Research Toolkit

Nikita Tkachenko

2023-09-20

Table of contents

Preface

> Optimizing Research Toolkit: A Guide for What School Forgot to Teach.

This book was born from my frustrations and experiences in higher education. Students typically learn about models, experiments, and theories during their classes and frequently return to that knowledge. However, executing high-quality research and analysis necessitates a deeper understanding of the tools and the "How" rather than just the "What" and "Why". The knowledge required often transcends what is taught within the constraints of a standard curriculum. In this book, I aim to bridge that gap between knowing what you want to do and understanding how to do it. I've distilled hundreds of hours of frustrations into these chapters, so you won't have to traverse that path yourself.

Chapter 1

Introduction

1.1 About the book

This book's genesis was a collection of notes and reading materials for a class I taught on Survey Design in Spring 2023. The positive reception and plethora of insightful questions from the students spurred me to systematize and weave these resources into this book.

This book is not a comprehensive guide; if that's what you're seeking, you may want to look elsewhere. Instead, this book serves as a map that outlines the necessary tools and topics for your research journey. We'll delve into the basics of data collection, cleaning, validation, and presentation, all while underpinning these concepts with relevant theory. 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, as my objective isn't to bore you, but to enlighten.

1.2 Who Should Read This Book

While my experiences in Economics graduate school served as the primary motivation for writing this book, I've aimed to make it universally applicable for anyone working with data. That's why you'll find dedicated chapters on topics like thesis writing and literature reviews. If your work involves humanities, data analytics, or anything in-between, I'm confident that you'll find this book valuable. Whether you're a seasoned professional brushing up on some skills or a beginner hoping to learn something new, I promise you'll discover something useful here.

Although the examples in this book are written in R, which I believe to be one of the most powerful statistical languages, you don't need prior knowledge of

it to benefit from the content. The book focuses on the underlying concepts behind the tools, making it framework-agnostic. Some of the more theoretical chapters do not even require any programming knowledge. Over time, individuals and teams from diverse domains, such as web development, mathematics, data analytics, and economics, have found the content valuable for their work. So regardless of your coding skills or professional background, there's something in here for you.

The book does not need to be read sequentially, but the chapters are ordered and build upon each other for those who prefer a more structured approach. Initially, the data manipulation part was positioned in the middle, but it was moved to the beginning to aid those who need to understand R code earlier.

Chapter 2

Set up

2.1 R and RStudio Set Up

To begin our data analysis journey, we will utilize the R language, which was specifically designed by statisticians for statisticians. R boasts an intuitive syntax and workflow, enabling us to create readable and **reproducible** code, making it ideal for data analysis tasks.

Before we dive into the exciting world of data exploration and modeling, it's essential to understand the distinction between R and RStudio IDE. R is the actual programming language itself, while RStudio is an Integrated Development Environment (IDE) that provides a user-friendly interface to work with R efficiently. In other words, RStudio serves as a helpful tool to interact with R and enhances the overall coding experience.

To get started, you'll need to download both R and RStudio. R acts as the backbone, and RStudio complements it, making the coding process smoother and more accessible, especially for beginners. So, make sure to install both R and RStudio on your computer to embark on your data analysis journey seamlessly. Let's dive in and explore the incredible possibilities that R and RStudio have to offer! ## Download R

R is maintained through The Comprehensive R Archive Network.

2.1.1 For macOS

1. Go to https://cran.r-project.org/
2. Select "Download R for macOS"
3. If you have Apple Silicon mac (M1,M2…) download the latest version that has -arm64 in its name (R-4.2.2-arm64.pkg)
4. If you have Intel Max Download one without -arm64 (R-4.2.2.pkg)
5. Follow the steps from the installation wizzard

6. The installer lets you customize your installation, but the defaults will be suitable for most users
7. Your computer might ask for your password before installing new programs

2.1.2 For Windows

1. Go to https://cran.r-project.org/
2. Select "Download R for Windows"
3. Click on "base"
4. Click the first link at the top of the new page (Download R-4.2.2 for Windows)
5. ink downloads an installer program, which installs the most up-to-date version of R for Windows.
6. Follow the steps from the installation wizzard
7. The installer lets you customize your installation, but the defaults will be suitable for most users
8. You might need administration privileges to install new software on your machine

2.2 Download RStudio

Rstudio is like a Microsoft Word for writing text, but instead of text RStudio helps you write in R. RStudio is also free and easy to install! Go to RStudio Website, click on DOWNLOAD RSTUDIO or select your operation system below, and follow the isntallation instructions.

2.3 Configure RStudio

First watch an introduction to RStudio: https://www.youtube.com/watch?v=FIrsOBy5k58 and introduction to projects: https://www.youtube.com/watch?v=MdTtTN8PUqU

Now the most crucial part!!! Change the default theme of RStudio from light to dark! To do this

1. Go to the top panel
2. Select the Tools tab
3. Navigate to Global Options
4. Select Appearance
5. In Editor Theme, select "Dracula"
6. Click Apply

Now let's install a cooler font with ligatures! Install fira-code: https://github.com/tonsky/FiraCode/wiki/Installing. **Restart** RStudio for the font to load. Then go back to Appearance, choose FiraCode, and hit apply.

2.4 Install Packages

An R package is a collection of useful functions, documentation, and data sets that can be used in your own R code after it is loaded. These packages typically focus on a specific task and make use of pre-written routines for various data science tasks.

You can install packages with a single line of code:

```
install.packages("tidyverse")
```

You can also install multiple packages at the same time using c():

```
install.packages(c("tidyverse","gapminder"))
```

You can load packages using the library() function:

```
library("tidyverse")
```

Run the following command to install packages we will use in class

```
install.packages(c("tidyverse", "janitor", "esquisse", "modelsummary", "styler"))
```

Part I

Working with Data

Chapter 3

Data Manipulation

In the realm of data analysis, data visualization and manipulation are indispensable tools for understanding and communicating complex information. R, being a potent programming language for data analysis, provides an array of packages that allow creation of visually striking plots and facilitate efficient data manipulation. Tidyverse, one of the most user-friendly and widely used collections of R packages[1], has been developed by Hadley Wickham, the Chief Scientist at Posit (RStudio). Tidyverse encompasses packages catering to all common tasks, which can be installed and activated using `install.packages("tidyverse")` and `library("tidyverse")` respectively. In this introduction, we will delve into the basics of Tidyverse, utilizing `readr` for data reading, `tidyr` for tidying, `dplyr` for manipulation, and `ggplot2` for visualization. To explore more about Tidyverse, visit their website https://www.tidyverse.org/.

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

3.1 Basics

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

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

```
[1] 4
```

```
2 * 4
```

```
[1] 8
```

```
2^8
```

```
[1] 256
```

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

```
[1] 0.5
```

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

```
[1] 2.302585
```

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

```
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 substitutes `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.

```
x
```

```
[1] 2
```

```
x <- x * 5
```

3.1.1 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"
```

```
(name <- "Parsa Rahimi")                                    ①
class(name)
```

① Wrapping with (…) prints the variable

```
[1] "Parsa Rahimi"
[1] "character"
```

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

```
[1] TRUE
```

```
class(true_or_false)
```

```
[1] "logical"
```

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 class into a vector.

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

```
[1] "Parsa"  "Rahimi"
```

```
length(name)
```

```
[1] 1
```

```
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!

```
(i <- c(1, 2, 3, 4))
i + 10                                                      ①
i * 10                                                      ②
i + c(2, 4, 6, 8)                                           ③
```

① Adds 10 to each element
② Multiplies each element by 10
③ Adds together elements in corresponding positions

```
[1] 1 2 3 4
[1] 11 12 13 14
```

```
[1] 10 20 30 40
[1]  3  6  9 12
```

The operations performed above 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.

```
name
```

```
[1] "Parsa Rahimi"
```

```
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. Take care to perform numeric operations only on numeric objects to avoid encountering errors. The `str()` function can be utilized to obtain the structure of the object, such as type, length, etc.

```
name_surname + 2
```

```
Error in name_surname + 2: non-numeric argument to binary operator
```

```
str(name_surname)
```

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

3.2 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 10 times faster than `read.csv`, but it also produces a tibble instead of a data frame and evades inconsistencies of `read.csv`. 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 non-standard 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 local path from the folder of the project (e.g., `"/data/file.csv"`). Let's

read the data! Using `readr::` specifies which package to use. Replace the text between the quotes with your path.

3.2.1 Example Data

In this tutorial, I'll be using a 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. Sessions below 30 Celsius were labeled as control, and sessions above 30 Celsius were labeled as treatment. We will primarily be using results from the 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 can be found in the `data` directory of the GitHub repository. We'll need the `tidyverse` package.

```
library(tidyverse)
```

```
data <- readr::read_csv("https://raw.githubusercontent.com/nikitoshina/ECON-623-Lab-2023/main/
```

① Download data from GitHub

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

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

```
Rows: 114
Columns: 70
$ id                 <chr> "001018001", "001018002", "001018005", "001018009",~
$ country_city       <chr> "mexico_chapingo", "mexico_chapingo", "mexico_chapi~
$ start_time         <chr> "12H 16M 0S", "12H 16M 0S", "12H 16M 0S", "12H 16M ~
$ end_time           <chr> "13H 14M 0S", "13H 14M 0S", "13H 14M 0S", "13H 14M ~
$ date               <date> 2022-06-20, 2022-06-20, 2022-06-20, 2022-06-20, 20~
$ mean_temp_celsius  <dbl> 28.58772, 28.58772, 28.58772, 28.58772, 28.58772, 2~
$ incentive_local    <dbl> 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 1~
$ exchange_usd_local <dbl> 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20,~
$ incentive_usd      <dbl> 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, ~
$ gender             <chr> "Female", "Male", "Male", "Male", "Male", "Female",~
```

```
$ point_value        <dbl> 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10,~
$ site_id            <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ~
$ session_n          <dbl> 18, 18, 18, 18, 18, 18, 18, 18, 19, 19, 19, 19, 19,~
$ subject_n          <dbl> 1, 2, 5, 9, 10, 13, 14, 15, 1, 2, 3, 4, 5, 6, 7, 8,~
$ tr_opponet_n       <dbl> 2, 1, 9, 5, 13, 10, 15, 14, 2, 1, 4, 3, 5, 4, 8, 7,~
$ ch_opponent_n      <dbl> 9, NA, NA, 1, 9, NA, NA, NA, NA, 3, 2, NA, NA, 8, N~
$ version            <chr> "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, ~
$ dc_cl_ps           <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ~
$ dc_cl_envy         <dbl> 2, 2, 2, 2, 1, 1, 2, 1, 2, 2, 2, 1, 1, 1, 1, 2, 2, ~
$ dc_c_ps            <dbl> 1, 1, 1, 1, 1, 2, 1, 1, 1, 2, 1, 2, 1, 2, 2, 2, 2, ~
$ dc_c_envy          <dbl> 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, ~
$ dc_cl_ps_points    <dbl> 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16,~
$ dc_cl_envy_points  <dbl> 20, 20, 20, 20, 16, 16, 16, 20, 20, 20, 16, 20, 16,~
$ dc_c_ps_points     <dbl> 16, 16, 16, 16, 12, 20, 16, 16, 12, 20, 12, 20, 16,~
$ dc_c_envy_points   <dbl> 23, 23, 23, 23, 21, 18, 23, 23, 23, 23, 18, 21, 23,~
$ pr_correct         <dbl> 7, 5, 6, 1, 7, 2, 6, 7, 5, 6, 8, 2, 2, 5, 6, 5, 2, ~
$ pr_wrong           <dbl> 0, 1, 0, 0, 0, 0, 0, 0, 0, 2, 0, 1, 1, 1, 0, 1, 0, ~
$ pr_not_attempted   <dbl> 5, 6, 6, 11, 5, 10, 6, 5, 7, 4, 4, 9, 9, 6, 6, 6, 1~
$ tr_correct         <dbl> 3, 6, 7, 5, 9, 7, 6, 7, 4, 7, 6, 3, 6, 7, 8, 6, 6, ~
$ tr_wrong           <dbl> 1, 1, 1, 4, 0, 0, 2, 1, 2, 2, 2, 0, 3, 1, 1, 4, 1, ~
$ tr_not_attempted   <dbl> 8, 5, 4, 3, 3, 5, 4, 4, 6, 3, 4, 9, 3, 4, 3, 2, 5, ~
$ tr_die             <dbl> 2, 5, 2, 6, 6, 2, 2, 6, 2, 4, 5, 3, 6, 6, 3, 6, 1, ~
$ tr_total           <dbl> 5, 11, 9, 11, 15, 9, 8, 13, 6, 11, 11, 6, 12, 13, 1~
$ tr_guess_correct   <dbl> 5, 7, 8, 6, 11, 5, 8, 5, 5, 6, 8, 5, 5, 8, 8, 7, 3,~
$ tr_guess_die       <dbl> 3, 4, 1, 4, 2, 2, 5, 5, 4, 3, 4, 2, 4, 2, 3, 1, 2, ~
$ tr_guess_total     <dbl> 8, 11, 9, 10, 13, 7, 13, 10, 9, 9, 12, 7, 9, 10, 11~
$ tr_other_correct   <dbl> 6, 3, 5, 7, 7, 9, 7, 6, 7, 4, 3, 6, 6, 3, 6, 8, 8, ~
$ tr_other_die       <dbl> 5, 2, 6, 2, 2, 6, 6, 2, 4, 2, 3, 5, 6, 3, 6, 3, 3, ~
$ tr_other_total     <dbl> 11, 5, 11, 9, 9, 15, 13, 8, 11, 6, 6, 11, 12, 6, 12~
$ tr_result          <dbl> 0, 2, 0, 2, 2, 0, 0, 2, 0, 2, 2, 0, 1, 2, 0, 2, 0, ~
$ tr_points          <dbl> 0, 22, 0, 22, 30, 0, 0, 26, 0, 22, 22, 0, 12, 26, 0~
$ f_happy            <dbl> 5, 9, 8, 4, 8, 8, 3, 7, 6, 10, 8, 9, 4, 5, 9, 8, 6,~
$ f_energetic        <dbl> 7, 8, 7, 3, 10, 9, 3, 5, 7, 10, 9, 8, 5, 6, 9, 6, 8~
$ f_frustrated       <dbl> 8, 3, 3, 0, 4, 8, 8, 2, 7, 0, 3, 0, 5, 0, 1, 2, 0, ~
$ f_last_meal_min    <dbl> 180, 240, 240, 90, 30, 120, 90, 120, 30, 120, 30, 3~
$ ch_correct         <dbl> 9, 4, 7, 7, 10, 5, 7, 6, 4, 4, 8, 6, 5, 6, 6, 6, 5,~
$ ch_wrong           <dbl> 0, 0, 0, 5, 0, 2, 1, 2, 4, 2, 3, 0, 1, 2, 2, 2, 0, ~
$ ch_not_attempted   <dbl> 3, 8, 5, 0, 2, 5, 4, 4, 4, 6, 1, 6, 6, 4, 4, 4, 7, ~
$ ch_die             <dbl> 3, 3, 5, 6, 2, 4, 1, 3, 6, 1, 5, 4, 3, 2, 4, 2, 6, ~
$ ch_total           <dbl> 12, 7, 12, 13, 12, 9, 8, 9, 10, 5, 13, 10, 8, 8, 10~
$ ch_tournament      <dbl> 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, ~
$ ch_other_correct   <dbl> 9, 4, 4, 7, 9, 4, 4, 4, 4, 4, 8, 4, 4, 6, 4, 6, 5, ~
$ ch_other_die       <dbl> 3, 3, 3, 6, 3, 3, 3, 3, 6, 1, 5, 6, 6, 2, 6, 2, 6, ~
$ ch_other_total     <dbl> 12, 7, 7, 13, 12, 7, 7, 7, 10, 5, 13, 10, 10, 8, 10~
$ ch_result          <dbl> 1, NA, NA, 1, 1, NA, NA, NA, NA, 1, 1, NA, NA, 1, N~
```

```
$ ch_points          <dbl> 12, 7, 12, 13, 12, 9, 8, 9, 10, 5, 13, 10, 8, 8, 10~
$ ct_selected        <dbl> 2, 2, 3, 6, 6, 2, 1, 1, 1, 6, 2, 1, 1, 1, 6, 6, 6, ~
$ ct_tails           <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ~
$ ct_points          <dbl> 9.5, 9.5, 8.0, 1.0, 1.0, 9.5, 11.0, 11.0, 11.0, 1.0~
$ task_paid          <dbl> 4, 4, 4, 4, 4, 4, 4, 4, 6, 6, 6, 6, 6, 6, 6, 6, 6, ~
$ complete           <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ~
$ total_points       <dbl> 23, 23, 23, 23, 21, 18, 23, 23, 0, 22, 22, 0, 12, 2~
$ total_local_paid   <dbl> 330, 330, 330, 330, 310, 280, 330, 330, 100, 320, 3~
$ total_usd_paid     <dbl> 16.5, 16.5, 16.5, 16.5, 15.5, 14.0, 16.5, 16.5, 5.0~
$ final_version      <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ~
$ comment            <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,~
$ f_last_meal_time   <chr> "3 0", "4 0", "4 0", "1 30", "0 30", "2 0", "1 30",~
$ start_time_h       <dbl> 12, 12, 12, 12, 12, 12, 12, 12, 14, 14, 14, 14, 14,~
$ end_time_h         <dbl> 13, 13, 13, 13, 13, 13, 13, 13, 15, 15, 15, 15, 15,~
```

3.3 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 `magrittr` package. The way you use functions in base R is you wrap new function over the previous one, such as `k(g(f(x)))` this will become impossible to read very quickly as you stack up functions and their arguments. To solve this 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: base R now also has its own pipe `|>`, but we will stick to `%>%` for compatibility across packages.

3.3.1 `select()`

`select()` selects only the columns that you want, removing all other columns. You can use column position (with numbers) or name. The columns will be displayed in the order you list them. We will select subject_id, temperature, gender and results of raven's matrices games.

- `id` is a unique subject identification number, where site_id.session_n.subject_n (001.001.001).
- `mean_temp_celsius` is mean temperature through the session
- `gender` is gender of the subject.
- `pr_correct` is number of correct answers in *piece-rate* round.
- `tr_correct` is number of correct answers in *tournament* round.
- `ch_correct` is number of correct answers in *choice* round.
- `ch_tournament` is 1 if participant decided to play tournament and 0 if choice.

```
data_raven <- data %>% select(id, mean_temp_celsius, gender, pr_correct, tr_correct, ch_tourna
head(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 00101~              28.6 Female          7          3             1          9
2 00101~              28.6 Male            5          6             0          4
3 00101~              28.6 Male            6          7             0          7
4 00101~              28.6 Male            1          5             1          7
5 00101~              28.6 Male            7          9             1         10
6 00101~              28.6 Female          2          7             0          5
```

You can also exclude columns or select everything else with select using -

```
data_raven %>%
  select(-gender) %>%
  head()
```

```
# 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
```

3.3.2 `filter()`

The `filter()` function keeps only rows that satisfy specified criteria. Below, we use it to create two separate datasets for Males and Females.

```
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
```

We use == for comparisons.

You can chain multiple criteria. In the example below, 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.

```
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 0010~              28.6 Female          7          3             1          9
 2 0010~              28.6 Female          2          7             0          5
 3 0010~              28.6 Female          6          6             0          7
 4 0010~              28.6 Female          7          7             0          6
 5 0010~              30.7 Male            6          7             1          4
 6 0010~              30.7 Male            5          6             1          6
 7 0010~              30.7 Male            2          6             1          5
 8 0010~              30.7 Male            6          8             0          8
 9 0010~              30.7 Male            7          7             1          7
10 0010~              30.7 Male            4          5             1          6
# i 51 more rows
```

3.3.3 `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) %>%
  head()
```

```
# 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            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
6 00102~              30.4 Male            2          3             1          2
```

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

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

```
# 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~              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
6 00102~              32.2 Female          8          7             0          8
```

3.3.4 `mutate()`

The `mutate()` function adds new columns or modifies existing ones in the dataset. For instance, you can create a dataset with 10 rows and add three new variable columns:

```
tibble(rows = 1:10) %>% mutate(
  One = 1,
  Comment = "Something",
  Approved = TRUE
)
```

```
# A tibble: 10 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
 5     5     1 Something TRUE
 6     6     1 Something TRUE
 7     7     1 Something TRUE
 8     8     1 Something TRUE
 9     9     1 Something TRUE
10    10     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,
  impovement = tr_correct - pr_correct,
  pr_deviation = pr_correct - mean(pr_correct)
)
```

```
# A tibble: 114 x 10
   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
 6 0010~              28.6 Female          2          7             0          5
 7 0010~              28.6 Female          6          6             0          7
 8 0010~              28.6 Female          7          7             0          6
 9 0010~              30.7 Female          5          4             0          4
10 0010~              30.7 Male            6          7             1          4
# i 104 more rows
# i 3 more variables: mean_temp_fahrenheit <dbl>, impovement <dbl>,
#   pr_deviation <dbl>
```

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

3.3.5 `recode()`

The `recode()` function modifies values within a variable. For example, we can use `recode()` to change "Male" to "M" and "Female" to "F":

```
data_raven <- data_raven %>% mutate(gender = recode(gender, "Male" = "M", "Female" =
```

3.3.6 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 choice round, and the mean score in the choice round.

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

```
data_raven %>%
  summarize(
    perc_male = sum(gender == "Male", na.rm = T) / 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
  perc_male pr_median tr_max ch_ratio ch_mean
      <dbl>     <dbl>  <dbl>    <dbl>   <dbl>
1         0         5      9    0.456    6.01
```

3.3.7 group_by() and ungroup()

3.3.7.1 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()
```

```
    ) %>%
    ungroup()
```

```
# 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()
  ) %>%
  ungroup()
```

```
# A tibble: 4 x 5
  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
```

3.3.7.2 `ungroup()`

The `ungroup()` function removes grouping. Always ungroup your data after performing operations that required grouping to avoid confusion.

```
data_raven %>%
  drop_na(gender) %>%
  group_by(gender) %>%
  mutate(n = n()) %>%
  mutate(mean_male = mean(gender == "Male")) %>%
  ungroup() %>%
  select(id, gender, n, mean_male) %>%
  head(n = 5)
```

```
# A tibble: 5 x 4
  id        gender     n mean_male
```

```
  <chr>     <chr>  <int>     <dbl>
1 001018001 F         58         0
2 001018002 M         55         0
3 001018005 M         55         0
4 001018009 M         55         0
5 001018010 M         55         0
```

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

```
data_raven %>%
  drop_na(gender) %>%
  group_by(gender) %>%
  mutate(n = n()) %>%
  ungroup() %>%
  mutate(mean_male = mean(gender == "Male")) %>%
  select(id, gender, n, mean_male) %>%
  head(n = 5)
```

```
# A tibble: 5 x 4
  id        gender     n mean_male
  <chr>     <chr>  <int>     <dbl>
1 001018001 F         58         0
2 001018002 M         55         0
3 001018005 M         55         0
4 001018009 M         55         0
5 001018010 M         55         0
```

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

3.3.7.3 `.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.

```
# A tibble: 113 x 8
   id    mean_temp_celsius gender pr_correct tr_correct ch_tournament ch_correct
   <chr>             <dbl> <chr>       <dbl>      <dbl>         <dbl>      <dbl>
 1 0010~              28.6 F               7          3             1          9
 2 0010~              28.6 M               5          6             0          4
 3 0010~              28.6 M               6          7             0          7
 4 0010~              28.6 M               1          5             1          7
```

```
 5 0010~              28.6 M              7          9              1          10
 6 0010~              28.6 F              2          7              0           5
 7 0010~              28.6 F              6          6              0           7
 8 0010~              28.6 F              7          7              0           6
 9 0010~              30.7 F              5          4              0           4
10 0010~              30.7 M              6          7              1           4
# i 103 more rows
# i 1 more variable: n <int>
```

1. same as `group_by %>% mutate %>% ungroup`

3.3.8 `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 dataframe 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)
)

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 example above, 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. Alternatively, you can use `lengths()`, which applies `length()` to each list.

3.3.9 `count()`

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

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

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

Or with multiple columns:

```
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
```

3.3.10 `rename()`

The `rename()` function allows you to change column names, with the new name on the left and the old name on the right:

```
data_raven %>%
  rename(subject_id = id, sex = gender) %>%
  head(5)
```

```
# A tibble: 5 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 1 more variable: ch_correct <dbl>
```

3.3.11 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) %>%
  mutate(id = row_number()) %>%
  relocate(id) %>%
  head(5)
```

```
# A tibble: 5 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
```

3.3.12 skim()

`skim()` from the `skimr` package provides an extensive summary of a dataframe. It offers more than `summary()`, detailing quartiles, missing values, and histograms. Use it during exploratory data analysis to understand your data.

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

Table 3.1: 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	p0	p25	p50	p75	p100	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	

It provides a neat, comprehensive view of each variable, useful for further analysis.

Chapter 4

Tidy Data

```
data_raven <- readr::read_csv("https://raw.githubusercontent.com/nikitoshina/ECON-623-Lab-2023
```

Tidy data is a standard way of structuring a dataset to streamline its usability. This standard, as popularized by Hadley Wickham (Wickham 2014), depends on the organization of rows, columns, and tables, and how they correspond to observations, variables, and types.

Wickham, Hadley. 2014. "Tidy Data." *Journal of Statistical Software* 59 (10). https://doi.org/10.18637/jss.v059.i10.

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.

    What changes put it in a column

Data commonly takes two formats: wide and long.

Wide data, also referred to as unstacked data, is 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 its readability excels. A notable example of wide data is a panel where columns correspond to years Table **??**.

In contrast, long data, also known as stacked data, is organized so that each row represents a single observation of a variable, with columns representing the variable and the unit of observation identifier. 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 4.1: 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

Table 4.2: 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

From the first example, one might think that long and tidy data are synonyms. Let's consider another example where we wish to make our data wider. In Table **??**, 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.

Another common issue with messy data occurs when two variables are combined into one column, as seen in Table **??**, where 'Year' and 'Population' are in the same column. These need to be separated into two distinct columns: 'Year' and 'Population'.

4.1 Example

Tidy data is more than a theoretical concept; it has practical implications for data structuring. Consider this example of storing data on experiment payments:

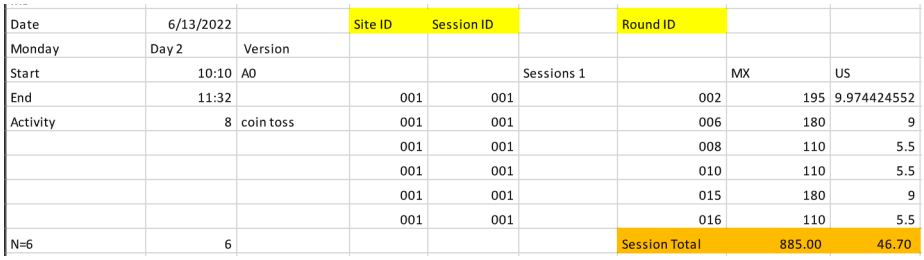
Table 4.3: Separate Data

(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



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

When a computer reads this 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 errors. Want a total for the entire experiment? We'd have 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

In this format, data input is straightforward, 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.

4.2 `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`, `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 two observations, not one.

To tidy such a dataset, we need to pivot the problematic columns into a new pair of variables. This operation requires:

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

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

```
    select(id, game, n_correct) %>%
    head(n = 5)
```

```
# 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
```

4.3 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 %>% head(n = 5)
```

```
# 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
```

To tidy this up, 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) %>%
    head(n = 5)
```

```
# 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
```

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.

4.4 separate() and unite()

Sometimes, data may come with columns united, necessitating us to `separate()` them to maintain tidy data.

```
data_raven_sep %>% head(n = 5)
```

```
# 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
```

```
data_raven_sep %>%
  separate(col = "gender/pr_correct", into = c("gender", "pr_correct"), sep = "/") %
  head(n = 5)
```

```
# 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
```

What if we have one column 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 %>% head(n = 5)
```

```
# 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
```

```
data_raven_uni %>%
  unite(c(site_id, session_n, subject_n), col = "id", sep = "") %>%
  head(n = 5)
```

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

4.5 tibble() and tribble()

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

- Like data frames, tibbles can hold tabular data, but they have some key differences.
- By default, tibbles display only the first 10 rows when printed, making them easier to work with large datasets.
- They use a consistent printing 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.
- Most importantly, tibbles can have columns that consist of lists.

In summary, Tibbles are a more modern and consistent version of data frames. They are less prone to errors and more readable, making them an excellent choice for data manipulation and exploration tasks.

```
  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 the Row column — represents the data in a row by row layout.

```
  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
```

4.6 `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 this vignette! If you are interested in creating summary tables with `janitor`, refer to this vignette!

4.6.1 `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

meaning, and resolves issues of duplicate or empty names. It is recommended to call this function every time data is read.

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

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

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

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

4.6.2 remove_empty()

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

```
test_df2 <- data.frame(
  numbers = c(1, NA, 3),
  food = c(NA, NA, NA),
  letters = c("a", NA, "c")
)
test_df2
```

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

```
test_df2 %>%
  remove_empty(c("rows", "cols"))
```

```
  numbers letters
```

```
1       1       a
3       3       c
```

4.6.3 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).

```
test_df3 <- data.frame(cool_numbers = 1:3, boring = "the same")
test_df3
```

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

```
test_df3 %>% remove_constant()
```

```
  cool_numbers
1            1
2            2
3            3
```

4.6.4 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 `excel_numeric_to_date()`.

```
convert_to_date(36922.75)
```

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

```
convert_to_datetime(36922.75)
```

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

4.6.5 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.

```
test_df4 <- data.frame(
  x_1 = c(NA, "Names", 1:3),
  x_2 = c(NA, "Value", 4:6)
)
test_df4
```

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

```
row_to_names(test_df4, 2)
```

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

Chapter 5

Relational Databases

A database is an organized collection of data that can be easily stored, managed, updated, and retrieved. One such type is the relational database, which structures data into tables consisting of rows and columns. Each row, also known as a record, encapsulates information about a single entity, while each column, or attribute, defines a specific aspect of that entity.

Relational databases employ tables and relationships between these tables to store data. A row signifies a unique instance of a table's subject, while a column represents a distinct characteristic of the subject.

For example, consider the following table:

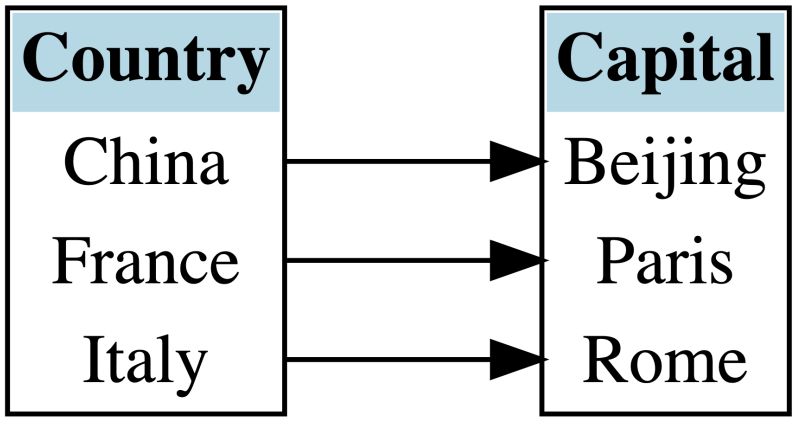
Student ID	Student Name	HW1	HW2	MidTerm
20101002	Nikita	88	100	76
20101003	Nelson	78	98	100
20101004	Parsa	98	99	95
20101005	Shivani	56	80	76

When rows from one table can be associated with rows in another table, the tables are understood to share a relationship.

5.1 Relationship Types

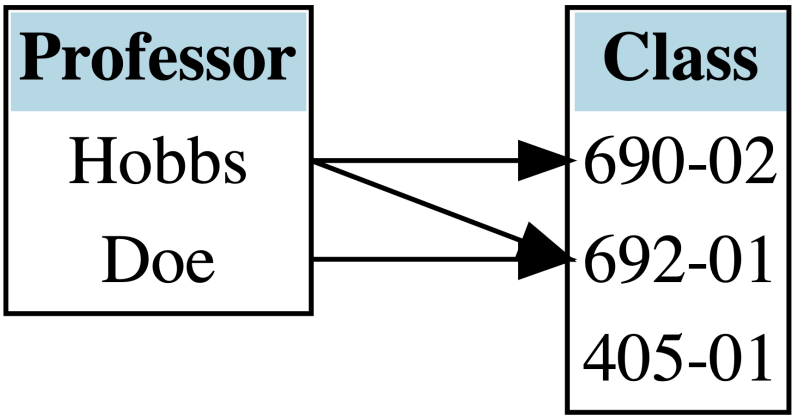
5.1.1 One to One (1:1)

In a one-to-one 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:



5.1.2 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:

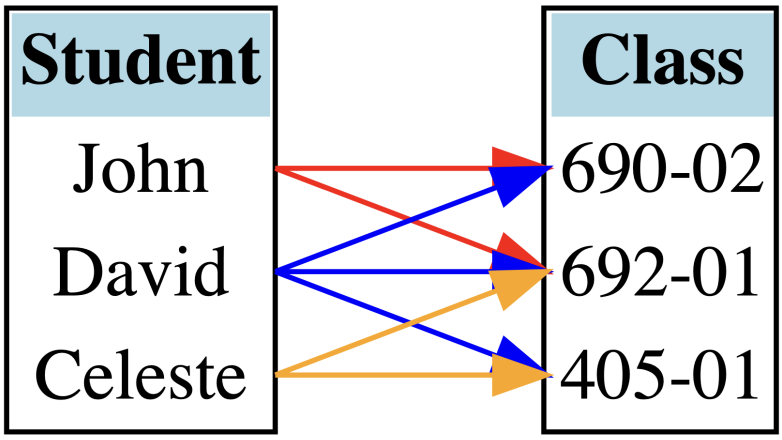


5.1.3 Many to Many (M:N)

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:

Table 5.2: Keys

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



5.2 The Concept of Keys

A Primary Key (PK) is a unique identifier for each row within a table; every table should possess a primary key. To establish relationships between tables, we integrate PKs from one table into another, where they become Foreign Keys (FKs). These FKs allow us to draw connections between related entities across different tables.

5.3 Types of Joins

In relational databases, a join operation is employed 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` (Primary Key) and `name`, while the `projects` table contains `project_id` (Primary Key) and `employee_id` (Foreign Key).

```
knitr::kable(employees, caption = "Employees Table")
knitr::kable(projects, caption = "Projects Table")
```

Table 5.5: Employees Table

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

Table 5.6: Projects Table

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

5.3.1 Outer Joins

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

5.3.1.1 Left Join

A left join retrieves all the rows from the left table and only the matched rows from the right table. Essentially, it enriches the left table with additional information.

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

knitr::kable(left_join_result, caption = "Result of Left Join")
```

Table 5.7: 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

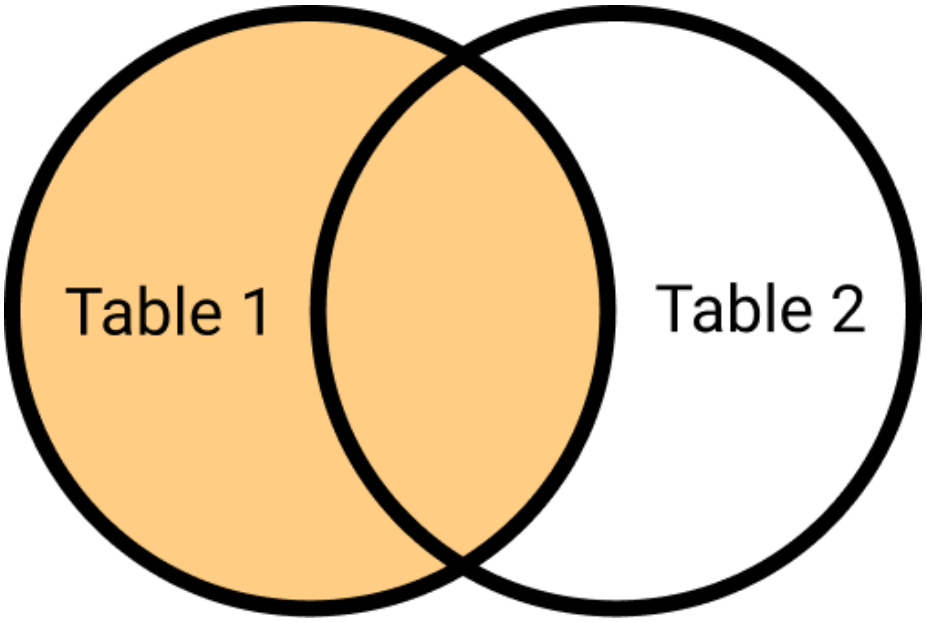


Figure 5.1: Left Join Illustration

5.3.1.2 Right Join

A right join operates similarly to a left join, but it retrieves all the rows from the right table and only the matched rows from the left table.

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

knitr::kable(right_join_result, caption = "Result of Right Join")
```

Table 5.8: 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

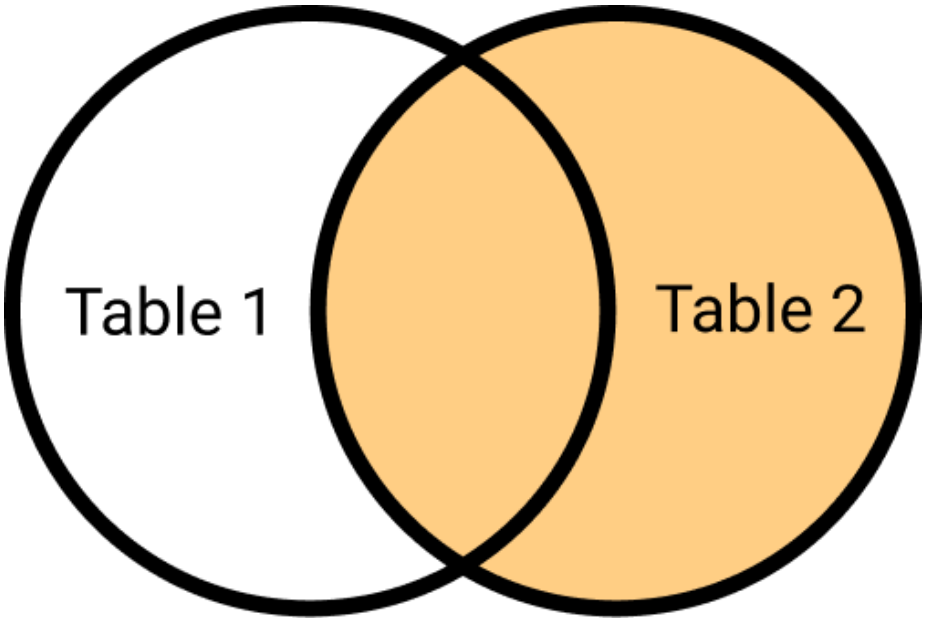


Figure 5.2: Right Join Illustration

5.3.1.3 Full Join

A full join returns all the rows from both tables, filling non-matching rows with null values. It essentially merges both tables.

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

knitr::kable(full_join_result, caption = "Result of Full Join")
```

Table 5.9: 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

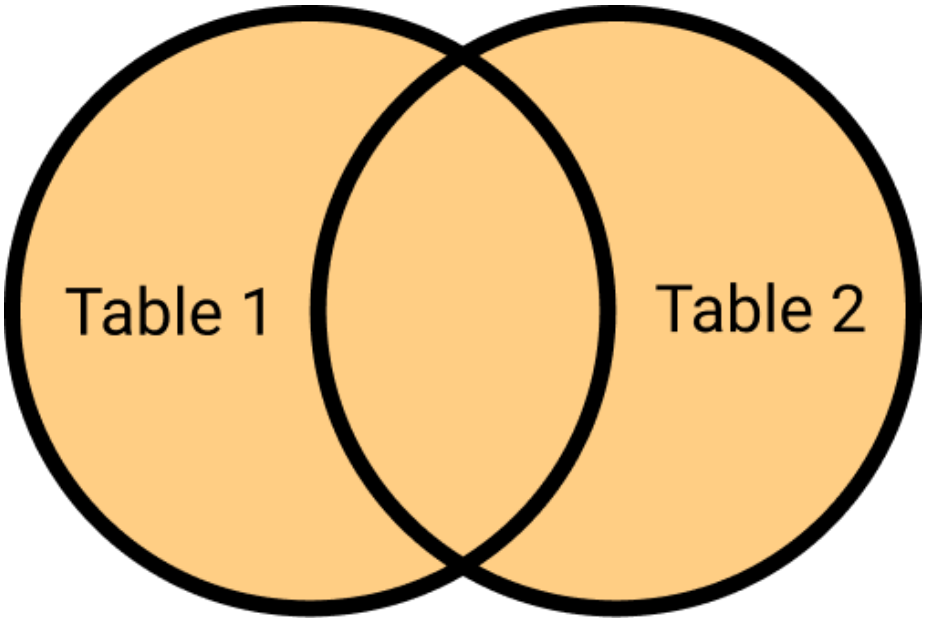


Figure 5.3: Full Join Illustration

5.3.1.4 Inner Join

An inner join only returns the matched rows between two tables. Thus, only the rows that found a match in both tables will be retained.

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

knitr::kable(inner_join_result, caption = "Result of Inner Join")
```

Table 5.10: Result of Inner Join

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

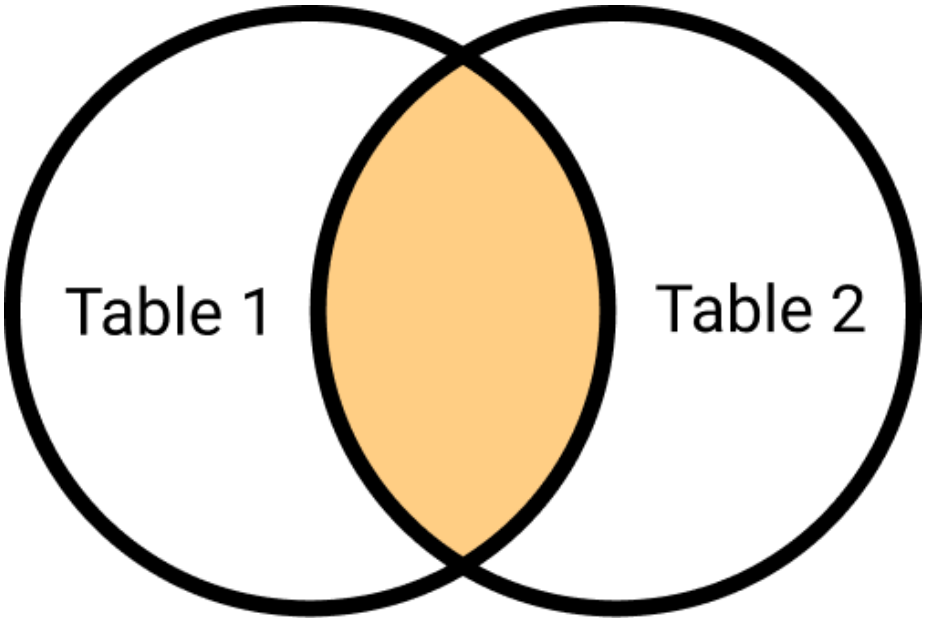


Figure 5.4: Inner Join Illustration

5.3.2 Filtering Joins

5.3.2.1 Anti Join

An anti join returns the rows from the left table that do not find a corresponding match in the right table, without adding any new columns to the output. It's useful when you want to filter rows based on the absence of matching entries in the other table.

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

knitr::kable(anti_join_result, caption = "Result of Anti Join")
```

Table 5.11: Result of Anti Join

employee_id	name
5	Tom

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

5.3.2.2 Semi Join

A semi join is akin to an inner join in identifying matching rows between two tables. However, unlike an inner join, it does not add any 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.

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

knitr::kable(semi_join_result, caption = "Result of Semi Join")
```

Table 5.12: Result of Semi Join

employee_id	name
1	John
2	Jane
3	Bob
4	Alice

5.4 Visualizing Databases

Remember our discussion on UML? We're going to use it to construct an Entity-Relationship Model to understand how tables in our database are related and how they interact, simplifying our later work. UML coding tools like Mermaid and Graphviz are 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, which is an object (place, person, thing) that we want to track in our database. In our case, these will be a customer, order, and product. Each entity possesses attributes, for example, a customer has `customer_id`, `name`, `email`, `address`, etc., 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 a line between the tables. Cardinality is used to describe these relationships in numeric terms, akin to our discussion in Section **??**.

For instance, let's examine the relationship between a customer and an order. We ask: what is 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 from 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).

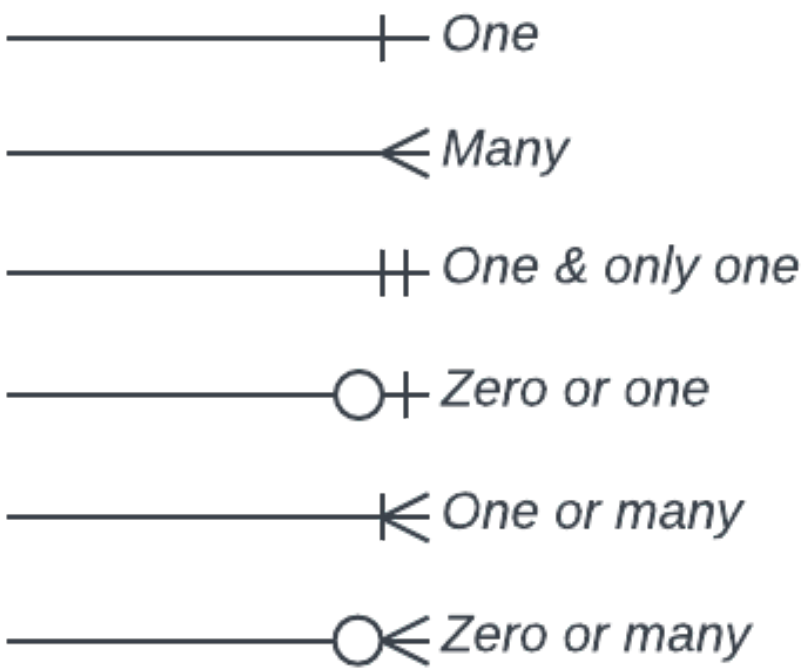


Figure 5.5: Cardinality

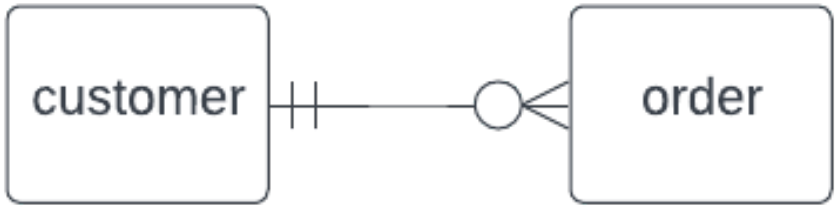


Figure 5.6: Customer-Order Relationship

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:

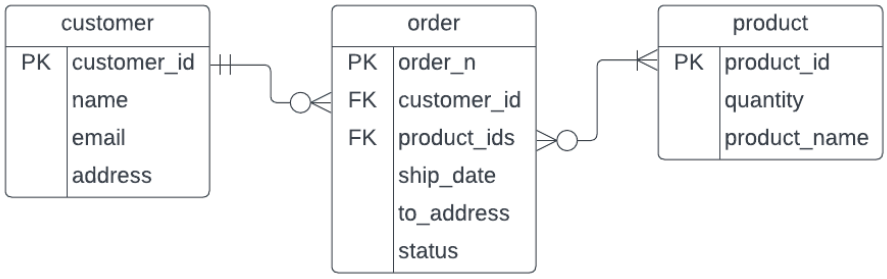


Figure 5.7: Entity-Relationship Diagram

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 data set. If you'd like to delve deeper into this topic, check out the Lucid Software YouTube guides and Neso Academy's Database Management Systems course.

Chapter 6

Optimizing 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. With dynamic data, automating these checks becomes the solution. 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.

6.1 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. This approach not only gives you a comprehensive understanding of the data at hand but can also save time in the long run by helping you avoid unfit data for analysis. Lastly, always verify your results by reviewing the output table, an important yet simple step to remain fully engaged with the raw data.

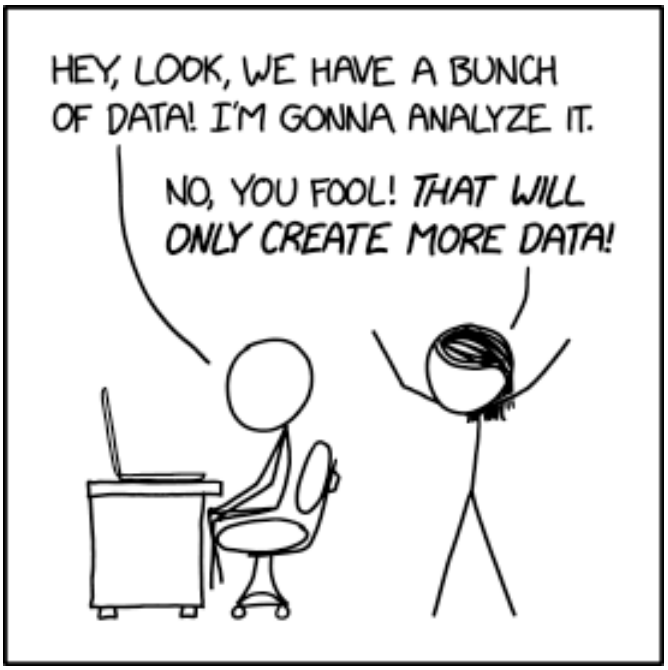


Figure 6.1: https://xkcd.com/2582/

6.2 Handling Data Issues

6.2.1 Base R

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. Lastly, don't forget the `duplicated()`, `unique()`, and `dplyr`'s `distinct()` functions.

```
x <- c(1, 2, 3)
df <- data.frame(x = c(x, 1), y = c(x * 2, 2))
if (!is.numeric(x)) stop("x is not numeric!")                    (1)
stopifnot(is.numeric(x))                                         (2)
if (!all(x > 0)) stop("x contains non-positive values!")         (3)
if (any(duplicated(x))) stop("x contains duplicated values!")    (4)
if (nrow(dplyr::distinct(df)) != nrow(df)) warning("df has duplicated rows!")  (5)
```

(1) Stops execution if `x` isn't numeric.
(2) Halts if `x` isn't numeric.
(3) Stops the process if `x` contains non-positive values.

④ Halts execution if `x` has duplicates.
⑤ Issues a warning if `df` has duplicate rows.

6.2.2 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 such as `verify()`, `assert()`, `insist()`, and `chain()`, all of 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 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 for specifying a proportion or number of values that must meet the predicate function condition.
- `chain()`: This function is used when you want to specify more than one predicate function in the same `assert()` or `insist()` call. The data passes validation only if all predicates are met.

The syntax of the package integrates smoothly into a typical `dplyr` pipeline. Here's a brief example:

```
library(assertr)
library(dplyr)
```

① Verify dataset isn't empty.
② Assert that specified Engine Types are present in row names.
③ Ensure that at least one 'mpg' value is within two standard deviations.

```
Attaching package: 'dplyr'

The following objects are masked from 'package:stats':

    filter, lag

The following objects are masked from 'package:base':

    intersect, setdiff, setequal, union
```

```
data(mtcars)

mtcars %>%
  tibble::rownames_to_column() %>%
  verify(nrow(.) > 0) %>%                    ①
  assert(in_set(0, 1), vs) %>%               ②
  insist(within_n_sds(2), mpg)               ③
```

```
Error: assertr stopped execution
```

```
Column 'mpg' violates assertion 'within_n_sds(2)' 2 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
```

In this example, `verify()` ensures `mtcars` has more than one row. `assert()` checks for the presence of certain row names, and `insist()` ensures at least one `mpg` value lies within two standard deviations. If any check fails, the pipeline stops and throws an error.

"Helper" functions in `assertr` are predicate functions that return a logical true or false. These are used in conjunction with `assert()`, `insist()`, or `verify()`. Examples of helper functions include `within_bounds()`, `not_na()`, `is.numeric()`, `in_set()`, etc. You can also create and use your own predicate functions if needed.

Replace your in-console data check with assertions, and if you're not already conducting data checks, start now.

6.2.3 Precise Validation with Pointblank

The `pointblank` package in R is specifically tailored for data validation. It's designed with features to make data validation more reliable, better documented, interactive, and adaptable to various scenarios. Its key features include:

- **Assertion functions**: These functions allow setting quality standards for your data and halting execution if these standards are not met. They can check data types, value ranges, set memberships, distribution properties, etc.
- **Informative interruptions**: Pointblank provides detailed error messages when data issues arise, causing a halt in the R process.
- **Report generation**: A unique feature of Pointblank is that it creates comprehensive reports about the data validation process and its results.
- **Integration with `dplyr` and `tidyverse`**: Pointblank complements the `tidyverse` suite of packages, especially `dplyr`, making it easy to apply data validation alongside data manipulation and visualization tools.
- **Agent objects for ongoing validation**: The concept of 'agent' objects is introduced for continuous data validation. An agent can hold various