. Share this cleaned data with stakeholders.
4. **Analysis Code**: Apply various analytical techniques to derive meaningful insights from the analytic data, uncovering trends, patterns, and correlations. Document this analysis code comprehensively.
5. **Computational Results**: Generate computational results highlighting key findings such as average satisfaction scores, common complaints, and customer segments.

[1] More on this in Chapter Document

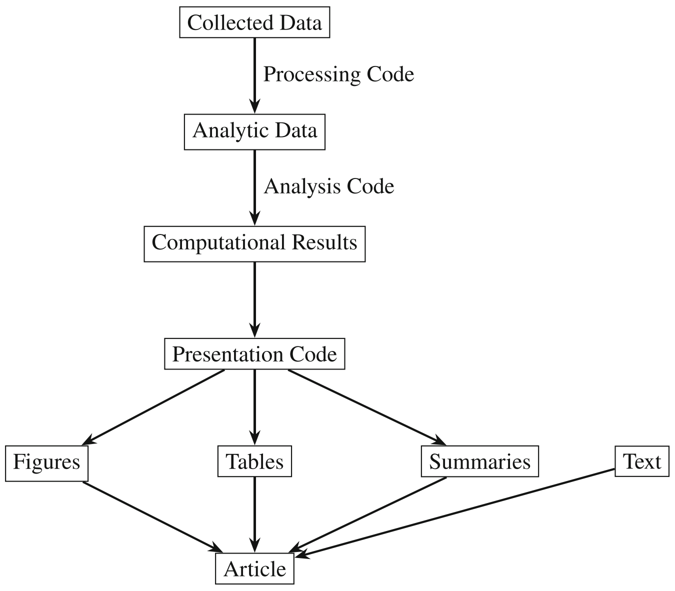


Figure 6-2 Literate Programming Flow

6. **Presentation Code**: Develop code to create visualizations, charts, and graphs that effectively communicate survey results to stakeholders and other analysts:
   - **Figures:** Visual representations illustrate trends and distributions, for instance, the evolution of Net Promoter Scores over time and customer segment distribution.
   - **Tables**: Use tables to showcase snippets of data, including summaries and visual information.
   - **Summaries**: Summarize significant findings from tables, such as regression summaries and five number summaries.[2]
7. **Text**: Incorporate descriptive text to provide context and explanations for visual and tabular components.
8. **Article**: Communicate your findings using the figures, tables, and summaries you produced with your code, and tell a story through the textual narrative.

Remember that while this is a simplified representation, real projects may involve more intricate steps and considerations.

6.2 Summary

I hope you now appreciate the importance of reproducible research in scientific studies and recognize the challenges associated with replicating experiments. Hopefully, you are also eager to adopt literate programming, which we will cover in the next part. But first, let's explore the concept of a reproducible environment, a foundational aspect of research reliability.

[2] Minimum, First Quartile (Q1), Median (Q2), Third Quartile (Q3), Maximum

Chapter 7
Reproducible Environment

Reproducible environments are crucial in modern software development, as they guarantee consistent outputs, regardless of the system or execution time. These environments are key to achieving reliability, scalability, and robustness, which are essential for smooth collaboration and efficient problem diagnosis and maintenance.

In today's complex software projects, burdened with intricate dependencies, the necessity for reproducibility is undeniable. Changes or updates in these dependencies can lead to code malfunctions or portability issues.

Understanding package versioning is helpful in this context. Version numbers, such as 1.2.3, indicate the scale and nature of updates. Major versions (1.x.x) introduce breaking changes, minor versions (x.2.x) add backward-compatible features, and patch versions (x.x.3) fix bugs while maintaining backward compatibility.

The term "dependency hell," often encountered in machine learning projects, refers to the challenge of managing conflicting, unreliable, or incompatible dependencies. Reproducible environments, which record all code dependencies, offer a robust solution to this issue.

Package updates that introduce breaking changes can significantly disrupt a codebase. This risk can be mitigated by diligently recording specific package versions and staying aware of release notes and news.

The R package `renv` (**r**eproducible **env**ironment) addresses these challenges by creating isolated, project-specific environments. It effectively manages R package dependencies, documenting the exact versions needed for your project and restoring them as required. This ensures the consistency of package versions across different R sessions and projects.

7.1 `renv`

`renv` greatly enhances the isolation, portability, and reproducibility of R projects. It ensures **isolation** by assigning a private library to each project, preventing conflicts between packages across different projects. The **portability** of `renv` facilitates the easy transfer of projects between various computers and platforms, streamlining package installation. Moreover, `renv` aids in **reproducibility** by recording exact package versions, which is crucial for maintaining consistency and reliability in different environments and enabling the precise replication of project setups.

N. Tkachenko, *Data Insight Foundations*, https://doi.org/10.1007/979-8-8688-0580-6_7

7.1.1 *Workflow*

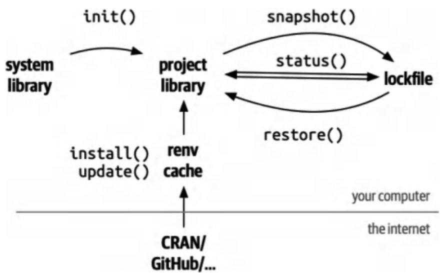


Figure 7.1 `renv` Workflow Image from Package Documentation

Integrating `renv` into an existing project or starting a new one is a straightforward process. Initiate the environment using `renv::init()`, which creates a lock file, "renv.lock," and detects your code dependencies. To capture the current state of dependencies, use `renv::snapshot()`.

Note 7.1 `renv` automatically excludes files listed in your `.gitignore`.

To install specific packages within an `renv`-managed project, use `renv::install()`, which installs packages and records them in your project's lock file. When updates are available for your packages, `renv::update()` helps you seamlessly apply these updates while also updating the lock file to reflect the new versions.

For a quick check on your project's dependency status, `renv::status()` provides an immediate overview of any discrepancies between your current project environment and the recorded state in "renv.lock."

For sharing code or running it in a new environment, `renv::restore()` reinstalls packages from the lock file. Implementing reproducible environments in your projects is highly beneficial. As you become more familiar with `renv`, visit its documentation to discover more about its capabilities and additional functionalities.

The value of setting up a reproducible environment becomes crystal clear after you've spent hours resolving bugs caused by package updates.

7.2 Computational Environments

In the realm of computational research, simply tracking package versions might not suffice. Factors like the operating system version and interactions between external packages can significantly affect the outcomes of analyses. For those aiming to elevate their reproducibility practices, exploring virtual machines and containerization is a worthwhile next step.

Virtual machines, essentially simulated computers, offer a versatile solution. They make it possible to create a "virtual" computer, choosing its operating system and other attributes, and operate it much like any regular application. Inside this virtual environment you'll find a desktop, file system, standard software libraries, and more, all of which behave as they would on a physical computer. Researchers can configure a virtual machine, conduct their research within this controlled environment, and then preserve its state—complete with files, settings, and outputs. This preserved state can then be distributed as a comprehensive, fully functional project. If you would like to learn more about virtual machines, go to VirtualBox.[1]

Containers share many advantages with virtual machines but are distinct in their approach and efficiency. While virtual machines replicate an entire operating system and its bundled software (regardless of whether its needed for a project), containers are more selective. They encapsulate only the specific software and files required for a project, making them considerably more lightweight and efficient than virtual machines. More information on containers is available at Docker.[2]

7.3 Summary

Navigating the intricacies of creating reproducible analyses can quickly become complex. It's important to gauge the level of technical expertise required and the likely technical proficiency of your collaborators. If there's no pressing need for advanced solutions like virtual machines, containers, or declarative operating systems, it might be more practical to avoid going too deep into these areas. Many collaborators may not be sufficiently tech-savvy to comfortably navigate these more complex solutions.

Begin by building a habit of using tools like `renv`. Once you and your team are comfortable with these foundational practices, you can gradually explore more advanced options like containers and virtual machines.

The next two chapters will present a brief introduction to the command line and then use it to learn version control with Git and GitHub.

[1] https://www.virtualbox.com/

[2] https://www.docker.com/

Chapter 8
Introduction to Command Line

The command-line interface (CLI), also known as the terminal or shell, is a tool for manipulating files, executing programs, and managing system settings. Unlike a graphical user interface (GUI), where users interact with visual elements, in a CLI, you interact with the computer by typing text commands. Despite its daunting appearance, understanding the command line can unlock almost magical powers.

In the command line, your workspace is a `current directory` or `working directory`. You navigate between directories using `cd` (change directory). For example, `cd Documents` moves you into the Documents directory within your current directory.

Directories in the command line have both relative and absolute paths. The relative path to a directory is its name. For instance, `cd Documents` uses the relative path `Documents`. Absolute paths specify the complete path from the root directory to the directory in question. For instance, `cd/home/username/Documents` uses the absolute path `/home/username/Documents`.

8.1 Learning Basic Commands

Commands form the core of CLI interactions, serving as instructions we type into the terminal to perform certain tasks, like file manipulation and directory navigation.

- `pwd`: “Print Working Directory” shows your current directory.
- `cd`: “Change Directory” allows you to navigate between directories. The use of `cd` with different arguments lets you navigate more efficiently:
  - `cd ..`: Takes you one directory up.
  - `cd /`: Takes you to the root directory.
  - `cd ~`: Takes you to your home directory.
  - `cd .`: Refers to the current directory.
- `ls`: “List” displays files and directories in the current directory.
- `touch [filename]`: Creates a new file.
- `cat [filename]`: Displays file content.
- `cp [source] [destination]`: “Copy” duplicates files or directories.
- `mv [source] [destination]`: “Move” renames or relocates files.
- `rm [filename]`: “Remove” deletes files.
- `rm -r [directory]`: “Remove recursively” deletes directories and their contents, including all files and subdirectories.

- `mkdir [directoryname]`: "Make Directory" creates a new directory.
- `rmdir [directoryname]`: "Remove Directory" deletes an empty directory.

You can chain directory names together with a forward slash `/` for navigating. This applies to both absolute and relative paths. For instance, `cd /home/username/projects/my_project` navigates to `my_project` by providing an absolute path. On the other hand, `cd projects/my_project` navigates to the same location using a relative path. If you want to move up to the parent directory and then into a sibling directory, you could use `cd ../sibling_directory`. This command means "go up one level, then down into `sibling_directory`."

These basic commands are the starting point for interacting with a CLI. As you gain familiarity, you'll learn more complex commands and combinations for powerful functionalities. Remember, don't hesitate to explore and experiment. The command line is a tool for you to harness to your advantage!

8.2 Getting Started with Nano

Nano is a simple and user-friendly text editor commonly used in Unix-like operating systems. It's an excellent choice for beginners due to its straightforward interface and commands.

To open a file in nano, type `nano` followed by the filename:

```
nano filename.txt
```

If the file doesn't exist, nano will create a new file with that name. Once you've opened a file, you can move around using the arrow keys. You can also use `Ctrl + A` to move to the beginning of the line, `Ctrl + E` to move to the end of the line, `Ctrl + Y` to move to the previous page, and `Ctrl + V` to move to the next page. Editing text in nano is as straightforward as typing. To cut the current line of text, use `Ctrl + K`. If you want to paste the cut text, use `Ctrl + U`. When you're done editing, you can save your changes using `Ctrl + O`, which will prompt nano to confirm or change the filename. To exit, use `Ctrl + X`. If you've made changes, nano will ask if you want to save them. If you're stuck or need to know more commands, you can access nano's help menu by pressing `Ctrl + G`.

We've covered basic commands for file and directory management, as well as text manipulation. I highly encourage you to use your terminal for daily activities. While it might be slow and frustrating at first, you will become more familiar and less reluctant to use it. As you grow more comfortable, explore various CLI tools such as `fzf` for fuzzy file finding, `grep` for text search and replacement, and `ffmpeg` for interacting with video and audio files. In the next chapter, we will delve into version control using Git and GitHub.

Chapter 9
Version Control with Git and Github

Picture yourself working tirelessly on a thesis or final paper. You've invested days in brainstorming ideas, writing the story, coding, reviewing, and adding visuals. After hours of refining, it's nearly perfect—until your trusty laptop decides to self-destruct. If only you'd saved your work online, you wouldn't be scrambling to find the latest emailed version or rewriting from a printed copy annotated by your advisor. Or even worse, starting from scratch!

Undeterred, you begin storing everything on Google Drive, thinking you're now safe. You tackle a big coding project, crafting an impressive model with a sleek frontend. But when you try to neaten your messy code, the entire application crashes and becomes a bug-infested nightmare. If only there was an easy way to revert to a previous version or work on new changes without disrupting the main codebase.

If any of this sounds familiar, I feel your pain. But if not, it's not too late to prevent such catastrophes with version control! Picture version control as the ultimate folder with labels like "final_version," "final_version2.0," and "final_final_version1.2.5." It tracks every change you make to documents, code, and data files, acting like a time machine that whisks you back to any point in your project's history. With it, you can recover any previous version of your work. Sounds like magic, right?

Here's how version control works:

1. **Snapshots**: Every time you save your work, version control takes a snapshot, preserving that particular version with a comment on what you did.
2. **Branching**: Want to try out a bold new idea, but afraid it might not work? No problem! With version control, you can create a separate "branch" and experiment without affecting the main version. If your idea works, you can "merge" the changes back into the main branch. If it doesn't, just discard the experimental branch and pretend it never happened.
3. **Collaboration**: Working with a team? Version control makes collaborating a breeze. All team members can work on their own individual parts of the project and later "merge" all the changes into a cohesive whole, replacing the email chains with "Final_Version_3_revised_edited_FINAL _review."
4. **Backup**: Version control also acts as a backup system, ensuring your work is safely stored in a remote location. So even if your computer decides to give up on you, your project remains secure and accessible.

N. Tkachenko, *Data Insight Foundations*, https://doi.org/10.1007/979-8-8688-0580-6_9

9.1 Git and GitHub

Git is a widely used version control system that efficiently manages and distributes projects of various sizes and complexities. Essential for developers, scientists, analysts, and writers, Git excels in tracking changes, fostering collaboration, and handling code. As a command-line interface (CLI), Git is accessed via the terminal.

GitHub, a web-based platform, enhances Git's functionality by offering a user-friendly interface for project management and collaboration. Additional features include issue tracking, code review, and project management tools. GitHub also serves as a developer's social network, facilitating the sharing of work, discovery of projects, and open-source contributions.

Note 9.1 The difference between Git and GitHub is the same as the difference between porn and PornHub.

Starting to use Git and GitHub can be a little overwhelming and confusing, especially when you only need a small subset of the functionality. My goal is to make you feel comfortable enough with the basics to be able to store, update, collaborate, and share your work on GitHub.

There have been many guides on installing Git on a machine and connecting it to RStudio. I will spare the world my version. I recommend using the guide from Hester [16] Happy Git and GitHub for the useR as a reference for your other adventures with Git. Once you go through the whole process of installation, you may want to connect it to RStudio, but that's optional. Because the commands can be hard to memorize and connect to the actions and the state of your repository, you should install a Git client, which will obtain an overview of the recent commit history, branches, and a GUI that simplifies Git operations. Try GitHub Destop or GitKraken.

Note 9.2 When inside a GitHub repository, press "." to start a web version of VSCode.

9.2 Basics

Now, let's quickly go over some basics of Git and GitHub. My aim is to introduce you to the key concepts and show you that getting started with version control is quite straightforward.

Repositories on GitHub are storage spaces for projects, where you can manage, organize, and collaborate on files while keeping track of changes using Git's version control system. To create a repository, start with either Github first or local-first. In other words, with Github first you start by initiating a repository on GitHub and then cloning it into your machine, and with local-first you write some code and connect your repository to the GitHub. I generally use local-first as I often start a project and only later decide whether I want it online. With GitHub desktop it is extremely easy to add a repo by just going to File → New Repository. There are also two types of repositories, Private and Public. You can change the status in your repository's settings.

Say you have done some work and want to save it. You want to commit! To do that, select all the files you want to add to the commit `git add file.md` or add all files with `git add .` and then `git commit -m "message"`; you're required to add a message to your commit, so you don't forget what changes you made. All of this can be done in your client and even RStudio with the click of a few buttons. Nonetheless, I advise you to get comfortable using the CLI. So you've committed the changes, but they're not online yet; they're just recorded, so you can load them if you mess up. You can have multiple commits, so you can keep on adding more and more changes before you are ready to share your work with the world. Once you are ready, you'll want to `git push`, and everything will appear on your GitHub!

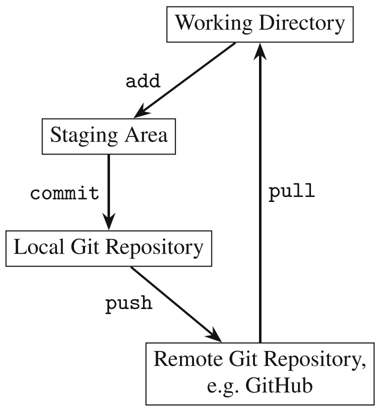


Figure 9.1 `git add, commit, push, pull`

What if you want to download the version from GitHub because, say, you want to work on a different machine? `git pull` is a command that combines retrieving updates from a remote repository and merging them into your local branch. It also allows you to synchronize your local branch with the latest changes from the remote, ensuring your project stays up to date and facilitating collaboration with other contributors.

Note 9.3 Using `add`, `commit`, `push`, and `pull` commands can prevent about 80% of common headaches!

But what if your partner updates the version on GitHub while you are working? Now you need to incorporate the changes into your work somehow. To do that, you first pull, and if there are no conflicts, meaning you didn't change the same line, it will automatically merge both versions. But what if you worked on the document and there were conflicts? You need to resolve them, just like in middle school.

Resolving a merge conflict in Git involves identifying the conflicting changes and manually deciding which changes to keep or modify within the file with the conflict. This will appear as follows inside the file:

```
#>   <<<<<<< HEAD
#>   This is the line in your current branch
#>   ========
#>   This is the line in the branch you're merging.
#>   >>>>>>> <branch-name>
```

To resolve the conflict, edit the file to keep the desired changes and remove the conflict markers. You might keep the change from the current branch or the branch you're merging, or you might to create a new line altogether:

```
#>   This is the new, resolved line.
```

Save the changes in the file and stage the file using `git add`. Commit the resolved merge conflict with `git commit -m "Resolved merge conflict in file"`.

Remember when my attempt to tidy up the code caused a crash (second story)? To prevent this, use branches in Git. Branches let you work on multiple features or fixes without affecting the main codebase. Once these are complete and tested, merge the branch back into the main branch.

Forking creates a personal copy of a repository under your account, while cloning creates a local copy on your computer. Forking is useful in open-source projects, allowing contributors to make changes without affecting the original project. After testing, submit a pull request to propose updates to the original repository.

A pull request proposes merging changes from one branch to another, typically from a forked repository to the original repository. It facilitates discussion, review, and collaboration, enabling project maintainers to approve, modify, or reject proposed changes.

GitHub issues help track bugs, requests, or discussions related to a project. They serve as a centralized forum for communication, task assignment, and progress monitoring, ensuring effective project management and timely resolution of concerns.

9.3 Guide to Using .gitignore

The `.gitignore` file instructs Git which files should not be tracked or ignored before you make a commit. It is especially useful in team-based projects where different developers have their own configurations, settings, and integrated development environments (IDEs), or when you have temporary files that are generated when a program runs. Let's walk through how to create and manage a `.gitignore` file.

To create a `.gitignore` file, navigate to your project's root directory and create a new file named `.gitignore`. This can be done directly via your code editor or in the terminal:

```
touch .gitignore
```

9.3.1 Specifying Files to Ignore

Note 9.4 Always include your logins, passwords, secrets, and other sensitive information in `.gitignore`! Replacing credentials and removing old ones from the repository after they have been published can be problematic.

Open the `.gitignore` file in your text editor and specify the files, directories, or file patterns to exclude. Each new line should contain a new rule.

- To ignore a directory, simply add the directory name, e.g., `node_modules/`
- To ignore a file, add the full file name, e.g., `debug.log`
- To ignore a file type, use `*.` followed by the file extension, e.g., `*.log`
- Comments can be added by starting a line with a `"#"`.

Here's an example `.gitignore` file:

```
# Ignore node_modules directory
node_modules/

# Ignore all .log files
*.log

# Ignore the secret.json file
secret.json
```

For files you'd like to ignore globally, across all projects, you can create a global `.gitignore` file:

```
git config --global core.excludesfile "~/.gitignore"
```

Now you can define all rules in the file, and these will be applied across all repositories.

9.3.2 .gitignore in Other Programs

While `.gitignore` was originated from and is most commonly associated with Git, the concept of defining files and directories to ignore has been adopted by other software tools. These tools include various IDEs, text editors, and even some operating systems. Many of these tools support using `.gitignore` as a way to exclude files and directories. It's always best to check the documentation of your specific tools to see how they handle file ignoring.

For most programming languages and popular software tools, standard `.gitignore` files have been created and maintained by the open-source community. GitHub maintains a repository of these files. You can use them as a starting point for your projects.

9.4 Summary

Note 9.5 Do not delete the `.git/` folder, as it contains your entire history!

This chapter introduced the essentials of version control with Git and GitHub, focusing on tracking changes, managing projects, and collaboration. We covered basic Git commands, the use of repositories, branching, merge conflicts, and the importance of the `.gitignore` file. I hope you are convinced of the importance of version control and see that it is not overly complicated. I encourage you to use the resources referenced or find your own to learn more! In the next chapter, we will explore how to style your code.

Chapter 10
Style and Lint Your Code

While immersing yourself in coding, you might neglect aesthetics like proper indentation or line breaks, which could make your code less readable and more error-prone. It's crucial to adopt certain coding practices to enhance readability and minimize errors. To help with this, we will explore R code conventions and introduce `styler` and `lintr`, packages that improve code aesthetics and detect potential errors.

A good coding style, much like correct punctuation, significantly enhances readability. It's important to remember that while some guidelines improve usability, others may be subjective. However, their true value lies in fostering consistency, which simplifies the coding process.

You may wonder, if you're comfortable with your coding style, why use these tools? While there are subjective benefits, the primary reason is that they enhance your code's accessibility and eliminate irrelevant stylistic changes from git commits.

Note 10.1 Mac and Windows represent tabs with different symbols. If your code contains tabs, it may fail on a different OS. To ensure compatibility, use spaces (your editor can convert tabs into spaces).

10.1 Tidyverse Style Guide

We will now go over some basic guidelines that you can begin applying immediately. These conventions are sourced from the tidyverse style guide, which can be read in no more than 20 minutes. I highly recommend going through it: https://style.tidyverse.org.

10.1.1 White Spaces and Indentation

Adhere to a two-space (or four-space) indentation to illustrate the structure and hierarchy in your code. Function contents should also follow this two-space rule. For functions with pipes, start a new line for each pipe and indent them accordingly, ensuring clarity and readability.

N. Tkachenko, *Data Insight Foundations*, https://doi.org/10.1007/979-8-8688-0580-6_10

Good

```
for (i in 1:10) {
  print(i)
}

iris %>%
  group_by(Species) %>%
  summarize_if(is.numeric, mean) %>%
  ungroup() %>%
  gather(measure, value, -Species) %>%
  arrange(value)
```

Bad

```
for (i in 1:10){
print(i)}

iris %>% group_by(Species) %>% summarize_all(mean) %>%
ungroup %>% gather(measure, value, -Species) %>%
arrange(value)
```

10.1.2 Naming Conventions

R has a unique operation <- used for assigning variables. You should always use <- as an assignment operator over =. One of the hardest things in all of programming is naming your variables. In R, we use `snake_case` over `camelCase` or anything else. For variables, use nouns and don't hesitate to make the names a bit longer. Aim for the perfect balance where your code can be readable yet concise. For functions, always use verbs, since functions perform actions.

Good

```
average_height <- 1.70
add_row()
permute()
```

Bad

```
averageHeight = 1.70
AverageHeight = 1.70
row_adder()
permutation()
```

10.1.3 Braces

Use braces `{}` in your R code strategically to boost readability, grouping together commands that operate in tandem.

For if-else statements, place the opening brace `{` on the same line as the condition, and position the closing brace `}` on a new line. Ensure that the `else` statement shares a line with the closing brace of the preceding `if` section. This configuration facilitates easy identification of distinct code blocks.

Good

```
if (condition) {
  action1()
} else {
  action2()
}
```

Bad

```
if (condition)
{
action1()
}
else
{
action2()
}
```

10.1.4 Comments

Commenting on your code can save you time and prevent confusion later. Even though your code should aim to be self-explanatory, comments provide invaluable context about the reasoning behind your code and help document key findings and decisions in data analysis. Think about your future self coming back and feeling confused.

Use # and a space to write a comment. Stick to sentence case and only use a full stop at the end if your comment spans multiple sentences. Begin each line of the comment with # and a space.

If you find you need many comments to explain your code, consider refining it for clarity. Code that requires excessive comments may be better suited to a verbose platform, such as Quarto.

Good

```
# Calculate average height - this metric is used for normalization
```

Bad

```
# We are finding the average height

#We will sum all of the heights and divide by the number of heights
```

10.1.5 Long Functions

Aim to limit your code to no more than 80 characters per line. This fits well on a standard printed page using a readable font size. If you find your code exceeding this limit, take it as a hint to encapsulate some of your code in a separate function.

For function calls that extend beyond a single line, allocate separate lines for the function name, each of its arguments, and the closing parenthesis. This practice enhances readability and future edits.

Good

```r
long_function_name <- function(argument1,
                               argument2,
                               argument3,
                               argument4) {
  # function body
}
```

Bad

```r
long_function_name <- function(argument1, argument2, argument3, argument4) {
  # function body
}
```

Fortunately, you don't need to obsess over formatting, as there are tools available to automate this process. The R packages `styler` and `lintr` can automatically enforce these guidelines, making them beneficial for learning these conventions.

10.2 Formatter

A formatter is a tool designed to process code and ensure consistent formatting for elements such as semicolons, whitespaces, and other details that do not affect the code's functionality.

Like many other languages, R also has its own formatter! `styler`, an R package, formats your code in line with the tidyverse style guide. It's straightforward to install using `install.packages("styler")`. After installation, you'll find a new function in the `Addins` menu at the top of your RStudio window. I recommend you set a keyboard shortcut, such as "Command + Shift + a," for the "Style Selection" function for easy access.

Before

```r
iris %>% ggplot(aes(Sepal.Length, Sepal.Width)) + geom_point()
```

After

```r
iris %>%
  ggplot(aes(Sepal.Length, Sepal.Width)) +
  geom_point()
```

10.3 Linter

A linter is a tool that performs static code analysis of your code to identify potential errors, enforce coding standards, and improve code quality. It helps developers by automatically checking for common coding mistakes, such as typos or incorrect use of language features, and by ensuring the code adheres to best practices.

R has its own linter, the `lintr` package, which is a static code analysis tool that flags style, coding, and syntax issues. Here's a brief explanation of these three components:

1. **Style Issues**: `lintr` checks if the code adheres to the style guidelines, such as those defined in the tidyverse style guide. It checks for correct indentation, line lengths, usage of spaces around operators, and more.
2. **Coding Issues**: `lintr` looks for potential coding errors and suboptimal code, such as use of undefined variables, use of = instead of `<-` for assignments, and the presence of TODO comments, for example.
3. **Syntax Issues**: `lintr` checks for any syntax errors in the code, like missing parentheses or incorrect usage of language keywords.

Static code analysis with `lintr` can be performed automatically within certain development environments, such as RStudio, or it can be invoked manually from the R console with `lintr::lint(filename = "path/bad.R")`. It greatly helps in improving code quality, readability, and maintainability.

Formatters and linters might appear similar, but they have different functions. A formatter is faster and tidies up your code for better readability, while a linter is a bit slower but essential for spotting and avoiding errors in your code. So a formatter makes your code look neat, and a linter keeps it bug-free.

10.4 Summary

This chapter emphasized the importance of good coding practices in R, guided by the tidyverse style guide. It also introduced the `styler` and `lintr` packages, which help in formatting code for better readability and detecting potential errors. In the next chapter, you will learn how to modularize your code.

Chapter 11
Modular Code

Working within a single file is a smooth ride until you hit a couple of hundred lines. Then, searching for bugs, making edits, and simply understanding your location in the file become rather problematic. While adding comments and creating a table of contents or using search to navigate the file can help, these are just temporary solutions. They become completely ineffective when sharing the code with others. More problems arise when you have to download multiple libraries, leading to name-space conflicts, or when all the sensible names for variables and functions are taken and you end up with something like `very_important_variable_2_winz`.

11.1 Reuse Functions

One of the easiest steps to improve your code and make it easier to maintain and understand is to put all the repeating actions into functions. While it might be tempting to break your code down into small functions for each tiny task, such as creating separate functions for mean, standard deviation, t-score, and p-value calculations, the rule of thumb should be to focus on the goal of a function.

If you find that you need to reuse a portion of the larger function, it's a good practice to go ahead and split it into a separate function. By abstracting your code into functions, you gain the benefits of code reusability and improved modularity. Additionally, abstracting code into functions helps break down deeply nested code into more manageable pieces, making it easier to read and understand.

The goal is to strike a balance between creating functions that serve a specific purpose and avoiding an excessive number of small functions. The purpose-driven approach ensures that each function communicates its intended goal effectively, leading to improved code readability. By abstracting your code and organizing it into purposeful functions, you'll achieve code that is easier to comprehend and maintain. If you are doing something more then a few times, it likely belongs in a function.

We want to calculate the mean and standard deviation on our a multiple datasets. We could run mean and sd on each separately, but let's define a single function.

There are benefits to encapsulating code in functions for reusability and abstraction, for example, creating a function to calculate the mean and standard deviation of a numeric vector.

```r
calculate_stats <- function(data) {
  mean_value <- mean(data)
  sd_value <- sd(data)
  return(list(mean = mean_value, sd = sd_value))
}
```

N. Tkachenko, *Data Insight Foundations*, https://doi.org/10.1007/979-8-8688-0580-6_11

11.2 Split It

Once your file exceeds a hundred lines, it makes sense to separate it into multiple files based on the function. This will drastically improve your code's readability. You can use `source()` to run your scripts. For instance, you first source a `load_data` script that saves the loaded data and then run the `data_preprocessing` and `data_visualization` scripts. Additionally, you can load the scripts into the same local environment with `local = TRUE`.

data_loading.r

```r
load_data <- function(file_path) {
  # Code to load data from a file
}
```

data_processing.r

```r
preprocess_data <- function(data) {
  # Code to preprocess data
}
```

data_visualization.r

```r
visualize_data <- function(data) {
  # Code to visualize data
}
```

main.r

```r
source(data_loading.r, local = TRUE)
source(data_preprocessing.r, local = TRUE)
source(data_visualization.r, local = TRUE)
```

Now let's look at a simple application of `source()` when dealing with multiple files. The example shows the process through snippets encompassed in distinct files: "name.csv" holds a single entry, "Dima"; "load_data.r" loads the data from the csv; "add_title.r" defines the `add_title` function, combining a name and a title; "say_hi.r" defines `say_hi`, appending "Hi" to a name; and "main.r" utilizes the source command to load these modules and showcases their utilization in a sequence of chained operations. Ultimately, the code yields the output "Hi Mr. Dima," highlighting how `source` integrates functions across multiple files.

name.csv

```
Dima # CSV with a single entry
```

load_data.r

```r
# Read `name.csv` and get the name
my_name <- read.csv("name.csv", header = FALSE)
```

add_title.r

```r
source("load_data.r")
# `add_title`, concatenates a title and name
add_title <- function(name, title) {
  return(paste(title, name))
}
```

say_hi.r

```r
# `say_hi`, which concatenates a name and "Hi"
say_hi <- function(name) {
  return(paste("Hi", name))
}
```

main.r

```r
# "main.r," sources the `say_hi` and `add_title`
source("say_hi.r")
source("add_title.r")

title <- "Mr."

add_title(myName, title) |>
  say_hi() |>
  print()
```

```
#>  [1] "Hi Mr. Dima"
```

11.3 box It

The “box” system in R allows you to convert regular R files into reusable modules, enabling simplified code sharing without packaging. You can export functions by placing the `#' @export` directive before the function’s name. For instance:

say_hi.r

```r
#' @export
say_hi <- function(name) {
  return(paste("Hi", name))
}
```

The `box::use()` function serves as a universal import declaration, superseding traditional library or require functions in base R. It provides more explicit and flexible loading of packages or modules, reducing errors. For example, instead of loading an entire library with `library(ggplot2)`, you can use `box::use(ggplot2[...])`.

allows specific function loading, which avoids name clashes and improves the traceability of a function's origin. It allows for aliasing to assign a different name to a package or its functions when imported:

load_data.r

```r
# Load `read.csv` function from `utils` and rename it to `read`
box::use(
  utils[read = read.csv]
)

#' @export
my_name <- read("name.csv", header = FALSE)
```

add_title.r

```r
#' @export
add_title <- function(name, title) {
  return(paste(title, name))
}
```

These modules can be stored in a central location or within individual projects and can be imported and used with the `box::use` function, like:

main.r

```r
box::use( # Load the files
  ./say_hi,
  ./add_title,
  ./load_data[my_name]
)

box::use(
  stringr[str_split_1] # Load function from `stringr` package
)

title <- "Mr."

# Access the function and the variable using `module$variable`
add_title$add_title(my_name, title) |>
  say_hi$say_hi() |>
  # `str_split_1` splits a string by space into a character vector
  str_split_1(pattern = " ")
```

```
#>   [1] "Hi"   "Mr."  "Dima" "!"
```

Modules in `box` share similarities with R packages but are simpler to create and use and offer features such as hierarchical nesting.

`box` is an amazing package that will greatly aid you on larger projects. It is also a part of the `rhino` shiny framework. There is a lot more to it, so make sure to read the official documentation.

11.4 Summary

In this chapter, we explored managing complex R scripts by modularizing them into functions and separate files. We discussed using functions for repetitive tasks, dividing lengthy scripts into smaller, targeted files, and utilizing the `box` system for creating reusable modules. These strategies enhance code readability and maintainability and prevent namespace conflicts, making it valuable for managing larger R projects. In the next section, we will cover the research process, beginning with a literature review.

Chapter 12
Literature Review

Every great adventure starts with a literature review. Literature review is an indispensable step in any research, identifying knowledge gaps and shaping your research question. This section introduces powerful tools to supercharge your literature review process, guiding you from initial topic exploration to organizing your final notes.

12.1 Search

Begin your literature hunt by seeking advice from a professor or field expert, who will point you in the right direction and even hold your hand during the early stages, saving you time and effort in the long run. An excellent starting point is to delve into review articles, as they offer comprehensive overviews of recent developments, summarize pivotal findings, and pinpoint existing gaps in the field. Also, consider picking up a primer article if you are new to the topic and reading a textbook for foundational knowledge.

Proceed by exploring peer-reviewed databases accessible through your university's subscription, such as *Scopus*[1] and *Web of Science*.[2] These databases have dedicated teams of peer reviewers to ensure the high quality of the articles they host. Additionally, do not overlook your university library, as it can provide access to expensive databases and reading materials at no cost.

Note 12.1 Beware of predatory journals! Although they may seem legitimate, these journals are known to publish any material for a fee. If you're uncertain, review the journal's website and consult *Beall's List*,[3] a repository of predatory journals.

Once you've tapped into these resources, expand your search with *Google Scholar*[4] for its extensive collection of articles. However, keep in mind it is on you to verify the quality of work. For book resources, consider visiting the author's personal website, which might offer free chapters or even a complete book.

1 https://www.scopus.com/home.uri

2 https://mjl.clarivate.com/search-results

3 https://beallslist.net/

4 https://scholar.google.com/

N. Tkachenko, *Data Insight Foundations*, https://doi.org/10.1007/979-8-8688-0580-6_12

A modern way to jump-start your review is to use tools such as *Elicit*[5] to get the first few papers and their summaries with topic overviews. Tools such as *Research Rabbit*[6] and *Lit Maps*[7] build networks of papers you have and allow you to find papers that are close in the network, helping to find those lesser-known but relevant articles. Research Rabbit has saved me many times from hitting a wall in finding new papers.

Lastly, for optimal organization of your research, look into the *Arc browser*.[8] This tool facilitates the creation of structured spaces for each project, allowing for efficient navigation through folders containing all your links and notes.

12.2 Reference Management

As you collect papers, where do you store them? In a reference manager! Reference managers are essential in the realm of academic research for organizing and managing your literature sources effectively. Among these, *Zotero*[9] stands out as the most comprehensive tool. It's a free, open-source, cross-platform application equipped with a wide range of extensions. Zotero allows you to add papers using DOI, ISBN, links, and files, read PDFs inside the app as well as maintain a shared online repository in the cloud, add tags, link related references, insert notes, and annotate, with the added convenience of exporting all this information into your preferred note-taking app. Its compatibility with numerous writing programs simplifies the citation process across different environments. Enhance your experience using the *Zotero web extension*[10] for easy reference collection from the web. For even greater functionality, install the *BetterBibTeX plugin*,[11] which improves Zotero by generating precise citation keys and facilitating the automatic export of your library to BibTeX. There is an extension for anything you might possibly need. I also highly recommend following a guide when setting up Zotero for the first time.

12.3 Reading

With your literature collection now ready, it's essential to sift through the papers, prioritizing them based on their quality, impact, and relevance. Perhaps counterintuitively, academic papers need not be read in the order in which they are presented. Begin by quickly reviewing each paper's title, abstract, conclusions, and figures to gauge its significance. This initial screening will help the most pertinent papers rise to the top of your virtual stack, and don't be afraid to drop papers that don't fit your needs. This approach allows you to grasp the core findings and implications before exploring the more detailed technical aspects. It can be hard to maintain focus and momentum, especially with lengthy or complex texts, so use *Speechify*,[12] a text-to-speech application that vocalizes articles. This tool not only helps maintain your reading pace but also results in better comprehension. It's a common

[5] https://elicit.org/

[6] https://www.researchrabbit.ai/

[7] https://app.litmaps.com/

[8] https://arc.net/

[9] https://www.zotero.org/

[10] https://www.zotero.org/download/connectors

[11] https://retorque.re/zotero-better-bibtex/

[12] https://speechify.com/

strategy to keep both Speechify and Zotero open simultaneously, so you can listen to an article while taking structured notes in Zotero.

To further enhance the process, try out *DocAnalyzer.ai*,[13] *Zotero QA* on Hugging Face,[14] or *ARIA*,[15] which was developed by the same author for asking questions about a Zotero library. Additionally, consider loading PDFs into ChatGPT to ask questions and browse collections of custom GPTs.

12.4 Note Taking

Effective note taking is a vital part of the literature review process, and its approach should align with the scope of your project. When taking notes, focus on making them concise, meaning they should be straight to the point and purposeful. You want to turn them into a coherent story, and don't forget to include references to the sources. When it comes to the act itself, for short to medium-length papers, a simple method like jotting down notes in a Google document may suffice. These notes can later be refined and structured into a cohesive essay. However, for more extensive research endeavors, a systematic approach to note taking and organization is crucial.

Consider using tools like *Obsidian*[16] for managing your notes. Obsidian is excellent for creating interconnected idea networks using Markdown files. It allows for a more organized and interconnected understanding of your research. Alternatively, *Notion*[17] offers a dynamic platform for organizing notes, tasks, and databases, catering to a variety of organizational needs. These platforms can be further enhanced by integrating with Zotero's export capabilities, allowing for a seamless transfer of references and notes to your chosen note-taking application.

In addition to these tools, AI solutions like *ChatGPT*[18] can significantly augment your literature review process. ChatGPT excels in summarizing text, enhancing writing styles, and converting text into structured Markdown tables. It's particularly useful for transforming stream-of-consciousness thoughts into polished academic language. It can also explain visual elements like tables, graphics, and diagrams. Furthermore, it assists in planning academic papers and providing critical feedback.

By leveraging these cutting-edge tools, you'll avoid slowing down your work and ensure a more efficient review process. However, do not overly depend on them to do your work; always proofread, and edit the output. They are just tools, not a replacement for your expertise.

12.5 Summary

Here is my typical workflow:

1. Talk to an expert.
2. Read review articles, primer articles, and textbooks.
3. Use Elicit to get initial papers.
4. Search using Google Scholar.
5. Put everything in Zotero.

[13] https://docanalyzer.ai/

[14] https://huggingface.co/spaces/lifan0127/zotero-qa

[15] https://github.com/lifan0127/ai-research-assistant

[16] https://obsidian.md/

[17] https://www.notion.so/

[18] https://openai.com/blog/chatgpt

6. Expand further with Research Rabbit.
7. Skim and filter the articles.
8. Read with Speechify.
9. Take notes in a Markdown file, citing works stored in Zotero.
10. Turn notes into a first draft using ChatGPT.
11. Write in Quarto and iterate.
12. Get feedback using Trackdown.
13. Export into LaTeX and typeset in Overleaf.[19]

In this chapter, we covered several useful resources that will drastically simplify your literature review process and help you organize all this knowledge. In the next chapter, we will cover technologies and tools for writing your paper!

[19] Covered in the next chapter

Chapter 13
Write

Following your comprehensive review of the existing literature, you should have developed a concept for your writing topic. This chapter delves into various technologies you'll likely encounter in your search for an effective text editor and provides an overview of the basic syntax for Quarto.

13.1 WYSIWYG

WYSIWYG (What You See Is What You Get) editors are programs that allow you to make and edit content visually without having to know how to code. This means that you can see how your content will look as you're creating it. They're very helpful for people who don't have experience with coding or markup languages. Examples of WYSIWYG editors that you may already be familiar with include Google Docs and Microsoft Word. Learning how to navigate these widely used tools will teach you valuable skills such as formatting and writing. Plus, after using them for a while, you'll start to appreciate the simplicity and efficiency of markup languages.

I strongly recommend that you develop expertise in these tools, as the skills you acquire are highly transferable to other similar editors. Additionally, it's a valuable investment as your colleagues are likely to use them as well. To get started, I suggest checking out the Microsoft Office Specialist (MOS) Certification Series; for instance, LinkedIn Learning has great options. These tutorials are designed for those preparing to take MOS exams and cover a broad range of functionality in Microsoft Word, Excel, and PowerPoint. Even if you don't plan on taking the exams, I highly encourage you to watch the tutorials. They're a great resource for building your skills and improving your proficiency with these essential tools!

13.2 Markup Languages

Markup languages are sets of codes that provide structure and formatting to documents, such as web pages, eBooks, and scientific papers. Unlike WYSIWYG editors, which allow users to create content visually, markup languages require the use of specific tags or codes to indicate how the content should be displayed. These tags define the document structure, formatting, and other attributes. Some of the most commonly used markup languages include HTML, LaTeX, and Markdown. By learning a markup language, you will have greater control over the appearance and functionality of your digital documents. Additionally, markup languages can help you focus more on writing, allowing templates

N. Tkachenko, *Data Insight Foundations*, https://doi.org/10.1007/979-8-8688-0580-6_13

to handle all the formatting. I promise after you switch, there is no going back! Your assignments and web pages will look beautiful every time.

13.2.1 HTML

The term “markup language” often brings to mind HTML (Hypertext Markup Language), a ubiquitous tool in web content creation. HTML employs tags to specify content display in web browsers. Tags such as `<br>` for line breaks and `<img>` for images are fundamental in HTML. Knowing HTML is a key skill in web development, providing the foundation for customizing web pages and applications. HTML is also frequently used for text manipulation in graphs.

13.2.2 LaTeX

Have you ever wondered why academic papers look so beautiful or why every textbook looks the same? They are all written in LaTeX, a typesetting system used for academic and scientific publishing. It is a markup language that enables precise formatting of complex technical documents. It is also famous for its beautiful mathematical equations. With LaTeX, users can create professional-looking documents with a high degree of customization and flexibility. It is easy to pick up but hard to master. Introductory tutorials online and ChatGPT will assist you with most LaTeX questions. In case you have an equation you want to transfer into LaTeX, *MathPix*[1] is a handy tool that transforms pictures or screenshots into code.

A commonly used LaTeX editor is *OverLeaf*,[2] an online platform that simplifies the writing process and offers extensive guides for any questions you might have.

13.2.3 Markdown

Markdown stands out for its attention to detail and user-friendly design. It’s a lightweight markup language created for generating formatted text in a plain-text editor. Its appeal lies in its simplicity and readability in source code form, in stark contrast to the complexity of traditional markup languages like HTML.

Popular among developers, writers, and bloggers, Markdown is celebrated for its simplicity and adaptability. It supports a range of elements, including links, images, and other multimedia features. Because it is widely supported, Markdown is compatible with numerous tools and platforms such as text editors, note-taking apps, and content management systems. Learning Markdown will enable you to produce well-structured, readable content efficiently, without relying on complex formatting tools or specialized software. In fact, this book was originally written entirely in this format.

For Markdown writing, I recommend MacDown for Mac, Markdown Pad for Windows, StackEdit for online use, or Typora for those who prefer a premium option.

[1] https://mathpix.com/image-to-latex

[2] https://www.overleaf.com/

13.2.4 *Yet Another Markup Language (YAML)*

YAML (Yet Another Markup Language) is a human-readable data serialization language often used for configuration files and data exchange. It uses indentation, key—value pairs, and maps and supports different data types. Maps are used to organize data as key—value pairs with indentation to indicate hierarchy. The key represents a unique identifier or name, and the value represents associated data or content. YAML is easy to read and write and is popular in web development and software configuration.

```
% Key value pairs
title: "Dead Souls"
author: "Nikolai Gogol"
date: 03/09/1842

% Map
chapters:
  - index.qmd
  - chapter1.qmd
```

13.2.5 *Pandoc*

Pandoc (literally meaning “all documents”) converts files from one markup format into another. It serves as a versatile tool for converting files between various markup formats, including HTML, Markdown, LaTeX, PDF, and Microsoft Word. This utility allows for swift and seamless document conversions across different languages without manual editing or reformatting.

Pandoc’s diverse applications range from converting Markdown files to HTML for web pages to transforming LaTeX documents into PDFs for printing. It also excels in merging multiple documents into a single file and extracting specific content formats. Its high customizability offers extensive control over the output’s format and appearance, making pandoc a staple in academic and scientific publishing, web development, and documentation. The automation of format conversions with pandoc significantly reduces time and effort.

Though a command-line tool, pandoc integrates well with certain Markdown editors, enhancing Markdown’s functionality with features like tables, footnotes, citations, and more. For instance, this book was composed in Markdown and then transformed into HTML and PDF formats using pandoc.

13.3 Quarto

At this point, you might be wondering why I had you read about these technologies. Because the combination of Markdown, YAML, and pandoc provides a powerful and flexible set for the creation of documents. Those familiar with Sweave, Rmarkdown, or Jupyter Notebooks will find Quarto to be a similar tool. Quarto builds upon the success of Rmarkdown and Jupyter Notebooks, combining the best features of Markdown with new functionality and eliminating the need for additional packages. It provides attractive default formatting options and easy customization. If you have experience writing in Markdown, Quarto will be a breeze to use for any of the following:

- Creating reproducible documents and reports.
- Generating dynamic content with Python, R, Julia, and Observable.
- Producing professional-grade content, such as articles, reports, presentations, websites, blogs, and books, in various formats including HTML, PDF, MS Word, ePub, and more.
- Authoring with scientific Markdown, featuring equations, citations, cross-references, figure panels, callouts, advanced layouts, and more.
- Developing interactive tutorials and notebooks.

13.3.1 Your First Document

To use Quarto, you don't need any special software; if you'd like, you can even use a text editor to create your .qmd files and command line to render the document. However, working in an integrated development environment (IDE) will make your life much easier. I prefer to use RStudio; after all, Quarto is a product of Posit (formerly RStudio), but Visual Studio Code also works well.

From within Rstudio you can install Quarto as you install any other package. Additionally, you should install TinyTeX, a minimal set of packages required to compile LaTeX documents. You can get both by running the following command in R:

```
install.packages(c("quarto", "tinytex"))
```

Write Create a Quarto project. Go to top panel → File → New Project → Directory → Quarto Project → Create File → Quarto Document → Write the Title of your document → Select the format you want (HTML is the default) → Create

An (optional) YAML header is demarcated by three dashes (`---`) on either end. You can learn more about YAML options on Quarto's website. There are plenty of useful options for display.

```
---
title: "Title"
author: Nikita Tkachenko
date: today
format: pdf
---
```

If you prefer the WYSIWYG style, you can switch to the visual editor in the top left corner. Additionally, if you tick "Render to Save," a new document preview will be updated after each save. Now it is time to get writing!

Note 13.1 To start, the H1 heading can be defined in yaml `title: "Title"` and as `# Title`.

Markdown Syntax	Output
# Header 1	**Chapter**
## Header 2	**Section**
### Header 3	**Subsection**
#### Header 4	**Subsubsection**
##### Header 5	Paragraph
###### Header 6	Subparagraph
italics	*italics*
bold	**bold**
superscript^2^	superscript^2
<https://nber.org>	https://nber.org
[NBER](https://nber.org)	NBER
![Caption](golden_gate.pdf)	Caption
unordered list + item - sub-item - sub-sub-item	unordered list • item – sub-item · sub-sub-item
ordered list 1. item I. sub-item i. sub-sub-item	ordered list 1. item I. sub-item i. sub-sub-item
inline math: $E = mc^{2}$	inline math: $E = mc^2$
display math: $$E = mc^{2}$$	display math: $E = mc^2$

As for writing regular prose, there is nothing special; you just write it as usual. If you want to add a line break, add an empty line between your paragraphs; otherwise it will continue as the same text.

Note 13.2 Use Visual Mode to conveniently access syntax options!

Use `` `...` `` to add inline code: `print(hi, friend)`.
Use ``` `` ` ``` to delimit blocks of source code:

~~~~
```
print(hi, friend)
```
~~~~

Making tables in Markdown is not complicated. The most frequently used table is the pipe table. It allows you to see alignment and captions with `:`. Tables can get complicated pretty quickly; if you ever get stuck, refer to Quarto's table documentation.

```
| Default | Left | Middle | Right   |
|---------|:-----|:------:|--------:|
| Hola    | Pitt | 3.141  | Nile    |
| Bonjour | Li   | 2.718  | Amazon  |
| Salut   | Roth | 4.123  | Yangtze |

: Table Demonstration
```

Table 13-1 Table Demonstration

Default	Left	Middle	Right
Hola	Pitt	3.141	Nile
Bonjour	Li	2.718	Amazon
Salut	Roth	4.123	Yangtze

If you ever feel lost or struggle with formatting, consider using the visual editor. It provides a familiar interface and is particularly useful for creating and previewing tables. To adjust options, for example, the number of list options, simply click on the circle icon with an ellipsis next to it, and a selection menu will appear. In addition to this, the visual editor offers extensive customization options for other elements, such as images and tables.

When you create a Quarto project, you might have noticed that `_quarto.yml` is also created. This is your project-wide specification document. While everything you specify in the YAML header of your document will be applied only to that particular page, all the settings you set up in `_quarto.yml` will be applied to all files. Additionally, this is where you can specify things such as the type of project, for instance, book/website/article. You might also have noticed that there is a `publish` button at the top of your screen. Click on it, and it will walk you through publishing your Quarto document on Quarto Pub, which is a simple and free way to share your work. You can learn more at Quarto's website (https://quarto.org/).

13.4 Summary

In this chapter, we explored various text-editing technologies, beginning with user-friendly WYSIWYG editors such as Google Docs and Microsoft Word, then progressing to markup languages like HTML, LaTeX, and Markdown. Additionally, the chapter introduced YAML and pandoc for format conversion, as well as Quarto, which integrates these technologies to provide a robust platform for creating diverse and professional documents. In the next chapter, we will dig deeper into Quarto's layout capabilities.

Chapter 14
Layout and References

Now we look at a few basic layout features of Quarto as well as how to reference literature with Zotero.

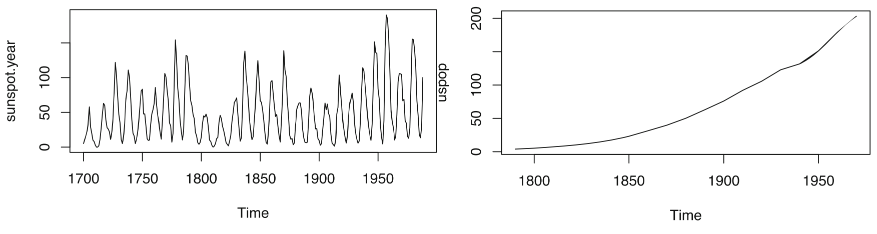
14.1 Knitr

`knitr` is a package that bridges the gap between code execution and the generation of PDF/HTML documents. It executes your code and integrates its output into a Markdown file, which pandoc then converts.

With `knitr`, you can set cell options to manage code blocks' behavior and output. These options are placed at the start of a block within comments. For instance:

~~~
```{r}
#| label: fig-plots
#| fig-cap: Plots
#| fig-subcap:
#|   - "Sunspot"
#|   - "US Population"
#| layout-ncol: 2

# sunspot.year and uspop are built-in datasets in R
plot(sunspot.year)
plot(uspop)
```
~~~



(a) Sunspot (b) US Population

Figure 14-1 Plots

This example uses several knit options: `label: fig-plots` assigns a label to the chunk, `fig-cap: Plots` sets the overall caption, `fig-subcap` specifies subcaptions for individual plots ("Sunspot" and "US Population"), and `layout-ncol: 2` arranges the plots in two columns. Other frequently used code block options include the following:

Note 14.1 You can use `knitr` options in the YAML header of a document or project to apply them to all code blocks within the document or all code blocks in the project!

Table 14-1 R Markdown Cheat Sheet

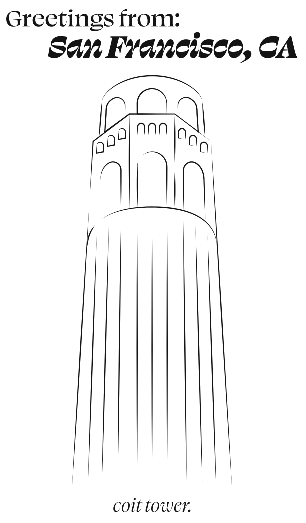
Option	Value	Description
eval	true	Dictates whether the code should be evaluated and included
echo	true	Determines whether to display code alongside results
warning	true	Controls the display of warnings
error	false	Governs the display of errors
message	true	Manages the display of messages
include	true	Prevents any output (code, results) from being included
tidy	false	Formats code neatly when displayed
results	"markup"	Specifies output format (markup, asis, hold, hide)
cache	false	Caches results for future rendering

14.2 Div Blocks

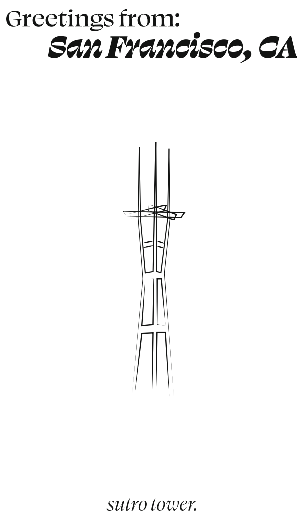
For those acquainted with HTML, `<div>` blocks will be familiar. You can create div blocks by wrapping text with three `:::` or more colons. This is useful for arranging images in a grid, as shown below:

```
::: {layout-ncol="2"}
![Coit Tower](coit_tower.pdf)

![Sutro Tower](sutro_tower.pdf)
:::
```



(a) Coit Tower



(a) Sutro Tower

Div blocks should be isolated from adjacent blocks by blank lines. They can also be nested within other divs.

The `pagebreak` short code allows for the insertion of a native page break, compatible across various formats:

```
First Page

{{< pagebreak >}}

Second Page
```

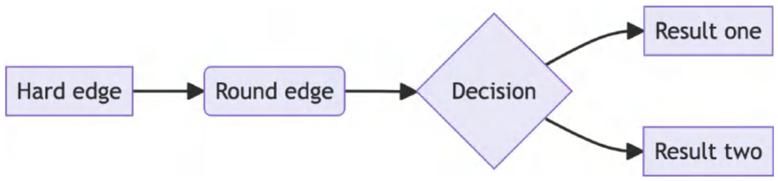
Different languages like R, YAML, HTML, and LaTeX have their own commenting syntaxes. The universal commenting syntax in Quarto is the HTML style `<!-- comment here -->`.[1]

[1] In RStudio, use Ctrl + Shift + C (or Command + Shift + C on macOS) to comment out a line of text.

14.3 Diagrams

Create elegant UML (Unified Modeling Language) diagrams within Quarto using tools like Mermaid and Graphviz. For example, the following flowchart was created with Mermaid:

```
flowchart LR
  A[Hard edge] --> B(Round edge)
  B --> C{Decision}
  C --> D[Result one]
  C --> E[Result two]
```



14.4 Citations

"Proper citation adds credibility to your work and acknowledges the work of others."
(ChatGPT)

Citing sources in your work is streamlined with Quarto's integration with Zotero. Quarto uses Pandoc to generate citations and bibliographies in your chosen style. To source your citations, include a .bib or .bibtex file and, optionally, a .csl file for citation style in your YAML header:

```
bibliography: references.bib
```

Cite sources in your document using `@yourcitation9999`; at your first source citation Quarto will make a citation file for you. Visual mode offers suggestions, and inputting an article's Digital Object Identifier (DOI) will insert it even if it is not in your bibliography. For detailed citation methods, see Quarto Citation and Pandoc Citations.

Markdown Format	Output (author-date format)
@abadie2017 says cluster your SE.	Abadie et al. [1] says cluster your SE.
Some thing smart [@abadie2017; @bai2009].	Some thing smart [1, 2].
Abadie says cluster [-@abadie2017].	Abadie says cluster 2017.

Zotero simplifies adding citations to your document. As you type, Zotero suggests citations for your bibliography file. For documents with over 10 citations, consider using Better Bibtex; just make sure Zotero is open.

To generate citations from a document (e.g., cited in Obsidian) without manual reciting, use the ***bbt_update_bib***`()` function from the `rbbt` package. Ensure Zotero is active and you are in the Markdown document you wish to update. The ***bbt_update_bib***`()` function will create a bibliography. With additional arguments, you can specify `path_rmd` as the path to your document and `path_bib` as the path to your bibliography file.

14.5 Summary

This chapter covered `knitr`, a tool for integrating code into documents, highlighting features like code block options and div block layouts. It also discussed creating UML diagrams with Mermaid and Graphviz and the use of Zotero for citations in Quarto. In the next chapter, you will learn how to effectively collaborate with less technical stakeholders using Google Docs and explore the various formatting and templating options available to you with Quarto.

Chapter 15
Collaboration and Templating

15.1 Streamlining Collaboration with `trackdown`

In projects that involve collaborators with varying levels of technical proficiency, the use of Markdown files may present a challenge to some stakeholders. While one might consider converting these files into a Microsoft Word document for easier editing and then transferring the edits back to Markdown, this process can be cumbersome. Our aim is to streamline workflows and enhance productivity efficiently. `trackdown` offers a compelling solution by enabling the collaborative editing of narrative text through Google Docs. For editing `.qmd` files, it's essential to have `trackdown` version 1.3 or higher, available for download from GitHub with the command:

```r
# install.packages("remotes")
remotes::install_github("claudiozandonella/trackdown")
library(trackdown)
```

Currently, the Google API credentials provided with the package are exhausted, requiring users to set up their own credentials. This setup is simplified by a guide from the developers, accessible here: https://claudiozandonella.github.io/trackdown/articles/oauth-app-configuration.html.

Note 15.1 If you want to use the files in a shared folder where you are not the owner, it will not work. You need either to be the owner of the folder or to use a shared drive.

Once it is configured, uploading your `.qmd` file to Google Docs is straightforward with `upload_file("your_file.qmd")`. For incorporating code, figures, tables, or analysis results, these cannot not be added directly in Google Docs. Instead, ensure that the document is first downloaded and all suggested edits from collaborators are reviewed and accepted (or rejected) in Google Docs. To streamline this process, you can use the option Tools → Review suggested edits → Accept all before downloading the document through `download_file(file = "your_file.qmd")`. Implement the necessary changes locally, and, if needed, use `update_file("your_file.qmd")` for further updates in Google Drive.

Note 15.2 Remember to save your file before uploading or updating to avoid data loss.

A useful tip is to employ `render_file("your_file.qmd")` to download and render the file simultaneously, allowing for a quick review of the rendered document and an efficient way to manage edits.

N. Tkachenko, *Data Insight Foundations*, https://doi.org/10.1007/979-8-8688-0580-6_15

Google Docs are recommended exclusively for narrative text, and while Git should be relied on for code management. This approach will prevent errors that may arise from editing code in Google Docs. Any coding should be done in an integrated development environment (IDE) like RStudio. Be aware that formatting applied in Google Docs will not be preserved in `trackdown`; use Markdown syntax for formatting instead.

This workflow encourages iterative collaboration, alternating between editing narrative text on Google Drive and coding locally with Git. It's important to note that `trackdown` does not support simultaneous editing of narrative text and code; thus, structuring the workflow sequentially is crucial for a smooth collaboration experience. To minimize potential conflicts between code and narrative text, limit the R code in the Quarto file and separate code into different files. This division creates two distinct systems for narrative and code, ensuring a seamless integration.

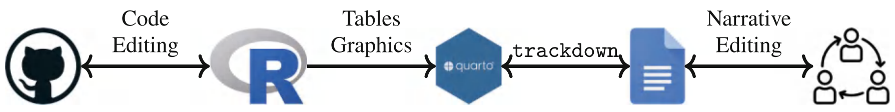


Figure 15-1 Collaboration Workflow

Once all the writing is done, it needs to be correctly formatted; we will briefly explore a few options in the following section.

15.2 Templating

Formatting your document to meet specific requirements, such as those for journals, theses, reports, or corporate styles, can be a frustrating but necessary task. There are numerous approaches to achieving this, and Quarto provides a range of default templates available for rendering a Quarto document. These templates are both editable and expandable, though creating a custom template is no small feat. Instead of diving deep into customization, it's advisable to explore templates created by others.

The Extensions section on Quarto's website is a great place to start looking for a specific journal template. If you're unsure which template to use, consider exploring those offered for *Elsevier*,[1] a major scientific publisher. If you already have a publication venue in mind, downloading its specific template early on can save you from a cumbersome transition later. Most templates come with clear installation instructions.

For those requiring further customization, especially with LaTeX documents, incorporating your Quarto LaTeX output into an Overleaf template offers much more control over the document's appearance by allowing direct edits to the LaTeX code. However, navigating through formatting options can be overwhelming, so it's beneficial to keep things simple and build on the work of others whenever possible.

If your organization mandates the use of Microsoft Word documents, while it may seem daunting, solutions exist to ease this process. The pandoc Microsoft Word reference document simplifies the conversion of Markdown to Microsoft Word, enabling relatively straightforward document styling. For more intricate styling needs, the `officedown` and `officer` packages provide tools to precisely meet styling requirements, allowing you to concentrate on content rather than formatting.

[1] https://github.com/quarto-journals/elsevier

15.3 Summary

`trackdown` enhances collaboration by integrating Google Docs for editing and simplifying the workflow, making it accessible and efficient for team members with diverse technical skills. Meeting specific formatting requirements for documents is streamlined by Quarto's templates and external resources like *Elsevier*[2] for journals. For advanced customization, Overleaf provides control to LaTeX users, while tools such as pandoc and `officedown` simplify styling for Microsoft Word documents, allowing authors to focus on content. Now we will transition to the data collection part, beginning with a chapter on total survey error.

[2] https://www.elsevier.com/researcher/author/policies-and-guidelines

Chapter 16
Total Survey Error (TSE)

If you are reading this book, you are likely familiar with basic statistics, especially the standard error. And if you are an economist, you know how challenging it can be to handle biases. We will explore the components of Total Survey Error (TSE) and methods to minimize them, enhancing survey results' quality and reliability. Additionally, we will delve into survey design, covering goal setting, sample selection, question design, and survey tools. Let's start with a question: What exactly is a survey? It might seem straightforward, but it's important to grasp the concept clearly. A formal definition from Groves et al. [14] is as follows:

The survey is a **systematic method**
for **gathering information** from
(a sample of) **entities**
for the purpose of
constructing quantitative descriptors
of the attributes of the larger population
of which the entities are members.

A more human-readable definition, and the one I prefer, comes from Caroline Jarrett [19]:

The survey is a **process**
for **getting answers to questions** from
(a sample of) **people**
for the purpose of
getting numbers
that **you can use to make decisions.**

Notice that the survey is not just a series of questions, which would be a questionnaire. It is an entire operation that, in addition to constructing a questionnaire, involves distributing the questions, gathering information from the target population, and processing the answers to extract meaningful insights. Each of these steps constitutes a distinct area of expertise.

What makes a survey credible? The answer lies in its accuracy, effective design, and minimal error. TSE [14] is a framework that enables researchers to identify and mitigate the various errors that can surface in surveys. A proper understanding of these errors will allow you to take proactive measures to minimize them. TSE consists of several components:

N. Tkachenko, *Data Insight Foundations*, https://doi.org/10.1007/979-8-8688-0580-6_16

$$\text{Total Survey Error} = \begin{array}{l} \overbrace{\text{Specification error} + \text{Measurement error} + \text{Processing error} +}^{\text{Measurement side}} \\ \underbrace{+ \text{ Coverage error} + \text{Sampling error} + \text{Non-response error}}_{\text{Representation side}} \end{array}$$

The first three—Specification, Measurement, and Processing errors—constitute measurement errors, which include all the errors introduced during the creation, measurement, and processing of questionnaires. The last three—Coverage, Sampling, and Nonresponse errors—are part of representation errors and are concerned with how accurately the respondents match the target population.

Measurement

1. **Specification error**: Specification errors manifest when there's no distinct link between theoretical concepts or constructs and the survey variables. It's vital to ensure that the survey questions aptly capture the intended concepts. Uninsightful questions yield uninformative results, so thorough literature reviews are critical. Specification error can be very nuanced. For example, measuring trust is challenging. Asking "How much do you trust on a scale of 1 to 10?" is too ambiguous. Instead, researchers ask questions about different aspects of trust—such as trust in strangers, the government, and friends. These responses have an underlying hidden cause: "the trust" that is measured by aggregating the responses into a comprehensive trust index.
2. **Measurement error**: Measurement errors arise from discrepancies between the estimated value and the "true" value, often attributed to survey design elements. Such errors can involve inaccuracies in question phrasing, response options, or respondent interpretation. To illustrate, food consumption surveys relying on respondents' memory might introduce measurement error because of inaccurate recall or understanding of portion sizes.
3. **Processing error**: These errors are related to issues encountered during the collection and management of survey data. They can include inaccuracies in data entry, coding mistakes, or other inconsistencies that may arise during data processing. Such errors are particularly common in physical surveys, where manual data entry may result in mistakes due to a slip of the finger or misinterpretation of handwriting. Additional errors may be introduced during the data cleaning and encoding stages. Transitioning to electronic surveys can help mitigate many of these common issues by reducing the likelihood of human error.

Representation

1. **Coverage error**: Coverage errors happen when individuals from the target population are absent from the sampling frame used to draw a representative sample. This can lead to underrepresentation or overrepresentation of certain groups within the survey. For example, research on general fitness levels based on data from fitness apps may significantly overestimate the average level of fitness, since it primarily captures a population of fitness enthusiasts.
2. **Sampling error**: Sampling errors occur because surveys typically gather data from a subset of individuals rather than the entire population of interest. Such errors stem from the fact that the sample results may not flawlessly reflect the true population values. For instance, a survey that randomly selects a thousand individuals from a city to estimate the average income of its residents may yield different results from the true average income of the entire city population, purely due to the variability inherent in sampling.
3. **Nonresponse error**: Nonresponse errors occur when individuals selected for the survey either choose not to participate or fail to provide comprehensive responses. This can introduce bias if nonrespondents differ significantly from respondents in crucial aspects. This is especially common

in online reviews, where people who had a good experience with a product won't leave a review but those unlucky few who didn't like the product leave negative reviews.

We will look into each of these stages and build an understanding of how to avoid common errors.

Let's begin with specification error and goal setting. The utility of a survey is determined by the insights it provides. The principle here is simple: "Ask stupid questions, get stupid answers." Before designing your survey, ask yourself these three questions:

1. What do you want to understand?
2. Why do you need this understanding?
3. What decisions will be guided by the answers?

In a business context, consider these three questions that focus on practical business goals:

1. What decisions will the survey guide?
2. What information is needed for these decisions?
3. How will the survey improve understanding?

If you are just starting out and are looking for a good research question, begin by conducting an exhaustive literature review. Identify gaps in the discussion and determine the importance of addressing these gaps. Then define a problem, use theory to form hypotheses, and decide on the testing methods.

It is vital to spend a significant amount of time reading the literature and building a solid understanding of the field, methods, and research. This will form the foundation of your entire project. By setting clear goals for your project, you will not only save time but also avoid specification errors.

16.1 Representation—The People You Ask

Achieving representativeness in surveys is crucial for obtaining a balanced reflection of the population. This ensures that the selected sample will accurately depict the characteristics, opinions, or behaviors of the larger population.

Striving for representativeness ensures a comprehensive depiction that will accurately reflect the true diversity of the study population. Consider whether a selection of birds is representative of a larger flock. Do you believe that the random sample in Figure 16-1 is representative of the full flock in Figure 16-2?

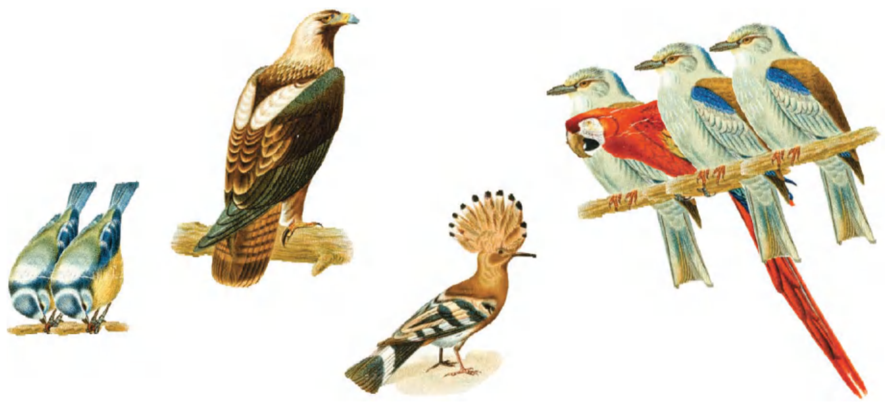


Figure 16-1 Random Sample of Birds

Figure 16-2 Bird Population

Definitely not! We are missing a number of different birds entirely, and we have several of others. They seem to like being in groups. There's also a cheeky macaw that isn't even part of the flock we're interested in.

16.1.1 Sampling

In fieldwork, we interact with participants and collect their responses. However, it's critical to address potential challenges in reaching the desired target group. To illustrate the considerations, we will use an example of conducting a survey to understand the coffee drinking habits of university students on campus.

1. **Target population**: The specific group we aim to understand or represent through the survey.
   - We're interested in all university students on campus, regardless of their field of study.
2. **Sample frame**: The list or source used to select our survey sample, ensuring it will align with the target population.
   - We decide to use the university's enrollment records to identify potential participants, ensuring they're actually students.
3. **People you ask**: These are individuals we approach to participate in the survey, selected based on the sample frame and target population.
   - Based on the enrollment records, we approach students in various campus locations—libraries, cafeterias, and study halls—to fill out our survey. We need to keep in mind that those who don't spend time at these locations are not sampled!

4. **People who respond**: The participants who voluntarily provide their responses.
   - Of the students we ask, only those who decide to complete our survey will provide the insights we're looking for. This group might have stronger opinions about coffee or more free time to participate in surveys, which is important to consider when analyzing our data.

By carefully thinking about the target population, using an appropriate sample frame, effectively engaging participants, and analyzing respondent characteristics, we enhance the reliability and validity of our survey findings.

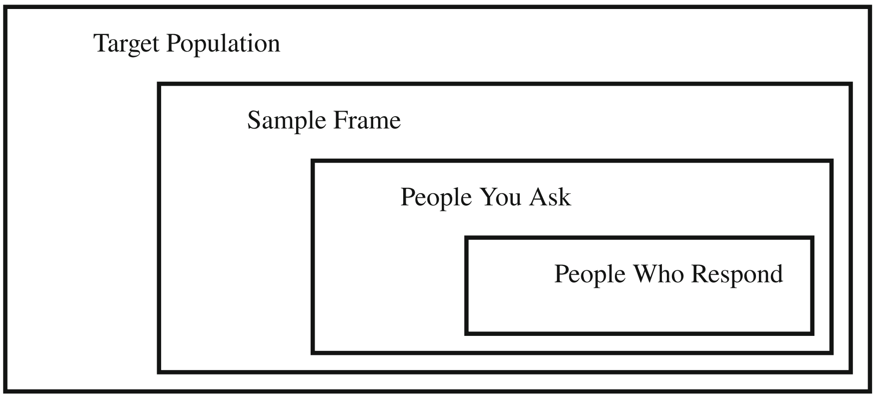


Figure 16-3 Sampling Error

For a successful survey, we must outline our target population and sampling frame. However, obtaining complete coverage can be challenging. Undercoverage happens when certain individuals or groups in the target population are inadequately represented in the sampling frame. This can result in biased outcomes. Conversely, overcoverage refers to the inclusion of individuals or groups in the sampling frame who are not part of the target population, potentially leading to extraneous data collection and biases.

Note 16.1 Better to ask the right people than many people.

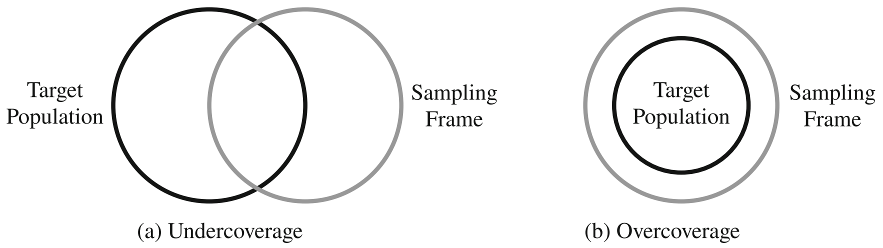


Figure 16-4 Target Population and Sampling Frame Coverage

To ensure a representative sample, it's crucial to minimize undercoverage and overcoverage. This can be achieved by methodically designing the sampling frame, considering aspects such as demographics, location, advertisement strategy, and time constraints. Frequent evaluation and adjustments of the sampling frame are key to addressing any coverage issues that might surface.

If you happen to miss your target population, you might find yourself taking a detour in the footsteps of Mr. Worldwide to Alaska. In a 2012 incident, Pitbull participated in a commercial in which Walmart customers could vote for the store where the artist would perform. Well, asking a question on the internet is like posing the question to everyone, and things took an unexpected turn. The vote was hijacked, and Pitbull ended up being sent to Alaska. So, it's a reminder that when conducting surveys, it's crucial to ensure you're reaching the right people and avoiding unintended outcomes.

16.1.2 What Responses Depend On

Survey participant responses are influenced by myriad factors; nonetheless, a few key ones stand out: effort, reward, and trust. By effectively addressing these three factors, you can not only decrease nonresponse error but also address measurement error.

1. **Effort**: The amount of effort required from participants to complete the survey can significantly impact their willingness to participate. Long, complex, or time-consuming surveys may discourage participation, leading to lower response rates.
2. **Reward**: Participants often seek some form of incentive or benefit for their taking part in the survey. This can include tangible rewards, such as monetary compensation or gift cards, or intangible rewards, such as the satisfaction of contributing to research or personal interest in the survey topic. Offering appropriate rewards can motivate participants to provide accurate and thoughtful responses.
3. **Trust**: Building trust with survey participants is important for encouraging open and honest responses. Participants need to feel confident that their privacy and confidentiality will be respected, their data handled securely, and the survey conducted by a reputable organization. Clear communication about data protection measures and ethical considerations helps establish trust and increases the likelihood of obtaining reliable responses.

By taking into account the effort required, providing appropriate rewards, and building trust in the survey process, you can foster an environment conducive to obtaining high-quality responses from participants.

In an intriguing study by John et al. [20], experiments were conducted to understand the factors that influenced people's willingness to divulge personal information. One notable experiment involved altering the design of a survey page and observing its impact on disclosure rates.

Students from Carnegie Mellon University were recruited for the study, and variations were made to the survey's title and interface. In the frivolous framing, the survey bore a humorous title: "How BAD are U??" and featured a bright yellow background, red text, and a cartoonish devil icon, resembling a lighthearted online quiz to downplay privacy concerns. Conversely, the baseline framing had the survey presented within a professional context: "Carnegie Mellon University Survey of Ethical Standards."

Interestingly, participants in the frivolous framing were, on average, 1.7 times more likely to confess to engaging in sensitive behaviors compared to those in the baseline framing. For example, individuals in the frivolous framing were over twice as likely to admit to taking nude photos of themselves or a partner, with a 31.8% admission rate in the frivolous framing versus a 15.7% rate in the baseline framing.

These findings suggest that individuals might feel more at ease sharing personal information on platforms that don't appear strictly professional, even though these platforms might pose a higher risk of data misuse. The results highlight the complex interplay between context, trust, and disclosure when it comes to personal information.

16.2 Question Design

We now shift our focus to the heart of any survey—the questions themselves. We'll tackle common concerns such as the quality of questions, the critical role of their order, the implications of including "N/A" (not applicable) responses, and the impact of survey length.

16.2.1 Answering Questions

According to Tourangeau et al. [29], answering a survey question involves four key steps. These steps provide valuable insights into the respondents' thought processes:

1. **Understand the question**: The first step in answering a survey question involves comprehension. Respondents need to accurately understand the question to provide a meaningful response. Therefore, the clarity and readability of the question are of utmost importance. For example, try to understand the following question:

   In the last five days at work, what percentage of the time did you use corporate-grade communication software?

2. **Find an answer**: Once the question is understood, respondents must search their memory or knowledge base to come up with a suitable answer. This answer should be based on the information available to them. For instance:

   Remember:

   1. What color shirt did you wear two Tuesdays ago?
   2. What color shirt did you wear on New Year's Eve?
   3. How did you celebrate your 18th birthday?
   4. What did you have for breakfast yesterday?
   5. When did the American Civil War start?

3. **Judge the answer**: After identifying a potential answer, respondents evaluate whether they feel comfortable sharing it. Factors such as privacy, social desirability, and the sensitivity of the information play a role in this judgment process. For example:

   Agree or disagree with the following statement: "I approve of the current management's actions."

   1. Strongly agree
   2. Agree
   3. Neither
   4. Disagree
   5. Strongly disagree

   Though the question seems straightforward, expressing disagreement with the current management's actions could have potential consequences, such as risk of dismissal if the management becomes aware of your lack of support.
4. **Place the answer**: Finally, respondents need to appropriately map their answer onto the response options provided. This could involve selecting a specific category, rating on a scale, or providing a written response. For example:

   What is your major?
   What is your career?

   What is your major? ________
   What is your career? ________

Refining survey questions can significantly enhance the quality of responses received. Let's practice using the question from the first list item. Notice your thought process while answering the question, and consider how you might improve it:

> Over the last five working days, what percentage of the time did you use corporate-grade communication software?

There are several issues with the preceding question. First, what exactly is "corporate-grade communication software"? Second, recalling the percentage of time over the last five working days is challenging, as it is a relatively long period, and daily activities often do not overlap. Therefore, the question can be revised into multiple, more specific questions to address these concerns:

> On your most recent workday, what percentage of the time did you use messaging software other than email (e.g., Slack, Discord, WhatsApp, Telegram)?
>
> What do you use for communication at work? (Select all that apply)

☐ Email
☐ Slack
☐ Discord
☐ WhatsApp
☐ Other

Insights from survey response psychology can aid researchers in designing high-quality questionnaires that minimize respondent burden, enhance data quality, and create a better overall survey experience.

16.2.2 Guidelines for Effective Question Design

Question design is fundamentally a human-centric exercise. It involves crafting questions that are easily understood, allowing respondents to effortlessly process, judge, and articulate their answers. Although much of the advice might seem generic and rather obvious, it is crucial to remind yourself of these simple points. I often refer back to the guidelines from Krosnick's work on questionnaire design [22]. To simplify this, I've compiled the key points into a list:

1. Use simple and familiar words to ensure clarity and comprehension. Avoid technical terms, jargon, and slang.
2. Avoid words with ambiguous meanings to ensure all respondents interpret the questions similarly.
3. Employ simple sentence structures and syntax for easy comprehension.
4. Make questions specific and concrete rather than general and abstract to obtain consistent and reliable responses.
5. Avoid leading or loaded questions that sway respondents toward a particular answer.
6. Ask about one thing at a time to prevent confusion and promote accurate responses. Avoid double-barrelled questions that combine multiple topics.
7. Steer clear of questions with single or double negations, as they can be confusing and lead to misinterpretation, even if some disciplines require a portion of questions to use negation.
8. Provide response options that are exhaustive (covering all possible choices) and mutually exclusive (no overlap between options).

There is much more to consider, but this should get you thinking in the right direction. I recommend looking into Krosnick's other works, including his articles, books, and workshops.

16.2.3 The Impact of Question Order

The order in which questions are presented in a survey can profoundly influence the quality of responses and the level of engagement from respondents. Effective sequencing of questions is crucial for gathering reliable data. Here are some considerations that I like to keep in mind:

1. Start with easy and pleasant questions to build rapport and boost respondent confidence.
2. Begin with questions directly addressing the main topic as described to the respondent prior to the survey.
3. Group related questions together to maintain continuity and coherence.
4. Sequence questions on a specific topic from general to specific, gradually narrowing the focus.
5. Place sensitive questions that may make respondents uncomfortable toward the end of the questionnaire, allowing participants to warm up before addressing personal or sensitive topics.
6. Include filter questions to avoid asking irrelevant queries. This ensures participants will only answer questions applicable to them.

If you're looking for tested questionnaires, I highly recommend the *repository created by Gabriel Gilmore & Sara Finney*,[1] which offers a comprehensive collection of validated questionnaires. Additionally, a great resource for questions is the *Handbook of Marketing Scales* series [8], which is the longest-running set of books that provides reviews of multi-item measures used in scholarly studies of consumer behavior.

16.2.4 Survey Says: Bacon!

Let's look at an example of how intentionally flawed design of survey questions and the interpretation of their results can lead to misleading conclusions.

Edward Bernays, a prominent figure in public relations, was approached by a bacon company seeking to boost sales. Initially, Bernays conducted research and found that a substantial breakfast was likely beneficial for health. Armed with this information, he approached the physician on his team and requested his endorsement, leading to a survey sent to 5,000 colleagues, all of whom agreed with the health benefits of a hearty breakfast.

The results were sent to the media, culminating in headlines proclaiming "4,500 Physicians Urge Americans to Eat Heavy Breakfasts to Improve Their Health." This endorsement, along with the suggestion of incorporating bacon and eggs into breakfast, had a significant impact on public perception, permanently changing American breakfast habits.

It's worth noting that the particular bacon product itself was not in question. Rather, Bernays utilized the opportunity to align the product with a broader health message, successfully increasing bacon sales and shaping American breakfast culture. What is wrong with the question they asked and their conclusion?

> What is better for health, a slight or substantial breakfast?
>
> 1. Slight
> 2. Substantial
>
> "4,500 physicians urge Americans to eat heavy breakfasts to improve their health":
> Have more eggs and bacon for breakfast!

[1] https://www.jmu.edu/assessment/sass/_files/database_of_existing_measures_3.8.21.pdf

The question suffers from a specification error because it broadly inquires about the size of breakfast rather than directly addressing the health implications of consuming eggs and bacon. Additionally, the term "substantial" is somewhat abstract and ambiguous, as it is synonymous with "sufficient," leading to potential misinterpretation where respondents might consider "substantial" to mean simply "adequate" rather than "large" or "hearty."

Furthermore, the choice of the target population is questionable. Physicians, while medically knowledgeable, may not have the specialized expertise in nutrition that dietitians or nutritionists possess. The survey's sampling frame was limited to a mailing list of an employee physician, which may not accurately represent the broader medical community's views on diet.

The conclusion derived from the survey's results is then misleadingly extrapolated from "substantial" to "heavy" breakfast and subsequently linked specifically to the consumption of eggs and bacon without sufficient data to support such a specific dietary recommendation. This leap in logic oversimplifies and misrepresents the original survey findings, potentially harming a diverse population with varying health needs and dietary restrictions.

16.2.5 Types of Question Formats

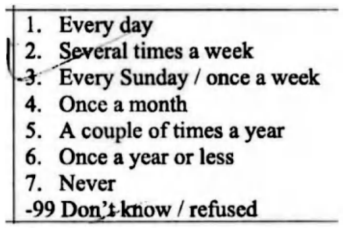
Different types of question formats in a questionnaire enable capturing diverse data types. Three commonly used question formats are the following:

1. **Single Choice**: Ideal for questions that require respondents to select only one answer from a list of options. For online surveys, the respondent chooses a single option by clicking on the corresponding radio button.
2. **Multiple Choice**: Suitable for questions that allow respondents to select multiple answers from a list of options. A common form is to have multiple checkboxes so respondents can check them to indicate their choices.
3. **Open-Ended Question**: Used when respondents need to provide a written response or input data not covered by predefined options. Usually, open-ended questions are represented by text boxes to allow users to type in their responses freely.

Properly utilizing these question formats ensures that your questionnaire will capture the necessary information and provide clear response options. However, it's essential to provide clear instructions on how participants should input their responses; otherwise, the data might be susceptible to interpretation.

The format in Figure 16-5 does not instruct participants on how to provide their answers. As a result, the method of indicating an answer—typically by checking—varies greatly among respondents: some check in the middle, some above, and some below, making it exceedingly difficult to interpret the selected answer. Additionally, handwriting can create ambiguity; for instance, the numbers 1 and 7 often appear very similar. A more effective approach would be to ask participants to circle the number, provide a checkbox next to each answer, or offer answer choices to avoid requiring respondents to write out their answers.

The choice of question format should align with the nature of the question and the type of response you are seeking. A thoughtful approach will decrease processing errors and make data entry effortless.


1. Every day
2. Several times a week
3. Every Sunday / once a week
4. Once a month
5. A couple of times a year
6. Once a year or less
7. Never
-99 Don't know / refused

(a) This is 2

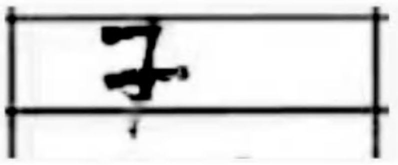

(b) 1 or 7?

Figure 16-5 Answers Without Instructions

16.2.6 Likert Scales

Likert scales are commonly used in surveys to measure attitudes, opinions, and perceptions. You have likely encountered them when answering questions that use an agree-disagree range. Here are some common considerations when choosing the number of scale points:

1. **5 or 7 Points**: Both 5- and 7-point scales are widely used and offer sufficient response options for capturing a range of opinions. There is generally no significant difference in response patterns between these two options.
2. **Consider Respondent Ease**: Using a lower number of scale points, such as 5, makes it easier for respondents to quickly choose their answer. This leads to higher response rates and less respondent fatigue.
3. **Higher Points for Specific Needs**: In some cases, a higher number of scale points may be necessary to capture nuanced responses or when greater precision is required. This could be relevant in specialized research or when examining highly specific attitudes or behaviors.
4. **Follow Established Protocols**: If your survey is part of a larger study or research field, it is advisable to use the scale points commonly used in that context. This ensures consistency and comparability with existing research.

Note 16.2 In the end, the person who decides the scale is the one who pays.

Ultimately, the decision on the number of scale points rests with the survey designer, who will have to take into account specific research goals, target audience, and practical considerations. For a guide on constructing questions with Likert scales, you can refer to *Likert Scale Response Options document*[2] based on Vagias [32].

16.2.7 "I don't know" and N/A Options

Including "I don't know" and N/A response options in surveys can impact data quality. Krosnick et al. [23] explored the impact of "no opinion" response options and their effects on data quality, highlighting potential tradeoffs associated with their inclusion. Some considerations are as follows:

1. **Potential Impact on Data Quality**: Inclusion of "I don't know" and N/A options may not necessarily improve data quality. It can sometimes limit the measurement of meaningful opinions by providing an "easy way out" for respondents who might otherwise give a thoughtful response.

[2] https://mwcc.edu/wp-content/uploads/2020/09/Likert-Scale-Response-Options_MWCC.pdf

2. **Influence of Mental Fatigue**: Respondents experiencing fatigue or uncertainty about their opinions may select neutral options like “I don’t know,” resulting in a higher proportion of neutral responses and potentially masking genuine attitudes.
3. **Optional Questions**: All questions in a survey are optional. Respondents can choose to skip questions they do not want to answer or that do not apply to them. Making questions optional allows individuals to provide answers only to the questions they feel comfortable or knowledgeable answering.

It’s crucial to consider these factors when designing surveys and deciding whether to include “I don’t know” and N/A response options.

16.2.8 Survey Length

The length of a survey can have significant implications for participant engagement and data quality. Below are some key points to keep in mind:

1. **Drop-Out Rates**: In web-based university studies [17], there is typically an instantaneous drop-out rate of around 10%, with an additional 2% drop for every 100 survey items. Consequently, longer surveys could lead to higher attrition rates as participants might become fatigued or lose interest.
2. **Satisficing and Quality of Responses**: Lengthy surveys, particularly in lab-based research, can result in satisficing behavior [33], where participants offer “good enough” responses instead of providing thoughtful and accurate answers. This could negatively impact data quality. However, these adverse effects may not significantly affect the overall results [4].
3. **Online Marketing Questionnaires**: For online marketing surveys, keeping the survey length short is advisable [37]. Surveys longer than a few questions are likely to experience significant drop-out rates. Surveys taking longer than ten minutes are strongly discouraged from maintaining participant engagement.

Survey designers should strive to balance acquiring the necessary information with keeping their survey manageable for respondents to ensure optimal participant engagement and response quality. Personally, I tend not to include an “N/A” option unless there is a chance something doesn’t apply to a particular person. In my experience, making questions optional (not forcing people to respond) and excluding “N/A” increases the number of answered questions. Additionally, leaving an open text option, where participants can write their own response, is extremely useful in the early stages, when you might not know much about the population. This allows people to indicate if you missed a common answer choice. However, it might backfire later at a larger scale, as some might forgo classifying their answer into prespecified bins and just put it in open text, necessitating later manual reclassification.

16.2.9 Survey Invitation

When inviting participants to a survey, it’s crucial to establish trust and provide the necessary information to encourage participation. Below is an outline of best practices:

1. **Introduction**: Clearly identifying yourself and explaining the reason for reaching out to the specific person can help build trust and credibility.

2. **Purpose and Benefits**: Outlining the purpose of the survey and the benefits of participation can motivate respondents. Explain how their responses will contribute to the research or decision-making process.
3. **Incentives**: If applicable, mentioning any incentives or rewards offered for participating in the survey can help motivate individuals to complete the survey.
4. **Survey Overview**: Providing an overview of the topics or questions covered in the survey helps manage expectations and assist participants in managing their time.
5. **Closing Time**: Specifying the deadline or closing time for survey completion can create a sense of urgency and ensure timely responses.

In addition to these, consider adding a "Thank You" page after participants complete the survey. This shows appreciation for their time and participation. A clear and concise privacy statement addressing privacy concerns can assure participants that their responses will remain confidential and used solely for research purposes. Lastly, outlining your follow-up plan can let participants know if and how you will share the survey results or any actions that will be taken based on the findings. Incorporating these elements into your survey invitation can improve participant engagement and enhance the overall survey experience.

16.2.10 Iterative Design

Iterative design involves repeated cycles of learning, refining, and scaling, aimed at increasing the effectiveness of surveys. The initial step is to create a rough draft of the survey and then interview someone from the target audience to determine whether the questions are clear and relevant to them. Make sure to pay close attention to all aspects of the survey, such as questions, instructions, design, and distribution. Based on the feedback, the survey will be edited and further refined through discussions with a larger group of people (ten would be a good number). It's important to avoid distributing the survey to the entire target pool initially. If changes are necessary after the initial distribution, there may not be enough participants left for a revised survey, not to mention the potential waste of resources. This iterative approach allows researchers to extract valuable insights from each interaction, continuously improve their survey design, and eventually scale up to a wider audience for comprehensive analysis. Having covered the creation of a questionnaire, we now consider how to distribute a survey to participants.

16.3 Survey Tools

Researchers have a wide array of survey tools to choose from, each offering unique features and advantages. Understanding these options is useful for data collection. Now we will look at different ways for physical and digital surveying, as well as platforms, to recruit participants.

16.3.1 Physical Survey Methods

Traditional physical survey methods provide diverse ways to collect data, which is especially beneficial in situations where digital access may be limited.

1. **Paper Surveys**: These are versatile and can be distributed in various settings and in addition don't require familiarity with laptops or phones.
2. **In-Person Interviews**: This method allows for direct participant interaction, clarification of complex questions, and collection of nuanced responses.
3. **Telephone Surveys**: These surveys are efficient for collecting data from geographically dispersed samples, but potential biases should be considered, such as excluding individuals without telephones.
4. **Mail Surveys**: This approach reaches a wide range of individuals, especially those without internet access. Response rates and turnaround times may vary.
5. **Intercept Surveys**: These surveys involve approaching individuals in public spaces for participation, allowing for immediate data collection, yet potential issues with representativeness and privacy may arise.

For all physical methods, it is imperative to adhere to ethical guidelines for data collection and ensure participant privacy. Additionally, training and supervision of assistants who conduct physical surveys are essential. Some hybrid approaches are also possible, where participants fill out a survey on a laptop or tablet, or a research assistant completes it during the interview process. Although cumbersome from an operational standpoint, physical surveys are often the only way to gather responses from rural areas.

16.3.2 Digital Survey Tools

A variety of digital survey platforms cater to different research needs, providing an array of features, usability levels, and costs.

1. **Google Forms**: Offers user-friendly and free services, suitable for simple surveys or budget-constrained projects.
2. **Typeform**: Provides an interactive interface for engaging surveys, enhancing user experience and response rates. It creates the best-looking surveys by far.
3. **Qualtrics**: A comprehensive platform favoring complex survey design with advanced features like conditional logic, built-in distribution capabilities, collaboration features, and extensive customization. Many universities have institutional subscriptions to Qualtrics, so make sure to check whether you have access!
4. **SMARTRIQS**: An open-source tool that integrates with Qualtrics for conducting interactive step-based online experiments.
5. **LimeSurvey**: A customizable, open-source survey tool ideal for complex survey flows, especially when Qualtrics is not available.
6. **O-tree**: Designed specifically for economic experiments and social science research with features like decision-making games, real-time data collection, and payment processing.
7. **Z-Tree (Zurich Toolbox for Readymade Economic Experiments)**: Software for developing and conducting economic experiments. It provides a control language designed to be easy to learn and use, enabling users to quickly create complex experiments.
8. **ChoiceFlow**: A framework for designing and running complex behavioral experiments online. It supports a wide range of experimental designs and integrates seamlessly with major online labor marketplaces, enabling researchers to efficiently recruit and pay participants.

16.3.3 Participant Recruitment Platforms

Platforms like Amazon Mechanical Turk (MTurk), Prolific, and Respondent offer an additional dimension to survey tools, serving as spaces for recruiting and compensating participants.

1. Amazon Mechanical Turk (MTurk): A crowdsourcing platform providing a large, diverse participant pool at a relatively low cost, albeit with considerations for fair compensation and data quality.
2. Prolific: Similar to MTurk, but specifically designed for academic research with prescreened participants and emphasis on ethical standards.
3. Respondent: Specializes in finding professionals for high-quality research, particularly useful for B2B market research or studies requiring professional expertise.

16.4 Summary

In this chapter, we explored the concept of TSE and its critical components: measurement and representation errors. Understanding these errors and implementing strategies to minimize them is essential for designing reliable and effective surveys. We also discussed the importance of clear goal setting, achieving representativeness, and crafting well-designed questions to ensure accurate data collection. By adopting best practices in survey design and leveraging both physical and digital survey tools, researchers can enhance the quality and credibility of their survey results. In the next chapter, we will underscore the importance of comprehensive research documentation, emphasizing traceability, accessibility, and security, with practical tips for maintaining high-quality, reproducible data.

Chapter 17
Document

Comprehensive documentation is pivotal to the success of any research project, ensuring consistent data collection and adherence to established procedures by field agents. This chapter outlines the importance of documenting every process related to sampling, administration, data collection, and data handling.

Note 17.1 Although universally disliked, bureaucracy is undeniably essential!

17.1 Principles of Documentation

A robust documentation system rests on several foundational principles:

Document Design and Formatting

1. **Physical and Digital Accessibility**: Ensure that all documentation is easily accessible in both physical and digital formats. Organize physical documents for easy retrieval and ensure that digital documents are stored in a way that allows quick and efficient access.
2. **Convertibility**: Enable the seamless conversion of physical documents, such as surveys and forms, into electronic format. This facilitates efficient storage, quick retrieval, and easier sharing.
3. **Consistent Formatting**: Maintain a uniform appearance and structure across all documents. Use standardized templates and consistent typography to enhance readability and ensure that users can easily understand and navigate the documentation.
4. **Update and Review Process**: Establish a clear procedure for regularly updating and reviewing documents. This ensures that the information will remain relevant, accurate, and up to date. Regular reviews help identify and correct errors or outdated content.

Document System

1. **Centralization**: Centralize all documentation in a single, unified system. This minimizes ambiguity and ensures that everyone has access to the most current and accurate information.
2. **Hierarchical Structure**: Adopt a clear hierarchical structure in documentation that reflects the different levels, such as site, session, and subject levels. This helps users navigate through the documents easily and understand the relationships between different sections and topics.
3. **Redundancy**: Introduce strategic redundancy by replicating key information across multiple formats or locations. This reduces the risk of data loss and ensures that crucial information is always available, even if one source becomes inaccessible.

N. Tkachenko, *Data Insight Foundations*, https://doi.org/10.1007/979-8-8688-0580-6_17

4. **Resilience**: Design the documentation to withstand chaotic situations. This includes creating backups, using formats that are less prone to corruption or loss, and ensuring that documents can be easily reorganized if they become disordered.
5. **Security and Privacy**: Place a strong emphasis on securing sensitive information within the documentation. This involves adhering to data protection laws, implementing robust encryption methods, and ensuring that only authorized personnel have access to sensitive data.

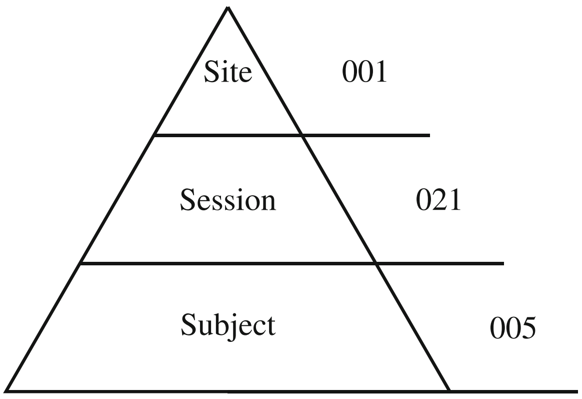
Working with Colleagues

1. **Collaboration and Sharing Protocols**: Define clear protocols for collaborative document creation and sharing. This ensures consistency, prevents conflicts, and reduces the risk of data loss. Effective collaboration protocols help maintain the integrity and coherence of the documentation.
2. **User Training and Support**: Provide comprehensive training and support for all stakeholders involved in the documentation process. This ensures that everyone can effectively use and contribute to the documentation system, leading to higher quality and more reliable documents.

17.1.1 Hierarchical Structure of Documentation

The hierarchical structure of project's documentation is a crucial element in ensuring the efficient management and retrieval of data during research. This structure is typically divided into three primary levels: site, session, and subject, each marked by unique identifiers and variables to maintain order and clarity.

Figure 17-1
Documentation Hierarchy

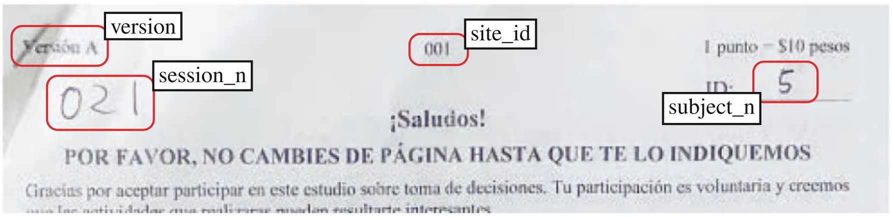


At the broadest scope, the project encompasses all aspects of the research, including analysis, goals, methodologies, responsibilities, and documentation. It provides a comprehensive overview of the entire research initiative, outlining its scope and objectives.

The **site level**, following the project tier, represents the broadest categorization within the project. It usually pertains to the physical or conceptual location of the research, such as the site of an experiment. This level includes specific information that distinguishes one site from another, such as location details, dates of research activities, responsible personnel, local exchange rates, and population characteristics. The identifiers at this level are broad, offering a macro view of the research landscape.

Under the site level is the **session level**. This level offers a more focused categorization and is essential for documenting the chronological progression of the research. It may represent specific phases of the project or a series of observations. The session-level details include specific experiments conducted, timeframes, methodologies employed, and treatments administered. It acts as a conduit, bridging the comprehensive scope of the site level with the detailed focus of the subsequent subject level.

The most granular tier in the hierarchy is the **subject level**. This level is concerned with individual subjects or units within the research, such as study participants or specific experimental samples. Each subject is assigned a unique identifier to ensure precise tracking and analysis. The subject level encompasses detailed data for each unit, including demographic information in human studies, specific characteristics of samples in laboratory research, and granular observations in field studies. A subject can be uniquely identified by their associated site, session, and subject number.


version
site_id
session_n
subject_n

Versión A

001

1 punto = $10 pesos

021

ID: 5

¡Saludos!

POR FAVOR, NO CAMBIES DE PÁGINA HASTA QUE TE LO INDIQUEMOS

Gracias por aceptar participar en este estudio sobre toma de decisiones. Tu participación es voluntaria y creemos que las actividades que realizarás pueden resultarte interesantes

Figure 17-2 Labeling Example

This structured hierarchy effectively organizes all materials and systems, providing a solid framework for your project. By labeling each document with a unique ID, you ensure that in the event of a mix-up, you can easily recollect the materials. Additionally, including essential information on the labels, such as ID, number of subjects, payouts, time and date, for example, will help you quickly find and reference important data. For instance, a session package containing all participant materials with appropriate labeling can streamline your process. While it is acceptable to deviate from this structure, finding a system that suits your specific research needs is crucial for maintaining productive and organized work.

I suggest considering Google Suite, as its products are widely used and understood. Google Sheets and Google Drive offer seamless interaction, enabling cross-references between documents and folders. Google Drive supports the creation of a robust folder structure, and Google Docs provides excellent collaboration tools. Google Sheets offers extensive capabilities for data entry and initial data processing, allowing you to maintain a database within the platform. It also enables convenient data loading into R through an Application Programming Interface (API) (covered in the next chapter). Furthermore, you can use Google Sheets to build a simple dashboard to track your results as new data come in, helping you monitor potential issues effectively. Additionally, if you are working on a larger project, you can implement a permission system for better control.

17.2 Physical and Electronic Documentation

Physical documentation is a crucial component of research, serving to ensure reliable and consistent data collection. Attention should be paid to structured formatting and strategic management of physical documents. Below is a brief overview of best practices:

1. **Consistency and Clarity**: It's important for physical documents to feature uniform formatting and clearly phrased questions to reduce interpretation errors. Such consistency aids in accurate data gathering and ensures a uniform understanding among participants.
2. **Efficient Paper Usage**: Conscious paper use helps control costs and reduces environmental impact. Employing strategies like two-sided printing and preferring digital copies where feasible can

markedly cut down on paper waste. Moreover, excessive paper can become heavy and hinder the processing of results.
3. **Tracking Participation Agreements**: Assign unique subject identifiers on consent forms for effective tracking (Figure 17-2). After sessions, collate these for easy reference, thereby ensuring compliance with ethical protocols. Additionally, include the survey version to aid in future referencing. For electronic data collection, storage and labeling are mostly automatic. However, for paper-based data collection, careful labeling is crucial.
4. **Session-Specific Recordkeeping**: Maintain detailed notes during each session. Such immediate documentation should cover any variations or significant occurrences, thereby improving the accuracy and comprehensiveness of data collection.

Note 17.2 Maintain a field journal to record observations regarding the weather, political climate, experiment rounds, site, and other relevant factors.

Effective physical documentation is crucial for maintaining the integrity and traceability of research data. After data collection, it is advisable to preserve the original data sources for potential future clarification.

Equally important is electronic documentation, which safeguards research data by ensuring both security and accessibility. After each research session, promptly transfer all collected information to an electronic session sheet and scan physical documents to create digital copies.

In today's digital landscape, safeguarding data protection and privacy is paramount for research integrity. Implement robust data protection strategies to secure collected data, especially personal information about subjects. This includes anonymizing data, employing secure storage solutions, and adhering to data protection laws, particularly when handling personally identifiable information.

Quality assurance is equally important. Develop a comprehensive strategy for data quality assurance by implementing training programs for data collectors to ensure accuracy and consistency. Conduct regular data checks or audits to identify and correct errors or inconsistencies. Additionally, establish feedback mechanisms for continuous improvement.

17.3 Data Organization in Spreadsheets

Spreadsheets are ubiquitous, and they are often the first and the only way people learn to interact with data. And while it is possible to do pretty much anything in them, that doesn't mean you should build a roller coaster simulator in Excel.[1] Here we will focus on organizing spreadsheets for data entry and storage to reduce errors and simplify later analyses. It is largely based on my personal experience and advice of Broman and Woo [7].

17.3.1 Spreadsheet Organization

Consistency is the cornerstone of effective data entry and organization. It's crucial to maintain a uniform approach to data handling to prevent the need for time-consuming data harmonization later.

Regarding the organization and storage of your data, consider how the data will be entered, transferred, and stored. Google Sheets is a recommended tool due to its accessibility and no-cost nature, but Excel or other software can also be effective. The principles outlined here are applicable regardless of the software used.

[1] https://www.youtube.com/watch?v=IrVA1BBHFHw

For larger datasets, it's advisable to distribute data across multiple sheets or workbooks rather than cramming everything into a single sheet. This approach prevents clutter and simplifies data management and export. Remember to avoid placing multiple tables on one sheet, which complicates working with and exporting data.

Data should be organized in a tidy, rectangular layout, with rows used for subjects and columns for variables. The header (first row) should contain variable names—limit this to one row; otherwise, it will be confusing and hard to work with. If you would like to improve navigation, look at the formatting. Avoid using formatting to convey key information; instead, opt for separate columns. When it comes to naming variables, especially in cases with a large number of them, the following guidelines are essential:

1. **Avoid Spaces**: Always use underscores (`_`) instead of spaces in variable names, following the tidyverse style. For example, use `temperature_15_minutes` instead of `temperature 15 minutes`.
2. **Pick Descriptive and Concise Names**: Opt for names that are both short and informative. For instance, `tempc_15_min` is preferable to `temperature_15_minutes`, as it is concise and specifies the unit (Celsius). Although short names are preferred, don't be afraid to use longer names when needed.
3. **Impose Consistency Across Files**: Use the same variable names consistently across all files (e.g., always use `tempc` instead of alternating between variations like `temp_room_c` and `temp`).
4. **Avoid Special Characters**: Refrain from using emojis, LaTeX symbols, or other special characters, for example, ~,!,#,$,%,^,&,*,(,\, as they can cause errors.
5. **Structure Names for Easy Analysis**: Start variable names with a section code (e.g., "q#_" for question number, "p_" for personality questions, "s_" for status questions) to facilitate later analysis (easily select all personality questions `select(starts_with("p_"))`). For comment columns, append `_comment` to easily exclude them when necessary.

All the foregoing suggestions are also applicable to file names! Furthermore, it is also beneficial to name your files in a way that follows your preferred ordering. For example, if dates are crucial for sorting, placing the year-month-day format at the start of the file name can be efficient. When dealing with numerical sequences, include leading zeros to ensure correct ordering, such as `004 < 015 < 030 < 200` rather than `15 < 200 < 30 < 4`. However, exercise caution when sequentially naming files, as renaming multiple files to include new additions or to reorder them can become challenging.

Table 17-1 Examples of Good and Bad File Names

Bad	Good
table 1.tex	tbl01_temperature-vs-dictator.tex
My Thesis.docx	2023-12-24_thesis-for-review.docx

When setting up your spreadsheets, it's important to separate data entry sheets from final data sheets. Avoid performing calculations in raw data files due to potential rounding issues and unexpected errors in spreadsheet programs. The primary file should contain only the data, and any calculations or graphical analysis should be done on a copy of the file.

To ensure accuracy in data entry, use validation tools. Default validation capabilities are usually sufficient, but additional methods are available online. Apply validation to each column!

Finally, if multiple people are accessing the files, assign permissions judiciously. Protect cells that shouldn't be altered, and maintain regular backups. Investing time in setting up these safeguards can save weeks of potential rework.

17.3.2 Data Input

Once you have established your organizational structure, the next step is to input your data. Here, we will discuss best practices for data recording:

1. **Uniform Codes**: Use a consistent coding system for each category. For example, if categorizing by gender, choose one label (like "Male") and stick with it. Avoid variations such as "M," "male," or "MALE."
2. **Fixed Code for Missing Values**: Establish a standard symbol or code for missing data, like "NA" or a hyphen. Avoid using numbers (e.g., 77 or 999) for missing numeric data, and avoid leaving cells empty or using spaces, as these can be easily overlooked.
3. **Uniform Subject Identifiers**: Employ a consistent format for subject identifiers across all records. A composite key like "site_id_session_n_subject_n" is a good standard.
4. **Consistent Date Format**: To avoid confusion (like mistaking 2023/05/10 for either May 10 or October 5), use a consistent date format. The ISO 8601 standard (YYYY-MM-DD) is universally recognized and reduces ambiguity. Be vigilant with spreadsheet software like Excel or Google Sheets, which may auto-format dates; set column data types to text to avoid this issue.
5. **No Extra Spaces**: Pay attention to extra spaces within cells. They can lead to discrepancies, such as between "male" and "male."
6. **Complete Cells**: Ensure all cells are filled. Use a standard code for missing data to distinguish between genuinely missing data and accidental blanks. This clarity helps in understanding whether data are missing, were omitted due to an entry error, or carry a different implication.
7. **Single Datum per Cell**: Adhere to the principle of "tidy data" by ensuring each cell contains only one piece of information. If a column contains multiple values, consider splitting it into separate columns. Likewise, store notes or comments in distinct columns rather than embedding them within data cells. Avoid using font styles or colors to convey information; instead, encode these data in separate, clearly labeled columns.
8. **Consistent Terminology in Comments**: When adding comments, use uniform terminology to facilitate easy searching and categorization.
9. **No Merged Cells**: In preparation for data loading, unmerge any merged cells to prevent creating empty cells, which can complicate data processing and analysis.

After finalizing the structure and understanding your data, create a data dictionary. This should include the following items:

- Variable name
- Variable description
- Optionally, its category, name for visualization, and possible values.

Table 17-2 Data Dictionary Example

Category	Variable	Viz. Name	Values	Description
Personality	q1_p_agreeable	Agreeableness	1–5	Agreeableness assessment
Status	q2_s_perception	Self Perception	1–5	Self-rated social status

17.4 Administration

The administration process plays a pivotal role in shaping the efficiency, integrity, and overall success of research. The foundation of the success builds on a series of well-defined protocols and procedures that are critical to maintaining consistency and reliability in data collection. Key elements of this approach include the following:

1. **Experimenter Script**: Develop a detailed script for experimenters, outlining communication guidelines, participant interaction protocols, and responses to potential queries. This script should emphasize maintaining participant free will and effectively handling unforeseen circumstances.
2. **Adherence to Procedures**: On the day of data collection, it is imperative to strictly follow established procedures to ensure uniformity and efficiency across all sessions.
3. **Contingency Planning**: Prepare for unexpected scenarios with comprehensive contingency plans, ensuring they do not adversely affect data integrity.
4. **Administrative Flowchart**: Use a flowchart to clearly depict the research process, from participant arrival to data entry, delineating key steps and decision points.
5. **Feedback and Ethical Considerations**: Incorporate feedback mechanisms for continuous refinement and ensure all administration adheres to ethical standards, including participant confidentiality and data privacy.
6. **Team Training**: Conduct regular training and practice sessions for the research team to ensure proficiency with the research procedures and protocols.

Although the sampling and marketing strategies should ideally be included in the project plan, they are often decided on the spot. Therefore, it is important to document the sampling strategy you used, whether it be random sampling, stratified, snowball, volunteer, or something else. Additionally, record the marketing tools employed to reach participants and plan out sessions. Ensure that the sign-up process for events is simple and standardized, with the use of external services like Eventbrite recommended.

If these streamlined administrative practices are adopted, research projects can significantly improve in terms of operational efficiency, consistency in data collection, and adherence to ethical principles, leading to more reliable and valid research outcomes. As an example of administrative flow (Figure 17-3), I have included a version I used in an experiment, created in Lucid Chart. While it might seem confusing, it helps to understand the necessary steps and more effectively share them with others.

17.5 Accounting in Research

Effective accounting is as crucial in research as in running a startup. It demands rigorous bookkeeping and financial transparency. Key practices include the following:

1. **Transaction Documentation**: Meticulously document all transaction receipts. This involves keeping a detailed record of every financial transaction related to the research.
2. **Digital Recordkeeping**: Utilize a Google Sheet or similar tool provided by your organization for systematic recordkeeping. This should include detailed entries for each transaction.
3. **Cloud Storage**: Scan or take pictures of all receipts and financial documents. Upload these images to a cloud service, ensuring each file is named correspondingly to the table entries for easy cross-referencing.
4. **Currency Conversion**: Be diligent in recording currency conversions, especially when dealing with international transactions, to ensure accurate financial reporting.
5. **Regular Audits**: Conduct periodic audits to ensure accuracy and transparency of financial records.

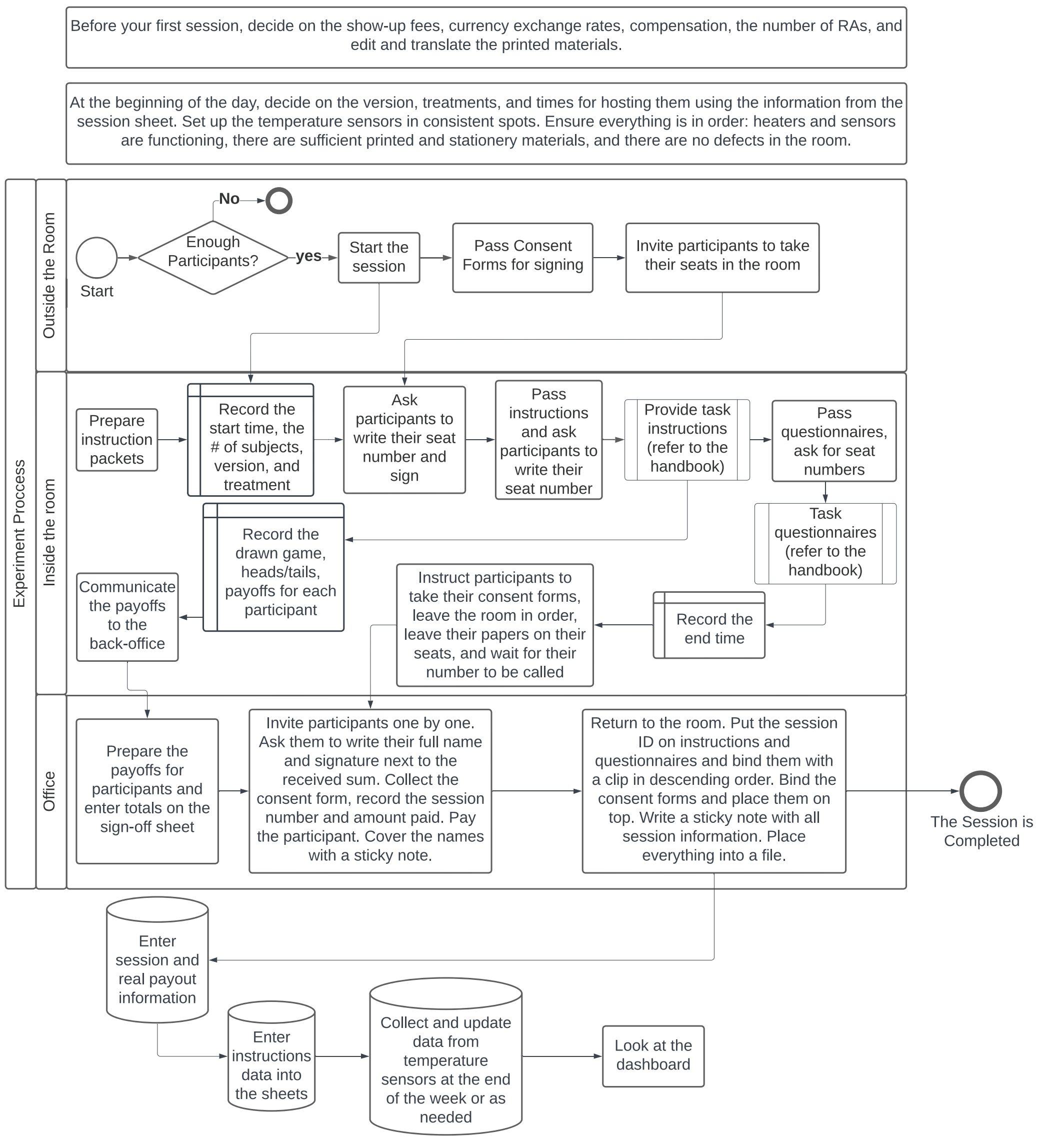

Before your first session, decide on the show-up fees, currency exchange rates, compensation, the number of RAs, and edit and translate the printed materials.
At the beginning of the day, decide on the version, treatments, and times for hosting them using the information from the session sheet. Set up the temperature sensors in consistent spots. Ensure everything is in order: heaters and sensors are functioning, there are sufficient printed and stationery materials, and there are no defects in the room.
Experiment Proccess
Outside the Room
Start
Enough Participants?
No
yes
Start the session
Pass Consent Forms for signing
Invite participants to take their seats in the room
Inside the room
Prepare instruction packets
Record the start time, the # of subjects, version, and treatment
Ask participants to write their seat number and sign
Pass instructions and ask participants to write their seat number
Provide task instructions (refer to the handbook)
Pass questionnaires, ask for seat numbers
Task questionnaires (refer to the handbook)
Record the end time
Instruct participants to take their consent forms, leave the room in order, leave their papers on their seats, and wait for their number to be called
Record the drawn game, heads/tails, payoffs for each participant
Communicate the payoffs to the back-office
Office
Prepare the payoffs for participants and enter totals on the sign-off sheet
Invite participants one by one. Ask them to write their full name and signature next to the received sum. Collect the consent form, record the session number and amount paid. Pay the participant. Cover the names with a sticky note.
Return to the room. Put the session ID on instructions and questionnaires and bind them with a clip in descending order. Bind the consent forms and place them on top. Write a sticky note with all session information. Place everything into a file.
The Session is Completed
Enter session and real payout information
Enter instructions data into the sheets
Collect and update data from temperature sensors at the end of the week or as needed
Look at the dashboard

Figure 17-3 Administration Flow

17.6 Communication

Effective communication is integral to the success of any experiment. Detailed documents should be created providing clear instructions on the use of each component of your experiment's system. These documents should include:

1. **Procedure Manuals**: Outlining step-by-step processes for sampling, administration, data collection, and handling in clear, concise language.
2. **Training Materials**: Practical demonstrations of procedures to help your field agents familiarize themselves with the processes before starting data collection.

3. **Reference Guides**: Quick-reference materials summarizing key points from the procedure manuals and training materials.
4. **Data Entry Guidelines**: Providing clear instructions on how to enter data into your electronic system.
5. **Data Protection Policies**: Explaining how personal data will be protected.
6. **Quality Control Procedures**: Outlining strategies for ensuring the quality of the data collected.
7. **Feedback Mechanisms**: Explaining how feedback will be collected and used to improve future research.

By ensuring these documents are readily accessible to your field agents, your experiment can run smoothly, generating high-quality, reproducible data.

A well-documented experiment is the key to successful research. By adhering to these guidelines, you can ensure your research is reproducible, traceable, and of high quality.

17.7 Summary

This chapter emphasized the importance of comprehensive documentation in research to ensure consistent data collection and adherence to procedures. Key principles include traceability, accessibility, consistent formatting, and security. Documentation should follow a hierarchical structure encompassing project, site, session, and subject levels. Practical tips cover both physical and electronic documentation, with a focus on data protection and quality control. Proper data organization in spreadsheets, along with thorough documentation of field activities, administration, and finances, is crucial. Effective communication and training are essential for maintaining high-quality, reproducible research. In the next chapter, we will explore the tools you can use to conduct surveys and examine APIs and how to utilize them for extracting data from the internet.

Chapter 18
APIs

Application Programming Interfaces (APIs) are essential tools for modern data analysis, allowing seamless interaction with servers to retrieve and manipulate data efficiently. In this chapter, we will cover the fundamentals of APIs, learn how to make simple API requests, and explore their application in R.

18.1 API Basics

What exactly is an API? Consider this analogy: when you enter a restaurant, you sit down and order food from the menu. You don't go into the kitchen to instruct the chef—that would be chaotic and inefficient. Instead, your order is delivered by a waiter.

In this analogy, you're the client, the kitchen is the server, your order is the request, the food prepared is the response, and the waiter is the API. Just as a waiter knows how to take your order, convey it to the kitchen, and serve you, an API dictates how to make requests, which requests are valid, how responses are delivered, and in what format. The waiter shields you from the kitchen's complexity and heat, similarly to how an API shields you from the intricacies of server-side operations. You don't need to understand the chef's cooking process; you just need to relay your order to the waiter and wait for your meal. Likewise, using an API doesn't require you to understand server-side procedures; you just need to make the request correctly and handle the response received.

Communicating with an API involves sending a request with the required data. Here are several types of requests you can make:

1. **GET Request**: This is like asking a waiter for a meal. It involves requesting data without making changes. In the context of APIs, a GET request fetches information from the server.
2. **POST**: This is akin to adding a new dish to the menu. It involves giving the API specific instructions to ask the server to add a new item. In API terms, a POST request transmits data to the server, generating a new entry.
3. **PUT**: Suppose you want to change your order from a burger to a pizza. This action corresponds to a PUT request, which updates the entire record.
4. **PATCH**: Suppose instead of replacing your burger, you just want to change the cheese from American to Swiss. This action corresponds to a PATCH request, used to update a specific part of existing information.
5. **DELETE**: Imagine canceling and removing an item from your order after receiving your meal. This action aligns with a DELETE request, which deletes existing data from the server.

N. Tkachenko, *Data Insight Foundations*, https://doi.org/10.1007/979-8-8688-0580-6_18

These basic types of HTTP requests are similar to your interaction with a waiter at a restaurant. Just as clear communication with the waiter is necessary for a correct order, using the appropriate request type is crucial when dealing with APIs to obtain the required data or perform the desired operation.

Fundamentally, an API request is similar to your browser downloading a webpage when you open a link. For instance, you can fetch data from the US Census American Community Service 2021 by following this link:
https://api.census.gov/data/2021/acs/acs5?get=NAME,group(B01001)&for=us:1

API Uniform Resource Locators (URLs) are constructed in a structured format consisting of several components. The **protocol** (e.g., `https://`) indicates how data are transferred securely. The **base URL** (e.g., `api.census.gov`) points to the main address of the API server. The **path** (e.g., `/data/2021/acs/acs5`) specifies the resource being accessed, in this case year, and specific dataset. Finally, **query parameters** (e.g., `?get=NAME,group(B01001)&for=us:1`) provide additional instructions to the server, structured as `key=value` pairs and separated by the `&` symbol. Together, these components form a complete URL.

Usually, an interface is used to interact with an API, rather than constructing a text link. There's a plethora of them, but I'm partial to HTTPie,[1] a CLI tool that also provides online and local applications. Follow their installation instructions on the website.

Whether you use `https` in your terminal or go to HTTPie's website, you can easily make your first API request. This command will return a header with information and a return message formatted as a JSON string:

```
https httpie.io/hello
```

```
{
    "ahoy": [
        "Hello, World! Thank you for trying out HTTPie",
        "We hope this will become a friendship
    ],
    "links": {
        "discord": "https://httpie.io/discord",
        "github": "https://github.com/httpie",
        "homepage": "https://httpie.io",
        "twitter": "https://twitter.com/httpie"
    }
}
```

Note 18.1 JavaScript Object Notation (JSON) is a light, readable data format that both humans and machines can readily parse and generate. Its structure consists of key—value pairs and arrays, making it a common choice in web APIs and data interchange.

An API request mainly includes three components: the request endpoint, header, and body.

1. **API Endpoint**: This refers to a specific URL or Uniform Resource Identifier (URI) of a web service, serving as the access point for making requests and receiving responses from the API.
2. **Request Header**: This contains supplemental details about the request, such as metadata or server instructions. It includes information like the HTTP method, content type, authentication credentials, and more. Headers provide context or control the behavior of the API request.

[1] https://httpie.io/cli

3. **Request Body**: This carries any data or parameters that need to be sent to the server with the API request. Primarily used for methods like POST, PUT, or PATCH, the body can contain various data formats such as JSON, XML, form data, or binary data, depending on the API's requirements.

Together, the request header and body allow for effective communication with an API, specifying the necessary information, desired action, data to be sent, and any additional instructions or authentication details. Let's add a header and body to a request:

```
https POST pie.dev/post X-API-Token:123 name=John
```

Here we're using a POST method to send data to an API endpoint `pie.dev/post`, adding an `X-API-Token:123` header, and specifying a `name` field value as `John` in the body. You will receive a JSON response with details on the successful POST request!

If this seems overwhelming, or you prefer a more convenient way to interact with APIs, I recommend using the HTTPie application or a service like PostMan, which provides a GUI.

18.2 Utilizing APIs in R

In R, APIs open the door to a wealth of intriguing datasets for your data analysis projects. Let's explore API implementation in R using the `httr2` package.

```
# install.packages("httr2")
library(httr2)
```

Let's start interacting with APIs using the `request()` to build a request and `req_perform()` to send a request from the `httr2` package. We'll extract data from the Gutendex API,[2] offering access to Project Gutenberg[3] ebook metadata. The following code fetches data from the API, particularly authors alive before 500 BCE (`?author_year_end=-499`):

```
# Create a request to the Base URL
req <- request("https://gutendex.com/books") |>
  req_url_query(author_year_end = -499) # Add a query parameter

req_dry_run(req) # Show the request being sent (dry run)
```

```
#>  GET /books?author_year_end=-499 HTTP/1.1
#>  Host: gutendex.com
#>  User-Agent: httr2/1.0.0 r-curl/5.2.0 libcurl/8.6.0
#>  Accept: */*
#>  Accept-Encoding: deflate, gzip
```

```
res <- req |> req_perform() # Perform the request
```

The `request()` function initializes a new request object, and `req_url_query()` adds query parameters to the URL. The `req_dry_run()` function shows the request being sent without actually

[2] https://gutendex.com/

[3] https://www.gutenberg.org/

sending it. The `req_perform()` function dispatches the HTTP GET request and stores the API response in the `res` variable. Inspect the response by printing the `res` object:

```
print(res)
```

```
#> <httr2_response>
#> GET https://gutendex.com/books/?author_year_end=-499
#> Status: 200 OK
#> Content-Type: application/json
#> Body: In memory (34502 bytes)
```

The next step is to parse the JSON response:

```
data <- res |> resp_body_json()
```

The `resp_body_json()` function directly parses the response content into a structured R object, such as a list or a data frame.

Note 18.2 Use `glimpse()` to examine the structure of the response. However, be aware that sometimes the response is extremely long, as it is in this case, so I didn't include it.

Look into the data by accessing its elements. For instance, you can display the following book titles:

```
head(purrr::map_chr(data$results, "title"), n = 5)
```

```
#> [1] "The Iliad"
#> [2] "The Odyssey: Rendered into English prose for the use of those who cannot read
#>     the original"
#> [3] "La Odisea"
#> [4] "The Iliad"
#> [5] "The Odyssey"
```

Voilà! You've successfully made an API request in R and fetched data for further exploration.

To fine-tune our API request, we can use additional query parameters to target authors alive before 500 BCE and their works in epic poetry in French. This can be achieved by adding `topic = "Epic Poetry"` and `languages = "fr"` to your URL query header, similar to how we added `author_year_end = -499`.

In some cases, you may need to use a `POST` request, especially when sending a request body. In `httr2`, you can achieve this by incorporating a request body using the `req_body_*()` family of functions, such as `req_body_json(req, list(key1 = "value1", key2 = "value2"))`, or by using `req_method()` to specify a different method. Always refer to the API documentation for specific requirements and the headers available.

Now that you have some understanding of what APIs are and how to communicate with them, we will look at different packages that let us interact with useful services such as Qualtrics, Google Sheets, and OpenAI.

18.3 Qualtrics API

Instead of downloading data through Qualtrics every time we want to update our survey results, we will load the data using Qualtrics API. For a complete guide, refer to the package's documentation and vignettes.[4] If your university provides access to Qualtrics, you might still need to separately request API access from IT services. First, we need to install a package, `qualtRics`.

```r
# install.packages("qualtRics")
library("qualtRics")
```

The package contains three core functions:

- `all_surveys()` shows surveys you can access.
- `fetch_survey()` downloads the survey.
- `read_survey()` reads CSV files you downloaded manually from Qualtrics.

It also contains a number of helper functions, including the following:

- `qualtrics_api_credentials()` stores your API key and base URL in environment variables.
- `survey_questions()` retrieves a data frame containing questions and question IDs for a survey;
- `extract_colmap()` retrieves a similar data frame with more detailed mapping from columns to labels.
- `metadata()` retrieves metadata about your survey, for example, questions, survey flow, and number of responses.

Note 18.3 You can only export surveys that you own or to which you have been given administrative rights.

If you have received API access, you can now connect to the API. To get `api_key` and `base_url`, go to the Qualtrics homepage → Account Settings → Qualtrics IDs (for older version). Under "API" click "Generate Token," and you will be issued a token. Copy this string and enter "YOUR_API_KEY." Then look at "User" module, copy "Datacenter ID," and add ".qualtrics.com" after it. Your `base_url` should look something like this: "lad2.qualtrics.com."

```r
qualtrics_api_credentials(
  api_key = "YOUR_API_KEY",
  base_url = "YOUR_BASE_URL",
  install = TRUE,
  # overwrite = TRUE # If you need to update your credentials
)
```

After `qualtrics_api_credintials` stores your credentials, you can use `all_surveys()` to fetch information on your surveys.

[4] https://cran.r-project.org/web/packages/qualtRics

```
head(all_surveys(), 5)
```

```
#>   # A tibble: 9 x 6
#>     id                 name              ownerId lastModified creationDate isActive
#>     <chr>              <chr>             <chr>   <chr>        <chr>        <lgl>
#>   1 SV_0rearXjH2Ri6umq Student Satisfa~ UR_2tz~ 2023-01-29T~ 2023-01-17T~ FALSE
#>   2 SV_2gWVWLn2vCCDJGu Luvuyo            UR_2tz~ 2022-05-03T~ 2022-05-03T~ FALSE
#>   3 SV_3h0HPHQbJStqb8a Pilot (USA) - G~ UR_2tz~ 2022-05-03T~ 2022-05-03T~ FALSE
#>   4 SV_3Q011O6gNqwn1Nc Pen Experiment    UR_2tz~ 2022-12-09T~ 2022-12-06T~ FALSE
#>   5 SV_4YDoTFLN61nKecS Pen_Showcase      UR_2tz~ 2023-01-29T~ 2023-01-29T~ TRUE
```

Once you select the questionnaire you want, you can refer to it using `id`. If you wish, redownload the dataset `force_request = TRUE`; otherwise, it will load previously saved downloads. The dataset will likely have many columns depending on the number of questions.

```
survey_data <- fetch_survey(
  surveyID = "SV_bJIs8lwz4CfAAgS",
  verbose = FALSE,
  force_request = TRUE
)

survey_data %>%
  select(StartDate, `Duration (in seconds)`,
         Finished, Gender, offer,
         decision...81, participantRole) %>%
  glimpse()
```

```
#>   Rows: 22
#>   Columns: 89
#>   $ StartDate           <dttm> 2023-02-07 16:00:52, 2023-02-07 16:01:17, 2~
#>   $ EndDate             <dttm> 2023-02-07 16:04:08, 2023-02-07 16:04:08, 2~
#>   $ Status              <chr> "IP Address", "IP Address", "IP Address", "I~
#>   $ IPAddress           <chr> "138.202.129.171", "138.202.129.164", "138.2~
```

If you want to see the text of the questions, use `survey_questions()`.

```
head(survey_questions(surveyID = "SV_bJIs8lwz4CfAAgS"), n = 5)
```

```
#>   # A tibble: 5 x 4
#>     qid   qname        question                                      force_resp
#>     <chr> <chr>        <chr>                                         <lgl>
#>   1 QID25 Introduction "Welcome to the <strong>University of San F~ FALSE
#>   2 QID26 Consent      "Do you agree to participate in the survey?" TRUE
#>   3 QID21 Gender       "What is your gender"                         TRUE
#>   4 QID23 Name         "What is your Name?"                          FALSE
#>   5 QID22 Competitive  "Would you consider yourself competitive?"    FALSE
```

18.4 Integrating Google Services with R

In this section, we will explore the integration of Google services with R, leveraging the versatility of Google's tools to enhance our data collection and analysis capabilities.

Google Sheets is a highly versatile, user-friendly, cloud-based tool, particularly effective for storing data collected via Google Forms. It is a great solution for data entry in projects that involve gathering physical data. `googlesheets4` is an R package that interacts with the Google Sheets API, offering a host of features to analysts and data scientists. Here are some common scenarios:

1. **Data Input**: Data are often input into Google Sheets by various people, thanks to its collaborative nature. `googlesheets4` allows you to pull these data directly into R.
2. **Real-Time Data Analysis**: If data are being continuously updated, `googlesheets4` allows real-time analysis by accessing and analyzing the latest data.
3. **Data Reporting**: Results from your analysis or model predictions can be written back to Google Sheets, making them accessible to nontechnical stakeholders.
4. **Data Sharing**: Google Sheets makes data sharing easy. Your data can be accessed from anywhere, anytime.

The `gargle` package simplifies authentication with Google APIs and safely manages authentication tokens. It is used by all R packages interacting with Google's services. Using `gargle` enhances workflow efficiency and security by storing tokens securely and simplifying reauthentication in future sessions.

```r
options(gargle_oauth_cache = ".secrets")
googlesheets4::gs4_auth()
list.files(".secrets/")
gs4_auth(cache = ".secrets", email="name@mail.com")
```

Note 18.4 Don't forget to add a .secrets folder to .gitignore!

To pull data from Google Sheets into R, use the `read_sheet()` function, providing the document's URL or identifier and the specific Sheet's name.

```r
library(googlesheets4)

url <- "https://docs.google.com/spreadsheets/d/your_spreadsheet_id_here"
data <- read_sheet(url, sheet = "Your Sheet Name")
head(data)
```

Writing data back to Google Sheets is just as straightforward using the `write_sheet()` function.

```r
data_to_write <- data.frame(
  column1 = c("Data1", "Data2"),
  column2 = c("Data3", "Data4")
)

write_sheet(data = data_to_write, ss = url, sheet = "Your Sheet Name")
```

The `googledrive` package in R allows you to interact with Google Drive, making it easy to read, write, and manage files.

```r
library(googledrive)
drive_auth(cache = ".secrets")
print(drive_ls())
file <- drive_get(path = "Your File Name")
data <- read_csv(drive_download(file, overwrite = TRUE))
```

To write files to Google Drive, save the file locally, then upload it using `drive_upload()`:

```r
data_to_write <- data.frame(
  column1 = c("Data1", "Data2"),
  column2 = c("Data3", "Data4")
)

write.csv(data_to_write, "data_to_write.csv")
drive_upload("data_to_write.csv", path = "FolderName/data_to_write.csv")
```

Note 18.5 You can also have a Google Drive folder on your computer and simply write and read files from it.

Remember to responsibly handle your files, especially those containing sensitive data. Numerous Google services like Google Maps, Google Earth, Google Cloud, and others are readily accessible from within R. A quick online search will reveal various resources for your projects, and there's likely a preexisting R package to simplify your access.

18.5 OpenAI's API

At the time of publication, there are now many more options for using LLMs in R, including `elmer`, `mall`, `tidychatmodels`, and `gander`. It would be a criminal offense not to show how to use OpenAI's API's inside of R. Although I have to admit package implementation in Python is fuller and all around better, I still recommend seeing what you can do in R.

`openai` is an R package that serves as a wrapper for OpenAI's API services. It includes support for various endpoints such as Models, Completions, Chat, Edits, Images, Embeddings, Audio, Files, Fine-tunes, and Moderations.

```r
# install.packages("openai")
library("openai")
```

Note 18.6 Read the official OpenAI documentation—it's not too technical and is well worth your time.

To access the OpenAI API, you must have an API key. Start by registering for the OpenAI API at https://openai.com/api/. After signing up and logging in, navigate to https://platform.openai.com > **Personal** > **View API keys** > green **Copy** button to copy the key.

The package will look for `OPENAI_API_KEY` in the environment variables. To set this as a global environment variable, execute the following command, replacing `YOUR_KEY` with your actual key:

```
Sys.setenv(
    OPENAI_API_KEY = 'YOUR_KEY'
)
```

Note 18.7 When using GitHub, you should specify the key in `.Renviron` and add it to your `.gitignore` file to keep the key secure.

With GPT-4, you can unlock a wide range of functionalities like summarizing, classifying, and editing text. Here's how to generate a chat completion:

```
chat <- create_chat_completion(
  # Specify a model to use
  model = "gpt-4o",
  # Pass a list of messages based on roles.
  messages = list(
    list(
      "role" = "system",
      "content" = "As an expert summarizer, condense the following paragraph into
↪   one
      succinct sentence."
    ),
    list(
      "role" = "user",
      "content" = "R is a programming language for statistical computing and
↪   graphics
        supported by the R Core Team and the R Foundation for Statistical
↪   Computing.
        Created by statisticians Ross Ihaka and Robert Gentleman, R is used among
↪   data
        miners, bioinformaticians and statisticians for data analysis and
↪   developing
        statistical software. The core R language is augmented by a large number
↪   of
        extension packages containing reusable code and documentation."
    )
  )
)

chat$choices$message.content
```

```
#>  [1] "R is a programming language developed by Ross Ihaka and Robert Gentleman, widely used
     ↪  for statistical computing and graphics, enhanced by numerous extension packages for
     ↪  data analysis and statistical software development."
```

You can create images based on textual descriptions. Here's an example of generating a grayscale cartoon:

```
image <- create_image("Frog-blossom cartoon in greyscle color",
                      size = "256x256")
download.file(image$data$url, "images/openai_image.png", mode = 'wb')
```

Figure 18-1 OPENAI Image Generation

Moreover, you can get audio transcription using the Whisper model:

```
create_translation(file = "data/whisper_demo.m4a", model = "whisper-1")
```

```
#>   $text
#>   [1] "Can you hear the whisper? Whispering?"
```

18.6 Summary

APIs offer a powerful and efficient way to interact with various data sources and services, making them indispensable for modern data analysis. This chapter has provided a comprehensive overview of what APIs are, how to make different types of API requests, and how to utilize them effectively in R. In the next part, we will explore the world of data visualization!

Chapter 19
Data Visualization Fundamentals

We are about to embark on a wonderful journey into the world of data visualization. This chapter was not originally planned, but the more I thought about the importance of basic psychophysics concepts, their relationship to design, and data visualization, the more I felt compelled to include them.

Let's start with a simple question: What is data visualization? Data visualization is the graphical representation of information and data.

Next question: What are the goals of data visualization?

1. Effective communication of information and insights
2. Analysis and exploration
3. Decision-making

It may sound obvious, but the complexity lies in how we achieve these goals. The main power of data visualization lies in its ability to present large, multidimensional datasets in a succinct, easy-to-consume form. Our memory has limits, as we can typically hold about 5 ± 2 things in our heads at one time. Visuals can help us reduce and simplify information. So how do we reach the point where our visuals help with better understanding of the data?

19.1 Perceptual Processing

To answer that question, we'll start with how humans perceive the world or, more accurately, how our brains process visual information. Understanding perceptual processing will help us distinguish why certain visual techniques are more effective than others and reveal the inherent limitations in human visual perception. This knowledge of perceptual processing is instrumental in guiding design decisions.

Visual perception occurs in two phases, similar to Daniel Kahneman's System 1 and System 2 thinking [21]: The first phase is quick and automatic, while the second is slower and more deliberate. The human visual system has an extraordinary ability to detect certain elements instantly and simultaneously as soon as an object is seen without deliberate attention or conscious effort. This **preattentive processing** stage is primarily concerned with the immediate and parallel perception of fundamental visual attributes such as orientation, color, texture, and motion [30]. During this phase, visual data are temporarily held in iconic memory, a form of transient storage that is constantly refreshed with new visual input.

In contrast to the instantaneous nature of preattentive processing, the subsequent stage of **serial processing** involves a more deliberate and sequential approach to processing visual information. It is slower, allowing for a more thorough and detailed analysis of the visual data. This stage engages

N. Tkachenko, *Data Insight Foundations*, https://doi.org/10.1007/979-8-8688-0580-6_19

both working memory and long-term memory, facilitating the assimilation of new information with existing knowledge stored in the brain.

19.1.1 Preattentive Processing

Our current focus is on System 1, as it is the system that “sees” and directs our attention. Intriguingly, certain elements within a visual scene are processed preattentively, without conscious attention, in less than 200–250 milliseconds, which is the same duration as eye movement. This rapid processing is managed in parallel with the low-level vision system, which allows for the simultaneous analysis of multiple visual elements. The preattentive system performs several tasks:

- **Target detection**: Accurately detect the presence or absence of a “target” element with a unique visual feature within a field of distractor elements.
- **Boundary detection**: Rapidly and accurately detect a texture boundary between two groups of elements, where all the elements in each group have a common visual property.
- **Region tracking**: Track one or more elements with a unique visual feature as they move in time and space.
- **Counting and estimation**: Count or estimate the number of elements with a unique visual feature.

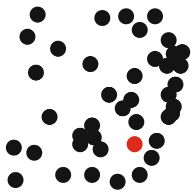
Let’s consider an example with a wall of numbers. Try to count all the occurrences of the number 2. Now try it again with the number 2 highlighted. Finding the number 2 becomes much faster, doesn’t it? Similarly, can you rapidly detect the presence of a red circle? You probably noticed it even before I asked you to look for it.

1	2	7	9	8	3
5	6	4	1	7	9
6	7	4	2	3	6
2	7	5	7	9	0
1	3	8	5	8	3
2	0	0	3	7	4

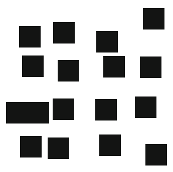
(a) Count all 2

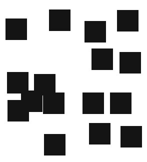
1	**2**	7	9	8	3
5	6	4	1	7	9
6	7	4	**2**	3	6
2	7	5	7	9	0
1	3	8	5	8	3
2	0	0	3	7	4

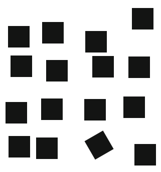
(b) Count all 2 (bold)

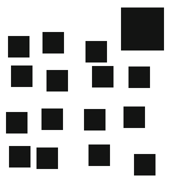

(c) Find a red circle

The variety of visual features that we can detect is remarkably large [15], and we already covered hue with the red dot example. Here are some types that, in my opinion, are most often used in visualizations:


(a) Length


(b) Density


(c) Orientation

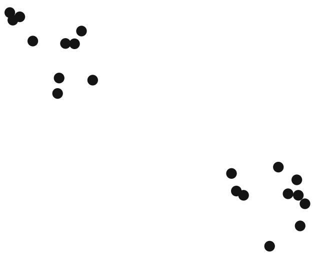

(d) Size

A few other notable types are intensity, curvature, terminators, intersections, and everything related to 3D and motion. I highly encourage you to visit an online version of Healey’s paper[1] to learn more about preattentive processing.

[1] https://www.csc2.ncsu.edu/faculty/healey/PP/

19.1.1.1 Gestalt Principles

Gestalt principles explain how the human brain perceives visual patterns from grouped elements. These principles encompass concepts like proximity, similarity, continuity, closure, connection, and enclosure.



Proximity: When objects are close together, we often perceive them as a group

Figure 19-1 Proximity

Similarity: When objects share similar attributes (color, shape, etc.), we often perceive them as a group

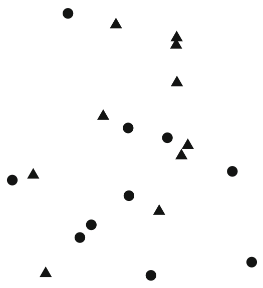
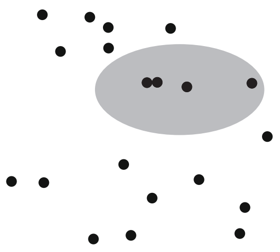


Figure 19-2 Similarity



Enclosure: When objects are surrounded by a boundary, we often perceive them as a group

Figure 19-3 Enclosure

Closure: Sometimes partially open structures can still be perceived as a grouping metaphor (e.g., "[...]")

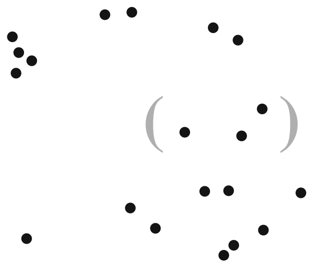
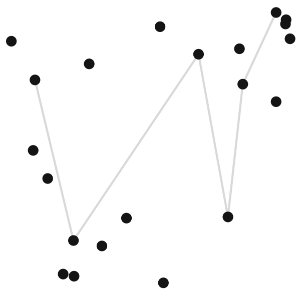


Figure 19-4 Closure



Connectivity: When you draw curves or lines through data elements, this is often perceived as creating a connection between them

Figure 19-5 Connectivity

19.1.1.2 Feature Hierarchy

Multiple features such as color and shape can represent multiple types of data in a single image. However, it's important to make sure that these visual features don't mix up and hide the data we want to show. Think of it as trying to find a red apple in a bowl full of green apples—it's easy because the color stands out.

Sometimes our eyes favor one visual feature over another. For example, when we observe shapes, colors can be distracting and make it harder to discern shape patterns. However, if the colors are uniform, the shapes become more prominent. Try to identify groups of points in Figure 19-6. Which groups do you notice first?

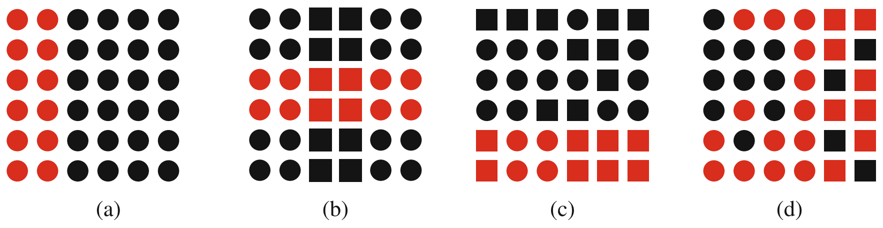


Figure 19-6 Examples of Feature Hierarchy

Figure 19-6 illustrates the hierarchy of hue and form features in perception:

(a) Vertical hue boundaries are easily noticed with constant form, demonstrating the preattentive detection of hue differences.
(b) Color and shape boundaries are easily identifiable to the human eye. However, you might have noticed that color is perceived before shape.

(c) Vertical hue boundaries stand out against varied forms, showing hue's preattentive visibility despite form changes.
(d) However, vertical form boundaries are not as easily preattentively identifiable amid random hue variations, highlighting the challenges in detecting form changes without focused attention.

So when we're deciding how to visually represent our data, we should choose features that make the most important information stand out. This way, we avoid obscuring the data we want to highlight. Common ways to visually encode numbers, from most easily perceived to least, include the following:

1. Position along a common scale, axis, and baseline
2. Position along nonaligned axes
3. Length, direction, angles of relative lines/slope
4. Area
5. Volume, curvature, arcs/angles within a shape
6. Color or shading

Consider a scenario where we encode two separate variables in a discrete dataset using two distinct visual properties, handling three is more complex, say, color and shape. For instance, let's go back to Figure 19-6. When presented with red squares, black squares, red circles, and black circles, will our perception categorize them into four distinct groups based on both color and shape, or will we primarily perceive them as two broader groups divided by color (red/black) and shape (square/circle)? The critical question is how these different properties interact. Will they make it easier or harder to differentiate and understand the data? There are two categories of how the two properties relate:

- **Integral**: Here, the two properties are perceived as a whole. The interaction between them is such that they are viewed as a single, unified element.
- **Separable**: In this case, each property is judged independently. The viewer can easily separate and evaluate each characteristic without interference from the other.

As an example, I have prepared four cases ranging from integral to separable in Figure 19-7. The first example uses the LAB color system, which we will cover in a few chapters. It encodes two values on the scales of red–green (A) and blue–yellow (B).[2] This system is so convoluted as to seem like a random collection of colors (which it certainly is). The second example is slightly better, as values are encoded by the height and width of the rectangles, making it appear as if there are small, tall, big, and wide rectangles as separate groups. The third example, using shape and color, is much better, but it can still be perceived as having four distinct groups. Finally, the fourth example, using grouping and color, leaves no doubt that there are two properties: group membership and color.

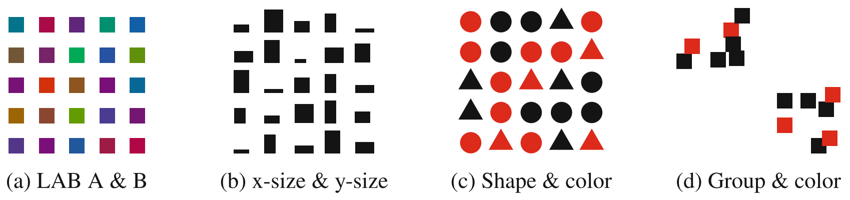


Figure 19-7 Integral vs. Separable

[2] L is for lightness, but this parameter isn't used in the visualization.

19.1.1.3 Nonlinear Perception

Perception is not uniformly linear. We perceive some things accurately, like length, while we tend to underestimate other things, such as the true difference between two values due to our ability to sense ratios.

For example, we are pretty good at estimating lengths and temperatures after a little practice. However, in some domains, we tend to underestimate differences and miss the ratios—for instance, when comparing the brightness of two light sources, a doubling in intensity may not be perceived as twice as bright. Our perception adjusts to the strength of the signal; for instance, our eyes adjust to bright daylight and to darkness in a room.

Visualization is about turning numbers into pictures. However, the goal is for the user to be able to translate these pictures back into numbers accurately. One tricky task in visualization is translating numbers into areas. This is not only difficult to decipher but also to encode. What are we comparing when we look at areas—radius, area, or sensation? Let's consider an example. Say there are three gray circles and one white circle. Which gray circle represents a number that is twice as big as the white one?

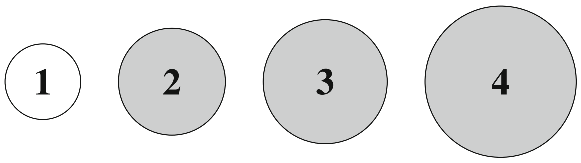


Figure 19-8 Nonlinear Perception

Curiously enough, all three are correct! The second circle has twice the area of the original, the third circle appears to have twice the area according to the Stevens law (which we will discuss shortly), and the fourth circle has twice the radius. Yes, it is indeed confusing! The way we perceive proportional differences in sensation is not a one-to-one relationship with the measurement. This is precisely the idea that Stevens was interested in and formulated Stevens' law in 1960:

$$s(x) = ax^b$$

where s is sensation, x is attribute intensity (ratio), a is a multiplicative constant (usually 1 or close to 1), and b is the exponent.

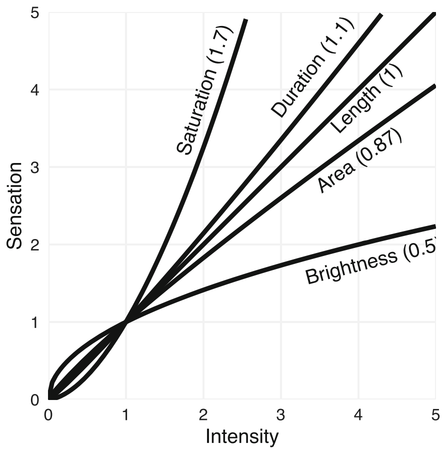
If $b > 1$: overestimate

If $b < 1$: underestimate

Experimental results for b center around 1 for lengths, 0.9 for area, and 0.7 for volume.

So, how would we apply this apparent scaling in practice? Let's consider an example where we want to draw circles of different areas. Imagine the largest circle has an area twelve times bigger than the smallest one. To counteract our tendency to underestimate, we could multiply the area by approximately $\sqrt[b]{s(x)} \div x$ or $\sqrt[0.87]{12} \div 12 \approx 1.44$. However, it's important to consider the context and whether these adjustments will truly benefit our visualization. I haven't seen anyone apply this outside of mapping software, as there are a few areas (pun intended) where you can't avoid using circles for visualization. Nonetheless, if you were to make these adjustments, this is how it would look, using beta from [13].

Figure 19-9 Stevens' Law



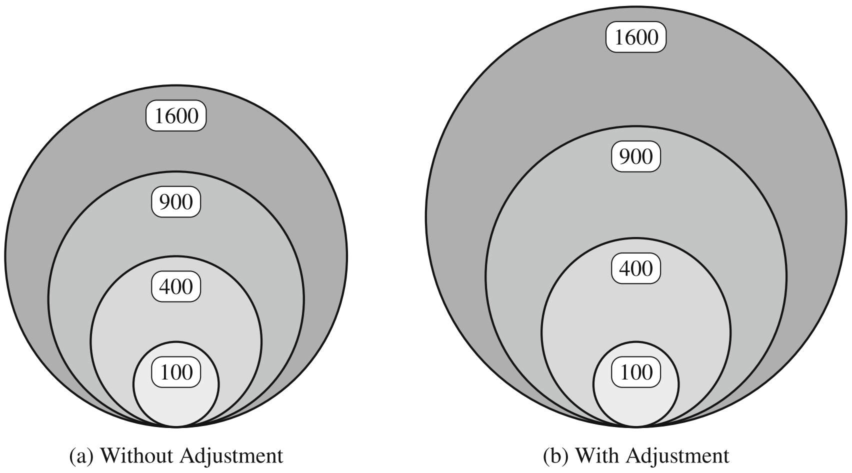


Figure 19-10 Stevens' Law Area Example

19.2 Visual Encoding

After learning about the basics of how our eyes and brain quickly process visuals, it's time to talk about visual encoding and associated tasks:

- Turning numeric data into visuals
- Turning categorical data into visuals
- Showing the differences between pieces of information
- Showing how data or information relates to some context

Humans have different types of memory like long-term, working, verbal, and visual memory, each stored in various parts of the brain. Our working memory, which holds information temporarily, can only retain around 5 ± 2 chunks of information at a time. Visualizations can help group or "chunk" information together, making it easier for us to process and remember.

It's essential to keep related information close together in a visualization to avoid fragmentation, which is when we separate things that should be remembered together. By doing this, we help people remember and understand the information better. We can highlight or annotate important points to draw attention to them.

Good design in visualizations helps people quickly understand what they're looking at. It's not just about putting numbers into shapes but making those shapes tell a story. A well-designed visualization will help people easily scan through information and go deeper if they want to.

Note 19.1 The goal is to make it easy for the reader to decode visual information without making errors.

Different visual attributes like position, length, angle, and color help represent data. Some attributes, like position and length, are better for showing precise data, while others, like color and size, are less precise. It's crucial to match the right attribute with the type of data we're showing.

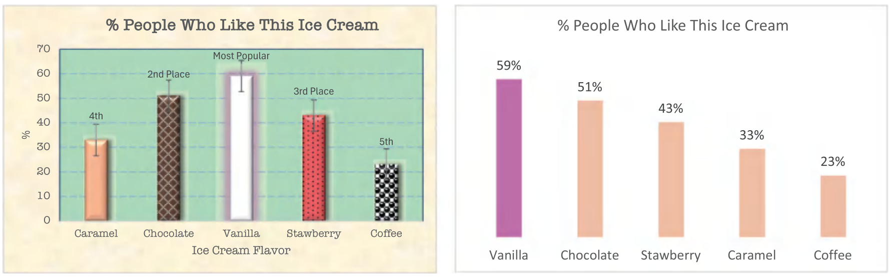
Using familiar chart types, intuitive colors, and shapes helps make the visualization easy to understand. Avoid making people remember too many new symbols or having long legends, as it can be overwhelming.

Lastly, knowing who will be looking at the visualization will inform your decisions, resulting in visuals that are easy to understand, remember, and interpret.

19.2.1 Evaluating Graphs

How do we evaluate our graphics to enlighten and engage our audience, rather than deceive them? Several practical frameworks have been proposed for this purpose.

One of the popular ideas in data visualization is the data-ink ratio [31], introduced by Edward Tufte. This idea is all about keeping things simple and getting rid of any extras that don't help convey the main message. As Tufte suggests, it's good to "erase non-data-ink, within reason" and "erase redundant data-ink, within reason." It might be tempting to remove too much, but it's better to take it slow. Trust your gut feeling on whether the chart still makes sense. The suggestions we'll discuss next are based on having clean and clear graphics.



(a) Before (b) After

Figure 19-11 Data Ink Example

In "Levers of Chart-Making" [11], Ware proposes several critical factors that influence the effectiveness of a graph. These factors include the **speed to primary insight**, which measures how quickly an audience can understand the key message from a graph. **Granularity** refers to the level of detail presented in chart data. The dichotomy between **explore and explain** reflects whether the visualization is designed for interactive exploration by users or is accompanied by explanatory content. The **dry or emotional** aspect deals with the tone of data presentation, ranging from serious to informal, with the potential to make presentations more emotionally engaging to attract a less data-savvy audience. Lastly, the balance between **ambiguity and accuracy** is crucial, determining how clearly a chart conveys its message versus any intentional ambiguity.

The “cognitive load” framework, introduced by Sibinga [27], categorizes cognitive demands into three types. The **intrinsic load** concerns the complexity inherent in the data itself, with aspects like the type of data (quantitative vs. qualitative), data certainty (certain vs. uncertain), clarity of data categories (precise vs. ambiguous), and the data’s relatability to everyday life (concrete vs. abstract). The **germane load** focuses on the audience’s readiness to process information, including people’s initial connection to the visualization (intentional vs. coincidental), the time they have to view it (slow vs. fast), their familiarity with the subject (expert vs. novice), and their comfort with the data format (confident vs. anxious). Finally, the **extraneous load** addresses how new information is presented considering aspects like the commonality of the chart type (common vs. rare), precision of chart values (accurate vs. approximate), information density (concise vs. detailed), and whether the reporting of the data is self-explanatory or requires exploration (explanatory vs. exploratory).

These frameworks complement each other in the broad field of data visualization. While data-ink ratio principles offer a strong start, blending these frameworks will enhance your approach, addressing varied needs and audiences. The key to effectively integrating these frameworks lies in understanding the visualization’s context, audience, message, and medium. Don’t be afraid to experiment and spend time analyzing the work of others to develop your taste and build experience.

Note 19.2 Finally, the most tried and true method of testing graphics is asking others to take a look at them!

19.3 Summary

This chapter on data visualization fundamentals has guided us through the essentials of visual perception in data presentation, highlighting the importance of preattentive processing and serial processing in understanding visuals. We explored how to effectively use visual attributes like color, shape, and size to highlight key information, and we discussed the interplay of integral and separable perception in design. Emphasizing the role of cognitive load, we examined how to tailor visualizations to the audience’s context and needs. In the next chapter, we will build upon this fundamental understanding by exploring how graphics are created.

Chapter 20
Data Visualization

Data visualization is undeniably the most thrilling aspect of analysis—everyone loves colorful graphs! It's also a crucial tool for conveying complex information. In this chapter, we'll cover two use cases of data visualization, explore the grammar of graphics, and introduce methods for creating interactive plots. Let's begin by examining the two main types.

20.1 Exploratory vs. Explanatory

When you start working with a new dataset, you're essentially learning about it and the relationships between its many variables. This approach is known as exploratory data analysis, where we test assumptions about the data and search for potentially interesting connections, patterns, and anomalies. While it's common to rely on summary statistics like mean and standard deviation, these numbers can obscure individual data points, masking the true shape of our datasets. Therefore, directly observing the actual data is irreplaceable. For example, Matejka and Fitzmaurice [24] created datasets with identical statistics that appear distinctly different. These diverse patterns are accessible through the `datasauRus` package.

mean_x	mean_y	std_dev_x	std_dev_y	corr_x_y
54.26	47.83	16.77	26.94	-0.06

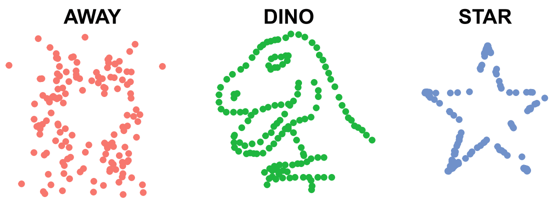


Figure 20-1 Datasaurus Dirty Dozen

Once you've compiled your results, the challenge becomes presenting them to a nontechnical audience. Such an audience won't be concerned with the technicalities, such as whether your model used cross-validation or how you optimized your gradient-boosted forest. What they will be seeking is a clear and convincing message. That is why you won't find fancy, overloaded graphs in client-facing presentations; it's all about delivering the message effectively. Consider the simplicity of a graph a fictional tech company used to demonstrate the superiority of their P100 processors. They focused on conveying the key message in an accessible and compelling way. Personally, I may not know the exact extent of the P100's improvement, but I am completely convinced that it is indeed better.

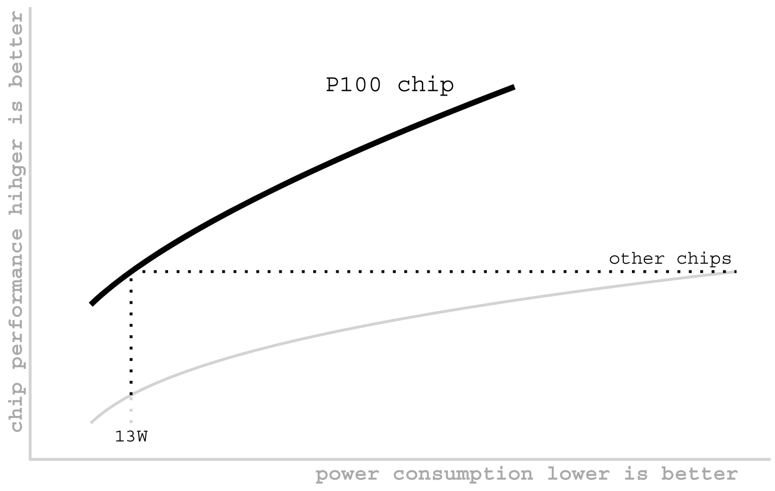


Figure 20-2 Explanatory Example

R offers a variety of packages for creating visually appealing and informative plots. One of the most popular and versatile packages for data visualization in R is `ggplot2`. We will explore the basics of using `ggplot2` to create different types of plots and customize them to suit your needs. We can load it separately, `libary(ggplot2)`, or with `libary(tidyverse)`.

20.2 The Grammar of Graphics

The `ggplot2` package in R is based on the principles of the grammar of graphics, a framework developed by Leland Wilkinson in his 1999 book *The Grammar of Graphics* [38]. This system outlines rules for visually representing data through a combination of graphical elements and mappings between data variables and visual properties. Hadley Wickham later adapted these concepts for R [35].

Excel users might be accustomed to its straightforward plotting workflow: you simply choose a plot type, and it generates the graph for you.

Figure 20-3 Excel Plotting Interface

In traditional plotting frameworks, scatter and bar plots appear entirely distinct:

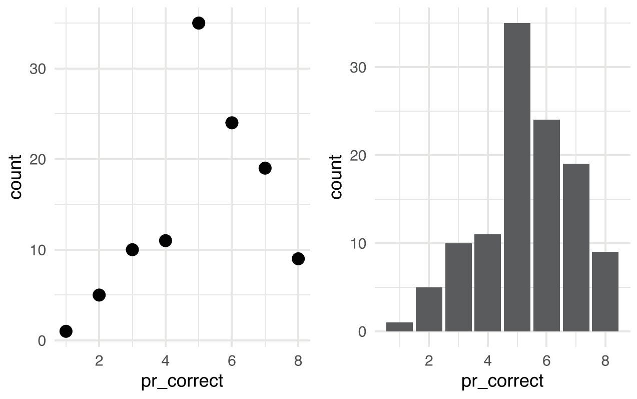


Figure 20-4 Scatter and Bar Plots

However, through the lens of the grammar of graphics, we observe the striking similarity between these charts. They differ only in their geometries—the first uses points and the second uses columns to represent data!

The `ggplot()` function establishes the base structure of a plot in `ggplot2`. Additional layers, such as points, lines, and facets, are added using the "+" operator. This layering approach simplifies the understanding, modification, and construction of complex, multilayer plots. Gaining proficiency in `ggplot2` is invaluable, as it not only empowers you to create visualizations but also enhances your understanding of graphical construction. For instance, let's refine the earlier histogram:

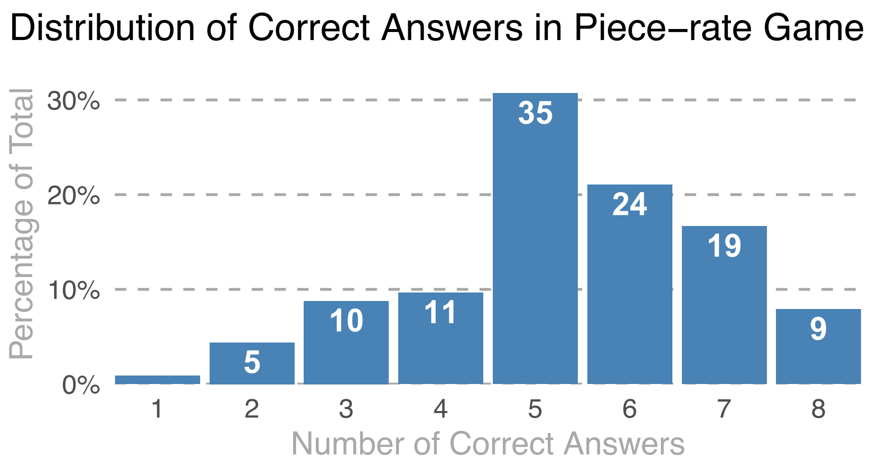


Figure 20-5 Improved Histogram

This enhanced version now includes labels, minimized grid lines, and a splash of color. For those eager to dig deeper into `ggplot2`, resources like *ggplot2: Elegant Graphics for Data Analysis*[1] and the *ggplot2 cheatsheet*[2] are highly recommended.

[1] https://ggplot2-book.org

[2] https://posit.co/wp-content/uploads/2022/10/data-visualization-1.pdf

Next, let's decompose the plot construction process into the grammar of graphics steps, demonstrating how to build a visualization layer by layer:

1. **Data**: This is the foundational layer where we specify the dataset to visualize. For example, we'll use the `iris` dataset, a built-in collection of iris flower (species *Iris setosa*) measurements.

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

At the data layer, we essentially have a blank canvas as no graphical elements have yet been defined.

```
data_layer <- ggplot(data = iris)
data_layer
```

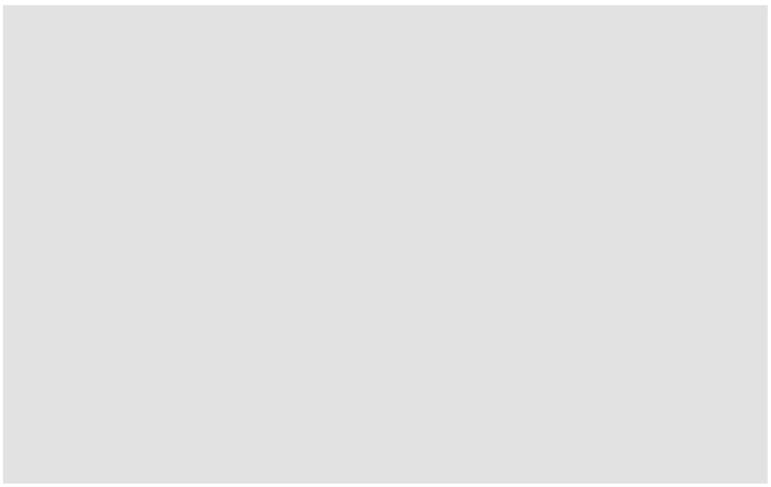


Figure 20-6 Data Layer

2. **Aesthetics**: This layer involves defining the visual properties that represent the data, such as x, y, color, or size. When aesthetics are added, the plotting area is set up. For example, mapping `Sepal.Length` to `x` and `Sepal.Width` to `y`:

```
aes_layer <- ggplot(
  data = iris,
  aes(x = Sepal.Length, y = Sepal.Width)
)
aes_layer
aes_layer mapping
```

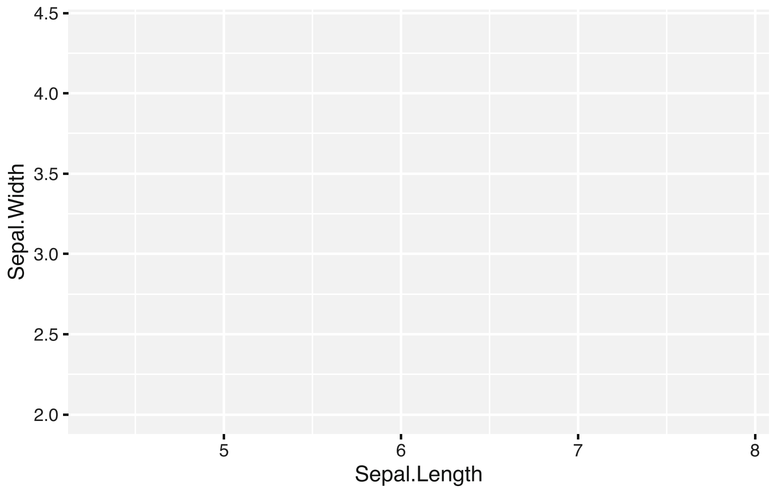


Figure 20-7 Aesthetics Layer

```
#>   Aesthetic mapping:
#>   * `x` -> `Sepal.Length`
#>   * `y` -> `Sepal.Width`
```

3. **Geometries**: Here, we decide the type of visual elements used to display the data, like points or bars. Adding geometry brings the data to visual life. The `geom_*()` functions, such as `geom_point()` for scatter plots, `geom_line()` for line charts, and `geom_bar()` for bar charts, are used to specify these visual elements in a plot.

```
geometry_layer <- aes_layer + geom_point()
geometry_layer
```

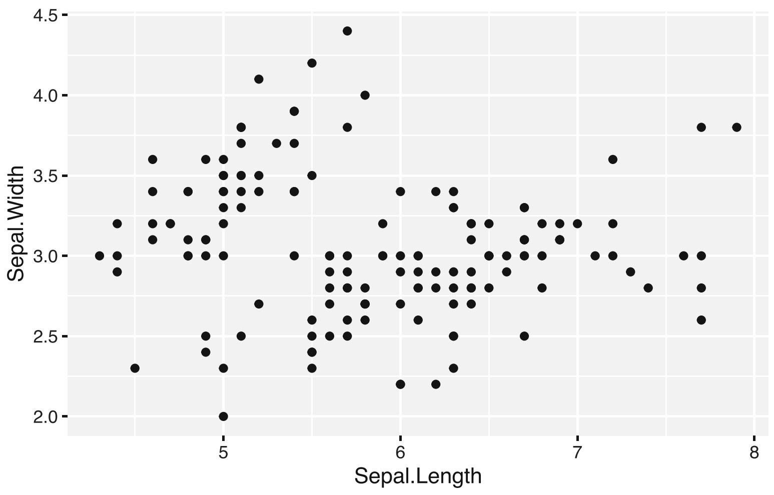


Figure 20-8 Geometry Layer

4. **Scales**: This layer defines how data values are translated into visual properties. There are different scales for aspects like color, size, or log(x). Adding a scale can change the appearance based on data attributes. For instance, the `scale_*()` functions, such as `scale_color_manual()` for custom colors or `scale_size_continuous()` for size, adjust these properties.

```
scales_layer <- ggplot(iris, aes(x = Sepal.Length,
                                 y = Sepal.Width,
                                 color = Species)) +
  geom_point() +
  scale_color_manual(values = c("red", "orange", "pink"))

scales_layer
scales_layer mapping
```

```
#>  Aesthetic mapping:
#>  * `x`     -> `Sepal.Length`
#>  * `y`     -> `Sepal.Width`
#>  * `color` -> `Species`
```

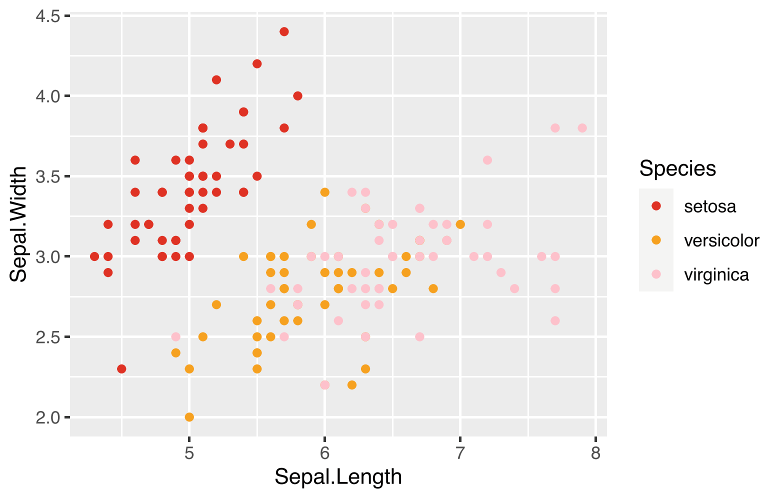


Figure 20-9 Scales Layer

5. **Statistics**: Before visualizing, the data can undergo mathematical transformations, such as computing summary statistics. The `stat_*()` functions perform these transformations, like `stat_summary()` for summarizing data. Some `geom_*()` functions, such as `geom_histogram()`, inherently use statistical transformations to divide data into bins and count observations.

```
stat_layer <- ggplot(data = iris, aes(x = Sepal.Length)) +
  geom_histogram(bins = 20, color = "white")
stat_layer
```

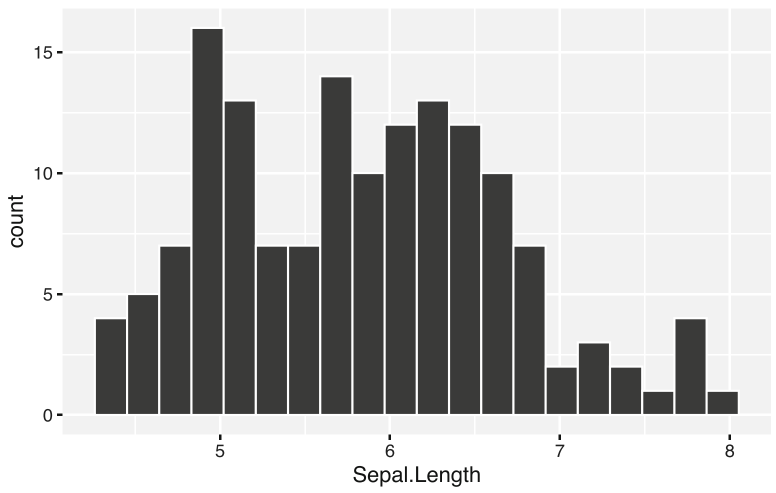


Figure 20-10 Statistics Layer

6. **Facets**: This involves breaking down the data into subgroups for separate visualizations. Facets can provide a clearer comparison between different categories. The `facet_*()` functions, such as *`facet_wrap`*`()` for wrapping panels into a grid or *`facet_grid`*`()` for creating a matrix of panels, are used to create these subplots.

```
facets_layer <- geometry_layer + facet_wrap(vars(Species), ncol = 3)
facets_layer
```

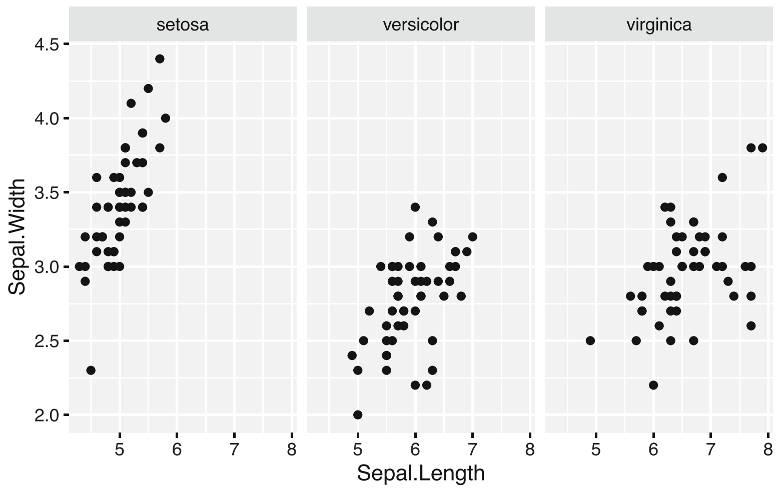


Figure 20-11 Facet Layer

7. **Theme**: Final touches to tailor the plot's appearance, including fonts, colors, and grid lines, and enhancing aesthetics and readability. Customize using the `theme()` function, or use premade themes like `theme_minimal()` for a clean look or `theme_bw()` for a classic style.

```
theme_layer <- facets_layer +
  # Apply a minimal theme with a base font size of 18
  theme_minimal(base_size = 18) +
  # Add points with size 2 and color coral
  geom_point(size = 2, color = "#ffb86c") +
  # Customize the theme settings
  theme(
    # Set plot background color to dark gray and border color to gray
    plot.background = element_rect(fill = "#282a36", color = "#44475A"),
    # Remove minor grid lines
    panel.grid.minor = element_blank(),
    # Set major grid lines color to gray
    panel.grid.major = element_line(colour = "#44475a"),
    # Set axis text and title as well as facet strip text color to white
    axis.text = element_text(color = "#f8f8f2"),
    axis.title = element_text(color = "#f8f8f2"),
    strip.text = element_text(color = "#f8f8f2")
  ) +
  # Add labels for the x and y axes
  labs(x = "Sepal Length", y = "Sepal Width") +
  # Set the x-axis limits from 4 to 8
  xlim(4, 8)

theme_layer
```

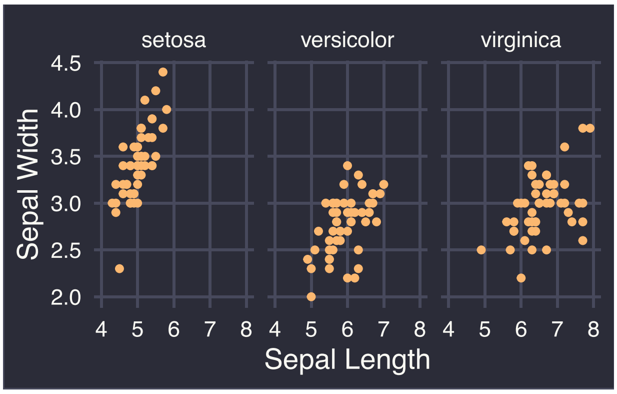
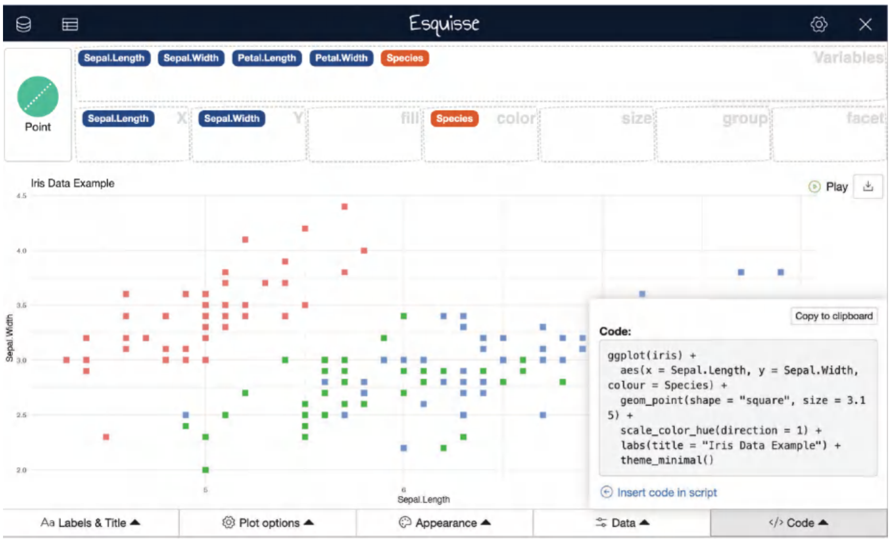


Figure 20-12 Theme Layer

20.2.1 Graphical Interfaces for `ggplot2`

If this feels a bit overwhelming, you don't have to start from scratch. We can use the powerful `esquisse` package to build our plots with a simple drag-and-drop interface. This tool allows you to select data, choose geometries, and make almost any edits you need. By the end, you'll have a fully customized, ready-to-go plot! You can access `esquisse` by going to "Addins" in the top panel or with `esquisser(your_data)`.



Code:

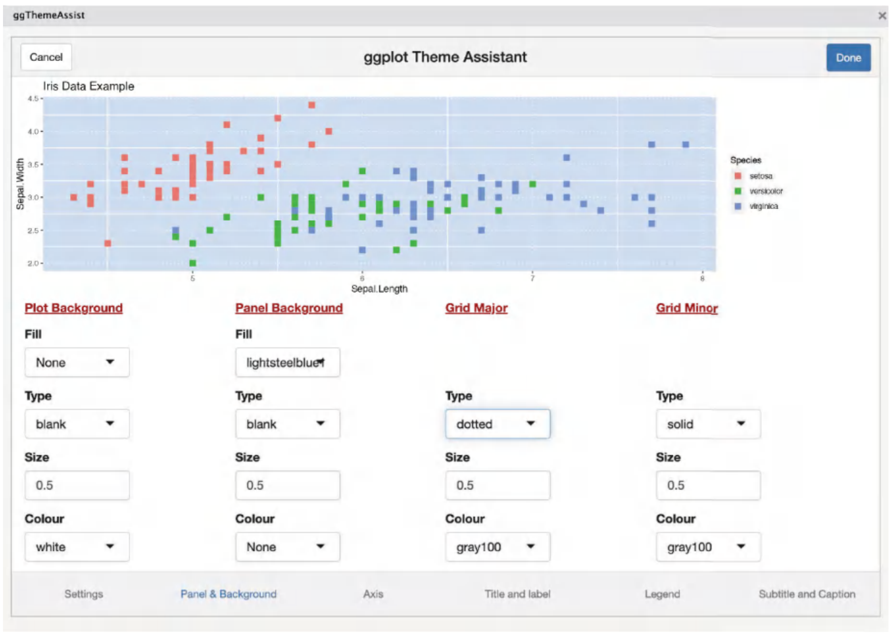
```
ggplot(iris) +
  aes(x = Sepal.Length, y = Sepal.Width,
colour = Species) +
  geom_point(shape = "square", size = 3.1
5) +
  scale_color_hue(direction = 1) +
  labs(title = "Iris Data Example") +
  theme_minimal()
```

Figure 20-13 `esquisse` UI

```
# install.packages('esquisse')
library(esquisse)
esquisser(iris)
```

And if you're looking for more theming capabilities, you can style the plot with a GUI using `ggThemeAssist`! You can access `ggThemeAssist` by selecting the output `ggplot2` plot object and going to "Addins" and clicking on "ggplot Theme Assistant."

```
# install.packages("ggThemeAssist")
library(ggThemeAssist)
```



Plot Background
Fill: None
Type: blank
Size: 0.5
Colour: white

Panel Background
Fill: lightsteelblue
Type: blank
Size: 0.5
Colour: None

Grid Major
Type: dotted
Size: 0.5
Colour: gray100

Grid Minor
Type: solid
Size: 0.5
Colour: gray100

Figure 20-14 `ggThemeAssist` UI

20.3 Interactive Plots

While `ggplot2` is fantastic for creating static visualizations, it doesn't offer much in terms of interactivity by default, which could come in handy for more dense plots. However, I urge you to master the art of static visualizations and not rely solely on interactivity to convey your message.

If you would like to add minimal interactivity, check out the `ggiraph` package. It has a number of `ggplot2`-like functions that add an additional level of interactivity, such as tooltips, selections, and more.

The easiest way to add interactivity is the `ggplotly` function in the `plotly` package. It offers an easy way to convert `ggplot2` figures into interactive `plotly` visuals. This feature is particularly handy when you're dealing with a large number of data points that would be difficult to distinguish in a static plot.

```
library(plotly)

p <- ggplot(iris, aes(x = Sepal.Length,
                      y = Sepal.Width,
                      color = Species)) +
  geom_point(size = 1, alpha = 1) +
  theme_minimal()

# Convert the `ggplot2` plot to a `plotly` plot
pp <- ggplotly(p)
pp
```

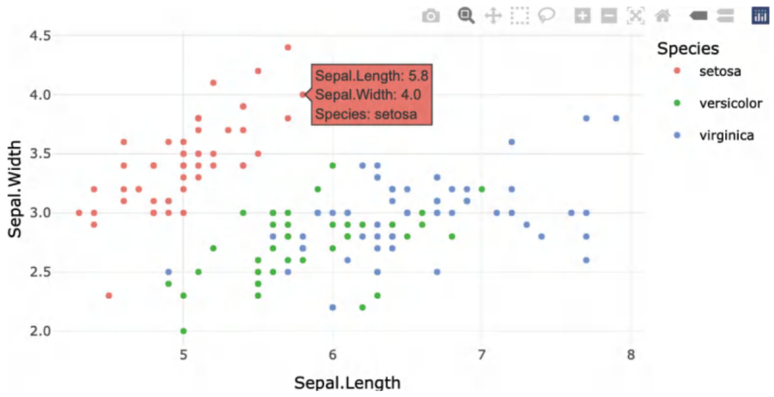


Figure 20-15 `ggplotly` Example

20.3.1 HTML Widgets

For those seeking advanced web interactivity in R, `htmlwidgets` is a must-see project. It seamlessly integrates R with powerful JavaScript libraries, enabling the use of prominent visualization tools like `plotly`, `leaflet`, and `DT` (DataTables).

`ggplotly()` lets you convert `ggplot2` figures into interactive plots, but `plotly` also allows you to create these visuals from scratch. However, directly creating visuals via `plotly` offers enhanced flexibility and access to features that might be unavailable through `ggplot2` conversion.

```r
library(plotly)

# Scatter plot: Sepal Length vs Sepal Width, colored by Species
p <- plot_ly(
  data = iris,
  x = ~Sepal.Length,
  y = ~Sepal.Width,
  color = ~Species,
  type = "scatter",
  mode = "markers",
  marker = list(size = 6),
  hovertemplate =
  'Sepal Length: %{x}<br>Sepal Width: %{y}<br><extra></extra>'
)

# Customize the plot layout
p <- layout(
  p,
  title = "Sepal Measurements",
  xaxis = list(title = "Sepal Length"),
  yaxis = list(title = "Sepal Width")
)

p
```

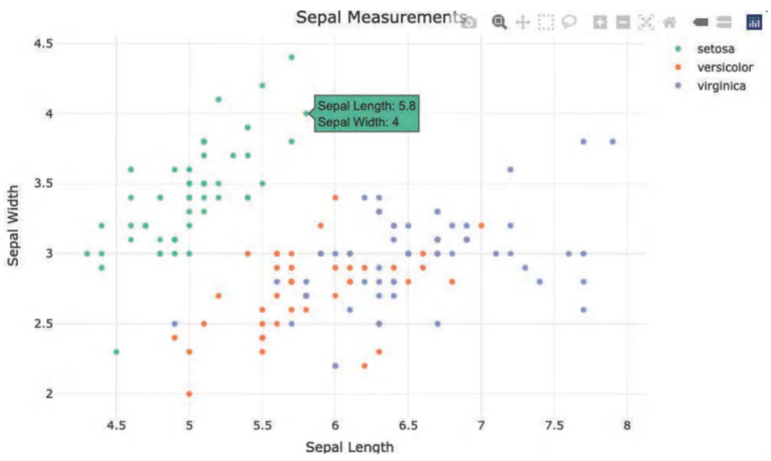


Figure 20-16 Plotly Example

The `echarts4r` package in R serves as a wrapper for ECharts, a versatile JavaScript library. It supports an array of chart types such as bar, line, scatter, pie, and radar. ECharts is particularly powerful for creating complex, multiseries, interactive charts and supports a broad range of customization options. The package offers impressive animations and interactivity, making it a great alternative to `plotly`.

```r
library(echarts4r)

e_chart <- iris %>%
  group_by(Species) %>% # Group by Species
  # Initialize plot with Sepal Length on x-axis
  e_charts(Sepal.Length) %>%
  # Add scatter plot with Sepal Width on y-axis
  e_scatter(Sepal.Width, symbol_size = 7) %>%
  e_y_axis(min = 1.5, max = 5) %>% # Set y-axis range
  e_x_axis(min = 4, max = 8) %>% # Set x-axis range
  e_axis_labels(x = "Sepal Width", y = "Sepal Length") %>% # Label axes
  e_tooltip(
    trigger = "item",
    formatter = htmlwidgets::JS("
      function(params){
        return('Sepal Length: ' +
        params.value[0] +
        '<br />Sepal Width: ' +
        params.value[1])
      }
    ")
  ) # Add tooltip

e_chart
```

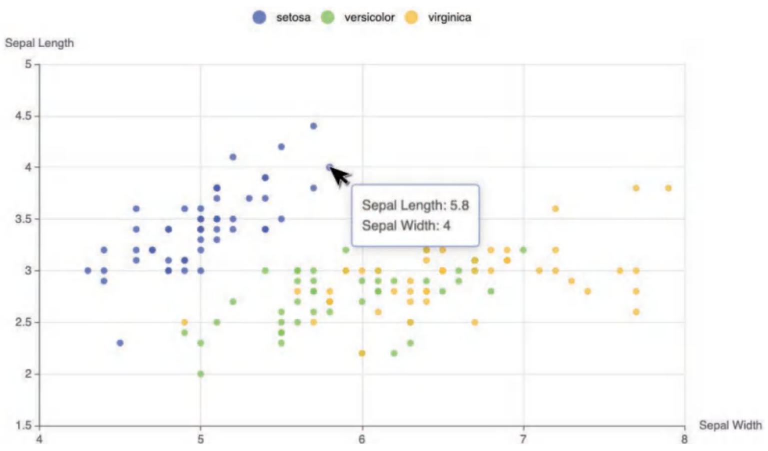


Figure 20-17 Echarts Example

20.4 Summary

This chapter has provided a comprehensive guide to data visualization in R, highlighting the importance of both exploratory and explanatory approaches. We explored the versatility of `ggplot2` based on the grammar of graphics and examined how to bring interactivity to visualizations. The upcoming chapter will showcase a variety of graph types and their practical applications.

Chapter 21
A Graph for the Job

When working on graphs, don't ask, "What chart should I use?" Instead, ask, "What am I trying to show?" In this section, we will explore different types of charts, their purposes, and when to use them. While we won't cover every possible graph, you will certainly expand your toolkit.

21.1 Category Comparison

Graphs for category comparison are a type of data visualization used to compare and contrast different categories or groups. The most common of these is the bar chart. The one below shows the top five countries by GDP per capita in 1997. It is evident that Norway ranks first, while Switzerland is fifth.

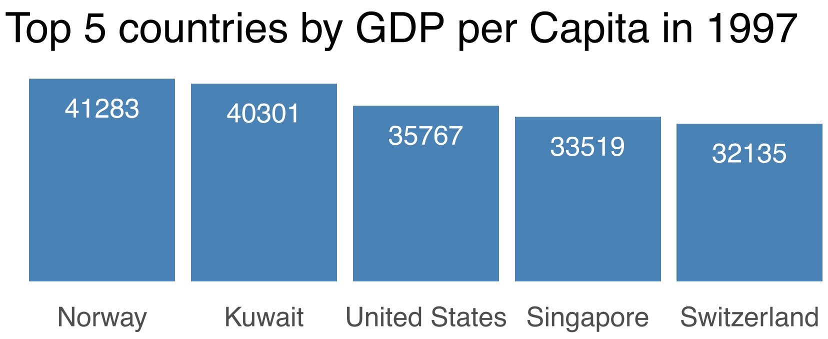


Figure 21-1 Bar Chart

Vertical bar charts are excellent for providing a quick comparison for a small number of categories (less than seven). However, if you need to show the ranking of more items, consider flipping the axis of the bar chart. An additional advantage is that horizontal bar charts are great for displaying long names. Below are the results of the 2021 London election, where British YouTuber Niko Omilana finished fifth. Another YouTuber, Max Fosh, also passed the cutoff.

© Nikita Tkachenko 2025
N. Tkachenko, *Data Insight Foundations*, https://doi.org/10.1007/979-8-8688-0580-6_21

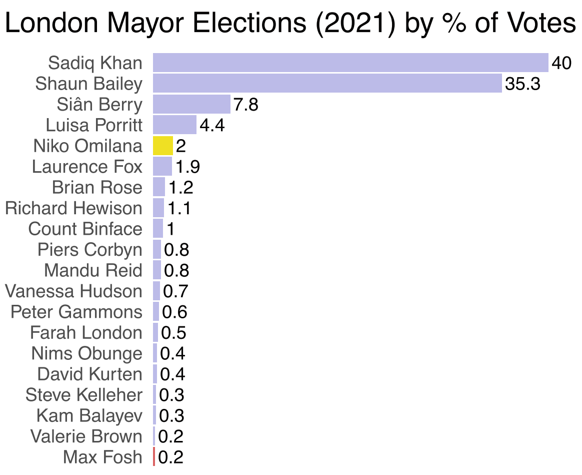


Figure 21-2 Vertical Bar Chart

21.1.1 *Lollipop Chart*

The lollipop chart stands out as a personal favorite, especially when contrasted with conventional bar plots. Its unique strength is its ability to clearly illustrate the position of the data point in a two-dimensional space. A helpful rule of thumb for choosing a lollipop chart over a bar chart is to consider the stackability of the data. For instance, since percentages cannot be stacked, a lollipop chart is more suitable.

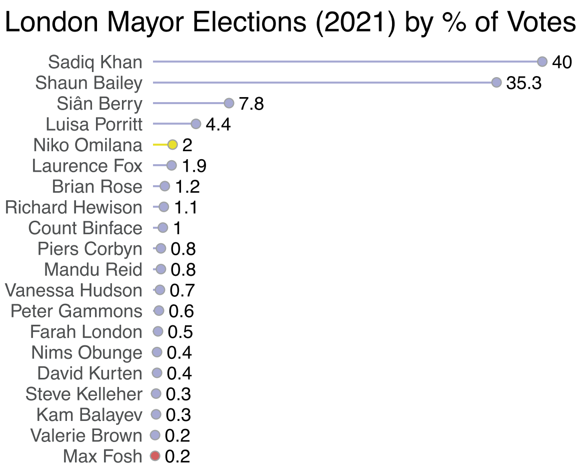


Figure 21-3 Lollipop Chart

21.1.2 *Bullet Graph*

A less common visual is the bullet graph. It is a powerful tool designed for comparing performance against a predefined target zone. Let's test it by visualizing the Net Promoter Score (NPS). The bullet graph clearly shows the target, various performance levels, and where our results stand in relation to the goal.

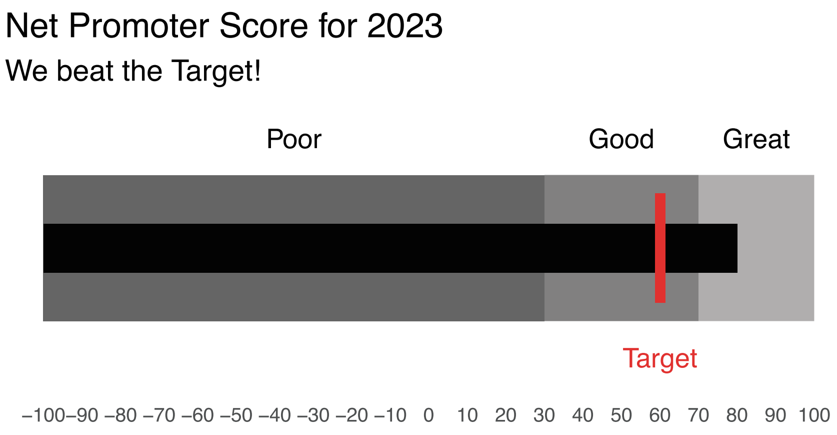


Figure 21-4 Bullet Graph

21.2 Distribution

What if you want to show the distribution of data? We can use a variation of a bar chart: the histogram. Histograms display the distribution of continuous data by grouping the data into bins and showing the frequency or proportion of observations within each bin. They are excellent for visualizing the shape of the distribution but are highly sensitive to the choice of bins. Let's use the `iris` dataset to demonstrate how the number of bins affects the sepal length plot. Notice how the shape of the distribution changes with the number of bins. It's important to strike a balance between too few and too many bins. Six bins make our distribution appear fairly normal, while thirty bins make it look erratic. Fifteen bins seem about right, preserving the bimodal feature of the distribution while keeping the picture legible.

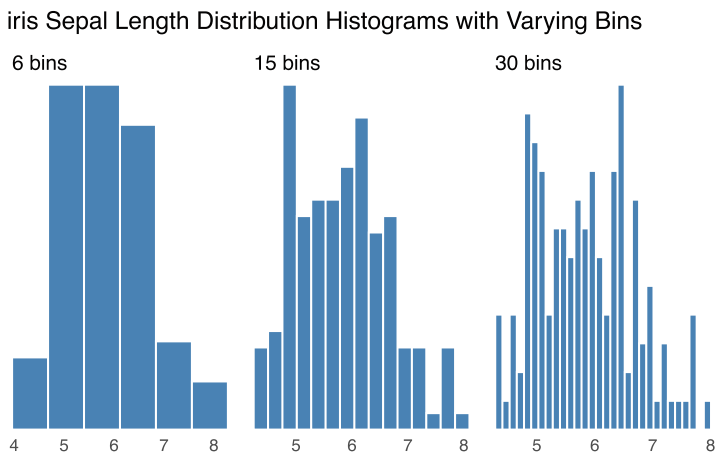


Figure 21-5 Histograms

21.2.1 Density Plot

Unlike histograms, density plots use a continuous line to represent the data instead of bars. This smooth curve provides a more detailed and nuanced representation of the distribution, allowing for easier detection of patterns and trends. Density plots construct this line by placing many small normal distributions at each data point, which are then used to weigh all points within their respective ranges and draw a curve connecting them. The width of these curves is controlled by the bandwidth of the density plot, determining how wide the curves span. A larger bandwidth will consider more points, resulting in a smoother curve, while a smaller bandwidth will lead to a jagged line.

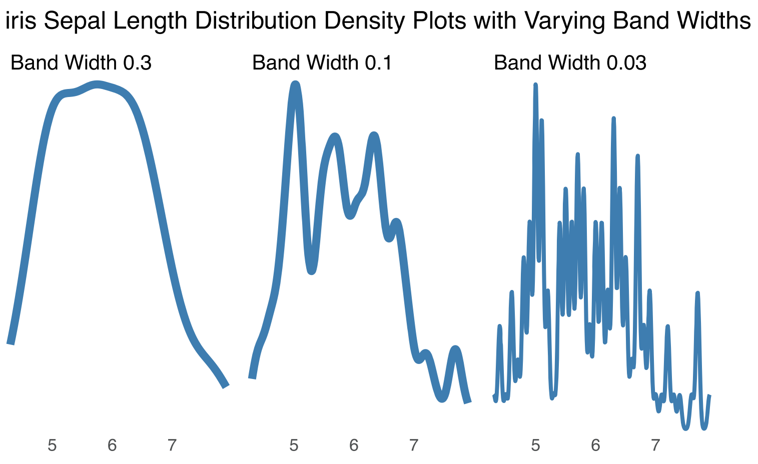


Figure 21-6 Density Plots

21.2.2 Frequency Polygon

A frequency polygon is similar to a histogram, but instead of bars, it uses a continuous line to connect points representing the frequencies. Frequency polygons are particularly useful when comparing two or more data sets on the same plot. Like histograms, they rely on the selection of bins.

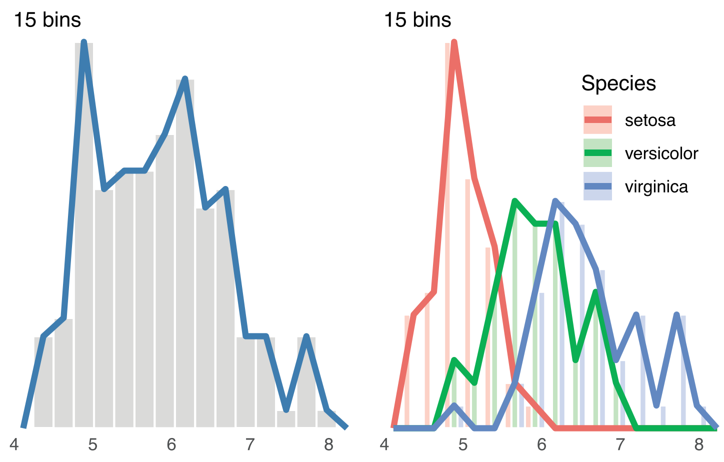


Figure 21-7 Frequency Polygon

21.2.3 Box Plot

Box plots provide a summary of the distribution of a dataset, showing the median, the lower and upper quartiles, and the minimum and maximum values. The box in the middle represents the interquartile range (IQR), which is the range of the middle 50% of the data. The line inside the box represents the median, the midpoint of the data. The whiskers on the top and bottom extend to the minimum and maximum values, excluding outliers. It is incredible how much information box plots contain! With just one plot, you can quickly identify outliers and gain a visual understanding of the data's distribution. In the context of the iris dataset, the box plot of sepal length across different species provides a clear picture of this variable's distribution. Notice how significantly the statistics differ across species.

However, like real boxes, box plots can also hide important information. To illustrate this point, we can use datasets with the same summary statistics but different distributions. The second graph displays three identical box plots. However, once we add data points to the plot, it becomes evident that the distributions are quite different.

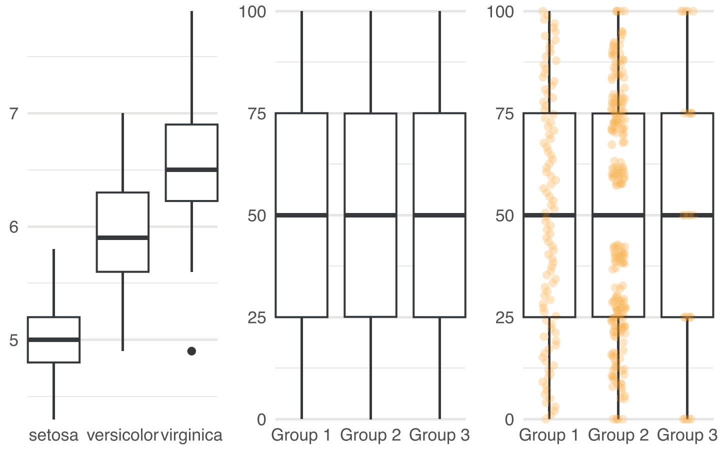


Figure 21-8 Box Plots

21.2.4 Violin Plot

One solution is to use a violin plot, which is essentially a vertical density plot. Look how much more we learn about the data distribution of iris species. We can see the density distribution, points, and quantiles. Do you remember how bandwidth is extremely important when making density plots? Setting an appropriate bandwidth reveals the true distribution.

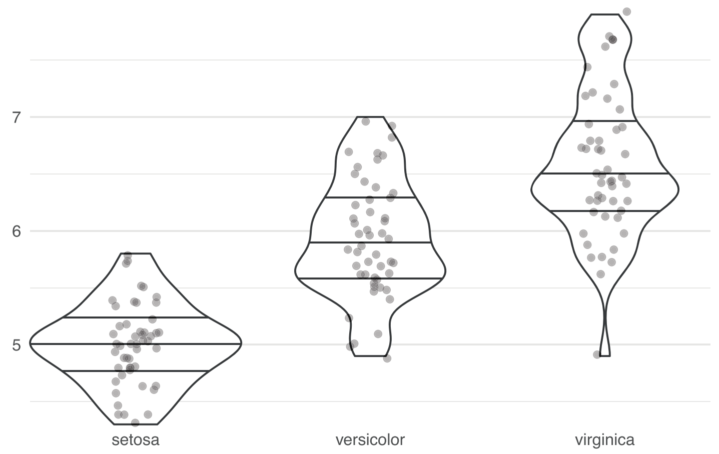


Figure 21-9 Violin Plots

21.2.5 *Bee Hive Plot*

The bee hive plot is a scatter plot that arranges data points as dots to minimize overlap. It's ideal for visualizing small datasets because it creates patterns like a density plot without hiding individual data points.

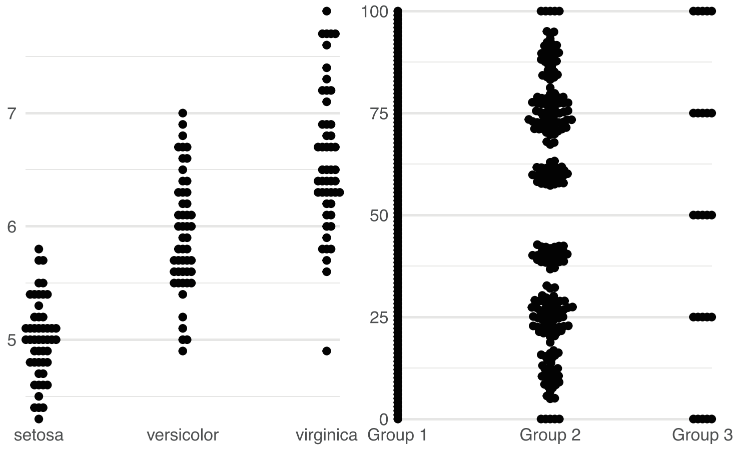


Figure 21-10 Beeswarm Plot

All of the plots we have covered so far have their advantages:

1. A box plot shows important statistics.
2. A density plot provides a high-level view of data shape.
3. A bee hive plot "shows" the actual data points.

While combining these plots might make for a crowded visual, with some modifications, it's possible to create a hybrid plot that captures the strengths of each.

21.2.6 Rain Cloud Plot

A rain cloud plot combines elements of box plots, violin plots, and bee hive plots. It uses a density plot to show the data distribution, a box plot to display the statistics, and individual data points represented as raindrops. The result is a visually appealing and informative way to visualize a large number of distributions side by side, allowing for easy comparisons and identification of patterns.

Isn't this beautiful? We have a box plot, density plot, and jittered points all in the same graph without looking cluttered. Moreover, we can clearly see how the distributions of our synthetic datasets differ.

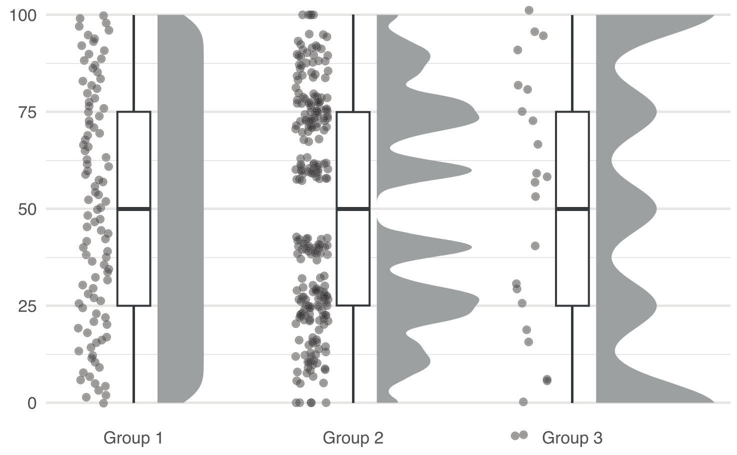


Figure 21-11 Rain Cloud Plots

My personal favorite is the horizontal version with vertical dashes instead of points, a density plot, and a box plot replaced by a slab plot. In the slab plot, a thick line represents the IQR, a dot in the middle signifies the median, and lighter lines serve as whiskers. While this alternative can be visually appealing, it is important to ensure that your audience understands the visualization. It may be necessary to provide additional context or include a note explaining the meaning of the slabs and dashes to avoid confusion.

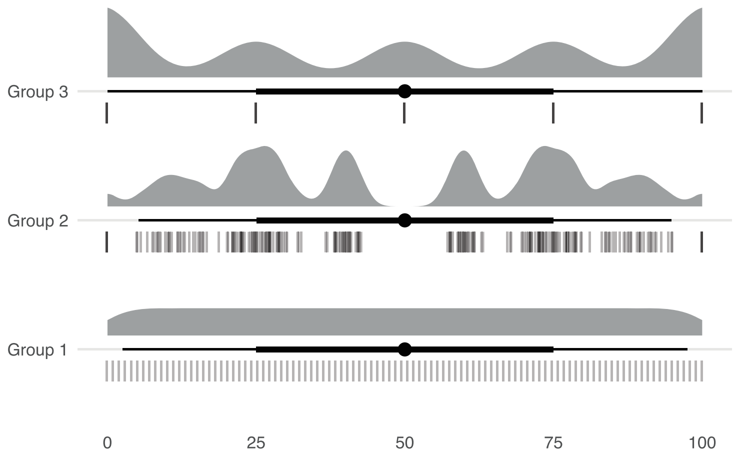


Figure 21-12 Horizontal Rain Cloud Plots

21.2.7 Margins

Marginal plots are a method for visualizing data distributions in relation to two variables. In this visualization, a density plot for each variable is positioned on the scatter plot's edges. This allows us to examine both the individual distributions of each variable and the relationship between the two. Observe how sepal width and length create distinct groupings and exhibit unique distributions for each of the iris species.

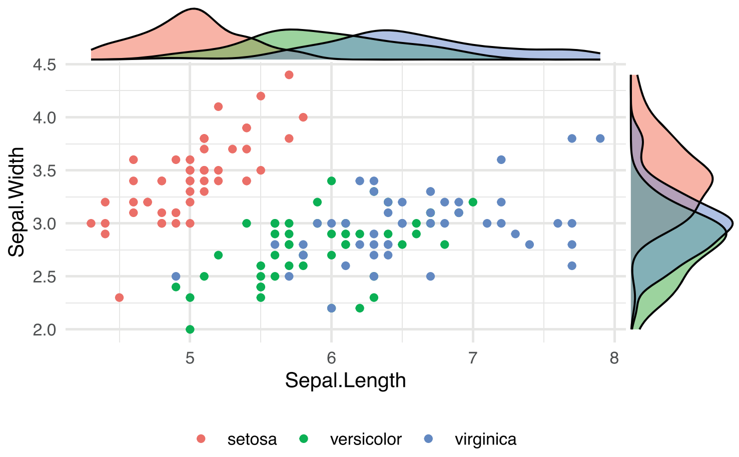


Figure 21-13 Margin Plots

21.3 Proportions

Another large collection of graphs is concerned with communicating proportions and composition.

21.3.1 Stacked Bar Charts

A stacked bar chart is a type of graph used to visualize the distribution of a categorical variable. It is similar to a regular bar chart, but in a stacked bar chart, each bar is divided into sections, with each section representing a different category within the variable. The height of each section corresponds to the proportion or frequency of the category within that bar. Stacked bar charts are particularly useful when comparing the distribution of a variable across different subgroups or time periods, as they allow for easy visualization of both the overall distribution as well as the relative proportions of each subgroup or category within the variable.

As an example, we will use U.S. expenditures across departments. Only the top four departments are shown; the rest are collected into "Other." The graph below shows absolute values and their components across years. Notice how the total spent increases year over year and the changes in composition. Can you tell which departments contributed most to this growth?

What if we are not concerned with absolute values, but with relative proportions? We can use a percentage-stacked chart. Stacked charts are useful for visualizing the distribution of categorical

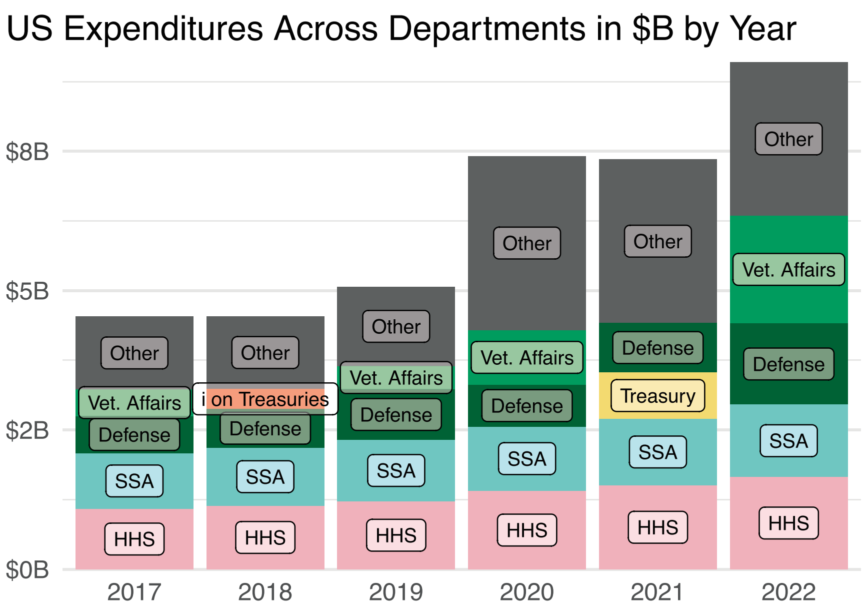


Figure 21-14 Stacked Bar Chart

variables, but they can be challenging to compare categories in the middle. Typically, the easiest categories to compare are the ones at the top and bottom of the stack. For example, suppose we want to compare the trends of the Department of Defense and the Social Security Administration (SSA) over time. In this case, we can move these categories to the top and bottom positions of the stacked chart to make it easier to compare their relative sizes and trends.

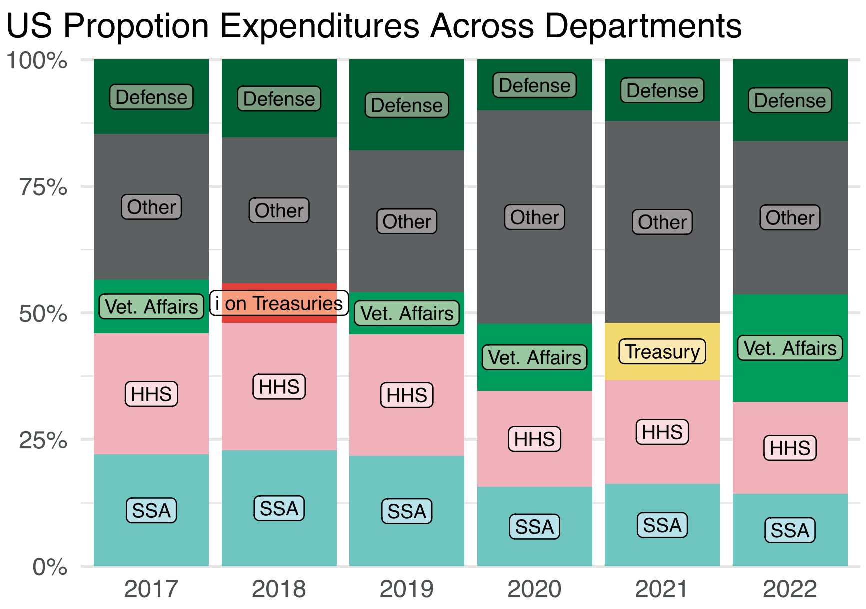


Figure 21-15 Percentage-Stacked Bar Chart

21.3.2 *Pie Chart*

Pie charts are essentially bar charts transformed into a polar coordinate system, where each category is represented as a slice of a circle. While pie charts can effectively communicate when one category is significantly larger or smaller than the others, they become difficult to read and compare accurately when there are many categories or when the differences between them are small. Comparing angles and areas of the slices can be confusing, leading to misinterpretation of the data. As an example dataset, we will use Japan's export basket from 2020.[1] Can you identify Japan's largest and smallest exports? What about the third largest?

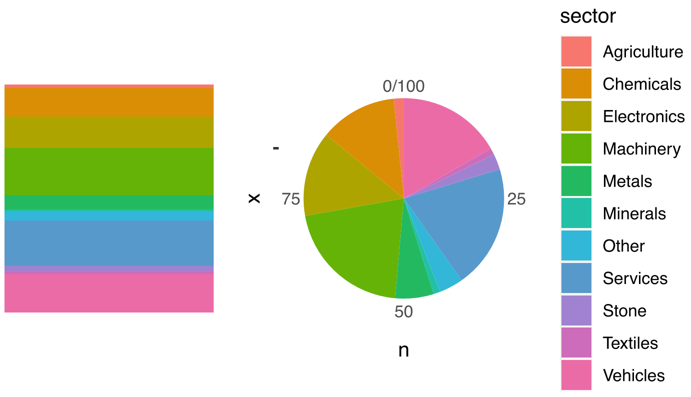


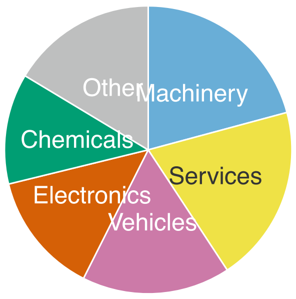
Figure 21-16 Pie Chart

But if you absolutely must use a pie chart, here are some rules to keep in mind:

1. Limit the number of categories to five to seven at most.
2. Consider grouping small categories into an "Other" category to avoid clutter.
3. Arrange the slices in decreasing order of size, starting at 12 o'clock to aid in comparing them.
4. Include the category labels directly on the chart instead of relying solely on a legend.
5. Add separators between slices to help distinguish between them.

However, keep in mind that this can also add visual clutter, so use with discretion.

Figure 21-17 Improved Pie Chart



[1] Data used from: The Growth Lab at Harvard University, The Atlas of Economic Complexity. http://www.atlas.cid.harvard.edu.

21.3.3 *Waffle Chart*

One alternative to a pie chart could be a waffle chart (these food names make me hungry). It is a gridlike visualization that resembles a waffle or a checkerboard. Each square in the grid represents a proportion of the total data, making it a useful way to visualize proportions or percentages in a visually appealing way. However, they are also vulnerable to large numbers of categories. But what they are truly great at is giving a sense of proportions and sizes. Waffle charts significantly benefit from interactivity.

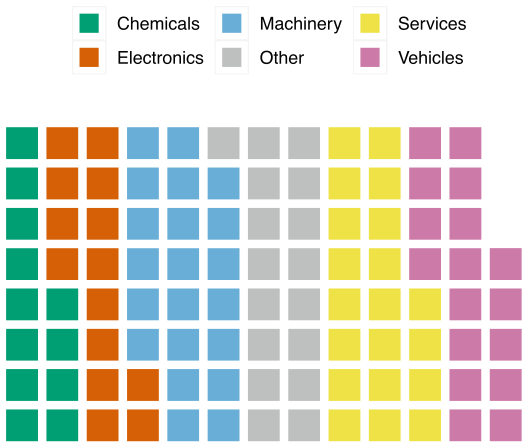


Figure 21-18 Waffle Chart

21.3.4 *Treemaps*

What if we have a lot of hierarchical data? Treemaps! Treemaps are a type of visualization that allows you to display hierarchical data in a way that is easy to understand. Each node in the hierarchy is represented by a rectangle, and the size of the rectangle corresponds to the proportion of the total data. The nodes are organized in a way that preserves the hierarchy, with parent nodes containing smaller child nodes. This allows you to quickly identify which nodes are the largest and which are the smallest, as well as the relationships between them. Treemaps are especially useful for displaying large amounts of data in a compact and intuitive way. Treemaps can become very cluttered, and interactivity is almost always necessary for such detailed plots. Check out the same plot on the source website at https://atlas.cid.harvard.edu/countries/114/export-basket.

21.4 Correlation

In addition to understanding the distribution of individual variables, it is important to examine the relationship between pairs of variables. Correlation plots are a useful tool for visualizing many aspects of data: relationships between variables (or lack thereof), clustering, and outliers, for example.

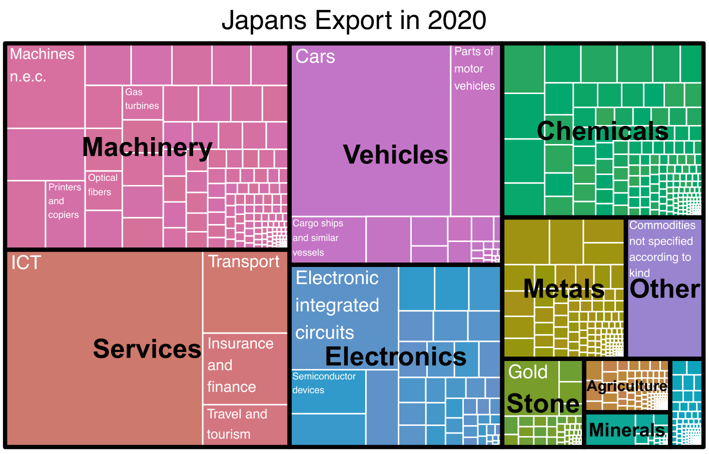


Figure 21-19 Treemap

21.4.1 Scatter Plot

The most common visualization is the scatter plot. It is no secret that scatter plots are amazing and perhaps the most persuasive types of plot. We can add fitted lines to the plot to better show the relationships between the variables. Returning to the iris flowers, observe the distinct angle of the fitted line for *Iris setosa* compared to the other two species. What does this tell you about the growth patterns of this species?

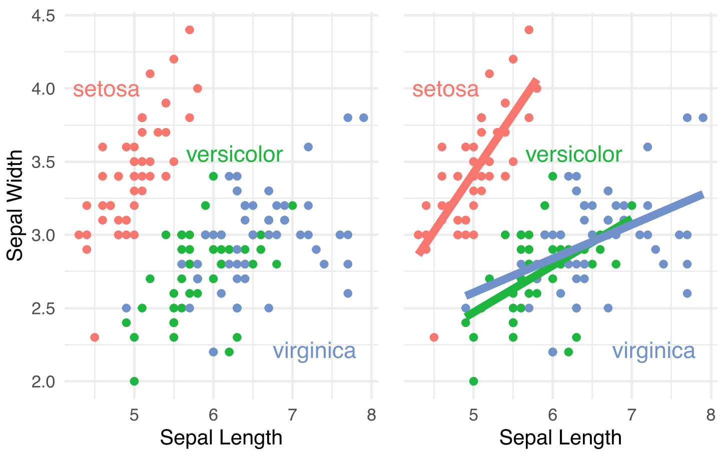


Figure 21-20 Scatter Plot

21.4.2 Correlograms

Correlograms serve as efficient tools for swiftly visualizing the relationships within a dataset. They enable us to visualize correlations between all pairs of variables, which is especially handy when exploring multidimensional datasets. There are several ways to structure correlograms, but the most common one is presented in Figure 21-21. Although it may look cluttered at first glance, it will help you quickly identify any existing relationships in the data. When examining the graphic, consider what characteristics can be helpful for distinguishing between the three species.

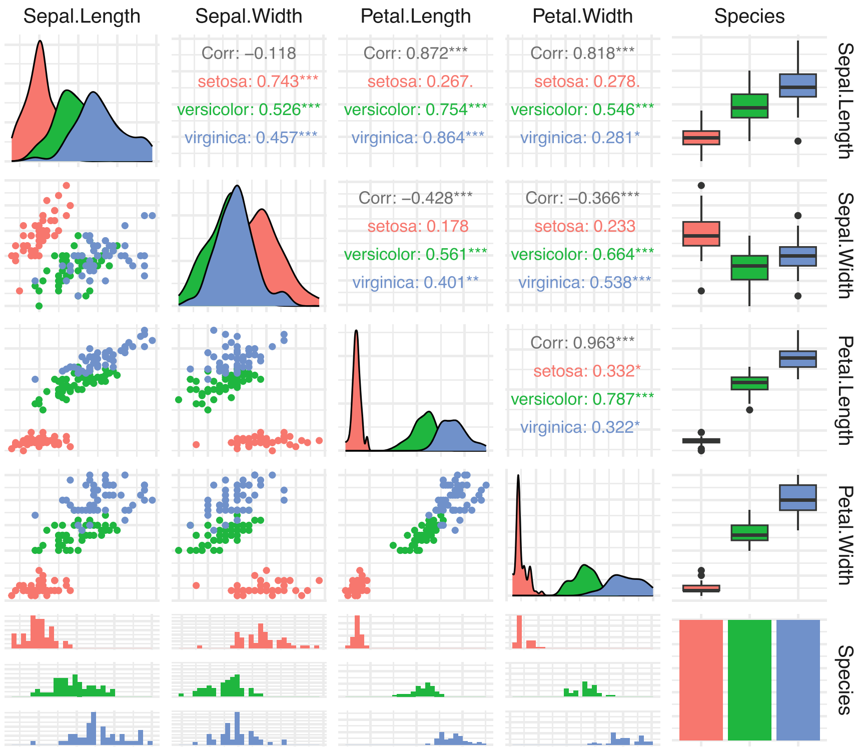


Figure 21-21 Correlogram Plot

21.5 Change Over Time

We have already seen a few plots that incorporate time change. Time series plots typically have time on the x-axis and the variable being measured on the y-axis. They can show trends, patterns, and seasonal fluctuations in the data.

21.5.1 Line Chart

Line charts are perhaps the most intuitive out there. To me, they are closely associated with finance. So let's have a look at the S&P 500 stock market index since 1927. The historical data are inflation-adjusted using the headline CPI, and each data point represents the month-end closing value. Compared to scatter plots, these make itsignificantly easier to see the general pattern at a glance.

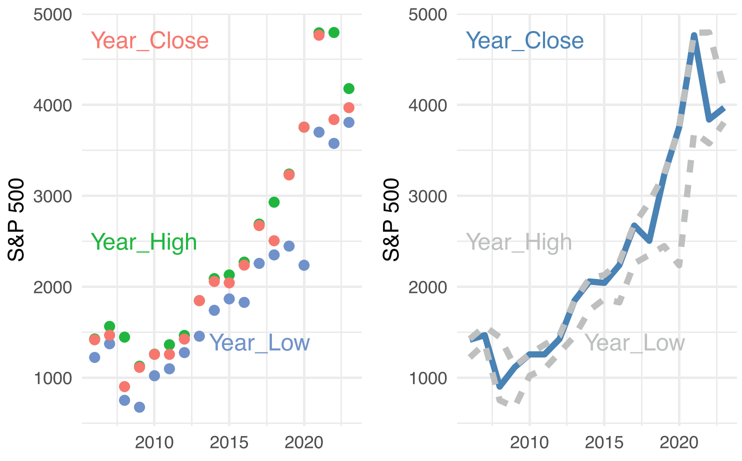


Figure 21-22 Line Chart

21.5.2 *Waterfall Chart*

Waterfall charts, also known as bridge charts, are a type of bar chart used to visualize the cumulative effect of sequentially introduced positive or negative values. The graph is called "waterfall" because it resembles a series of falling water droplets. Each bar in the chart represents a value and is color-coded to indicate whether it contributes to an increase or decrease in the cumulative total. They are useful for visualizing the relative contributions of positive and negative factors that affect the net change in the value being analyzed. For instance, we can use a waterfall chart to show how a bank's balance changed throughout the years.

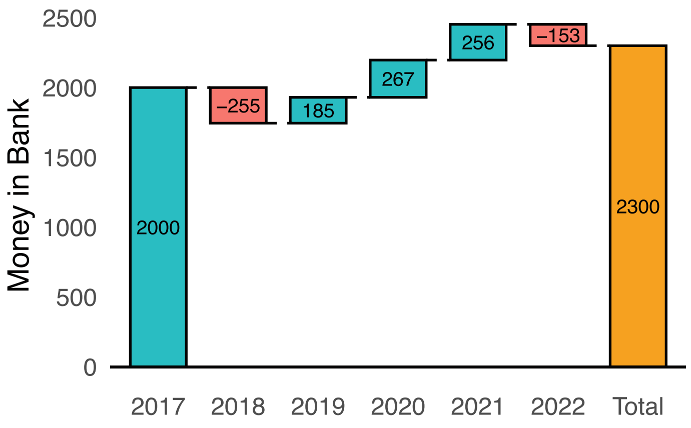


Figure 21-23 Waterfall Chart

21.6 Summary

In this chapter, we focused on selecting appropriate graphs for particular tasks. As you must have noticed, there is a vast array of tasks and an equally diverse range of graphs to choose from. Choosing the right visualization tool is crucial for the success of a presentation. Equipped with this expanded kit, we will now move on to the next chapter, which will teach you how to use colors to enhance your visuals.

Chapter 22
Color Data

When it comes to data visualization, using color effectively is crucial. Color can draw attention to key data points, enhance visual appeal, and make complex information easier to understand. A thoughtful color scheme guides the viewer's eye, highlights important details, and helps differentiate categories within the data, making the overall message clearer and more memorable.

Colors do more than just make charts look good; they can evoke emotions and convey meanings that might be hard to express otherwise. In the United States, for example, red often signals danger or passion, blue represents calm or sadness, and green stands for nature or health. But these meanings can vary widely across different cultures, so it's important to consider your audience when choosing colors. The impact of color is so significant that since 1979, the locker room at Iowa's Kinnick Stadium has been painted pink, supposedly to lower the testosterone levels of visiting teams.

By using color strategically, you can make your data visualizations more engaging and effective, ensuring that your audience not only understands but also remembers the information you present.

22.1 What Colors to Choose

Color is employed to underscore specific data and provide context. Consider the two graphs below, both of which compare the gross domestic product (GDP) of European countries in 1997. The first graph assigns each country a distinct color, resulting in a visual cacophony akin to an "explosion at a candy factory." The second graph improves upon this by minimizing distractions. It highlights "Greece" and grays out the remaining countries. The use of "red" for Greece serves as a signal, hinting that its economic performance may be subpar.

22.1.1 Complementary Harmony with a Positive/Negative Connotation

Complementary harmony involves the use of colors directly opposite each other on the color wheel (e.g., Figure 22-2), creating a stark contrast. This method effectively conveys a positive/negative connotation, ideal for emphasizing differences. While colors located near each other on the wheel can also complement each other, those placed in opposition offer the most substantial reinforcement for a key color. The following illustration compares the population growth of Asia and Europe. Here, the use of bright purple underscores the remarkable population surge in Asia, while the green tone underlines the comparatively slower growth in Europe.

N. Tkachenko, *Data Insight Foundations*, https://doi.org/10.1007/979-8-8688-0580-6_22

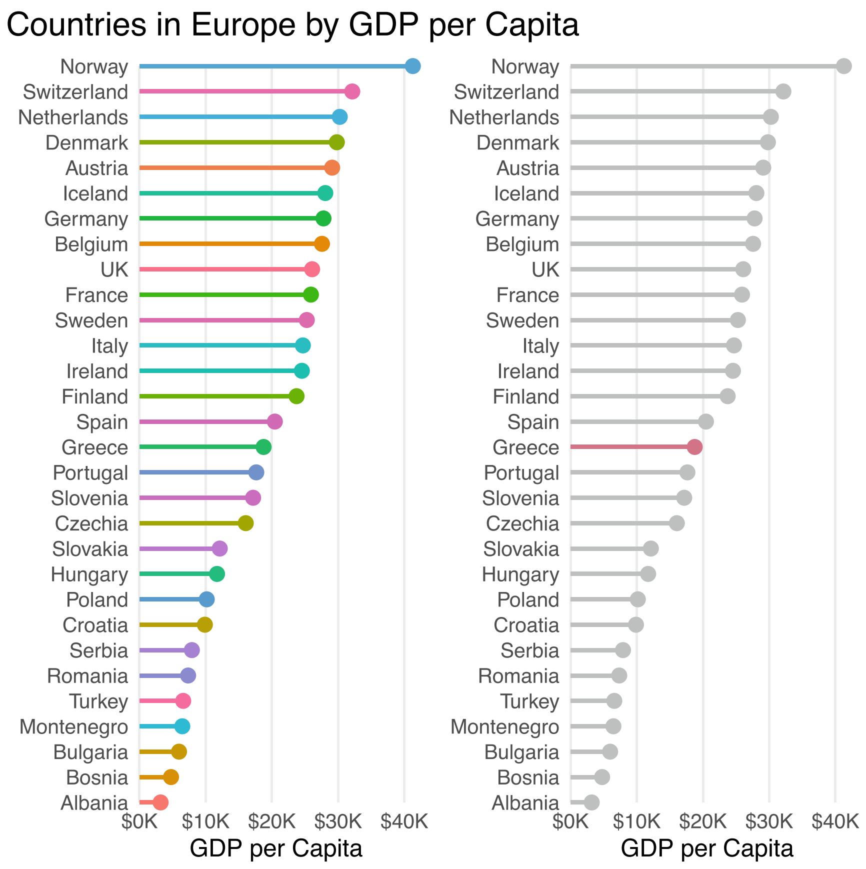
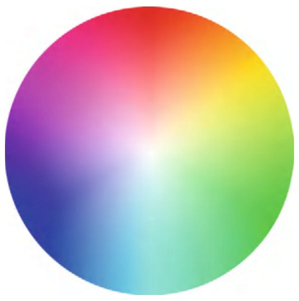


Figure 22-1 Emphasis with Color

Figure 22-2 Color Wheel



22.1.2 *Near Complementary Harmony for Highlighting Two Series Where One Is the Primary Focus*

Near complementary harmony is a color scheme that achieves substantial contrast without resorting to using colors diametrically opposite on the color wheel. Instead, it involves choosing a color located 33% around the wheel from the principal color, rather than a full 50% away. This method is effective when highlighting two series, one of which is the primary focus. It's preferable to use warm colors for the key series and cool colors for the complementary ones. If required, the intensity of the complementary colors can be subdued by reducing their saturation or modifying their lightness,

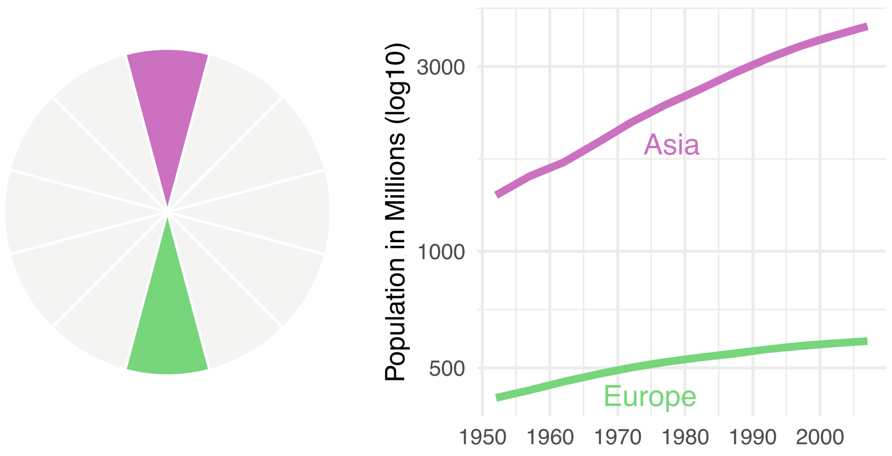


Figure 22-3 Complementary Harmony with a Positive/Negative Connotation

thereby lowering the contrast with the background. The example below underscores the significance of Asia's population growth, while Europe is neutrally depicted as a comparative reference, not a region with slow growth.

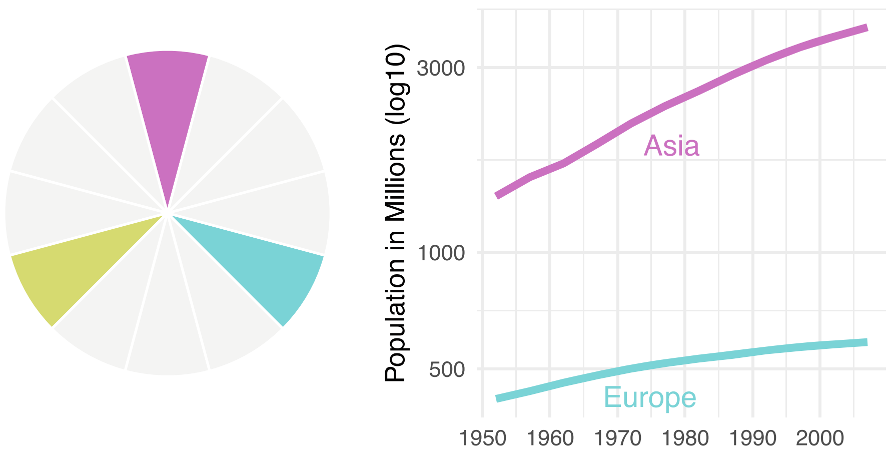


Figure 22-4 Near Complementary Harmony for Highlighting Two Series Where One Is the Primary Focus

22.1.3 Analogous/Triadic Harmony for Highlighting Three Series

Analogous harmony is effective for making simple distinctions between categories by using a key color and its two neighboring colors on the color wheel. This method, while simple, allows the key color to stand out slightly more. It is great for showing equally important categories, for example, showing population growth without emphasizing any aspects of it.

We have already introduced triadic harmony for the comparison of two categories. Here, triadic harmony can be used effectively to suggest a comparison between three continents. Remember

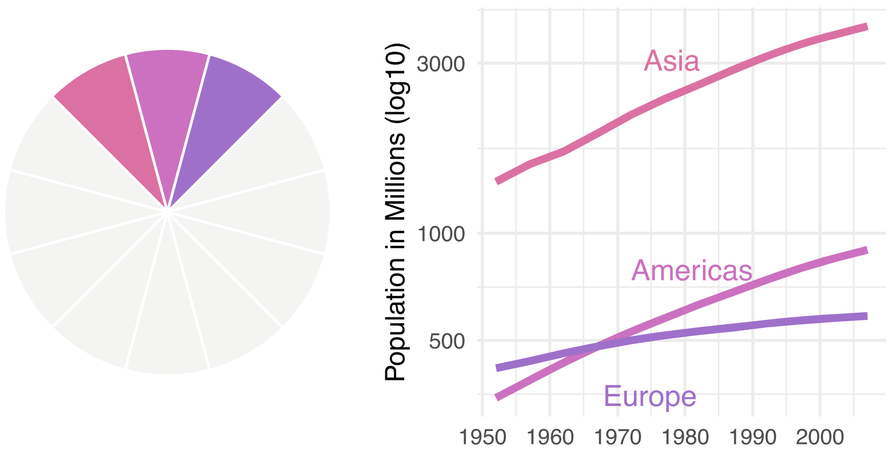


Figure 22-5 Analogous Harmony for Highlighting Three Series

that adjusting the tint of the colors can either increase or decrease emphasis as needed. Notice the difference in perception between analogous and triadic harmonies—what is it?

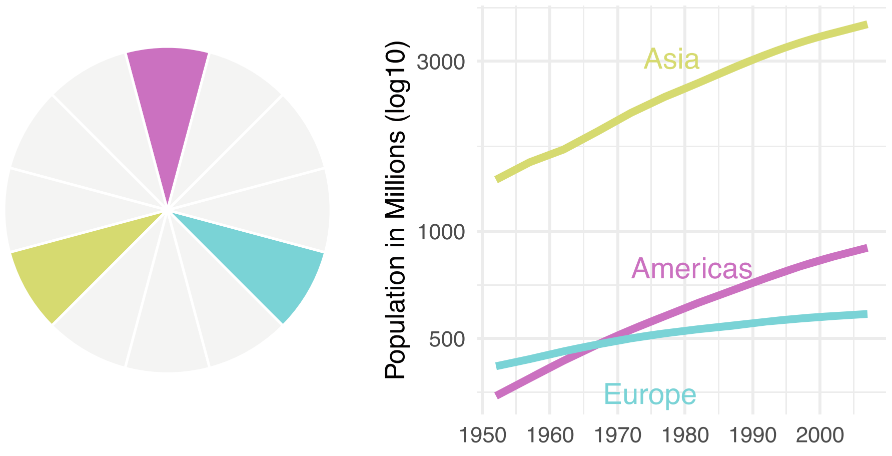


Figure 22-6 Triadic Harmony for Highlighting Three Series

22.1.4 Highlighting One Series Against Two Related Series

The near complementary harmony color scheme is adept at highlighting one series against others. The following chart clearly emphasizes Asia's GDP, represented by a vibrant purple. Conversely, Europe and the Americas, depicted in harmonizing greens, play a more subsidiary role in this narrative.

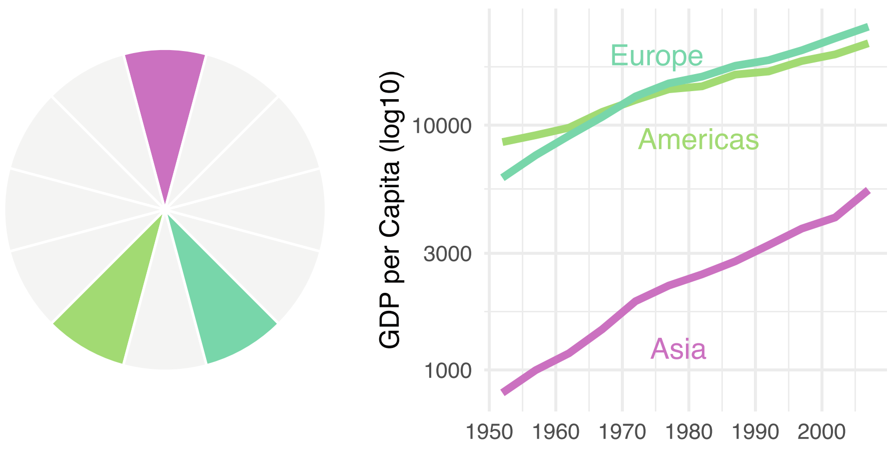


Figure 22-7 Highlighting One Series Against Two Related Series

22.1.5 Analogous Complementary for One Main Series and Its Three Secondary

The analogous complementary scheme, involving four distinct colors, provides an excellent platform for highlighting a primary series along with three related components. With this scheme, the key color stands out due to the similarities among the three complementary colors. An illustration of this can be seen in the subsequent example showcasing the so-called Malaysian Economic Miracle alongside three neighboring countries.

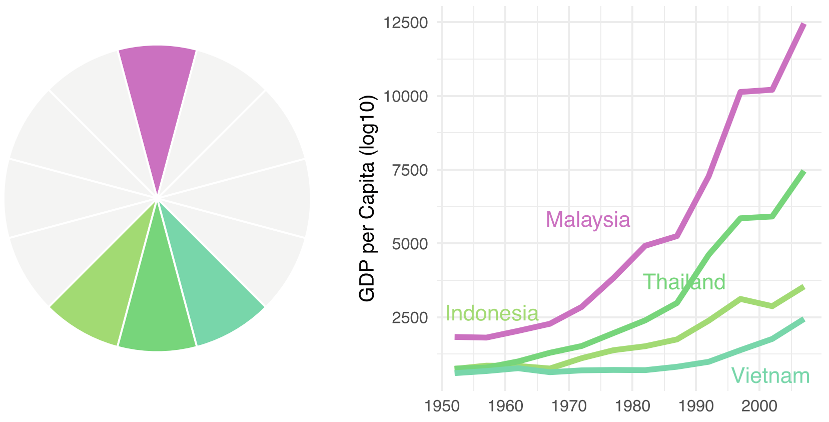


Figure 22-8 Analogous Complementary Scheme for One Main Series and Its Three Components

22.1.6 Double Complementary for Two Pairs Where One Pair Is Dominant

The double complementary harmony scheme is ideal for visualizing four data series divided into two distinct pairs. It involves the key color, an adjacent color, and their respective opposites on the color wheel. Warmer colors are suggested for the key and adjacent colors, with their complementary counterparts in cooler tones. This color arrangement effectively highlights one pair over the other. As demonstrated below, the 1952 GDPs of Switzerland and Norway form one group, shown in purple hues, while Bosnia and Albania, differentiated in green-blue, form the other group.

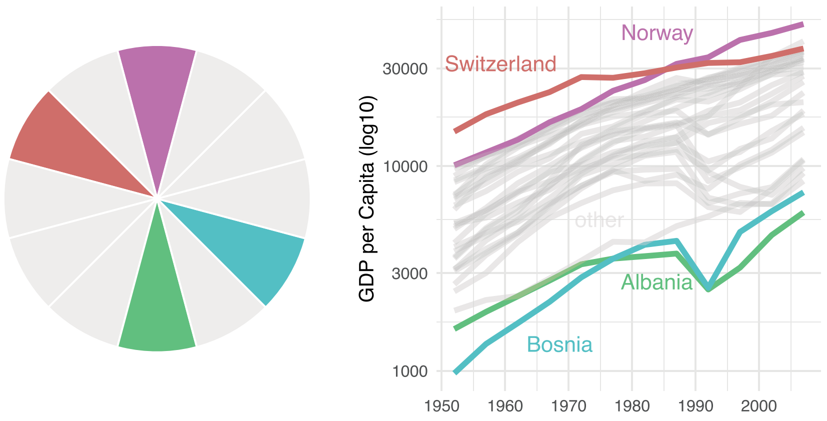


Figure 22-9 Double Complementary Harmony Scheme for Two Pairs Where One Pair Is Dominant

22.1.7 *Rectangular or Square Complementary Scheme for Four Series of Equal Emphasis*

The rectangular or square complementary scheme suits data visualization of four series with equal emphasis. Unlike the double complementary scheme, it includes the key color, its complement, and two additional colors to form a rectangle or square on the color wheel. This results in distinctive colors for each of the four series. While similar to the double complementary scheme, this scheme is optimal when all series share equal importance. For instance, the following graph shows the so-called Four Asian Tigers—Hong Kong, Singapore, South Korea, and Taiwan. These economies, rapidly developed from the 1960s to 1990s, are all equally significant in illustrating the dynamism of East Asia's growth.

22.1.8 *Sequential*

Sequential colors utilize a gradient from light to dark, mapping numeric values based on hue or lightness. Depending on the background, lower values receive lighter colors, while higher ones get darker shades. You can use a single hue or a sequence thereof. Let's apply our beloved purple to visualize COVID-19 deaths per 100,000 population in the United States.[1] With gray representing the minimum and vibrant purple indicating the maximum, can you identify the states with the fewest and most deaths? Do you find it easy to match the color to the corresponding value?

[1] Data from https://covid.cdc.gov/covid-data-tracker/#maps_deaths-rate-total

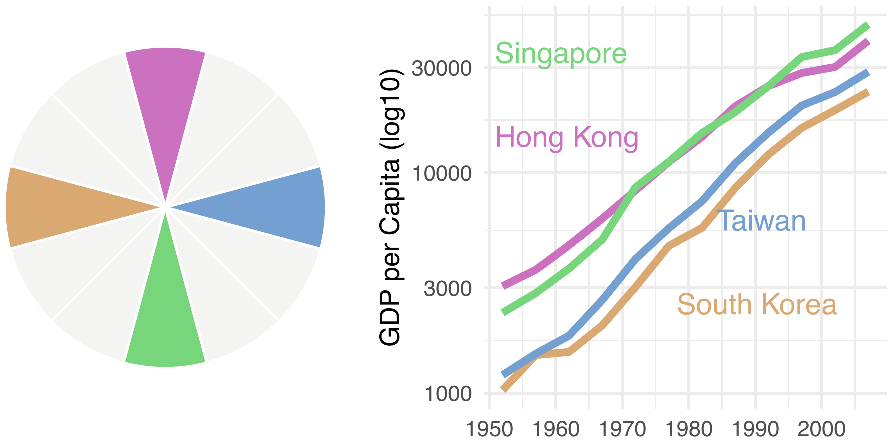


Figure 22-10 Rectangular or Square Complementary Scheme for Four Series of Equal Emphasis

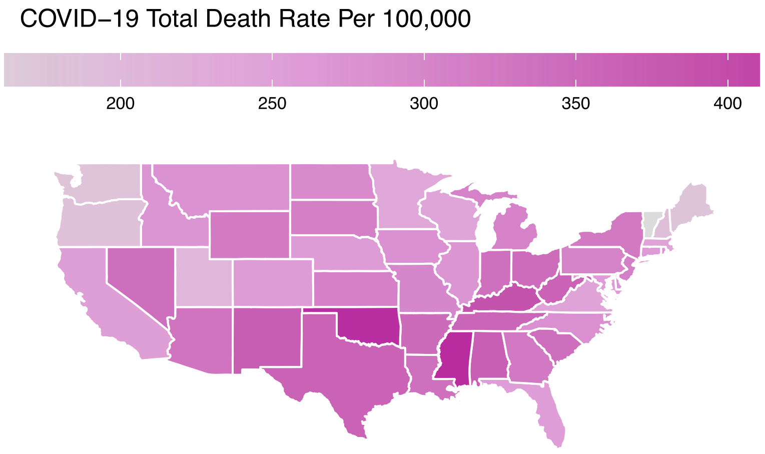


Figure 22-11 Sequential Color Scheme

22.1.9 Divergent

Diverging color schemes are employed when the numeric variable possesses a significant central value like zero. This scheme combines two sequential palettes with a common end, centering on the central value. Positive values receive colors from one side of the spectrum, while negative ones are designated colors from the other. The central value should ideally be a light shade, allowing darker colors to signify greater deviation from the center. Simplicity is key here to prevent diluting the intended meaning and confusing viewers.

Let’s scale our numbers so that they represent standard deviation as a value. Notice how states with significantly fewer deaths are blue and states with significantly more deaths are yellow, while states with numbers close to the average are a grayish color. Do you find it easy to identify states that are close to the mean (zero standard deviations away)?

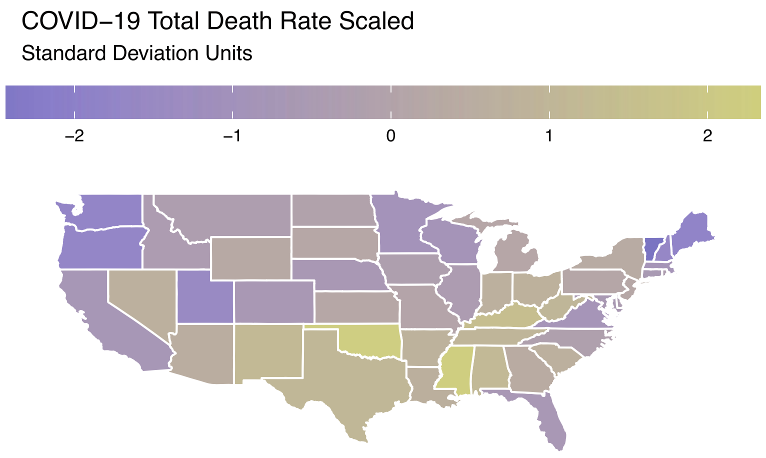


Figure 22-12 Divergent Color Scheme

22.1.10 Prebuilt

Prebuilt color scales such as Viridis are crafted for perceptual uniformity, ensuring visual appeal and ease of interpretation. These provide a standard, uniform color scheme, thereby obviating the need for custom creation and testing. Moreover, they aid individuals with color blindness in interpreting data visualizations, owing to their consistent visual contrast. By employing prebuilt color scales, data visualizations can be made accessible to a broad audience. Do you find it easier to match the colors to values and to group together states with similar values?

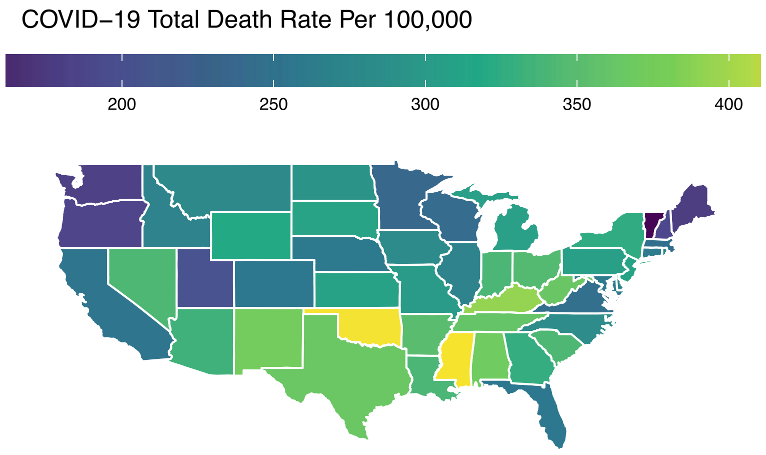


Figure 22-13 Prebuilt Viridis Color Scheme

22.2 Color Systems

Humans perceive colors differently than machines. For instance, colors that seem similar to a machine in red, green, blue (RGB) might not appear to be to the human eye. The following images illustrate the concept of perceptual uniformity using two color wheels: one RGB, which is perceptually nonuniform, and the other hue, chroma, lightness (HCL), which is more or less uniform. When viewed in grayscale, the nonuniform nature of the RGB color wheel becomes apparent. Notice how there are bright and dark sections, and compare this to the HCL version, which is uniform. Technically, a perceptually uniform color space ensures that the difference between two colors, as perceived by the human eye, is proportional to the Euclidean distance[2] within the given color space—in other words, similar looking colors have similar parameter values.

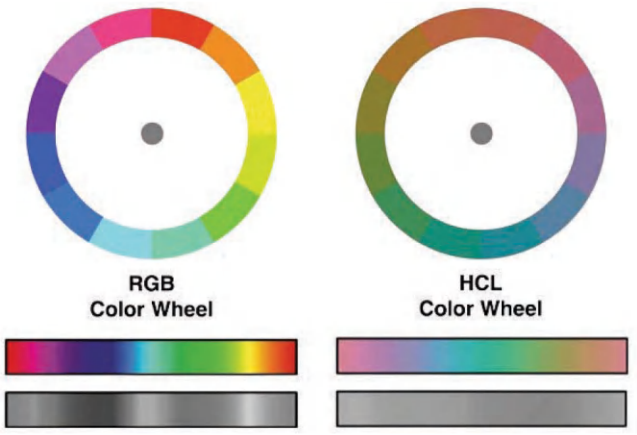


Figure 22-14 Uniform Perception

When it comes to selecting a color, you need a way to describe it beyond the English language. Several popular color systems are commonly used in digital design and data visualization to solve this problem.

The standard color space for displaying images and graphics on digital displays is the standard RGB (sRGB) color system, a device-dependent color space designed to provide consistent color reproduction across a wide range of devices. RGB is great for a single purpose: telling pixels in your display what they need to do and little else.

A considerable improvement is the HSV color system. You can think of HSV as a more user-friendly transformation of RGB, as it is significantly easier to find harmonic colors and alter human-understandable parameters: hue, saturation, and value. Hue, designated by a value from 0 to 360 degrees on the color wheel, determines the fundamental color of the pixel. Saturation denotes the purity of the hue, representing the degree of gray mixed into the color ranging from 0% (gray) to 100% (pure hue). Value signifies the brightness of the pixel, with 0% being black and 100% being the brightest possible color. Despite being an improvement over RGB, it suffers from the same perceptual nonuniformity.

An incremental improvement on HSV is the HSL color system, based on hue, saturation, and lightness. Hue and saturation function the same way as in the HSV system. Lightness, conversely,

[2] Euclidean distance is the straight-line distance between two points in a multidimensional space, calculated using the Pythagorean theorem.

symbolizes the proportion of white or black mixed with the color, with 0% being black, 50% being the pure color, and 100% being white. Despite its usefulness in graphic design and web development, the HSL system, like the HSV system, suffers from a lack of perceptual uniformity, although less so.

The HCL/LCH color system, based on hue, chroma, and lightness, is my go-to system for anything ranging from data visualization to graphic design. Often referred to as LCH or HCL depending on the field, this system is a significantly improved version of HSL, without the bias implicit in using varying saturation and being almost perceptually uniform. While hue and lightness behave the same as in HSL, chroma roughly represents the “amount of color” in a range between 0 and 150. It is often available in software, and its minor perceptual nonuniformity is outweighed by its ease of use.

Meanwhile, the LAB (L*a*b*) color system is a device-independent color space intended to accurately represent colors across various devices and environments. The LAB color system comprises three parameters: L* (lightness), a* (position between red/magenta and green), and b* (position between yellow and blue/cyan). This combination is based on the way our eyes perceive colors in these three complementary pairs. To better illustrate this, try the lilac chaser illusion, also known as the Pac-Man illusion, to experience the afterimage complementary color effect. To achieve this, follow the movement of the rotating blue-violettish (magenta) dots with your eyes, where the dots stay just one color, magenta. Now stare at the black × in the center. The magenta dots stop moving and a green dot starts flitting around the circle of magenta dots. This dot is green because green is the complement of pink. Stare at the central × long enough and reduce the saturation to around 20% and the magenta dots should disappear altogether. To experience the illusion, please visit the webpage.[3]

Note 22.1 The CIEDE2000 formula, used to calculate differences in color sensation in LAB, is perhaps one of the most bizarre mathematical equations in use. Look it up!

The LAB color space finds frequent use in quantitative applications of color such as spectrophotometers, color libraries, image editing programs, and data visualizations due to its ability to enable accurate color matching across diverse devices and environments. However, it is too complicated to use for manual color exploration.

OKLAB is a LAB-type color space engineered to be more perceptually uniform than other LAB variants, meaning that equal distances in the color space correspond to equal increments in perceived color difference. OKLAB is growing in popularity in digital design and data visualization due to its enhanced accuracy and consistency in color representation. A significant advantage of OKLAB over other color spaces is that you can use Euclidean distance to calculate perceptual color differences. For a deeper understanding of OKLAB and other color spaces, the blog posts on OKLAB[4] and Colorpicker[5] offer excellent insights.

Finally, there are also CMYK and Pantone, which are used in print. CMYK uses cyan, magenta, yellow, and key/black to construct colors. When printing, converting to CMYK is necessary, and if it is not done by you, the printer will handle it. The Pantone Matching System (PMS) offers standardized, precise colors for consistent printing results. Print is a whole other universe with its own problems. My advice: Just avoid overly saturated colors, and you will be fine.

Each of these color systems has its unique strengths and weaknesses, with the choice depending on the specific needs of the project.[6]

[3] https://michaelbach.de/ot/col-lilacChaser/index.html

[4] https://bottosson.github.io/posts/oklab/

[5] https://bottosson.github.io/posts/colorpicker/

[6] To see how some of these spaces look, check out this amazing video: https://youtu.be/HlDySNpGbyc.

22.2.1 *Warning: Colormaps Might Increase Risk of Death!*

In the 1990s, data visualization specialists adopted the rainbow color map, with the most renowned variation being the jet default palette. However, researchers expressed concerns about its nonuniform nature, which introduced transitions that could be misperceived. Notice how, in Figure 22-15d, the grayscale appears smooth and without visual breaks. In contrast, in Figures 22-15a and 22-15c, the colors change abruptly, creating breaks between them.

Rogowitz and Treinish [26] voiced concerns about the rainbow color map, while Borland and Taylor Ii [6] highlighted additional reasons why the rainbow color map is still considered harmful. Borkin et al. [5] conducted user studies on various color maps, including the rainbow color map, within medical visualization contexts and demonstrated that a perceptually uniform color map resulted in fewer diagnostic errors. Simply put, using an appropriate color palette can decrease diagnostic errors. However, Crameri et al. [12] concluded that the improper use of color still persists in science, emphasizing the importance of promptly adjusting our practices.

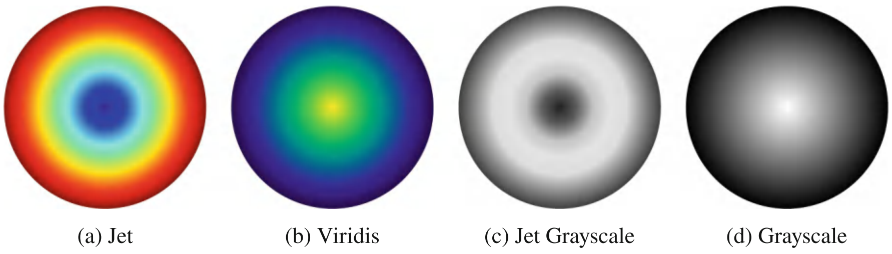


Figure 22-15 Rainbow (Jet) Color Map is Dangerous

These issues of using color for communication intensify when considering colorblind individuals. Approximately 8% of all men and 0.5% of all women are colorblind. There are three main forms of colorblindness: protan (red), deutan (green), and tritan (blue), each corresponding to color-sensitive cones in our eyes. Improving the readability of your colors involves varying their value and hue. Moreover, avoid including both red and green in your graphics as red-green color blindness is the most common form. To check whether your visualization is color-blind-friendly, you can use the Coblis simulator.[7]

22.2.2 *So What Should You Use?*

A simple and correct answer would be to use a scientific color map that you find appealing and make it your default. If you want to select colors yourself, use HCL/LCH, as it is the most intuitive and easiest to use in creating color palettes. You might also want to experiment with OKHSL, a child of OKLAB and HSL that produces a perceptually uniform HSL space. Try out both, and notice the differences.[8]

As general advice, when selecting colors, opt for pastel shades as they are generally softer and puts less strain on the eyes. Avoid high saturation because overly saturated colors can be distracting and may overshadow the data. Be mindful of using spectral colors, as they can cause afterimages and potentially distort the viewer's perception. Strategically utilizing color for grouping and searching

[7] https://www.color-blindness.com/coblis-color-blindness-simulator/

[8] https://bottosson.github.io/misc/colorpicker/

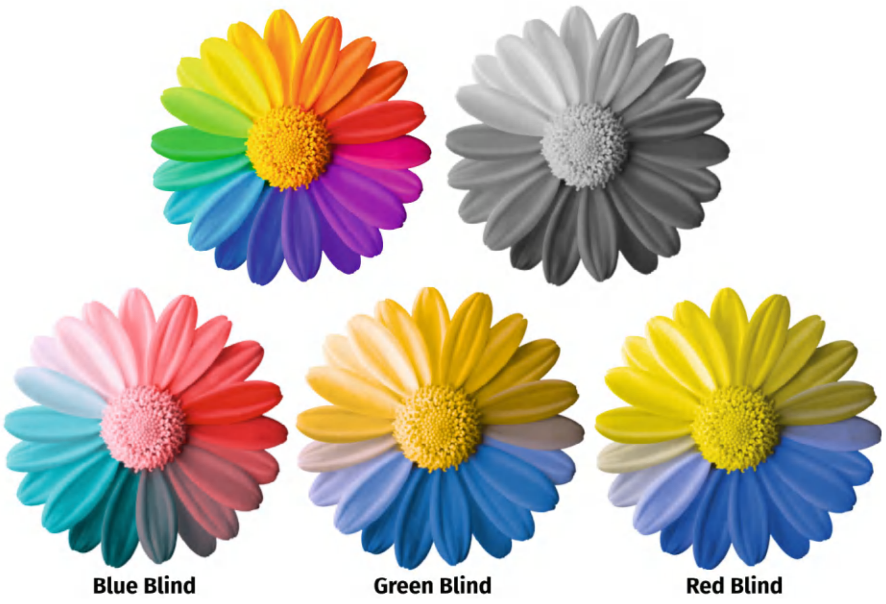


Figure 22-16 Color Blindness Example

purposes within a visualization can make it much easier for viewers to navigate and understand the information being presented. Skillful application of color in data visualization can enhance a viewer's understanding and engagement with the data, while inappropriate use may hide important details and lead to confusion. To assist you in the process of searching for more color options and creating palettes, refer to the following resources

- *Adobe Color*[9] allows you to create color palettes using different color harmony rules and color modes. You can also select colors from your image, create gradients from images, and test for accessibility.
- *Paletton*[10] is a fantastic tool for creating color palettes.
- *Color Brewer*[11] provides perceptually uniform color schemes for maps and data visualizations.
- *Color Thief*[12] lets you extract colors from your image to create nature-inspired palettes.
- *Viz Palette*[13] can be used to check your color palettes before creating visualizations. It allows you to view color sets in example plots, simulate color deficiencies, and modify the colors of your palette.
- *Scientific colour maps*[14] is a collection of uniform and readable color maps for scientific use.
- *Realtime Colors*[15] is a tool for visualizing and testing color palettes on websites, allowing users to apply and export colors efficiently in various formats.

[9] https://color.adobe.com/create/color-wheel
[10] https://paletton.com/
[11] http://colorbrewer2.org/
[12] https://lokeshdhakar.com/projects/color-thief/
[13] https://projects.susielu.com/viz-palette
[14] https://www.fabiocrameri.ch/colourmaps/
[15] https://realtimecolors.com

22.3 Summary

This chapter explored the strategic use of color schemes for organizing and presenting data effectively. Whether subtly distinguishing between categories or boldly contrasting them, the right application of color can make data more accessible and insightful. We also covered various color systems and the importance of choosing the right colors. In the next chapter, we will explore how to create effective tables!

Chapter 23
Make Tables

Honestly, I hate making tables. I do not know why, but they never turn out right. However, the `gt` (Grammar of Tables) package makes it easy and actually enjoyable. Therefore, this section will focus on creating simple summary tables using `gt`. It's essential to note that `gt` is just one of many available packages for table creation; others include `kableExtra` for more complex tables, `DT` tables for HTML interactive tables, `reactable` for reactive tables, and `flextable`, among others. Unlike visualization and data wrangling, where the use of `ggplot`, `plotly`, `dplyr`, and `data.table` is nearly universal, there is no default package for table creation. Hence, we'll look at `gt` due to its ease of use and ability to handle a range of basic tasks. The `gt` package simplifies the creation of elegant, customizable tables, making it a versatile tool for reports, presentations, and web applications.

Note 23.1 Ask ten R users about their preferred table package, and you'll likely receive twenty different answers.

23.1 `gt` Tables

To start, ensure you have the `gt` package installed. We'll also use the `gtExtras` package to expand the `gt` package's capabilities with custom themes, conditional formatting, and more. In addition, we'll use the `emojifont` package for accessing emojis and `dplyr` from `tidyverse` for data manipulation.

```r
# install.packages(c("gt","gtExtras","emojifont"))
library(dplyr)
library(gt)
library(gtExtras)
library(emojifont)
```

As an example we will use the built-in `gtcars` dataset, which contains information on various automobiles. Let's look at it using `gt()`.

Note 23.2 I will remove `trim` column, so the table fits on a page.

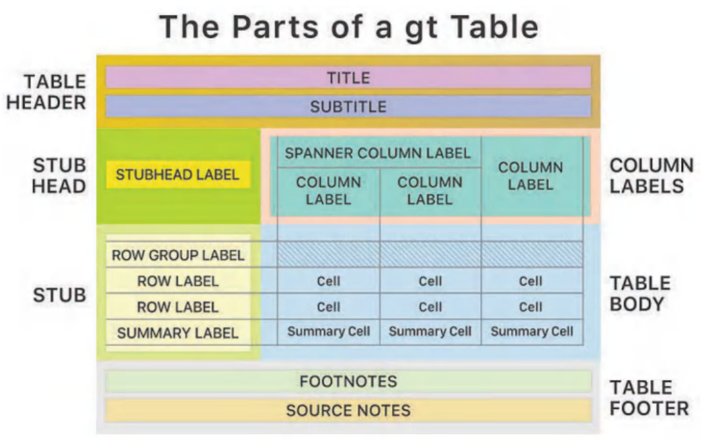


Figure 23-1 GT Table Parts

```r
gt::gtcars %>%
  select(-trim) %>%
  dplyr::sample_n(size = 4) %>%
  gt()
```

mfr	model	year	bdy_style	hp	hp_rpm	trq	trq_rpm	mpg_c	mpg_h	drivetrain	trsmn	ctry_origin	msrp
Ferrari	California	2015	convertible	553	7500	557	4750	16	23	rwd	7a	Italy	198973
Porsche	718 Boxster	2017	convertible	300	6500	280	1950	21	28	rwd	6m	Germany	56000
Nissan	GT-R	2016	coupe	545	6400	436	3200	16	22	awd	6a	Japan	101770
Maserati	Granturismo	2016	coupe	454	7600	384	4750	13	21	rwd	6am	Italy	132825

23.1.1 Prepare Your Data

The `gt` package works seamlessly with `dplyr`, allowing us to utilize familiar verbs for table formatting. Let's start by focusing on the top auto industry country, Germany. We'll group cars by their manufacturer (mfr) and sort them based on their price (msrp). During table creation, the `rowname_col` parameter allows us to designate a column as row names—for this example, we'll use the "model" column.

```r
table_grouped <- gt::gtcars %>%
  select(-trim) %>%
  filter(ctry_origin == "Germany") %>% # Filter to German cars
  group_by(mfr) %>% # Group by the manufacturer
  arrange(desc(msrp)) %>% # Arrange in descending order on price
  slice_head(n = 2) %>% # Select the top two cars
  filter(mfr %in% c("Audi", "BMW")) %>% # Filter to Audi and BMW cars
  gt::gt(rowname_col = "model") # Table with car models as row names

table_grouped
```

	Year	bdy_style	hp	hp_rpm	trq	trq_rpm	mpg_c	mpg_h	drivetrain	trsmn	ctry_origin	msrp
Audi												
R8	2015	coupe	430	7900	317	4500	11	20	awd	6m	Germany	115900
S8	2016	sedan	520	5800	481	1700	15	25	awd	8am	Germany	114900
BMW												
i8	2016	coupe	357	5800	420	3700	28	29	awd	6am	Germany	140700
M6	2016	coupe	560	6000	500	1500	15	22	rwd	7a	Germany	113400

The table contains many columns, and to make it more readable, we can hide some columns using ***`cols_hide`***`()`. Why use ***`cols_hide`***`()` instead of dropping the columns with `dplyr`? Sometimes, we may need to utilize a column in conditional statements, but we do not want to display it. To group performance-related columns together, we can use ***`cols_move`***`()`. Though this could be done with `dplyr`, integrating it into our table-creation process makes the workflow more streamlined. We can distinguish these grouped columns by adding a header spanner with ***`tab_spanner`***`()`, specifying the `columns` and setting the `label`.

```
table_span <- table_grouped %>%
  gt::cols_hide( # Hide certain columns
    columns = c(bdy_style, drivetrain, ctry_origin, trsmn)
  ) %>%
  gt::cols_move( # Move specific columns after the `year`
    columns = c(msrp, trsmn, mpg_c, mpg_h),
    after = year
  ) %>%
  gt::tab_spanner( # Create a spanner header
    columns = c(mpg_c, mpg_h, hp, hp_rpm, trq, trq_rpm),
    label = "Performance"
  )

table_span
```

			Performance					
	year	msrp	mpg_c	mpg_h	hp	hp_rpm	trq	trq_rpm
Audi								
R8	2015	115900	11	20	430	7900	317	4500
S8	2016	114900	15	25	520	5800	481	1700
BMW								
i8	2016	140700	28	29	357	5800	420	3700
M6	2016	113400	15	22	560	6000	500	1500

At times, we may want to combine pairs of related columns into a single column. For that, we can use the ***`cols_merge`***`()` function. We'll merge horsepower (hp) with associated rpm (hp_rpm), torque (trq) with associated rpm (trq_rpm), and city miles per gallon (mpg_c) with highway miles per gallon (mpg_h). The function takes `columns`, which can be referenced in the `text` with `{#}` of columns. Since we are working with HTML tables, we can use HTML tags, specifically <br> to add a line break. Alternatively, for LaTeX/PDF documents, you can use `\shortstack{{1}  {2}}` or, even better, `\makecell{{1}  {2}}` from the LaTeX `makecell` package. Note that the joined column will use the first column's name, and the labels are purely cosmetic and can't be used as reference. We'll then use the `cols_label()` function to assign custom labels to the columns, using the `column_name = "column_label"` syntax.

```
table_merge <- table_span %>%
  # Merge the horsepower (hp) and horsepower rpm (hp_rpm) columns.
  cols_merge(columns = c(hp, hp_rpm), pattern = "{1}<br>@{2}rpm") %>%
  # Merge the torque (trq) and torque rpm (trq_rpm) columns.
  cols_merge(columns = c(trq, trq_rpm), pattern = "{1}<br>@{2}rpm") %>%
  # Merge the city mpg (mpg_c) and highway mpg (mpg_h) columns.
  cols_merge(columns = c(mpg_c, mpg_h), pattern = "{1}c<br>{2}h") %>%
  cols_label( # Set custom labels for specific columns.
    year = "Year",
    msrp = "MSRP",
    mpg_c = "MPG",
    hp = "Horsepower",
    trq = "Torque"
  )

table_merge
```

			Performance		
	Year	MSRP	MPG	Horsepower	Torque
Audi					
R8	2015	115900	11c 20h	430 @7900rpm	317 @4500rpm
S8	2016	114900	15c 25h	520 @5800rpm	481 @1700rpm
BMW					
i8	2016	140700	28c 29h	357 @5800rpm	420 @3700rpm
M6	2016	113400	15c 22h	560 @6000rpm	50 @1500rpm

The `gt` package provides a variety of `fmt_*()` formatting functions that are useful for adjusting the display of numeric and text columns. For example, we can set the `msrp` column to display currency in USD without decimals. We can adjust the alignment of columns using `cols_align()`, allowing us to select columns and set alignments to "right," "left," or "center." Since the merged columns appear bulky with two lines, we can decrease the text size using `tab_style()`. This function utilizes a `style` definition, provided by helper functions such as `cell_styles()` that include supported style information. In this example, we use `cell_text()` to set the text size to "12px." The second argument, `locations`, uses another helper function, `cells_*()`. To target cells in the body, we apply `cells_body()` to the `mpg_c`, `hp`, and `trq` columns. To enhance our tables with color, we can use the `data_color` function, which supports the creation of a simple gradient or the application of a solid color to the data. The function can specify a `domain` and use prebuilt palettes. For more functionality, consider `gt_color_rows()` from `gtExtras`. For more information, refer to the package's documentation.

```r
table_format <- table_merge %>%
  # Format the msrp column as currency
  fmt_currency(columns = msrp, decimals = 0, currency = "USD") %>%
  # Center align specific columns
  cols_align(columns = c(mpg_c, hp, trq), align = "center") %>%
  tab_style( # Apply cell text style to specific columns
    style = cell_text(size = "12px"),
    locations = cells_body(columns = c(mpg_c, hp, trq))
  ) %>%
  # Apply a data-driven color scale to the msrp column
  data_color(columns = msrp, colors = c("white", "aquamarine"))

table_format
```

			Performance		
	Year	MSRP	MPG	Horse Power	Torque
Audi					
R8	2015	$115,900	11c 20h	430 @7900rpm	317 @4500rpm
S8	2016	$114,900	15c 25h	520 @5800rpm	481 @1700rpm
BMW					
i8	2016	$140,700	28c 29h	357 @5800rpm	420 @3700rpm
M6	2016	$113,400	15c 22h	560 @6000rpm	500 @1500rpm

Now the table is looking quite good! We can give it a title, “German Automobiles,” and an emoji-inclusive subtitle with the `tab_header()` function. To clarify the term “MSRP” for readers who might not know its meaning, we can use a footnote. Footnotes can be added to individual cells by either specifying the row number or using expressions—here, the most expensive car is the BMW i8, which is also electric! Like `tab_style()`, you need to provide the `location` and `footnote` text. Don’t forget to add your data source with the `tab_source_note()` function. Wrapping text in `md()` allows us to utilize Markdown syntax for formatting, including adding links.

```r
table_header <- table_format %>%
  # Add a title and a subtitle with an emoji.
  tab_header(
    title = "German Automobiles",
    subtitle = paste0("These are some nice ",
                      emojifont::emoji("car"), "s")
  ) %>%
  # Add a footnote to clarify what MSRP stands for.
  tab_footnote(
    locations = cells_column_labels(columns = msrp),
    footnote = "Manufacturer's Suggested Retail Price in USD"
  ) %>%
  # Add a footnote to the most expensive car
  tab_footnote(
    locations = cells_body(msrp, msrp == max(msrp)),
    footnote = "Electric cars used to be expensive"
  ) %>%
  # Add a source note to cite the source of the data.
  tab_source_note(source_note = md("Source: **gtcars** [dataset from gt
  package](https://gt.rstudio.com/articles/gt-datasets.html)"))

table_header
```

German Automobiles

These are some nice s

			Performance		
	Year	MSRP[1]	MPG	Horsepower	Torque
Audi					
R8	2015	$115,900	11c 20h	430 @7900rpm	317 @4500rpm
S8	2016	$114,900	15c 25h	520 @5800rpm	481 @1700rpm
BMW					
i8	2016	[2] $140,700	28c 29h	357 @5800rpm	420 @3700rpm
M6	2016	$113,400	15c 22h	560 @6000rpm	500 @1500rpm

[1] Manufacturer’s Suggested Retail Price in USD.
[2] Electric cars used to be expensive.
Source: **gtcars** dataset from gt package

To take your `gt` tables a step further, you can apply themes using the `gtExtras` package. This package offers a variety of prebuilt themes that can be easily added to your tables. After installing and loading the `gtExtras` package, you can use one of its theme functions to apply a specific style to your table. Let’s add a theme similar to that of the *FiveThirtyEight website*.[1]

[1] https://projects.fivethirtyeight.com/

```r
# Add Five Thirty Eight theme to the table
(table_themed <- table_header %>% gtExtras::gt_theme_538())
```

German Automobiles

These are some nice s

			Performance		
	Year	MSRP[1]	MPG	Horse Power	Torque
Audi					
R8	2015	$115,900	11c 20h	430 @7900rpm	317 @4500rpm
S8	2016	$114,900	15c 25h	520 @5800rpm	481 @1700rpm
BMW					
i8	2016	[2] $140,700	28c 29h	357 @5800rpm	420 @3700rpm
M6	2016	$113,400	15c 22h	560 @6000rpm	500 @1500rpm

[1]Manufacturer's Suggested Retail Price in USD.
[2]Electric cars used to be expensive.
Source: **gtcars** dataset from gt package

After creating your beautiful table, you'll likely want to save it. Although there isn't a direct method for saving to Excel, the `gt` package supports saving the table in various formats such as `.html` for embedding in websites or emails, `.png` for images, `.tex` for LaTeX documents, `.pdf` for PDF files, and `.docx` for Microsoft Word documents.

To enhance the usability and appearance of the saved tables, you can use additional options. For instance, saving the `gt` table as an HTML file with the `inline_css = TRUE` option is useful when embedding the table in an HTML email, ensuring the styles are applied directly within the email. Without this option, the HTML file will have embedded Cascading Style Sheets (CSS) styles. When saving a table as a PNG file, it creates a cropped image of the HTML table. You can adjust the whitespace around the image using the expand option, allowing for better presentation and integration into other documents or presentations.

Note 23.3 Note that you will need to install the `webshot2` package to save in some formats.

```r
gtsave(table_themed, filename = "tab_1.html", inline_css = TRUE)
gtsave(table_themed, "tab_1.png", expand = 10)
gtsave(table_themed, "tab_1.tex")
gtsave(table_themed, "tab_1.pdf")
gtsave(table_themed, "tab_1.docx")
```

And the final output will look something like this:

```r
library(dplyr)
library(gt)
library(gtExtras)
library(emojifont)

gt::gtcars %>%
  select(-trim) %>%
  filter(ctry_origin == "Germany") %>%
  group_by(mfr) %>%
  arrange(desc(msrp)) %>%
  slice_head(n = 2) %>%
  filter(mfr %in% c("Audi", "BMW")) %>%
  gt::gt(rowname_col = "model") %>%
  gt::cols_hide(
    columns = c(bdy_style, drivetrain, ctry_origin, trsmn)
  ) %>%
  gt::cols_move(
    columns = c(msrp, trsmn, mpg_c, mpg_h),
    after = year
  ) %>%
  gt::tab_spanner(
    columns = c(mpg_c, mpg_h, hp, hp_rpm, trq, trq_rpm),
    label = "Performance"
  ) %>%
  cols_merge(columns = c(hp, hp_rpm), pattern = "{1}<br>@{2}rpm") %>%
  cols_merge(columns = c(trq, trq_rpm), pattern="{1}<br>@{2}rpm") %>%
  cols_merge(columns = c(mpg_c, mpg_h), pattern = "{1}c<br>{2}h") %>%
  cols_label(
    year = "Year",
    msrp = "MSRP",
    mpg_c = "MPG",
    hp = "Horse Power",
    trq = "Torque"
  ) %>%
  fmt_currency(columns = msrp, decimals = 0, currency = "USD") %>%
  cols_align(columns = c(mpg_c, hp, trq), align = "center") %>%
  tab_style(
    style = cell_text(size = "12px"),
    locations = cells_body(columns = c(mpg_c, hp, trq))
  ) %>%
  data_color(columns = msrp, colors = c("white", "aquamarine")) %>%
  tab_header(
    title = "German Automobiles",
    subtitle = paste0("These are some nice ",
                      emojifont::emoji("car"), "s")
  ) %>%
  tab_footnote(
    locations = cells_column_labels(columns = msrp),
    footnote = "Manufacturer's Suggested Retail Price in USD"
  ) %>%
  tab_footnote(
    locations = cells_body(msrp, msrp == max(msrp)),
    footnote = "Electric cars used to be expensive"
  ) %>%
  tab_source_note(source_note = md("Source: **gtcars** [dataset from gt
  package](https://gt.rstudio.com/articles/gt-datasets.html)")) %>%
  gtExtras::gt_theme_538()
```

German Automobiles

These are some nice s

			Performance		
	Year	MSRP[1]	MPG	Horse Power	Torque
BMW					
i8	2016	[2] $140,700	28c 29h	357 @5800rpm	420 @3700rpm
M6	2016	$113,400	15c 22h	560 @6000rpm	500 @1500rpm
M5	2016	$94,100	15c 22h	560 @6000rpm	500 @1500rpm
6-Series	2016	$77,300	20c 30h	315 @5800rpm	330 @1400rpm
M4	2016	$65,700	17c 24h	425 @5500rpm	406 @1850rpm
Mercedes-Benz					
AMG GT	2016	$129,900	16c 22h	503 @6250rpm	479 @1750rpm
SL-Class	2016	$85,050	20c 27h	329 @5250rpm	354 @1600rpm
Audi					
R8	2015	$115,900	11c 20h	430 @7900rpm	317 @4500rpm
S8	2016	$114,900	15c 25h	520 @5800rpm	481 @1700rpm
RS 7	2016	$108,900	15c 25h	560 @5700rpm	516 @1750rpm
S7	2016	$82,900	17c 27h	450 @5800rpm	406 @1400rpm
S6	2016	$70,900	18c 27h	450 @5800rpm	406 @1400rpm
Porsche					
911	2016	$84,300	20c 28h	350 @7400rpm	287 @5600rpm
Panamera	2016	$78,100	18c 28h	310 @6200rpm	295 @3750rpm
718 Boxster	2017	$56,000	21c 28h	300 @6500rpm	280 @1950rpm
718 Cayman	2017	$53,900	20c 29h	300 @6500rpm	280 @1950rpm

[1]Manufacturer's Suggested Retail Price in USD.
[2]Electric cars used to be expensive.
Source: **gtcars** dataset from gt package

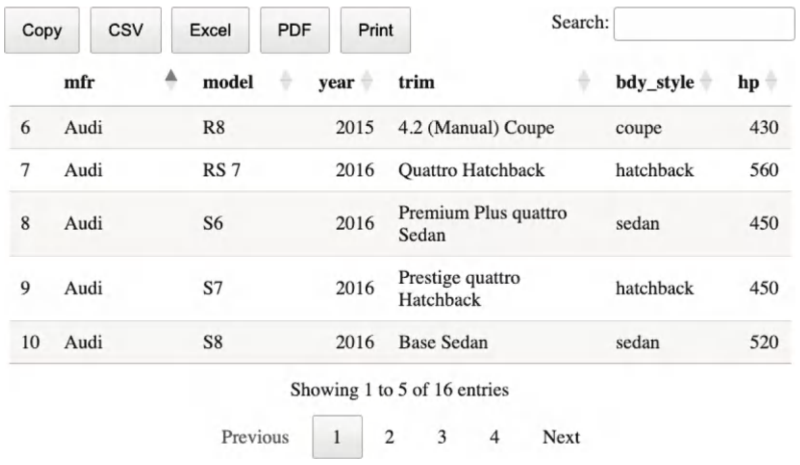
23.2 DT Tables

When you need to present data to important stakeholders, static HTML tables can sometimes fall short. The `DT` package in R can be a great solution for this because it lets you include interactive tables in your reports or analyses. `DT` is an interface to the JavaScript library DataTables. It's a flexible tool for displaying data in tables, with lots of customization options. Unlike the `gt` package, which is designed more for creating tables ready for publication, `DT` is really about making tables interactive for better data exploration as it lets you sort, filter, and paginate tables, which is really useful when dealing with larger datasets. It also has features like filters for individual columns and the ability to hide or rearrange columns. Plus, there are several options for how you present your data. Let's see how to use the `DT` package to create a table with a dataset of German automobiles:

```r
library(DT)

german_cars <- gt::gtcars %>%
  filter(ctry_origin == "Germany") %>%
  select(mfr, model, year, trim, bdy_style, hp)

DT::datatable(
  german_cars,
  extensions = "Buttons", # Enable Buttons extension
  options = list(
    dom = "Bfrtip", # Table control elements and their order
    buttons = c("copy", "csv", "excel", "pdf", "print"),
    pageLength = 5, # Rows per page
    autoWidth = TRUE, # Adjust column width
    order = list(list(1, "asc")), # Initial sorting order
    searching = TRUE, # Enable search
    searchHighlight = TRUE, # Highlight search matches
    lengthChange = TRUE # Allow changing entries per page
  )
```



	mfr	model	year	trim	bdy_style	hp
6	Audi	R8	2015	4.2 (Manual) Coupe	coupe	430
7	Audi	RS 7	2016	Quattro Hatchback	hatchback	560
8	Audi	S6	2016	Premium Plus quattro Sedan	sedan	450
9	Audi	S7	2016	Prestige quattro Hatchback	hatchback	450
10	Audi	S8	2016	Base Sedan	sedan	520

Showing 1 to 5 of 16 entries

Previous 1 2 3 4 Next

Figure 23-2 DT Table Example

23.3 Summary

In this chapter, we explored table creation in R using the `gt` package and walked through an example with the `gtcars` dataset, covering data manipulation, styling, and theming. Additionally, we introduced the `DT` package for interactive tables. At this point you should be ready to start on your own project!

Epilogue

In the realm of data science and analytics, the path to mastery is not a sprint but a marathon. Acquiring the skills and tools is not a one-time affair; it's a continuous journey of exploration, curiosity, and growth. Just as martial artists don't stop training upon receiving their black belt, data enthusiasts don't halt their learning once they've grasped the foundational tools. The black belt signifies the beginning of true mastery, not its culmination.

It's tempting, in our fast-paced world, to focus on immediate gains, to seek the quick wins that will propel us forward in the short term. But true expertise, the kind that sets you apart and allows you to make meaningful contributions, requires a longer view. It demands that we look beyond the horizon of the next month and envision where we want to be in five years, or even a decade.

As you embark on this journey, remember that the landscape of data science is ever evolving. New tools emerge, methodologies adapt, and the questions we seek to answer become more complex. Embrace the continuous learning process, nurture your curiosity, and set your sights on the long term for in this field, the journey itself is the destination, and every step you take enriches your understanding and expertise.

N. Tkachenko, *Data Insight Foundations*, https://doi.org/10.1007/979-8-8688-0580-6

References

[1] Abadie A, Athey S, Imbens G, Wooldridge J (2017) When should you adjust standard errors for clustering? NBER p w24003. https://doi.org/10.3386/w24003. http://www.nber.org/papers/w24003.pdf

[2] Bai J (2009) Panel data models with interactive fixed effects. Econometrica 77(4):1229–1279. https://doi.org/10.3982/ecta6135. MAG ID: 2128249713

[3] Baker M (2016) 1,500 scientists lift the lid on reproducibility. Nature 533(7604):452–454. https://doi.org/10.1038/533452a. https://www.nature.com/articles/533452a

[4] Bansak K, Hainmueller J, Hopkins DJ, Yamamoto T (2021) Beyond the breaking point? Survey satisficing in conjoint experiments. Polit Sci Res Methods 9(1):53–71

[5] Borkin M, Gajos K, Peters A, Mitsouras D, Melchionna S, Rybicki F, Feldman C, Pfister H (2011) Evaluation of artery visualizations for heart disease diagnosis. IEEE Trans Visualiz Comput Graph 17(12):2479–2488. https://doi.org/10.1109/TVCG.2011.192. http://ieeexplore.ieee.org/document/6065015/

[6] Borland D, Taylor Ii RM (2007) Rainbow color map (still) considered harmful. IEEE Comput Graph Appl 27(2):14–17. https://doi.org/10.1109/MCG.2007.323435. https://ieeexplore.ieee.org/document/4118486/

[7] Broman KW, Woo KH (2018) Data organization in spreadsheets. Am Stat 72(1):2–10. https://doi.org/10.1080/00031305.2017.1375989. https://www.tandfonline.com/doi/full/10.1080/00031305.2017.1375989

[8] Bruner GC (2019) Marketing scales handbook: multi-item measures for consumer insight research, vol 10. GCBII Productions, LLC, Fort Worth

[9] Buuren S, Groothuis-Oudshoorn C (2011) Mice: multivariate imputation by chained equations in R. J Stat Softw 45:1–67. https://doi.org/10.18637/jss.v045.i03

[10] Community TTW (2022) The turing way: a handbook for reproducible, ethical and collaborative research. Zenodo. https://doi.org/10.5281/ZENODO.3233853. https://zenodo.org/record/3233853

[11] Contgreave A (2022) The “Levers” of Chart-Making. https://www.linkedin.com/pulse/levers-chart-making-andy-cotgreave/

[12] Crameri F, Shephard GE, Heron PJ (2020) The misuse of colour in science communication. Nat Commun 11(1):5444. https://doi.org/10.1038/s41467-020-19160-7. https://www.nature.com/articles/s41467-020-19160-7

[13] Flannery JJ (1971) The relative effectiveness of some common graduated point symbols in the presentation of quantitative data. Cartograph Int J Geograph Inf Geovisualiz 8(2):96–109. https://doi.org/10.3138/J647-1776-745H-3667. https://utpjournals.press/doi/10.3138/J647-1776-745H-3667

[14] Groves RM, Fowler Jr FJ, Couper MP, Lepkowski JM, Singer E, Tourangeau R (2011) Survey methodology. Wiley, Hoboken

[15] Healey CG (2012) Perception in visualization. https://www.csc2.ncsu.edu/faculty/healey/PP/#jscript_search

[16] Hester J (2023) Let’s Git started | Happy Git and GitHub for the useR. Self-Published. https://happygitwithr.com/

[17] Hoerger M (2010) Participant dropout as a function of survey length in internet-mediated university studies: implications for study design and voluntary participation in psychological research. Cyberpsychol Behavior Soc Netw 13(6):697–700. Mary Ann Liebert, New Rochelle

[18] Jakobsen JC, Gluud C, Wetterslev J, Winkel P (2017) When and how should multiple imputation be used for handling missing data in randomised clinical trials – a practical guide with flowcharts. BMC Med Res Methodol 17(1):162. https://doi.org/10.1186/s12874-017-0442-1. https://bmcmedresmethodol.biomedcentral.com/articles/10.1186/s12874-017-0442-1

[19] Jarrett C (2021) Surveys that work: a practical guide for designing and running better surveys. Rosenfeld Media, Brooklyn

[20] John LK, Acquisti A, Loewenstein G (2011) Strangers on a plane: context-dependent willingness to divulge sensitive information. J Consumer Res 37(5):858–873
[21] Kahneman D (2011) Thinking, fast and slow, 1st edn. Farrar, Straus and Giroux, New York
[22] Krosnick JA (2018) Questionnaire design. In: The palgrave handbook of survey research, pp 439–455. Springer, Heidelberg
[23] Krosnick JA, Holbrook AL, Berent MK, Carson RT, Michael Hanemann W, Kopp RJ, Cameron Mitchell R, Presser S, Ruud PA, Kerry Smith V (2002) The impact of "no opinion" response options on data quality: non-attitude reduction or an invitation to satisfice? Public Opin Quarterly 66(3):371–403
[24] Matejka J, Fitzmaurice G (2017) Same stats, different graphs: generating datasets with varied appearance and identical statistics through simulated annealing. In: Proceedings of the 2017 CHI conference on human factors in computing systems, pp 1290–1294. https://doi.org/10.1145/3025453.3025912, Citation Key: Inproceedings
[25] Popper K (2002) The logic of scientific discovery. ISSR library, Routledge. https://books.google.com/books?id=Yq6xeupNStMC. Citation Key: popper2002logic tex.lccn: 92241375
[26] Rogowitz B, Treinish L (1998) Data visualization: the end of the rainbow. IEEE Spectrum 35(12):52–59. https://doi.org/10.1109/6.736450. https://ieeexplore.ieee.org/document/736450/
[27] Sibinga E, Waldron E (2021) Cognitive load as a guide: 12 spectrums to improve your data visualizations | nightingale. https://nightingaledvs.com/cognitive-load-as-a-guide-12-spectrums-to-improve-your-data-visualizations/
[28] Stodden V (2014) 2014: What scientific idea is ready for retirement? Edgeorg. https://www.edge.org/response-detail/25340. Citation Key: Victoria2014Reproducibility
[29] Tourangeau R, Rips LJ, Rasinski K (2000) The psychology of survey response. Cambridge University Press, Cambridge
[30] Treisman A, Gelade G (1980) A feature-integration theory of attention. Cognitive Psycholo
[31] Tufte ER (2001) The visual display of quantitative information, 2nd edn. Graphics Press, Cheshire
[32] Vagias WM (2006) Likert-type scale response anchors. Clemson International Institute for Tourism & Research Development, Department of Parks, Recreation and Tourism Management Clemson University
[33] Vriesema CC, Gehlbach H (2021) Assessing survey satisficing: the impact of unmotivated questionnaire responding on data quality. Edu Res 50(9):618–627
[34] Ware C (2021) Information visualization: perception for design, 4th edn. Morgan Kaufmann, Cambridge
[35] Wickham H (2010) A layered grammar of graphics. J Comput Graph Stat 19(1):3–28. https://doi.org/10.1198/jcgs.2009.07098. Taylor I& Francis tex.eprint: https://doi.org/10.1198/jcgs.2009.07098 Citation Key: Hadleygrammar2010
[36] Wickham H (2014) Tidy data. J Stat Softw 59(10). https://doi.org/10.18637/jss.v059.i10
[37] Wigmore S (2022) What is a good survey length for online research? https://www.kantar.com/north-america/inspiration/research-services/what-is-a-good-survey-length-for-online-research-pf
[38] Wilkinson L (1999) The grammar of graphics. Statistics and computing. Springer New York. https://books.google.com/books?id=5boZAQAAIAAJ

Index