R Essentials: A Beginner's Guide to Data Analysis

Tengku Muhammad Hanis Bin Tengku Mokhtar, PhD

Nov 23, 2025

Table of contents

Welcome

⚠ This book is currently in its drafting stages and has not yet been published. While all the chapters are complete, most of them are still undergoing further revision and refinement.

This book is intended for anyone interested in using R for applied statistical analysis and machine learning. I first encountered R (specifically RStudio) during my master's studies, and it wasn't an easy journey—especially as someone from a non-coding background. Initially, I was introduced to software like SPSS and STATA, and eventually R during my master's studies. R felt unfamiliar and complex by comparison to the previous software. However, those who could use R proficiently seemed impressive to me, which motivated me to keep going. Surprisingly, the learning process became much easier as I keep using R.

Looking back, I wish I had known certain things earlier in my journey of learning R. This book is my attempt to provide those insights, covering the key concepts and tips I wish I had had when I started. I hope it sparks interest in R for others and helps them fully utilize its capabilities.

Finally, I just want to say thanks to my lovely wife, Nurul Asmaq, my parents, Tengku Mokhtar and Nor Malaysia, and my in-laws, Mazalan and Salmeh, for all their understanding. Writing this book means taking the time I would usually spend with them, and while they might not fully get my obsession with R and data analysis (especially my wife!), they've supported me every step of the way.

Tengku Muhammad Hanis Bin Tengku Mokhtar, PhD

About the book

⚠ This book is currently in its drafting stages and has not yet been published. While all the chapters are complete, most of them are still undergoing further revision and refinement.

This book is designed for beginner and novice R users, with chapters structured sequentially to introduce R step-by-step, starting with foundational topics and progressing to more complex material.

Chapters 1 to 3 introduce readers to R and RStudio. For those already somewhat familiar with R, these chapters may seem straightforward and can be skipped without issue.

Chapters 4 and 5 provide essential knowledge on basic R coding, commonly used in data analysis projects. It is advised to read these chapters thoroughly before moving on, as they lay the groundwork for more advanced topics.

Chapter 6 focuses on basic visualizations and plotting techniques. Readers will learn how to use base R functions for visual representation and will be introduced to `ggplot2`, a highly regarded package for data visualisation in R.

Chapter 7 covers loop, apply family, and function, topics that may pose a challenge for beginners. This chapter aims to equip readers with the skills needed for more efficient R coding. While not critical for initial learning, understanding these concepts will become increasingly important as one progresses in data analysis.

Chapters 8 and 9 delve into essential skills for data exploration and descriptive statistics. Mastering these will enable readers to gain deeper insights into their data and prepare them for more advanced analysis techniques.

Chapter 10 concludes the book by summarizing previous content and offering guidance on next steps to further enhance R and data analysis skills.

Each chapter kicks off with a quote—I hope you enjoy it! To wrap things up, each chapter ends with chapter summary and revision questions.

Happy learning!

1 What is R?

"Busses are very easy to use, you just need to know which bus to get on, where to get on, and where to get off (and you need to pay your fare). Cars, on the other hand, require much more work: you need to have some type of map or directions (even if the map is in your head), you need to put gas in every now and then, you need to know the rules of the road (have some type of drivers license). The big advantage of the car is that it can take you a bunch of places that the bus does not go and it is quicker for some trips that would require transferring between busses.

Using this analogy, programs like SPSS are busses, easy to use for the standard things, but very frustrating if you want to do something that is not already pre-programmed.

R is a 4-wheel drive SUV (though environmentally friendly) with a bike on the back, a kayak on top, good walking and running shoes in the passenger seat, and mountain climbing and spelunking gear in the back.

R can take you anywhere you want to go if you take time to learn how to use the equipment, but that is going to take longer than learning where the bus stops are in SPSS."

– Greg Snow

1.1 R

R is a language and environment for statistical computing and graphics.

That basically summarises this whole chapter.

Figure 1.1: The logo of R software.

Readers who are not interested in knowing more about R and its history can skip this chapter and move on to more practical chapters.

Well, for those who keep reading this, I guess you are interested to know more about the story of how R came to be. Do not worry, this chapter is only going to cover a short version of the history of R, so, that we can appreciate this software.

R was a successor of S language. It was developed by Ross Ihaka and Robert Gentleman at the University of Auckland in 1991. R was made known to the public only in 1993. The R version 1.0.0 was released in 2000.

1.2 RStudio and Posit

Figure 1.2: The logo of RStudio.

In 2009, a company known as RStudio, Inc. was founded by Joseph J. Allaire, which later developed the RStudio software. RStudio software is an integrated development environment (IDE) which helps make R more user-friendly, especially for those without a programming background. RStudio IDE is unequivocally the most commonly used IDE for R software. The company, RStudio, Inc. later changed its corporation to a public benefit corporation (PBC) in 2020, thus, known as RStudio, PBC. Subsequently, in 2022, the company changed its name to Posit Software, PBC to cater to a larger demography of the data science community.

Figure 1.3: The logo of Posit Software, PBC.

Despite the changes in Posit company, they still strongly support the development and maintenance of RStudio IDE specifically and R in general.

1.3 Other IDEs

As you may have guessed, we going to use RStudio in this book. However, there are other IDEs available. A few that are more common are:

1. Jupyter Notebook
2. JupyterLab
3. Visual Studio

Given that RStudio is initially developed for R (currently we can use Python as well in RStudio), a lot of functionalities work seamlessly with R. Thus, make it easier for R beginners and novices to use it.

1.4 Clouds

There are a few options to use R in a cloud. Meaning that we do not need to install anything on our machines.

1. Posit Cloud
2. Google Colab
3. Kaggle

The first two clouds are free with limited use, though you need to make an account. Kaggle is totally free to use as far as I know. However, Kaggle does not have functions such as code completion which is very helpful to beginners. This function is available in the first two clouds.

However, if you are looking for something more familiar to RStudio, Posit Cloud is the best choice. The functionalities and the overall look of the Posit Cloud are identical to RStudio.

1.5 Chapter summary

In this chapter, we learn about what are R and RStudio.

1.6 Revision

1. What is the difference between R and RStudio IDE?
2. What is the difference between RStudio IDE, RStudio, Inc., and RStudio, PBC?
3. What are other IDEs for R?

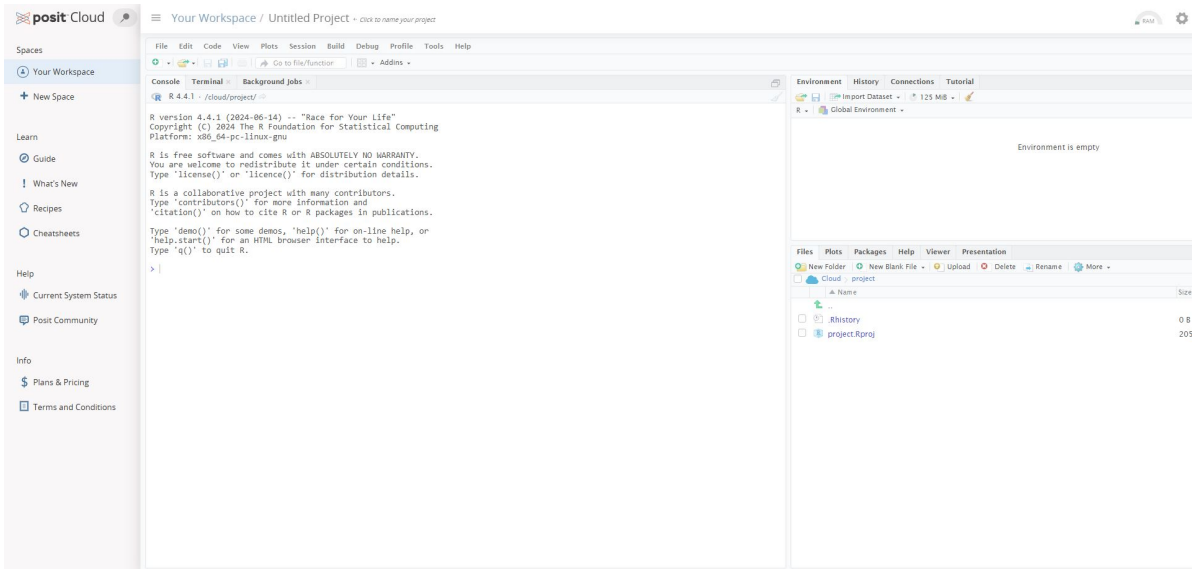


Figure 1.4: The interface of RStudio in Posit Cloud.

2 Installing R and RStudio

> "Everyone should learn how to code, it teaches you how to think."
>
> – Steve Jobs

2.1 Installing R

R can be installed regardless of your operating system. R is available for Windows, Mac, and Linux users. This page (https://cran.r-project.org/) contains all the necessary information needed to install R. Additionally, if you are stuck, I highly recommend you to watch a few YouTube videos on how to install R.

2.2 Installing RStudio

Once you have installed R, you need to install RStudio IDE. This page (https://posit.co/download/rstudio-desktop/) contains all the information including the related links necessary to download and install RStudio. Again, if you are stuck, I highly recommend you watch a few YouTube videos on how to install RStudio.

2.3 Installing RTools

This process is specific to Windows users only. If you are a Linux or Mac user, feel free to skip this part. Rtools is a collection of tools required to build R packages from source on Windows systems. Basically, you need RTools to install the unofficial packages from GitHub, GitLab or other repositories. We going to cover what are R packages in the Section 4.6. For now, just know that to fully utilise the capabilities of R, you need to have RTools in your machine.

This page (https://cran.r-project.org/bin/windows/Rtools/rtools40.html) contains all the necessary information on how to install RTools on your machine. The basic steps as outlined on the page are:

1. Download RTools
2. Install the RTools

3. Put RTools on the PATH

2.4 Other alternatives

On the rare occasion that you are unable to install R or RStudio, you always have the option to use the Posit Cloud. As long as you have a Google account, you should be able to use the Posit Cloud freely. The free account is limited, however, it is more than enough for you to use throughout this book. Additionally, you can use the Google Colab, though, the user interface is slightly different to RStudio.

2.5 Chapter summary

In this chapter, we learn how to install R and RStudio.

2.6 Revision

1. What is RTools?
2. What are the options if you can not install R and RStudio?

3 Introduction to RStudio

> "When I wrote this code, only God and I understood what I did. Now only God knows."
>
> – Anonymous

3.1 Panes in RStudio

The first time we open RStudio we will see three panes as in Figure 3.1.

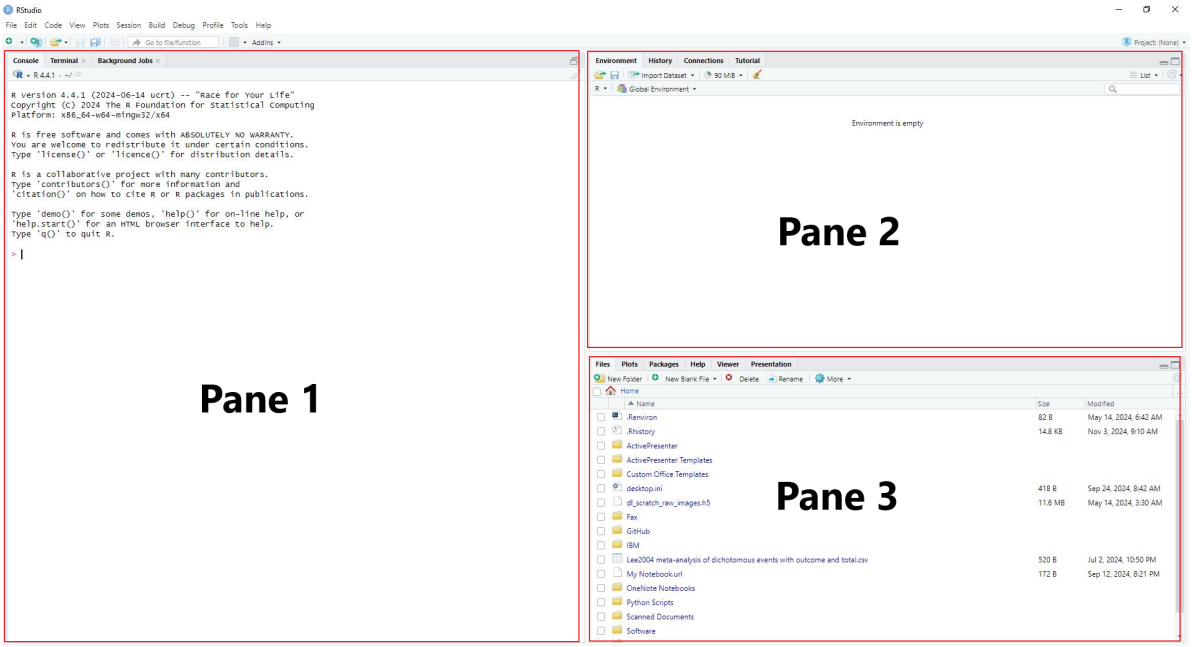


Figure 3.1: RStudio interface.

Pane 1 consists of three tabs:

1. Console - This pane is where R codes can be typed and executed. The console also is where the output will be displayed. However, any R codes typed into the console can not be saved.

2. Terminal - This is a command-line interface that allows users to interact with the system shell, much like other terminal applications on your computer. It supports tasks such as navigating the file system, running command-line tools, and managing files, which can be useful for integrating non-R tasks into your workflow. However, for beginners, this tab is relatively unimportant.
3. Background Jobs - This pane enables users to run R scripts in the background without interrupting ongoing work in the Console. This feature is helps run long or complex tasks, as you can continue coding or working in the RStudio environment while the script executes independently.

Next, Pane 2 consists of four tabs:

1. Environment - This pane displays all the active objects in the current R session, such as data frames, variables, functions, and vectors.
2. History - This pane keeps a record of all commands that have been executed in the R console during the session
3. Connections - This pane facilitates managing database connections within RStudio. It allows users to connect to external databases, view database contents, and run SQL queries directly from the IDE.
4. Tutorial - The Tutorial pane is part of RStudio's `learnr` package, which hosts interactive tutorials directly within the IDE.

Lastly, Pane 3 consists of five tabs:

1. Files - This pane allows users to navigate, create, delete, and manage files within the current working directory. It helps in organizing project files, accessing scripts, data files, and outputs.
2. Plots - This pane displays graphical output generated by R code, such as charts, graphs, and plots.
3. Packages - This pane shows a list of installed R packages and their status (loaded or not). It also allows users to install, remove, or update packages, providing an easy way to manage package dependencies for projects.
4. Help - This pane provides access to documentation for R functions, packages, and commands.
5. Viewer - The Viewer pane is used for displaying web content such as interactive visualizations, markdown files, and other HTML outputs directly within RStudio.
6. Presentation - This pane supports presenting R Markdown or Quarto documents in an interactive format. For example, if you are making a slide or HTML presentation, it will appear in this pane.

The information can be quite overwhelming, especially if you are new to R and RStudio. At this moment, you do not actually need to know every detail functionalities of each pane and tab yet. Once you become more familiar with RStudio, all this information will become second-hand to you.

3.2 Working directory

In Figure 3.1, we have a Files tab in Pane 3. This is usually where your working directory is located. However, we can also check using the R code below:

```
getwd()
```

Additionally, the working directory can also be changed to your preferred location.

```
setwd("C:/Users/tengk/OneDrive/Desktop")
```

Here, I changed my working directory to my desktop folder. A good practice when running the analysis in R is to set up your working directory before you start any analysis project. So, any outputs and figures generated during the analysis will be saved in your preferred working directory.

3.3 R script

One of the few things to do before running the analysis project (besides setting up the working directory) is to type the R codes in the R script. R scripts is a plain text file with the extension .R. R script can be saved and the R codes in it can be re-run if needed.

There are a few ways to open the R script.

1. Click on File (upper left side of RStudio) > New File > R Script.
2. Click on the green plus button (below the File tab) > R Script as shown in Figure 3.2.

Once you manage to open the R Script, you will see an additional pane as shown in Figure 3.3.

The R Script can be saved and additionally, at the lower right side of the pane, we can see `R Script`, confirming that this newly open file is an R Script.

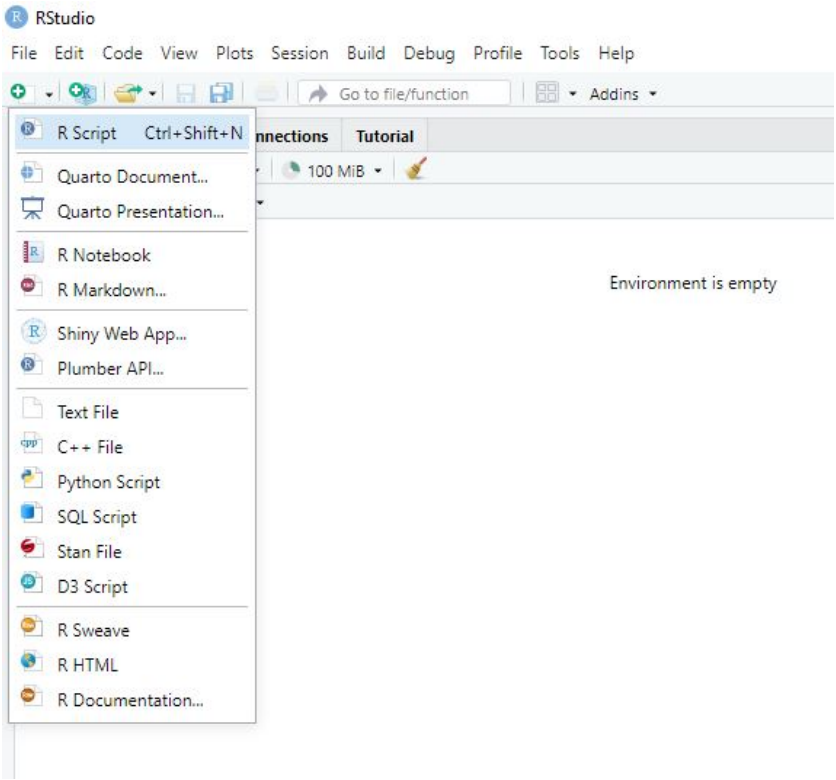


Figure 3.2: Opening the R Script.

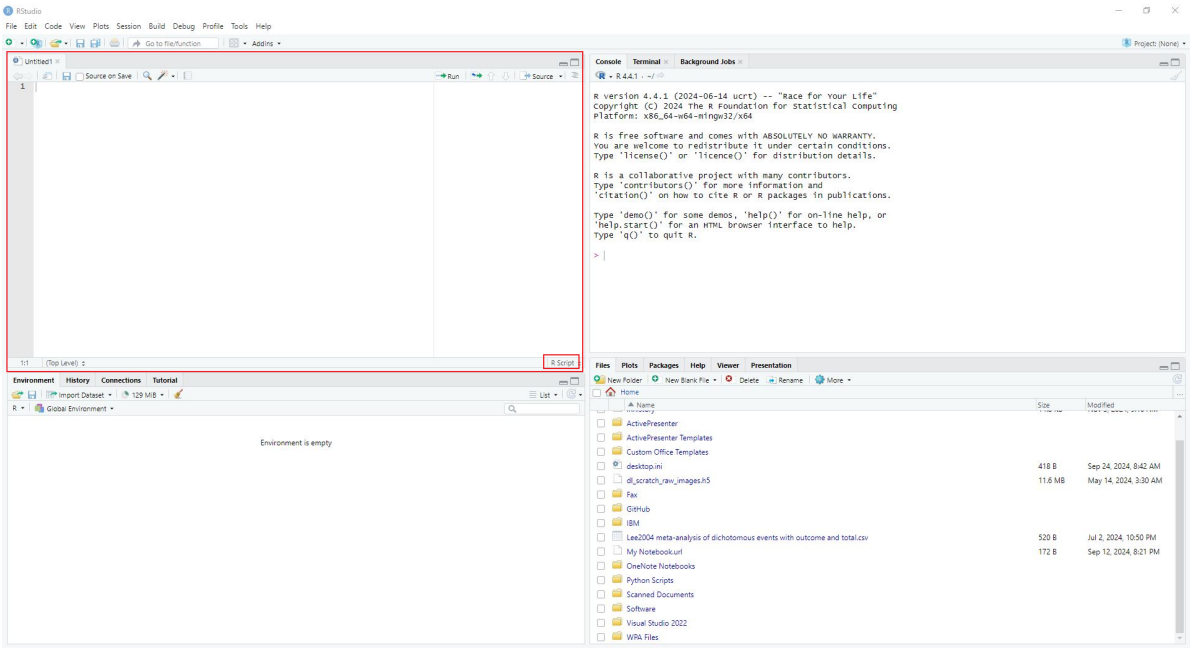


Figure 3.3: R Script interface.

3.4 Updating R and RStudio

`installr` package can be used to update R. You will what is an R package and how to install it in Chapter 5.

```
installr::updateR()
```

Once you run `installr::updateR()` either in the Console or R Script, this package will check whether there is a newer version of R or not. If the newer version of R is available, this package installs the newer version after asking a series of questions such as do you want to transfer your old packages to the new version of R and whether you want to update the packages or not.

To update the RStudio, you can click the Help tab (at the top of the RStudio) > Check for Updates.

3.5 Chapter summary

In this chapter, we learn about:

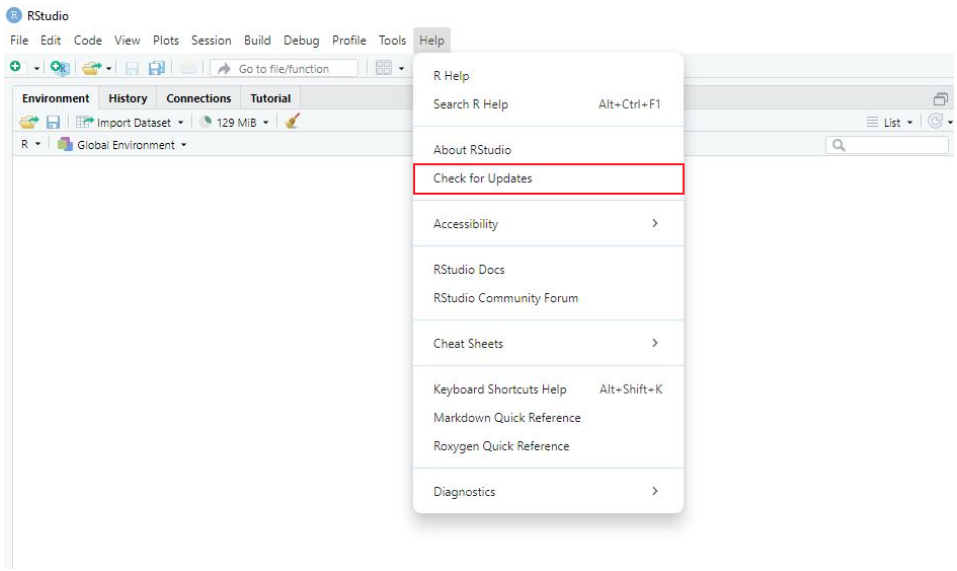


Figure 3.4: Updating RStudio

1. Basic interface of RStudio.
2. How to set up the working directory.
3. How to update R and RStudio.

3.6 Revision

1. Change your working directory to your desktop.
2. Try creating a new R Script, rename it as Test.R, and save it in your new working directory.

4 Basics of R

> "Confusion is part of programming."
>
> – Felienne Hermans

4.1 Getting help

Probably the most basic thing to know is how to get help in R. Besides a quick Google search or asking ChatGPT, R also provides a help function. The help function can be accessed using `?`.

```
?mean()
```

The code above will open the `Help` pane, which explains what the function `mean()` does.

4.2 Executing the code

The codes can be typed either in R Script or Console. The code in R Scripts can be executed by placing the cursor at any line of the codes and clicking `Ctrl` + `Enter` in Windows and `Cmd` + `Enter` in Macs.

```
# Example 1: A single line of code
mean(1:10)

# Example 2: A multiple lines of code
c(1:10, 10.6, 11.9) |>
  mean() |>
  round(digits = 1)
```

In the second example (example of multiple lines of code), a cursor can be placed at any line of code. Additionally, the codes in the Console can be executed by clicking `Enter` only. If you want to run any code in this book, you should copy and paste it into the R Script instead of the Console, especially if there are multiple lines of code. Notably, the codes should be run in sequence unless the current line of codes is independent from the previous line of codes.

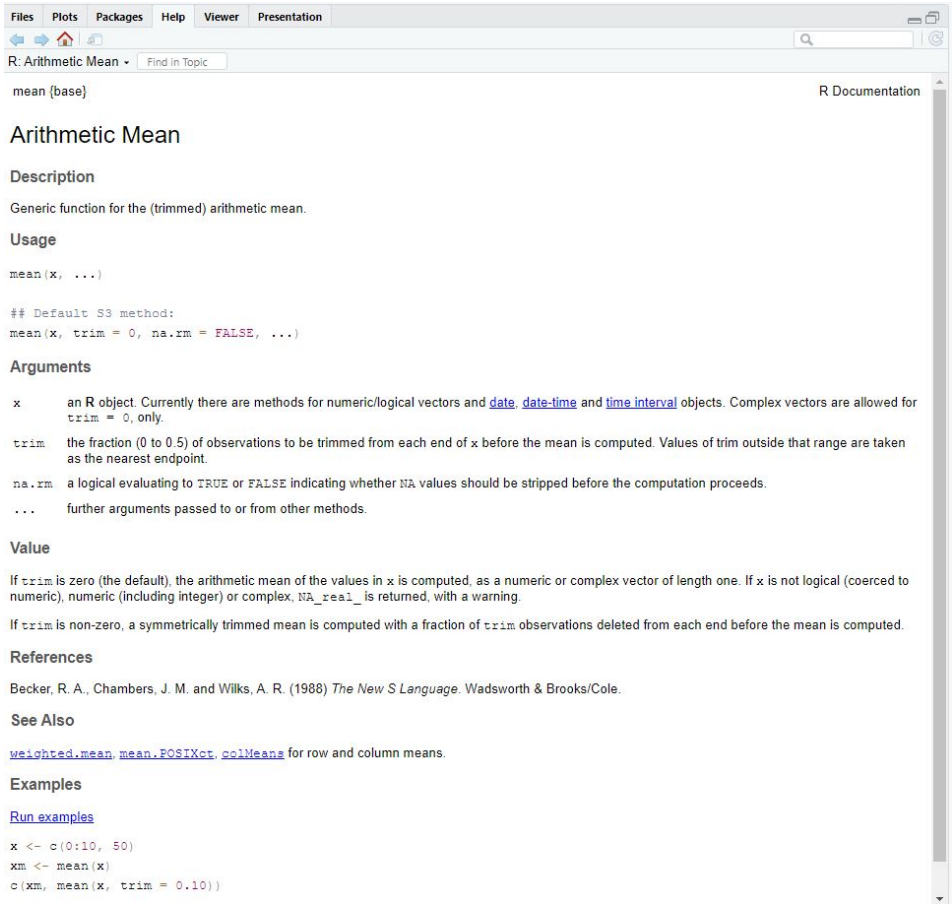


Figure 4.1: The Help pane in RStudio.

4.3 Data types in R

There are a few data types in R:

1. Numeric
2. Integer
3. Logical
4. Character
5. Complex

Let us see the example in R:

For the numeric:

```
x1 <- 11
x2 <- 11.9

class(x1); class(x2)
```

```
[1] "numeric"
```

```
[1] "numeric"
```

Both numbers are recognised as numeric in R. For integers, the number should be denoted by 'L' to be recognised as an integer.

```
x3 <- 11L

class(x3)
```

```
[1] "integer"
```

For logical values, the boolean operators such as 'FALSE' and 'TRUE' are examples of logical values.

```
x4 <- c(TRUE, FALSE)

class(x4)
```

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

Next, we have character values.

```
x5 <- c("fruit", "apple")

class(x5)
```

```
[1] "character"
```

Lastly, we have complex values. The type of data is usually used to store numbers and imaginary components (for example, `i` in the code below).

```
x6 <- 9 + 3i

class(x6)
```

```
[1] "complex"
```

It is important to note the data type for each value as the function for numeric values can only be applied to numeric values. For example, if we want to find a mean value.

```
numeric_val <- 1:10 #list out all numbers between 1 and 10
numeric_val
```

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

```
character_val <- letters[1:10] #list out the first 10 alphabets
character_val
```

```
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
```

Now, let us try applying the mean function to both data.

```
mean(numeric_val)
```

```
[1] 5.5
```

```
mean(character_val)
```

```
Warning in mean.default(character_val): argument is not numeric or logical:
returning NA
```

```
[1] NA
```

So, R gives us a warning that the 'character_val' is not a numeric or logical value. Thus, the returning NA mean not available.

4.4 Data structure in R

There are a few data structures in R:

1. Vector
2. Matrix
3. Array
4. Data frame
5. List

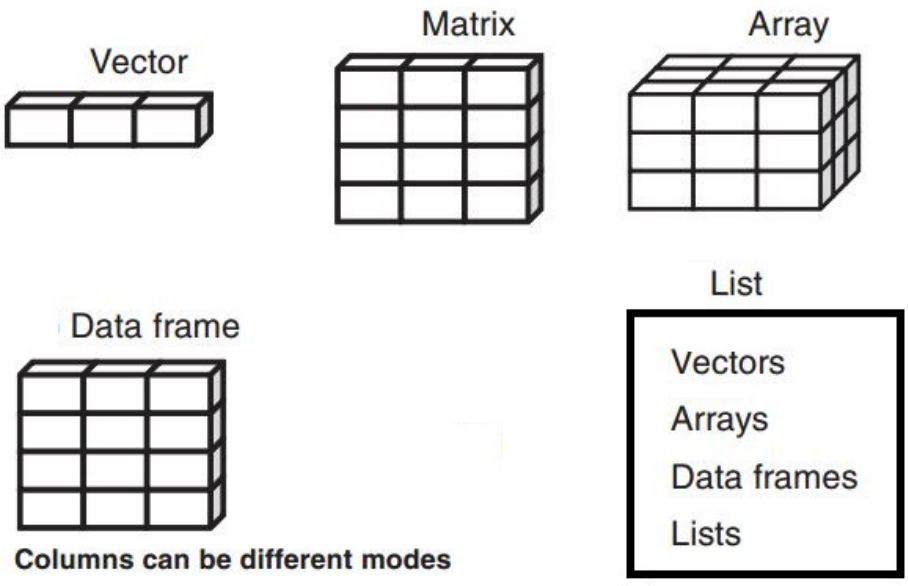


Figure 4.2: Data structures in R.

Depending on the fields, certain data structures are more common compared to others.

4.4.1 Vector

Vector is the most basic data structure in R. It can contain one data type at a time.

```
vec_data <- c(1, 2, 3, 4)
vec_data
```

```
[1] 1 2 3 4
```

The structure of the data can be checked using the function `str()`.

```
str(vec_data)
```

```
 num [1:4] 1 2 3 4
```

We can further confirm whether `vec_data` is a vector or not by using the `is.vector()` function.

```
is.vector(vec_data)
```

```
[1] TRUE
```

A `TRUE` result indicates that the data is a vector type.

4.4.2 Matrix

A matrix contains at least a single row and a single column. Contrary, to a vector which contains only a single row or a single column.

```
mat_data <- matrix(data = c(1, 2, 3, 4, 5, 6), nrow = 3, ncol = 2)
mat_data
```

```
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

By using `str()` function, we can see that the values are numerical and we have a matrix with 3 rows and 2 columns.

```
str(mat_data)
```

```
 num [1:3, 1:2] 1 2 3 4 5 6
```

Next, we can confirm that our data is a matrix by using `is.matrix()`.

```
is.matrix(mat_data)
```

```
[1] TRUE
```

4.4.3 Array

An array is quite similar to a matrix except that it can contain several layers of rows and columns.

```
arr_data <- array(data = c(1:6, 10:16), dim = c(2, 3, 2))
arr_data
```

```
, , 1

     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

, , 2

     [,1] [,2] [,3]
[1,]   10   12   14
[2,]   11   13   15
```

As we can see, we have an array of 2 layers with each layer having 2 rows and 3 columns. By using the `str()` function, we can see that R recognises the array has integer values with 3 dimensions. The first dimension, `1:2` refers to the rows, the second dimension, `1:3` refers to the columns, and the last dimension, `1:2` refers to the layers.

```
str(arr_data)
```

```
 int [1:2, 1:3, 1:2] 1 2 3 4 5 6 10 11 12 13 ...
```

`is.array()` can be used to ensure the data structure.

```
is.array(arr_data)
```

```
[1] TRUE
```

4.4.4 Data frame

A data frame is the extension of the matrix data structure. The difference between the former and the latter, the former contains the column names and each column may contain different data types.

```
df_data <- data.frame(
  ID = 1:5,
  Name = c("Mamat", "Abu", "Ali", "Chong", "Eva"),
  Age = c(25, 30, 22, 35, 28),
  Score = c(89, 95, 76, 88, 92)
)
df_data
```

```
  ID  Name Age Score
1  1 Mamat  25    89
2  2   Abu  30    95
3  3   Ali  22    76
4  4 Chong  35    88
5  5   Eva  28    92
```

We can further check the data structure and type using `str()`. So, our data structure is a data frame, consisting of 4 columns; the first column is an integer, the second column is a character, third and fourth columns are numeric.

```
str(df_data)
```

```
'data.frame':   5 obs. of  4 variables:
 $ ID   : int  1 2 3 4 5
 $ Name : chr  "Mamat" "Abu" "Ali" "Chong" ...
 $ Age  : num  25 30 22 35 28
 $ Score: num  89 95 76 88 92
```

We can double-check the data structure using `is.data.frame()`.

```
is.data.frame(df_data)
```

```
[1] TRUE
```

4.4.5 List

Lastly, we have a list. So, the list is a more advanced data structure in which we can have different data structures in a data structure. Let us see the example of a list in which we combine the previous vector and matrix data structures.

```
list_data <- list(
  "vector" = vec_data,
  "matrix" = mat_data
)
list_data
```

```
$vector
[1] 1 2 3 4

$matrix
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

We can further assess each data in the list using the `$` symbol.

```
list_data$vector
```

```
[1] 1 2 3 4
```

By using `str()`, we see our data is a list with 2 elements or components; a vector and matrix.

```
str(list_data)
```

```
List of 2
 $ vector: num [1:4] 1 2 3 4
 $ matrix: num [1:3, 1:2] 1 2 3 4 5 6
```

To further confirm our data is a list, we can use `is.list()`.

```
is.list(list_data)
```

```
[1] TRUE
```

4.5 Data in R

R itself contains the internal data that you can load for practice or other purposes. You can run `data()` in the Console to see what are the data available in R.

```
data()
```

So, for example, if you want to load any of these data, you can type the name of the datasets. You will see under the `Environment` pane, the chickWeight dataset is loaded to your environment.

```
data("ChickWeight")
```

Additionally, it is quite common that R packages have their own datasets as well. We will see this many times in this book going forward. Lastly, R is capable of reading different data formats.

Table 4.1: Common data formats and corresponding R codes for reading them.

Format	R codes
.csv	`read.csv()`
.sav (SPSS)	`haven::read_sav()`
.xlsx (Excel)	`readxl::read_excel()`
.txt	`read.table()`
.dta (STATA)	`haven::read_dta()`

The data formats that can be read are limited to the one listed in the table. Almost all data formats can be read in R. An efficient way to know whether R is capable of reading certain data formats by just a quick Google search.

For the rest of the chapter we going to the `iris` dataset, which is already available in R. To know more about this dataset, you can type the below code in the console.

```
?iris
```

4.6 Packages

A package is a collection of functions and sample data that can be utilised for certain tasks. Certain functions in R are already loaded when you open R or RStudio. However, to use more advanced functions we need to install and load a package.

The packages can installed using `install.packages()`. For example, the code below will install the `dplyr` package which is commonly utilised for data wrangling and manipulation.

```
install.packages("dplyr")
```

To use the installed packages, we need to load the packages using the `library()` function.

```
library(dplyr)
```

Now, that we know what is a package, we might wonder where exactly these packages coming from.

The official packages in R are located in the Comprehensive R Archive Network (CRAN). This link contains all available packages in CRAN. Additionally, there is CRAN Task Views. At the time of this writing, CRAN contains 21,606 R packages intended for various tasks.

Furthermore, for those interested in bioinformatics, the related packages are located in the Bioconductor. At the time of this writing, Bioconductor contains 2,289 R packages related to bioinformatics.

In addition to CRAN and Bioconductor, there are unofficial R packages, which are usually located in GitHub and GitLab. There are probably thousands of these unofficial packages. For example, `dmetar` which is located in GitHub, contains R functions and codes to facilitate the conduction of meta-analyses.

More often than not, the official packages in CRAN also have their GitHub repositories in which they contain the latest development of R functions and codes before they going to be released in CRAN. So, current bugs and errors in the package are corrected first using the unofficial packages from GitHub or GitLab before they are eventually released to the CRAN repositories.

Several packages can help in installing the unofficial packages. The two most commonly used packages are `devtools` and `remote` (or at least I commonly used them).

First, we need to install the packages.

```
install.packages("devtools")
install.packages("remote")
```

Next, we can install the unofficial packages.

```
devtools::install_github("MathiasHarrer/dmetar")
remotes::install_github("MathiasHarrer/dmetar")
```

`MathiasHarrer` refers to the GitHub account or usually the author's account on GitHub, followed by the name of the package. Depending on one's preference, we can choose to use either `devtools` or `remote`.

4.7 Functions

A function in R is a block of code designed to perform a specific task. It consists of an argument, in which we need to supply it. For example, the `mean()` function is designed to find the mean or average across the numeric values. The numeric values are the argument that we need to supply to the function.

```
num_values <- c(5, 6, 8, 10)
mean(num_values)
```

```
[1] 7.25
```

The base R itself has numerous functions that are accessible to us. These base R functions can be used immediately once we open R (or RStudio or any other IDEs). In contrast, we also have R functions from the packages that we installed. First, we need to install the packages.

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

We will learn about `tidyverse` in a little bit. Coming on to our current installation, once you install the package there are two ways to use the function inside the package. First, by loading the package, subsequently, all the functions in the package are ready to be used by us. For example below, `bind_rows()` is one of the functions from the `dplyr` package.

```r
# Load the package
library(dplyr)

# Example of functions from dplyr package
bind_rows()
bind_cols()
```

Secondly, we can use `::` to access a single function that we are interested in. However, in this approach, only a single function is loaded.

```r
# Example of dplyr functions
dplyr::bind_rows()
dplyr::bind_cols()
```

As we can see, every time we want to call a function from the `dplyr` package we type `dplyr::` as we do not load the package first.

4.8 Tidyverse package

We have learned that R packages contain a collection of functions and R codes that we utilise once we load the packages. So, tidyverse is an opinionated collection of R packages designed for data science (Wickham et al. 2019).

```r
# List of all packages in tidyverse
tidyverse::tidyverse_packages(include_self = FALSE) |>
  data.frame(Packages = _)
```

```
        Packages
1          broom
2     conflicted
3            cli
4         dbplyr
5          dplyr
6         dtplyr
7        forcats
8        ggplot2
9    googledrive
10 googlesheets4
11         haven
```

```
12          hms
13         httr
14     jsonlite
15    lubridate
16     magrittr
17       modelr
18       pillar
19        purrr
20         ragg
21        readr
22       readxl
23       reprex
24        rlang
25   rstudioapi
26        rvest
27      stringr
28       tibble
29        tidyr
30         xml2
```

Table 4.2 below summarises common packages in `tidyverse` and its uses.

Table 4.2: Common packages in tidyverse.

Packages	Summary
ggplot2	For data visualisation.
dplyr	Provides tools for data manipulation, including functions for filtering, selecting, grouping, and summarizing data.
tidyr	Specializes in data tidying, allowing users to transform datasets into a tidy format ready for analysis or visualisation.
readr	Used for reading rectangular data (e.g., CSV, TSV files) into R quickly and efficiently.
purr	Introduces a functional programming paradigm in R with tools for applying functions to data structures like lists and vectors.
tibble	Enhances data frames in R by making them more user-friendly with better printing options and stricter type checking

Packages	Summary
stringr	Facilitates consistent handling of strings, offering functions for string manipulation, pattern matching, and transformations.
forcats	Designed to work with categorical data (factors).

`Tidyverse` is commonly utilised for data analysis and throughout this book, we going to use `tidyverse` functions numerous times.

4.9 Pipe operators

There are two types of pipe in R:

1. `|>`: this pipe is from base R, first introduced in R version 4.1.0.
2. `%>%`: this pipe is from the `magrittr` package, which presented `tidyverse`.

To use `|>`, we do not actually need to load anything as it is already available in base R. However, to use `%>%`, you need to load `tidyverse`. Certain `tidyverse` associated packages such as `dplyr`, `forcats`, and `magrittr` also load the `%>%`.

The main function of these pipe operators (regardless of which one we use) is to make our R codes more readable and intuitive. So, let us compare the codes without and with the pipe.

```
mean_value <- mean(
  subset(
    data.frame(value = 1:10, group = rep(c("A", "B"), each = 5)),
    group == "A"
  )$value
)
mean_value
```

```
[1] 3
```

Now, compare the codes with the pipe.

```
mean_value <- data.frame(value = 1:10, group = rep(c("A", "B"), each = 5)) |>
  subset(group == "A") |>
  with(mean(value))
mean_value
```

```
[1] 3
```

Basically, what we do in both R codes are:

1. Create a data frame with ten rows and two columns.

```r
data.frame(value = 1:10, group = rep(c("A", "B"), each = 5))
```

```
   value group
1      1     A
2      2     A
3      3     A
4      4     A
5      5     A
6      6     B
7      7     B
8      8     B
9      9     B
10    10     B
```

2. Filter out the group column to value A.

```r
data.frame(value = 1:10, group = rep(c("A", "B"), each = 5)) |>
  subset(group == "A")
```

```
  value group
1     1     A
2     2     A
3     3     A
4     4     A
5     5     A
```

3. Calculate the mean.

```r
mean(c(1, 2, 3, 4, 5))
```

```
[1] 3
```

Coming back to both R codes, we can intuitively see that the codes with the pipe are more readable and clear compared to the other one. The codes with pipe can be read line by line, while the codes without the pipe need to be read inside out. As you can imagine, once our codes are more complex, the less readable the codes will become.

So, which pipe operators to choose?

In most cases, the differences are not significant enough to impact your code. Therefore, you can choose the pipe operator that best suits your preference or coding style. Most users will find that both options perform similarly for general tasks, so selecting one often comes down to familiarity or ease of use.

`|>` is a built-in pipe operator which can be used immediately without the need to load any package. `%>%` can be used once the `tidyverse` package is loaded. The latter pipe has short cut built-in in RStudio. For Windows user, `Ctrl` + `Shift` + `M` and for Mac user, `Cmd` + `Shift` + `M`. Additionally, we can set up the shortcut for `|>` or further change the shortcut for `%>%`, though, this will not be covered in this book.

4.10 Chapter summary

In this chapter, we learn about:

1. Data, its types and structures in R.
2. R packages and functions.
3. Pipe operators.

4.11 Revision

1. What this `?` actually do in R?
2. List all the data types that we learn in this chapter.
3. List all the data structures that we learn in this chapter.
4. What is the difference between installing the package and loading the package?
5. Which one to choose between `|>` and `%>%`?
6. List three packages in `tidyverse` and summarise their functionalities.

5 Data wrangling

> "Without clean data, or clean enough data, your data science is worthless."
>
> –Michael Stonebraker

Data wrangling refers to the process of transforming and preparing raw data into a clean and structured format suitable for analysis. This involves various steps, such as data cleaning, reshaping, merging, and filtering, to ensure the dataset is ready for statistical analysis or visualisation.

In most cases, data wrangling can occupy a significant portion of the time required for data analysis, often more than other stages. In this chapter, we will explore common operations in data wrangling. You will learn how to perform these operations using either base R functions, `tidyverse` functions, or both.

5.1 Load packages

The following packages will be used in this chapter. Please run these lines of code before proceeding to other sections.

```
library(tidyverse)
```

5.2 Indexing

Indexing involves selecting specific elements within data structures, which can be done using `[]`.

5.2.1 Vector

First, let's create a vector.

```
vec_data <- 1:10
vec_data
```

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

To select specific elements:

```
# Select a single element
vec_data[2]
```

```
[1] 2
```

```
# Select 3rd and 6th element
vec_data[c(3,6)]
```

```
[1] 3 6
```

5.2.2 Data frame

We will use `iris`, a built-in dataset in R, to demonstrate indexing within a data frame. Detailed information about this dataset can be accessed by typing `?iris` in the Console.

```
?iris
```

Below is a summary of this dataset.

```
summary(iris)
```

```
  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
 Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
 1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
 Median :5.800   Median :3.000   Median :4.350   Median :1.300
 Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
 3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
 Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
       Species
 setosa    :50
 versicolor:50
 virginica :50
```

We will use `[]`, where the general syntax is `[row, column]`.

```
# Selecting the 1st row
iris[1, ]
```

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
```

```
# Selecting the 1st and 2nd row
iris[c(1, 2), ]
```

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
```

Now let's see how to index columns.

```
# Selecting the 1st row
iris[, 1]

# Selecting the 1st and 2nd row
iris[, c(1, 2)]
```

```
[1] 5.1 4.9 4.7 4.6 5.0 5.4
```

```
  Sepal.Length Sepal.Width
1          5.1         3.5
2          4.9         3.0
3          4.7         3.2
4          4.6         3.1
5          5.0         3.6
6          5.4         3.9
```

By default, this shows the entire column, but here we'll limit the output to the first six items for clarity.

Instead of numbers, we can also use column names:

```r
# Selecting the 1st row
iris[, "Sepal.Length"]

# Selecting the 1st and 2nd row
iris[, c("Sepal.Length", "Sepal.Width")]
```

```
[1] 5.1 4.9 4.7 4.6 5.0 5.4
```

```
  Sepal.Length Sepal.Width
1          5.1         3.5
2          4.9         3.0
3          4.7         3.2
4          4.6         3.1
5          5.0         3.6
6          5.4         3.9
```

To select both a specific row and column, we can combine what we've learned. For example, to select the first row and the first column:

```r
# Approach 1
iris[1, 1]
```

```
[1] 5.1
```

```r
# Approach 2
iris[1, "Sepal.Length"]
```

```
[1] 5.1
```

To select the first five rows and the first two columns:

```r
# Approach 1
iris[1:5, 1:2]
```

```
  Sepal.Length Sepal.Width
1          5.1         3.5
2          4.9         3.0
3          4.7         3.2
4          4.6         3.1
5          5.0         3.6
```

```r
# Approach 2
iris[1:5, c("Sepal.Length", "Sepal.Width")]
```

```
  Sepal.Length Sepal.Width
1          5.1         3.5
2          4.9         3.0
3          4.7         3.2
4          4.6         3.1
5          5.0         3.6
```

For selecting a single column, an easier approach is to use `$`:

```r
iris$Petal.Length
```

```
[1] 1.4 1.4 1.3 1.5 1.4 1.7
```

5.2.3 Selecting and slicing

In R, there are many ways to perform tasks. Rather than using `[]`, `dplyr` provides `select()` and `slice()`, which are often preferred for their readability. The `dplyr` package is part of the `tidyverse`.

The `select()` function is used to choose specific columns.

```r
# Select a single column
iris %>%
  select(Sepal.Length)
```

```
  Sepal.Length
1          5.1
2          4.9
3          4.7
4          4.6
5          5.0
6          5.4
```

```r
# Select several columns
iris %>%
  select(Sepal.Length, Sepal.Width)
```

```
  Sepal.Length Sepal.Width
1          5.1         3.5
2          4.9         3.0
3          4.7         3.2
4          4.6         3.1
5          5.0         3.6
6          5.4         3.9
```

Similarly, `slice()` is used to extract specific rows.

```
# Select a single row
iris %>%
  slice(100)
```

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
1          5.7         2.8          4.1         1.3 versicolor
```

```
# Select several rows
iris %>%
  slice(2, 5, 100)
```

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
1          4.9         3.0          1.4         0.2     setosa
2          5.0         3.6          1.4         0.2     setosa
3          5.7         2.8          4.1         1.3 versicolor
```

By combining both `select()` and `slice()`, we can access specific rows and columns.

```
iris %>%
  select(Sepal.Length) %>%
  slice(1:5)
```

```
  Sepal.Length
1          5.1
2          4.9
3          4.7
4          4.6
5          5.0
```

5.3 Filtering

Filtering allows us to select rows based on a condition. For example, if we want to filter the `Species` column for the value `"setosa"` in the `iris` dataset, we start by creating an index:

```
ind <- iris$Species == "setosa"
```

Next, we apply the index to the dataset.

```
iris[ind, ]
```

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

The `==` symbol is a logical operator. Table 5.1 presents the most common logical operators in R.

Table 5.1: Common logical operators in R.

Operators	Description
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to

Alternatively, we can use `filter()` from `dplyr` for the same result.

```
iris %>%
  filter(Species == "setosa")
```

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
```

```
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

`filter()` is often more readable, especially for beginners, though both methods yield the same output.

Additionally, we can combine multiple conditions using `|` (or) and `&` (and). For instance, to filter the `iris` dataset for rows where:

1. `Species` is `"setosa"`, and
2. `Sepal.Length` is greater than 5.6 cm:

```r
# Make an index
ind2 <- iris$Species == "setosa" & iris$Sepal.Length > 5.6

# Apply the index to the dataset
iris[ind2, ]
```

```
   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
15          5.8         4.0          1.2         0.2  setosa
16          5.7         4.4          1.5         0.4  setosa
19          5.7         3.8          1.7         0.3  setosa
```

Or, with `filter()`:

```r
iris %>%
  filter(Species == "setosa" & Sepal.Length > 5.6)
```

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.8         4.0          1.2         0.2  setosa
2          5.7         4.4          1.5         0.4  setosa
3          5.7         3.8          1.7         0.3  setosa
```

5.4 Sorting

Sorting arranges the rows of a data frame by specific column values.

To demonstrate, let's limit `iris` to its first ten rows for easier display. Using `head()` will give the top ten rows, while `tail()` provides the last rows.

```
iris_top10 <- iris %>% head(10)
iris_top10
```

```
   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1           5.1         3.5          1.4         0.2  setosa
2           4.9         3.0          1.4         0.2  setosa
3           4.7         3.2          1.3         0.2  setosa
4           4.6         3.1          1.5         0.2  setosa
5           5.0         3.6          1.4         0.2  setosa
6           5.4         3.9          1.7         0.4  setosa
7           4.6         3.4          1.4         0.3  setosa
8           5.0         3.4          1.5         0.2  setosa
9           4.4         2.9          1.4         0.2  setosa
10          4.9         3.1          1.5         0.1  setosa
```

To sort by a single column, use `sort()`. Setting `decreasing = FALSE` sorts in ascending order.

```
sort(iris_top10$Sepal.Length, decreasing = FALSE)
```

```
 [1] 4.4 4.6 4.6 4.7 4.9 4.9 5.0 5.0 5.1 5.4
```

Another useful function is `order()`, which returns the indices and can arrange the entire dataset.

```
# Make an index
ind3 <- order(iris_top10$Sepal.Length, decreasing = FALSE)

# Apply the index to the dataset
iris_top10[ind3, ]
```

```
   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
9           4.4         2.9          1.4         0.2  setosa
4           4.6         3.1          1.5         0.2  setosa
7           4.6         3.4          1.4         0.3  setosa
3           4.7         3.2          1.3         0.2  setosa
2           4.9         3.0          1.4         0.2  setosa
10          4.9         3.1          1.5         0.1  setosa
5           5.0         3.6          1.4         0.2  setosa
8           5.0         3.4          1.5         0.2  setosa
1           5.1         3.5          1.4         0.2  setosa
6           5.4         3.9          1.7         0.4  setosa
```

With `tidyverse`, we have `arrange()`, equivalent to `order()` in base R:

```
iris_top10 %>%
  arrange(Sepal.Length)
```

```
   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1           4.4         2.9          1.4         0.2  setosa
2           4.6         3.1          1.5         0.2  setosa
3           4.6         3.4          1.4         0.3  setosa
4           4.7         3.2          1.3         0.2  setosa
5           4.9         3.0          1.4         0.2  setosa
6           4.9         3.1          1.5         0.1  setosa
7           5.0         3.6          1.4         0.2  setosa
8           5.0         3.4          1.5         0.2  setosa
9           5.1         3.5          1.4         0.2  setosa
10          5.4         3.9          1.7         0.4  setosa
```

To sort in descending order, use `desc()`:

```
iris_top10 %>%
  arrange(desc(Sepal.Length))
```

```
   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1           5.4         3.9          1.7         0.4  setosa
2           5.1         3.5          1.4         0.2  setosa
3           5.0         3.6          1.4         0.2  setosa
4           5.0         3.4          1.5         0.2  setosa
5           4.9         3.0          1.4         0.2  setosa
6           4.9         3.1          1.5         0.1  setosa
```

```
7           4.7         3.2          1.3         0.2  setosa
8           4.6         3.1          1.5         0.2  setosa
9           4.6         3.4          1.4         0.3  setosa
10          4.4         2.9          1.4         0.2  setosa
```

5.5 Rename

First, let's check the column names.

```
colnames(iris_top10)
```

```
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"
```

Suppose we want to rename the `species` column to `type`.

```
colnames(iris_top10)[5] <- "type"
```

Now, if we check the column names again:

```
colnames(iris_top10)
```

```
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "type"
```

Alternatively, we can use the `rename()` function from `dplyr`. Here, we will rename `Sepal.Length` to `LengthofSepal`.

```
# Change the column name
iris_top10 <-
  iris_top10 %>%
  rename(LengthofSepal = "Sepal.Length")

# Check the column names
colnames(iris_top10)
```

```
[1] "LengthofSepal" "Sepal.Width"   "Petal.Length"  "Petal.Width"
[5] "type"
```

5.6 Create new column

Let's create a smaller iris dataset first.

```
iris_bottom10 <- tail(iris, 10)
```

Using base R functions, we can create a new column as follows:

```
iris_bottom10$Sepal.Lengthx2 <- iris_bottom10$Sepal.Length * 2
head(iris_bottom10)
```

```
    Sepal.Length Sepal.Width Petal.Length Petal.Width   Species Sepal.Lengthx2
141          6.7         3.1          5.6         2.4 virginica           13.4
142          6.9         3.1          5.1         2.3 virginica           13.8
143          5.8         2.7          5.1         1.9 virginica           11.6
144          6.8         3.2          5.9         2.3 virginica           13.6
145          6.7         3.3          5.7         2.5 virginica           13.4
146          6.7         3.0          5.2         2.3 virginica           13.4
```

Here, we create a new variable `Sepal.Lengthx2` by multiplying `Sepal.Length` by 2. Using `tidyverse`, we can achieve this with `mutate()`.

```
iris_bottom10 <-
  iris_bottom10 %>%
  mutate(Sepal.Widthx2 = Sepal.Width * 2)
head(iris_bottom10)
```

```
    Sepal.Length Sepal.Width Petal.Length Petal.Width   Species Sepal.Lengthx2
141          6.7         3.1          5.6         2.4 virginica           13.4
142          6.9         3.1          5.1         2.3 virginica           13.8
143          5.8         2.7          5.1         1.9 virginica           11.6
144          6.8         3.2          5.9         2.3 virginica           13.6
145          6.7         3.3          5.7         2.5 virginica           13.4
146          6.7         3.0          5.2         2.3 virginica           13.4
    Sepal.Widthx2
141           6.2
142           6.2
143           5.4
144           6.4
145           6.6
146           6.0
```

5.7 Change data format

Data can be structured in two primary formats:

- **Long format**: Each row represents a single observation.
- **Wide format**: Each subject has one row, with variables across columns.

In long format, each row represents a single observation. This format is commonly used for data manipulation and analysis. Let's create an example of a long-format dataset. Here, five participants were given a dietary supplement to reduce weight, and the data contains:

1. `id`: participant ID
2. `time`: either pre- or post-supplement
3. `weight`: participant's weight

```
# Set seed for reproducibility
set.seed(123)

# Create a long format data
data_long <- data.frame(
  id = rep(1:5, each = 2),
  time = rep(c("Pre", "Post"), 5),
  weight =  sample(x = 60:80, size = 10, replace = TRUE)
  )

# View the data
data_long
```

```
   id time weight
1   1  Pre     74
2   1 Post     78
3   2  Pre     73
4   2 Post     62
5   3  Pre     69
6   3 Post     77
7   4  Pre     70
8   4 Post     64
9   5  Pre     79
10  5 Post     73
```

In wide format, each subject has a single row, with each measurement in a separate column. Here is `data_long` converted to wide format.

```r
# Set seed for reproducibility
set.seed(123)

# Create a wide format data
data_wide <- data.frame(
  id = 1:5,
  Pre = sample(60:80, 5, replace = TRUE),
  Post = sample(60:80, 5, replace = TRUE)
  )

# View the data
data_wide
```

```
  id Pre Post
1  1  74   77
2  2  78   70
3  3  73   64
4  4  62   79
5  5  69   73
```

In the wide format, data is often easier to interpret. Converting between long and wide formats is simple in R. To transform `data_wide` into long format, use `pivot_longer()`:

```r
data_long2 <-
  data_wide %>%
  pivot_longer(cols = 2:3, names_to = "time", values_to = "weight")
data_long2
```

```
# A tibble: 10 x 3
      id time  weight
   <int> <chr>  <int>
 1     1 Pre       74
 2     1 Post      77
 3     2 Pre       78
 4     2 Post      70
 5     3 Pre       73
 6     3 Post      64
 7     4 Pre       62
 8     4 Post      79
 9     5 Pre       69
10     5 Post      73
```

For `pivot_longer()`, we need to supply three arguments:

- `cols`: columns to be changed into a long format excluding the id column
- `names_to`: name of a new column which consist of column names from a wide format data
- `values_to`: name of a new column which consist of values from `cols`

To convert `data_long2` back to wide format, use `pivot_wider()`:

```
data_wide2 <-
  data_long2 %>%
  pivot_wider(id_cols = "id", names_from = "time", values_from = "weight")
data_wide2
```

```
# A tibble: 5 x 3
     id   Pre  Post
  <int> <int> <int>
1     1    74    77
2     2    78    70
3     3    73    64
4     4    62    79
5     5    69    73
```

To use `pivot_wider()`, the three basic arguments needed are:

- `id_cols`: ID column
- `names_from`: name of a column to get the column names for the wide data
- `values_from`: name of a column to get the values from

Both functions have additional arguments detailed in the `Help` pane (use `?` in front of the function name).

5.8 Change variable type

Here are the main variable types in R:

Table 5.2: Example of each of variable type in R.

Variables	Examples
Integer	100, 77
Numeric	100.2, 77.8

Variables	Examples
Character	“hello”, “ahmad”
Logical	TRUE, FALSE
Factor	“male”, “female”
Date	9/7/2024, 9 July 2024

The most common types in data are numeric, factor, and date. When importing data from software such as SPSS, STATA, or Excel, R may not always recognise the correct types. Let’s create a sample dataset and explore handling these issues.

```
# Create a data frame with mixed-up variable types
data_messed_up <- data.frame(
  id = as.character(1:6),
  score = as.character(sample(1:100, 6)),
  gender = rep(c("Male", "Female"), length.out = 6)
  )

# View the variable types
str(data_messed_up)
```

```
'data.frame':   6 obs. of  3 variables:
 $ id    : chr  "1" "2" "3" "4" ...
 $ score : chr  "25" "90" "91" "69" ...
 $ gender: chr  "Male" "Female" "Male" "Female" ...
```

```
# View the data
data_messed_up
```

```
  id score gender
1  1    25   Male
2  2    90 Female
3  3    91   Male
4  4    69 Female
5  5    98   Male
6  6    57 Female
```

We can see that `score` should be numeric and `gender` a factor. To convert these types, use `as.numeric()` and `as.factor()`. Let’s create a copy of `data_messed_up` for demonstration purposes.

```r
data_messed_up2 <- data_messed_up
```

Using base R functions, we can adjust the types:

```r
# Change score column to numeric
data_messed_up$score <- as.numeric(data_messed_up$score)

# Change gender column to factor
data_messed_up$gender <- as.factor(data_messed_up$gender)

# View variable type
str(data_messed_up)
```

```
'data.frame':   6 obs. of  3 variables:
 $ id    : chr  "1" "2" "3" "4" ...
 $ score : num  25 90 91 69 98 57
 $ gender: Factor w/ 2 levels "Female","Male": 2 1 2 1 2 1
```

Using `tidyverse`, we can achieve the same result with `mutate()`:

```r
# Change variable types for score and gender
data_messed_up2 <-
  data_messed_up2 %>%
  mutate(score = as.numeric(score),
         gender = as.factor(gender))

# View variable type
str(data_messed_up2)
```

```
'data.frame':   6 obs. of  3 variables:
 $ id    : chr  "1" "2" "3" "4" ...
 $ score : num  25 90 91 69 98 57
 $ gender: Factor w/ 2 levels "Female","Male": 2 1 2 1 2 1
```

5.8.1 Handling date

Dates can be tricky. For date variables, the standard format is `YYYY-MM-DD`. Using `lubridate`, we can work with various date formats easily. Let's look at a few examples:

```r
# Using base R
date_data <- as.Date("2024-11-30")
str(date_data)
```

```
 Date[1:1], format: "2024-11-30"
```

```r
# Using lubridate
date_data2 <- as_date("2024-11-30")
str(date_data2)
```

```
 Date[1:1], format: "2024-11-30"
```

For non-standard formats, use `lubridate` functions `ymd()`, `dmy()`, and `mdy()`.

```r
# DD-MM-YYYY
date_data3 <- dmy("30-11-2024")
str(date_data3)
```

```
 Date[1:1], format: "2024-11-30"
```

```r
# MM-DD-YY
date_data4 <- mdy("11-30-2024")
str(date_data4)
```

```
 Date[1:1], format: "2024-11-30"
```

Correctly formatted date variables allow for operations such as:

- Date calculation: adding days to a date.

  ```r
  date_data4 + 7
  ```

  ```
  [1] "2024-12-07"
  ```

- Extracting date components usch as month and year.

  ```r
  # Extract month
  month(date_data4)
  ```

  ```
  [1] 11
  ```

```r
# Extract year
year(date_data4)
```

```
[1] 2024
```

To demonstrate dates in a dataset, we'll recreate `data_messed_up` with a date column.

```r
# Create a data frame with mixed-up variable types
data_messed_up <- data.frame(
  id = as.character(1:6),
  score = as.character(sample(1:100, 6)),
  gender = rep(c("Male", "Female"), length.out = 6),
  date = as.character(Sys.Date() - 1:6)
  )

# View variable type
str(data_messed_up)
```

```
'data.frame':   6 obs. of  4 variables:
 $ id    : chr  "1" "2" "3" "4" ...
 $ score : chr  "92" "9" "93" "72" ...
 $ gender: chr  "Male" "Female" "Male" "Female" ...
 $ date  : chr  "2025-11-22" "2025-11-21" "2025-11-20" "2025-11-19" ...
```

The `date` column is recognised as a character. To convert `date` to a date format, use `as_date()`.

```r
# Change variable types for score, gender, and date
data_messed_up <-
  data_messed_up %>%
  mutate(score = as.numeric(score),
         gender = as.factor(gender),
         date = as_date(date))

# View variable type
str(data_messed_up)
```

```
'data.frame':   6 obs. of  4 variables:
 $ id    : chr  "1" "2" "3" "4" ...
 $ score : num  92 9 93 72 26 7
 $ gender: Factor w/ 2 levels "Female","Male": 2 1 2 1 2 1
 $ date  : Date, format: "2025-11-22" "2025-11-21" ...
```

The simplest solution for the date issue is to make sure we properly input the date according to the right format (`YYYY-MM-DD`) during the data collection process properly.

5.9 Merge datasets

If you collect data from different sources, you may need to combine datasets by row or column. `rbind()` combines datasets by row, while `cbind()` combines them by column.

Let us see how to combine two iris datasets.

```
# Data collected from area A
data1 <- iris

# Data collected from area B
data2 <- iris

# Combine both datasets by a row
data_combined_row <- rbind(data1, data2)
```

Now, we can check the dimensions of the data. The first element represents a row and the second element represents the column.

```
# Dimension of data1
dim(data1)
```

```
[1] 150   5
```

```
# Dimension of data2
dim(data2)
```

```
[1] 150   5
```

```
# Dimension of the combined data
dim(data_combined_row)
```

```
[1] 300   5
```

We can see that the rows of `data1` and `data2` each are 150. Combining both data by a row gives us 300 rows. It is to be noted that to use `rbind()`, both data should have the same column numbers and names. Additionally, `rbind()` is not limited to two data only.

Next, let us see the `cbind()`. Using the same data, we can combine both data by a column.

```
# Combine both datasets by a column
data_combined_col <- cbind(data1, data2)
```

Thus, by further checking the dimension, we can see that the total column of `data_combined_col` is 10, which is the sum of 5 columns in each of the `data1` and `data2`, while the row remained the same.

```
# Dimension of data1
dim(data1)
```

```
[1] 150   5
```

```
# Dimension of data2
dim(data2)
```

```
[1] 150   5
```

```
# Dimension of the combined data
dim(data_combined_col)
```

```
[1] 150  10
```

However, similar to `rbind()`, to use the `cbind()`, the rows of both data should be identical. If participant A is in the first row of `data1`, the, in `data2`, participant A should also be in the first row.

There is another set of functions that is more efficient than `cbind()`, in which we can combine two or more datasets according to the id. Let us create two datasets that are related and have the same ID.

```
# Set seed for reproducibility
set.seed(123)

# Create the first half of the data
data_half1 <- data.frame(
```

```
  name = c("Ahmad", "Ali", "Cheng", "Rama", "Wei"),
  height_cm = sample(160:180, 5, replace = FALSE)
  )

# Create the second half of the data
data_half2 <- data.frame(
  name = c("Ahmad", "Ali", "Cheng", "Rama", "Karim"),
  weight_kg = sample(70:90, 5, replace = FALSE)
  )

# The first dataset
data_half1
```

```
   name height_cm
1 Ahmad       174
2   Ali       178
3 Cheng       173
4  Rama       162
5   Wei       169
```

```
# The second dataset
data_half2
```

```
   name weight_kg
1 Ahmad        87
2   Ali        80
3 Cheng        74
4  Rama        83
5 Karim        88
```

Notice that the last row of both datasets is not similar. To combine both datasets, we can use either `left_join()` or `right_join()`. Both produce the same result. If there is a mismatch between the two datasets, `left_join()` will keep the first dataset as a reference, and any of the id of the second dataset that does not match the first one will be removed. `right_join()` works similar to the `left_join()` but in the opposite.

```
# Combine the data
data_full_left <-
  data_half1 %>% #first datasets
  left_join(data_half2, by = "name") #second datasets
```

```r
# View the combined data
data_full_left
```

```
   name height_cm weight_kg
1 Ahmad       174        87
2   Ali       178        80
3 Cheng       173        74
4  Rama       162        83
5   Wei       169        NA
```

We can see that `karim` in the second dataset is excluded. Let's see what will happen if we use `right_join()`.

```r
# Combine the data
data_full_right <-
  data_half1 %>% #first datasets
  right_join(data_half2, by = "name") #second datasets

# View the combined data
data_full_right
```

```
   name height_cm weight_kg
1 Ahmad       174        87
2   Ali       178        80
3 Cheng       173        74
4  Rama       162        83
5 Karim        NA        88
```

We can see that `Wei` in the first dataset is excluded. To include all the participants, despite the mismatch of the id is by using `full_join()`.

```r
# Combine the data
data_full <-
  data_half1 %>%
  full_join(data_half2, by = "name")

# View the combined data
data_full
```

```
   name height_cm weight_kg
1 Ahmad       174        87
2   Ali       178        80
3 Cheng       173        74
4  Rama       162        83
5   Wei       169        NA
6 Karim        NA        88
```

Both `Wei` and `Karim` are kept in the combined dataset. Additionally, there is `inner_join()`, which will exclude both `Wei` and `Karim`. This function keeps the data that the `name` is present in both datasets only.

```
# Combine the data
data_full_inner <-
  data_half1 %>%
  inner_join(data_half2, by = "name")

# View the combined data
data_full_inner
```

```
   name height_cm weight_kg
1 Ahmad       174        87
2   Ali       178        80
3 Cheng       173        74
4  Rama       162        83
```

5.10 Chapter summary

In this chapter, we covered the most common and basic operations in data wrangling. More operations were not covered in this chapter as this book is intended for beginners.

To summarise, we have covered:

- How to select a column and a row
- How to filter a dataset based on a condition applied to columns
- How to rename and create a column
- How to manage variable types
- How to combine several datasets into one

5.11 Revision

1. Load `mtcars` dataset in R.

```
head(mtcars)
```

```
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

2. Read about`mtcars`.

```
?mtcars
```

3. How many cars have `mpg` > 25?
4. How many cars have `mpg` > 25 and a number of carburetors of 2?
5. How many cars have a V-shaped engine?
6. Change `vs` and `am` into a factor.
7. Change the wide format data below into a long or tidy format.

```
# Set seed for reproducibility
set.seed(123)

# Create a wide format data
data_wide <- data.frame(
  id = 1:10,
  time1 = sample(1:100, 10, replace = TRUE),
  time2 = sample(1:100, 10, replace = TRUE),
  time3 = sample(1:100, 10, replace = TRUE),
  age = sample(18:80, 10, replace = TRUE)
  )

# View the data frame
head(data_wide)
```

```
  id time1 time2 time3 age
1  1    31    90    26  32
2  2    79    91     7  49
3  3    51    69    42  59
```

```
4  4    14    91     9  62
5  5    67    57    83  24
6  6    42    92    36  26
```

The result of the long format data should appear like this.

id	age	time_points	time_minute
1	32	time1	31
1	32	time2	90
1	32	time3	26
2	49	time1	79
2	49	time2	91
2	49	time3	7
3	59	time1	51
3	59	time2	69
3	59	time3	42
4	62	time1	14
4	62	time2	91
4	62	time3	9
5	24	time1	67
5	24	time2	57
5	24	time3	83
6	26	time1	42
6	26	time2	92
6	26	time3	36
7	58	time1	50
7	58	time2	9
7	58	time3	78
8	27	time1	43
8	27	time2	93
8	27	time3	81
9	40	time1	14
9	40	time2	99
9	40	time3	43
10	44	time1	25
10	44	time2	72
10	44	time3	76

6 Data visualisation

> "The simple graph has brought more information to the data analyst's mind than any other device."
>
> – John Tukey

Data visualization helps transform complex data sets into clear, insightful visuals, such as charts, graphs, maps, and infographics, to make it easier for people to understand, analyze, and interpret patterns, trends, and relationships within the data. By presenting data in a visual format, it allows users to quickly identify outliers and underlying patterns that might not be apparent in raw data alone.

In R itself, there are built-in functions for plotting various plots. Additionally, there are several packages available for plotting in R. Commonly utilised packages include `ggplot2`, `plotly`, `lattice`, and `leaflet`. `ggplot2` and `lattice` are commonly used for static plots, while packages such as `plotly` and `leaflet` are more common for interactive plots.

In this chapter, we going to cover how to create plots using base R and `ggplot2`. We going to learn the six most basic and commonly used plots in data analysis:

1. Barplot
2. Histogram
3. Boxplot
4. Violin plot
5. Scatter plot

We going to focus more on the `ggplot2` and only cover the basics for base R plotting. `ggplot2` is widely used for plotting and visualisation in the R community (Wickham 2016).

6.1 Load packages

Please load a `tidyverse` package before proceeding to the next section. We do not need to load the `ggplot2` package as it is part of the `tidyverse` meta package.

```
library(tidyverse)
```

6.2 Data

In this chapter, we going to utilise `ChickWeight` a built-in dataset in R. This dataset is about the effect of different diet regimens on the weight of the chicks.

```
# More info about the ChickWeight data
?ChickWeight
data("ChickWeight")
```

Let's create a new variable from the `Time` variable. In this dataset, the weight of the chicks was weighted each day until 21 days.

```
chick_data <-
  ChickWeight %>%
  mutate(Time_group = cut(Time, breaks = 3, labels = c("Period 1", "Period 2", "Period 3")))
```

We can check the variables.

```
str(chick_data)
```

```
Classes 'nfnGroupedData', 'nfGroupedData', 'groupedData' and 'data.frame':  578 obs. of  5 va
 $ weight    : num  42 51 59 64 76 93 106 125 149 171 ...
 $ Time      : num  0 2 4 6 8 10 12 14 16 18 ...
 $ Chick     : Ord.factor w/ 50 levels "18"<"16"<"15"<..: 15 15 15 15 15 15 15 15 15 15 ...
 $ Diet      : Factor w/ 4 levels "1","2","3","4": 1 1 1 1 1 1 1 1 1 1 ...
 $ Time_group: Factor w/ 3 levels "Period 1","Period 2",..: 1 1 1 1 2 2 2 2 3 3 ...
```

The second data we going to use is `ToothGrowth`. The data is about the effect of vitamin C on tooth growth in guinea pigs.

```
# More info about the data
?ToothGrowth
data("ToothGrowth")
```

Let's create a new variable based variable `dose`.

```
tooth_data <-
  ToothGrowth %>%
  mutate(dose_group = case_when(dose == 0.5 ~ "low",
                                dose == 1 ~ "intermediate",
                                .default = "high"),
         dose_group = as.factor(dose_group))
```

Here, we use `case_when()` to classify the dose into 3 groups:

1. 0.5 as low dose
2. 1 as intermediate dose
3. 2 as high dose

```
# Dose variable
table(tooth_data$dose)
```

```
0.5   1   2
 20  20  20
```

```
# Dose group variable
table(tooth_data$dose_group)
```

```
        high intermediate          low
          20           20           20
```

Then, we can check the variable types.

```
str(tooth_data)
```

```
'data.frame':   60 obs. of  4 variables:
 $ len       : num  4.2 11.5 7.3 5.8 6.4 10 11.2 11.2 5.2 7 ...
 $ supp      : Factor w/ 2 levels "OJ","VC": 2 2 2 2 2 2 2 2 2 2 ...
 $ dose      : num  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 ...
 $ dose_group: Factor w/ 3 levels "high","intermediate",..: 3 3 3 3 3 3 3 3 3 3 ...
```

Lastly, the third data that we going to use in this chapter is the `cars` dataset, another built-in dataset in R. The data is about the speed of the cars and the distances taken by the cars to stop.

```
# More info about the data
?cars
data("cars")
```

Next, we going to create a new variable, a type of car.

```r
# Set seed for reproducibility
set.seed(123)

# Create a new variable
cars_data <-
  cars %>%
  mutate(car = sample(c("A", "B", "C"), size = 50, replace = TRUE),
         car = as.factor(car))
```

We can check the new variable by using `table()`.

```r
table(cars_data$car)
```

```
 A  B  C
16 15 19
```

Also, we need to check the variable type.

```r
str(cars_data)
```

```
'data.frame':   50 obs. of  3 variables:
 $ speed: num  4 4 7 7 8 9 10 10 10 11 ...
 $ dist : num  2 10 4 22 16 10 18 26 34 17 ...
 $ car  : Factor w/ 3 levels "A","B","C": 3 3 3 2 3 2 2 2 3 1 ...
```

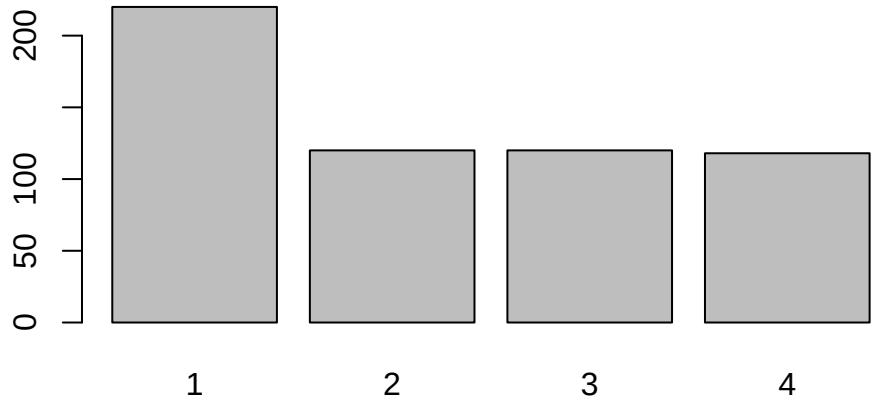
6.3 Barplot

Barplot is utilised when a variable is categorical. Let's explore the `ChickWeight` data. To use `barplot()`, the data should be a matrix.

```r
diet_table <-
  chick_data %>%
  select(Diet) %>%
  table()
```

Next, we can plot the `Diet` information.

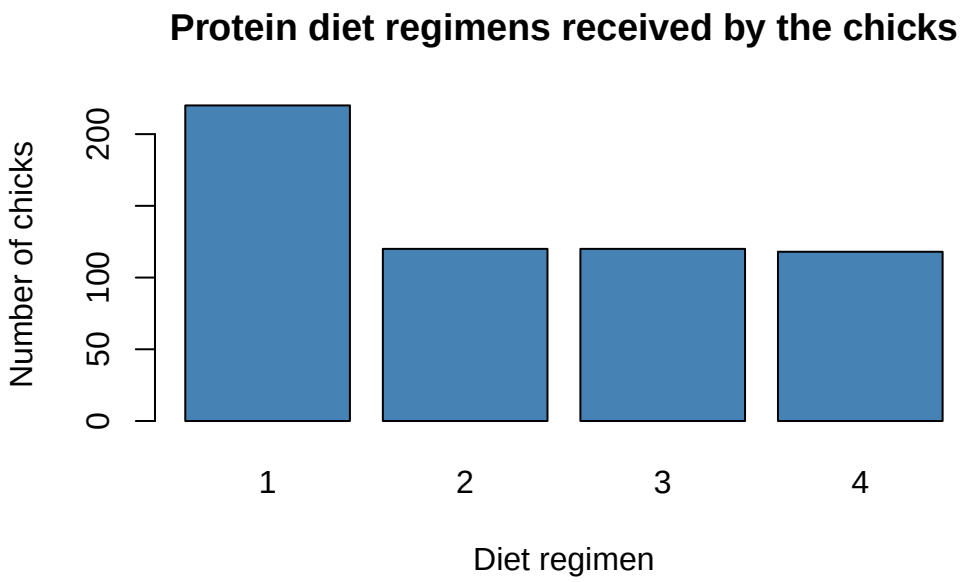
```r
barplot(diet_table)
```



Let's beautify the plot by setting up a colour, axis label and title. The arguments needed for those are:

- `col`: specifies the colour.
- `xlab`: name of the x-axis.
- `ylab`: name of the y-axis.
- `main`: the title of the barplot.

```r
barplot(diet_table, col = "steelblue",
        xlab = "Diet regimen",
        ylab = "Number of chicks",
        main = "Protein diet regimens received by the chicks")
```

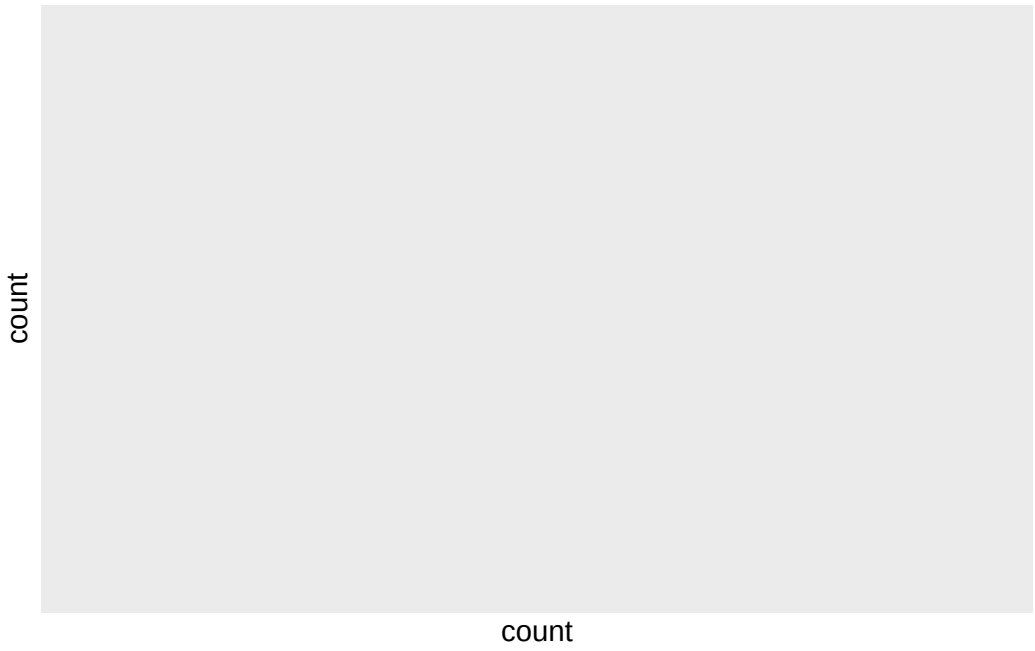


Now, let's do this in `ggplot2`. In `ggplot2`, the R codes are stacked on one over another. `ggplot()` initialises a `ggplot` object, and it sort of sets up the canvas for the plot. This part is aaplies to all the plots in `ggplot2`.

```
ggplot()
```

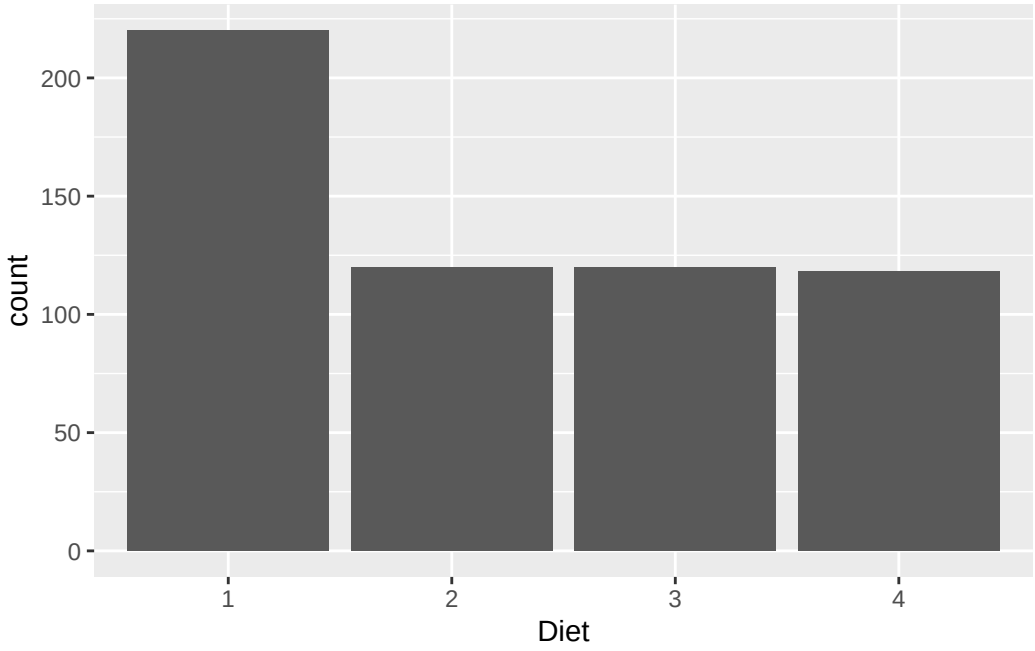
Next, we specify what type of plots that we want. Here, by using `geom_bar()`, we can see that the x- and y-axis are labelled.

```
ggplot() +
  geom_bar()
```



Let's supply the data for the barplot. The `x` argument specifies the column on the x-axis that we want to plot.

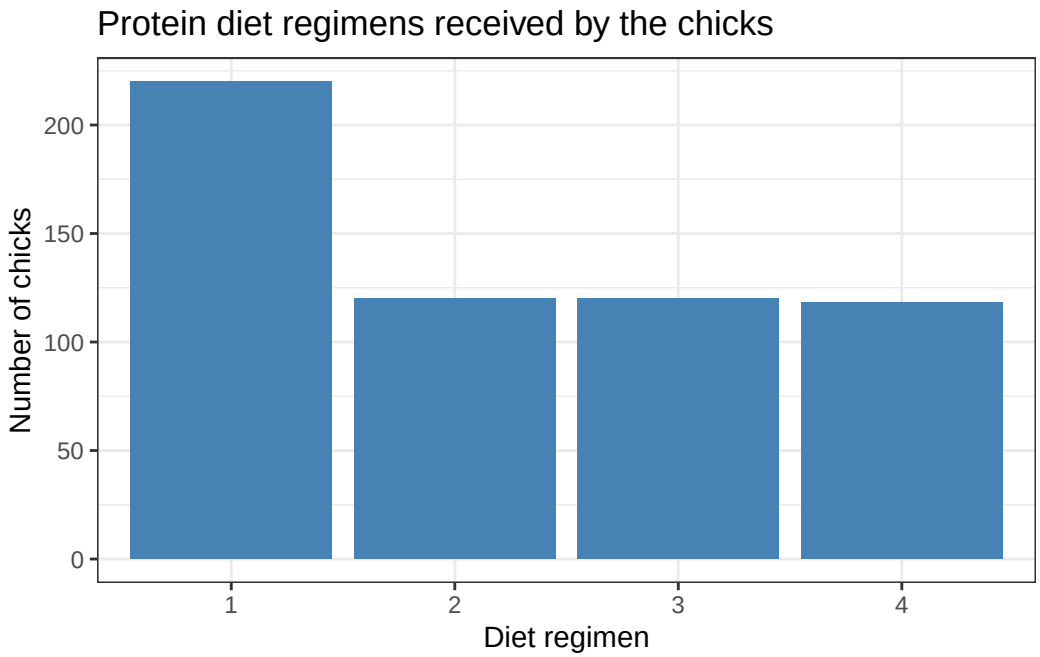
```
chick_data %>%
  ggplot(aes(x = Diet)) +
  geom_bar()
```



We can further beautify the plot by adding colour, labelling the axis, putting a title, and changing the theme of the plot. The arguments needed are:

- `fill`: specifies the colour.
- `x`: name of the x-axis.
- `y`: name of the y-axis.
- `title`: the title of the barplot.
- `theme_bw()`: specify the theme of the barplot.

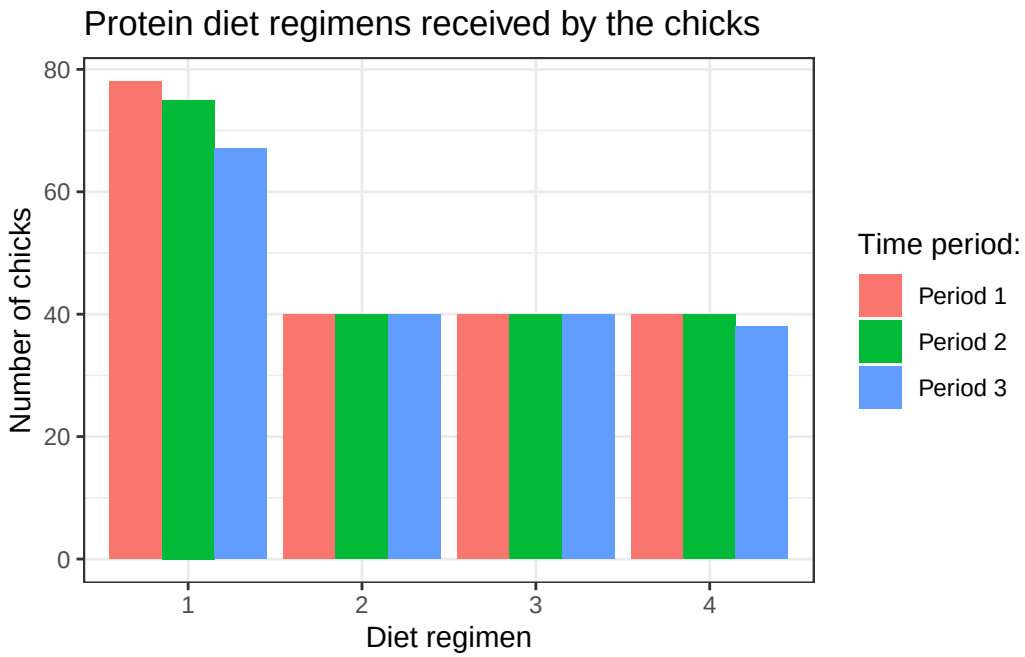
```
chick_data %>%
  ggplot(aes(x = Diet)) +
  geom_bar(fill = "steelblue") +
  labs(title = "Protein diet regimens received by the chicks",
       x = "Diet regimen",
       y = "Number of chicks") +
  theme_bw()
```



We can further add another variable, `Time_group`, that we created earlier. We remove the `colour` argument and add these arguments:

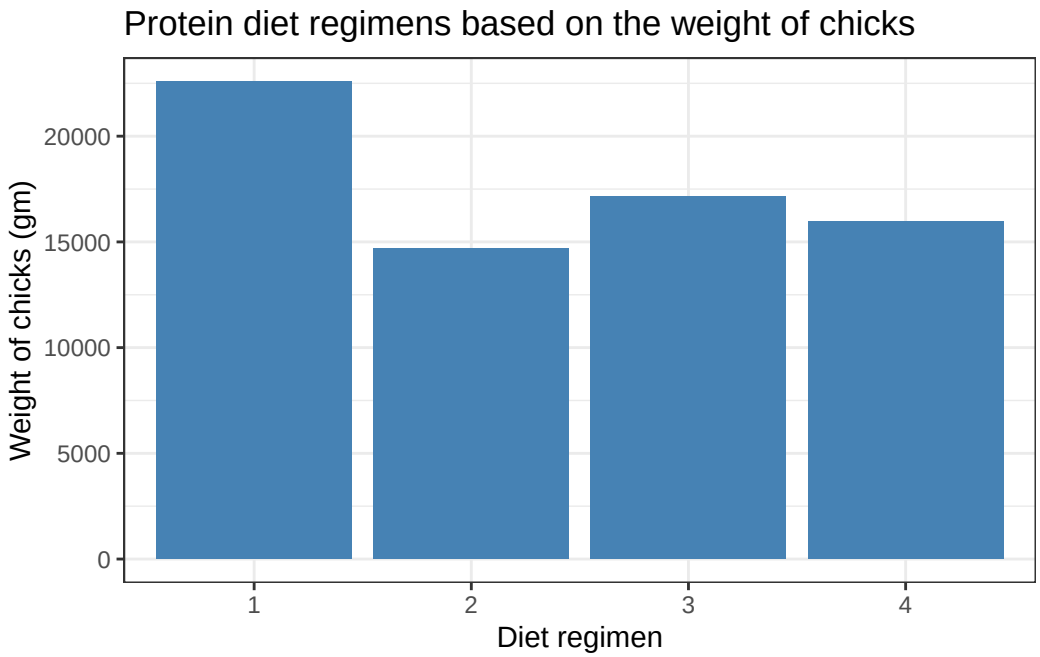
- `fill` in `ggplot()`: another group variable (must be a factor).
- `position`: dodge position allows us to maintain the vertical position of the plot.
- `fill` in `labs()`: the legend title.

```
chick_data %>%
  ggplot(aes(x = Diet, fill = Time_group)) +
  geom_bar(position = "dodge") +
  labs(title = "Protein diet regimens received by the chicks",
       x = "Diet regimen",
       y = "Number of chicks",
       fill = "Time period:") +
  theme_bw()
```



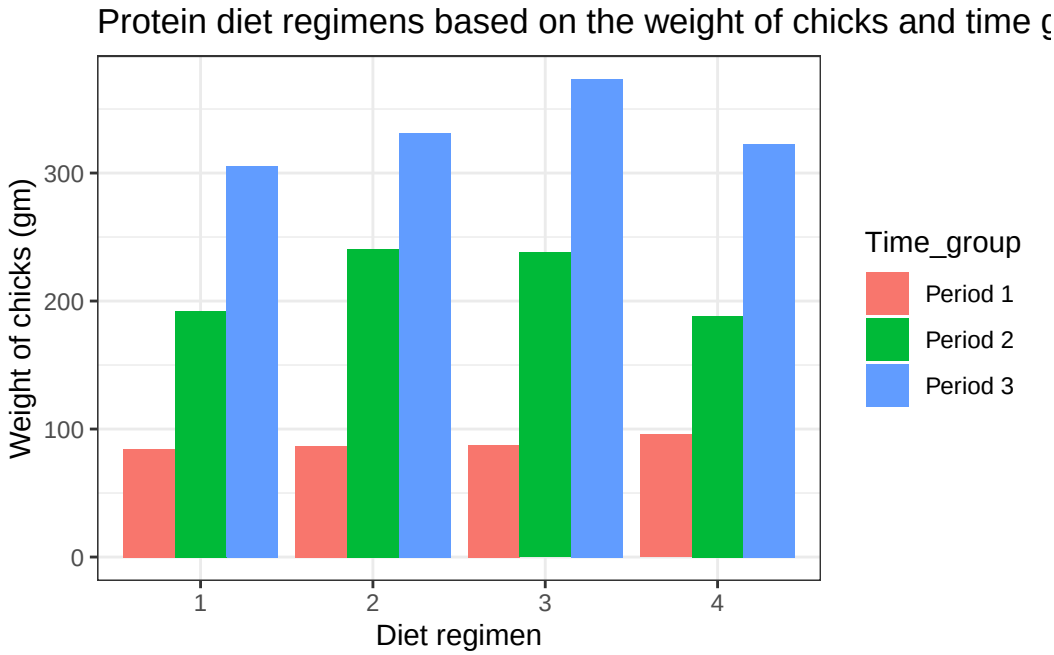
Additionally, we can do bar plot based on another numerical variable in the data. For example, let's plot diet regiemn based on the `weight`. Here, instead of `geom_bar()`, we are going to use `geom_col()`.

```
chick_data %>%
  ggplot(aes(x = Diet, y = weight)) +
  geom_col(fill = "steelblue") +
  labs(title = "Protein diet regimens based on the weight of chicks",
       x = "Diet regimen",
       y = "Weight of chicks (gm)") +
  theme_bw()
```



Also, we can add the third variable, `Time_group`. However, we need to specify `position_dodge()` in `geom_col()` to let R knows we want it side by side.

```
chick_data %>%
  ggplot(aes(x = Diet, y = weight, fill = Time_group)) +
  geom_col(position = position_dodge()) +
  labs(title = "Protein diet regimens based on the weight of chicks and time groups",
       x = "Diet regimen",
       y = "Weight of chicks (gm)") +
  theme_bw()
```



6.4 Histogram

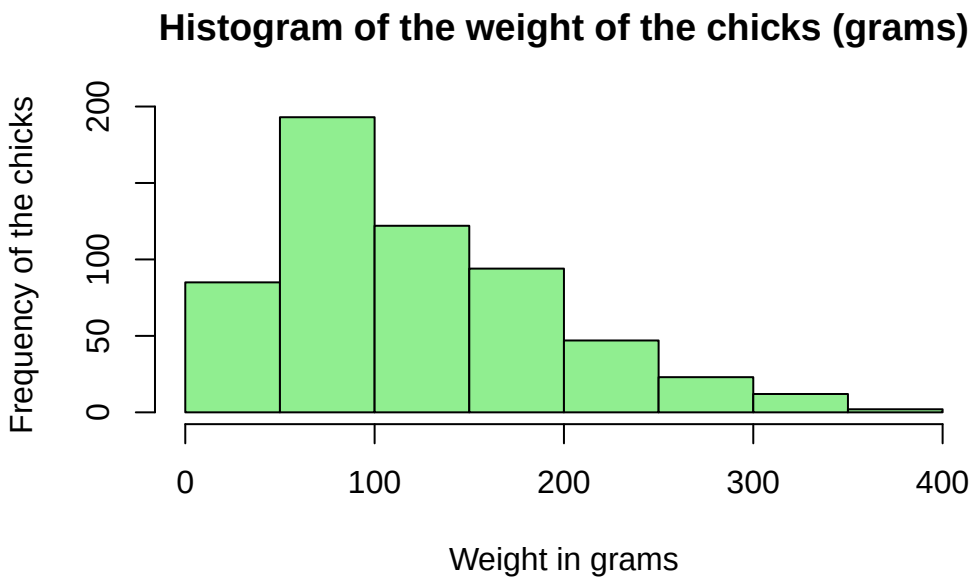
A histogram is used when a variable is numerical. It reflects the frequency distribution of the data.

Let's do the histogram in base R.

```
hist(chick_data$weight)
```

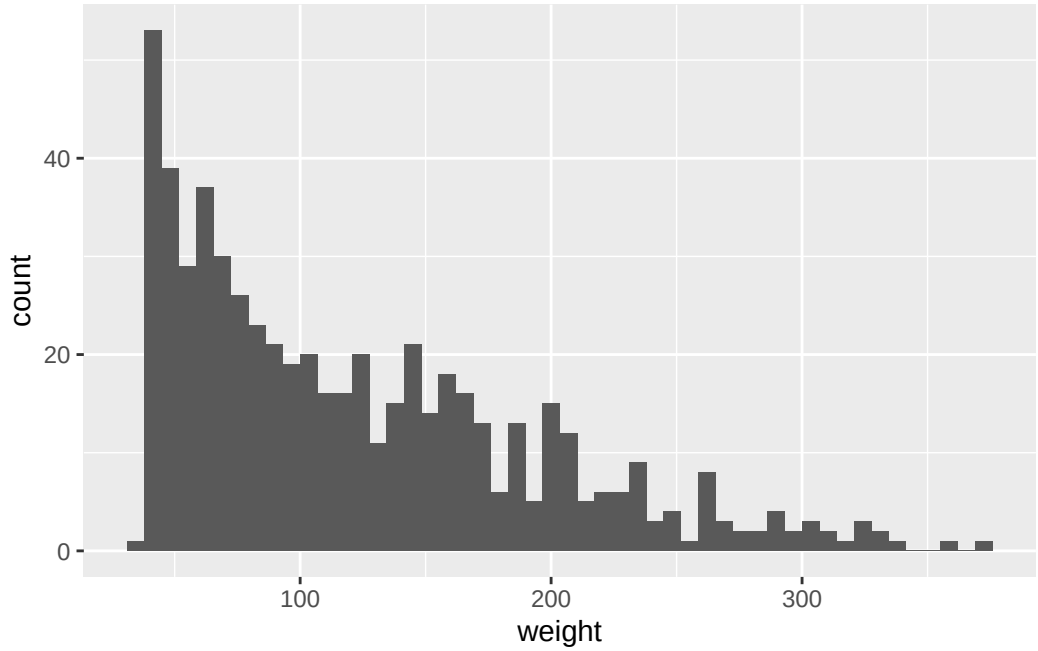
We can see that from the histogram, most chicks in our data weigh around 50-100 grams regardless of time and diet regimen. We can further beautify the histogram by adding a colour, and naming the x-axis and the title. The arguments are similar to the barplot previously.

```
hist(chick_data$weight,
     xlab = "Weight in grams",
     ylab = "Frequency of the chicks",
     main = "Histogram of the weight of the chicks (grams)",
     col = "lightgreen")
```



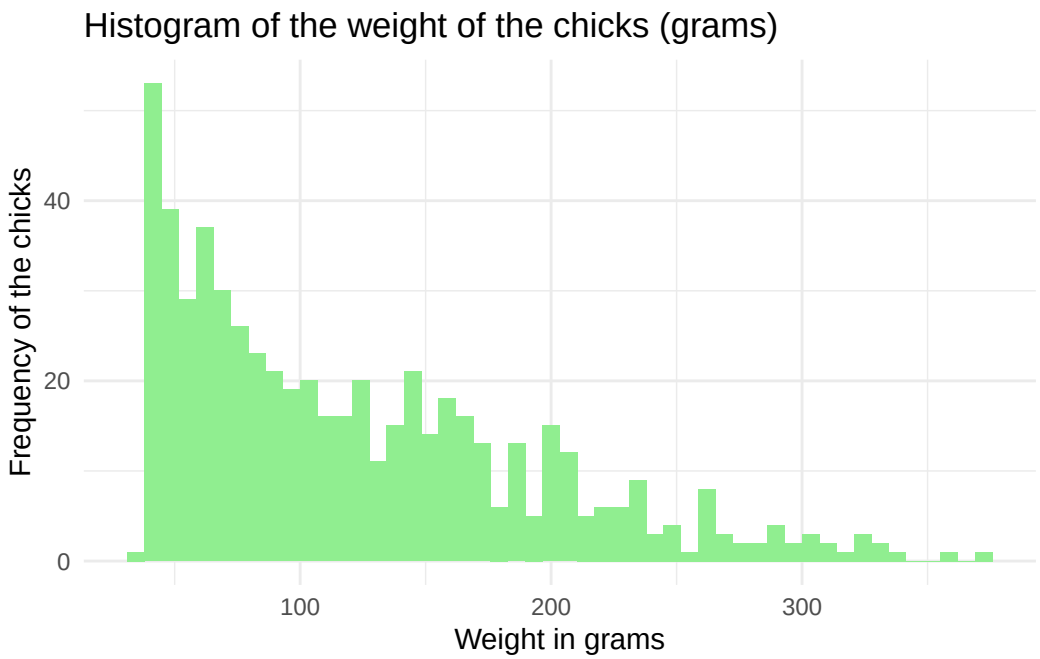
Next, let's do the basic histogram in `ggplot2` using `geom_histogram()`. The `bins` specifies the number of intervals used to group the data. By `bins = 50`, we divide the data into 50 equally spaced intervals or bins.

```
chick_data %>%
  ggplot(aes(x = weight)) +
  geom_histogram(bins = 50)
```



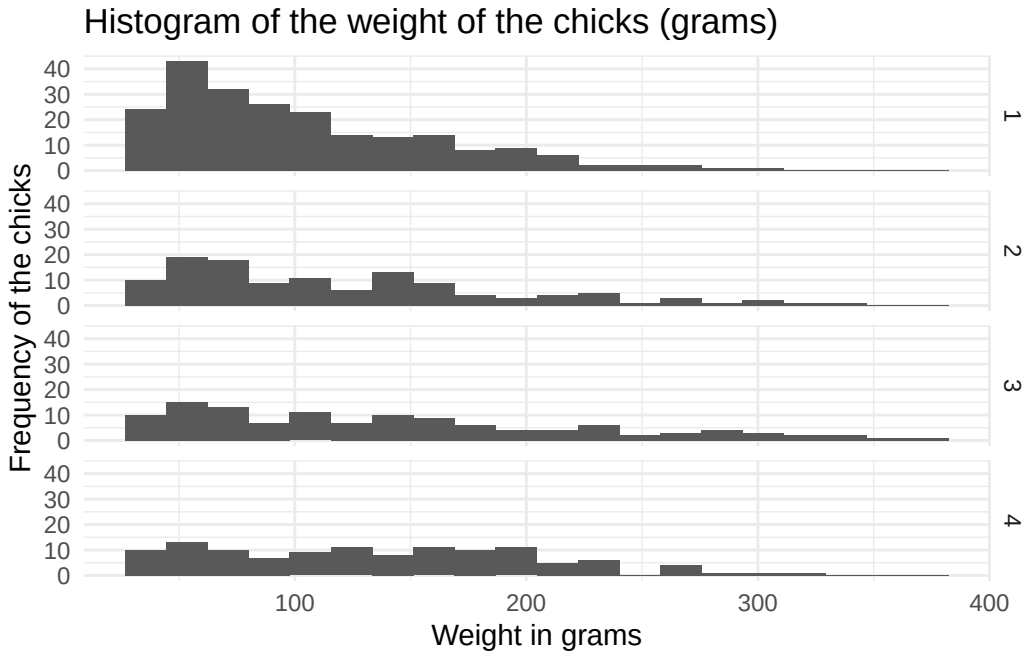
We can add a colour, title, and axis labels, and change the theme.

```
chick_data %>%
  ggplot(aes(x = weight)) +
  geom_histogram(bins = 50, fill = "lightgreen") +
  labs(title = "Histogram of the weight of the chicks (grams)",
       x = "Weight in grams",
       y = "Frequency of the chicks") +
  theme_minimal()
```



We can also add another variable (a factor variable). Let's say we want to see the distribution of the chick's weight across different diet regimens. However, instead of displaying all groups of diet regimens in a single plot, we display it across different plots in rows. `facet_grid()` specifies which column to use to divide across the rows.

```
chick_data %>%
  ggplot(aes(x = weight)) +
  geom_histogram(bins = 20) +
  labs(title = "Histogram of the weight of the chicks (grams)",
       x = "Weight in grams",
       y = "Frequency of the chicks") +
  facet_grid(rows = vars(Diet)) +
  theme_minimal()
```



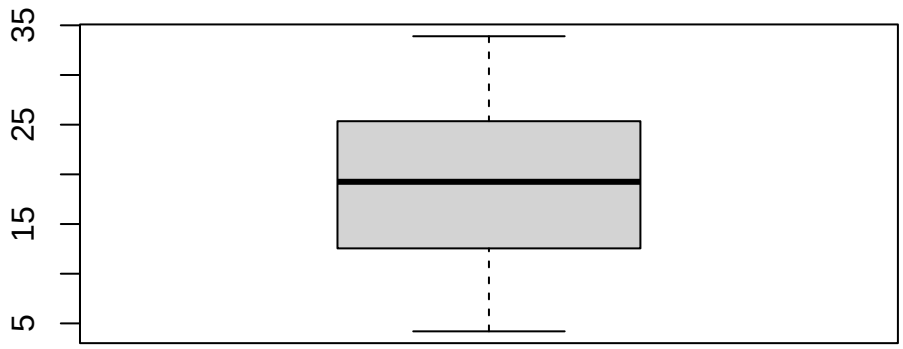
Here, we can see more clearly that the weight of the chicks in diet regimen 1 is skewed to the left side, while the weight of the chicks in diet regimen 4 is distributed more uniformly.

6.5 Boxplot

Boxplot is utilised when a variable is numerical. The main use of the boxplot is to display the spread of the data and further identify outliers and extreme values. Outliers are observations that lie far from the majority of the data. An extreme value may not be an outlier, but an outlier is always an extreme value.

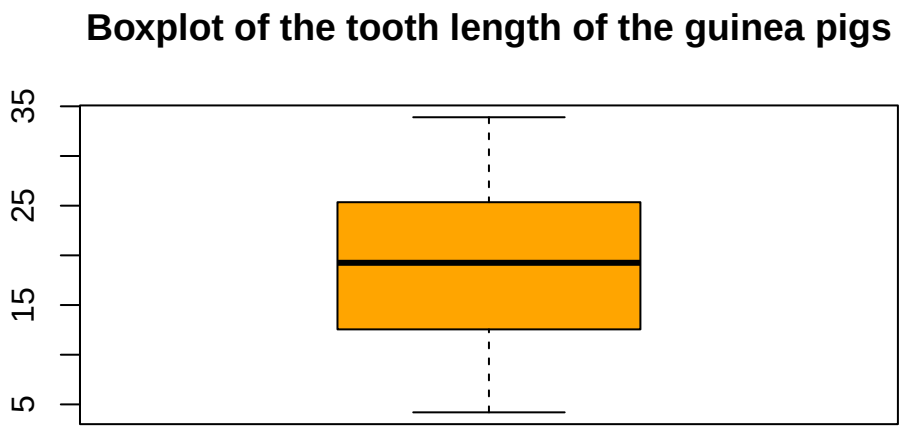
Let's plot the boxplot using base R.

```
boxplot(tooth_data$len)
```



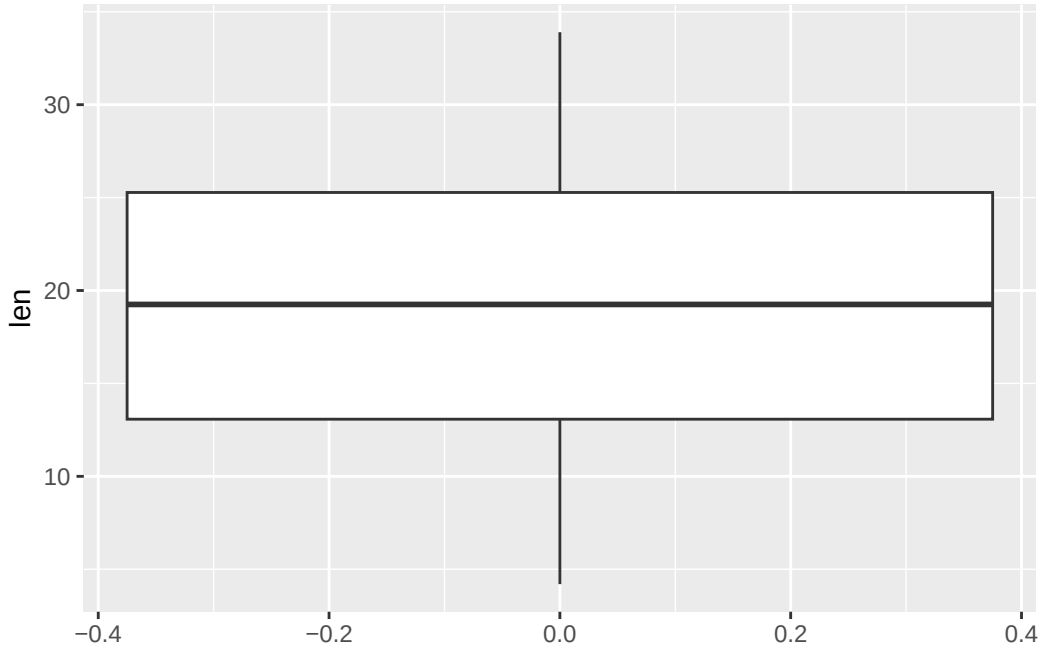
We can further beautify the plot.

```
boxplot(tooth_data$len,
        col = "orange",
        main = "Boxplot of the tooth length of the guinea pigs")
```



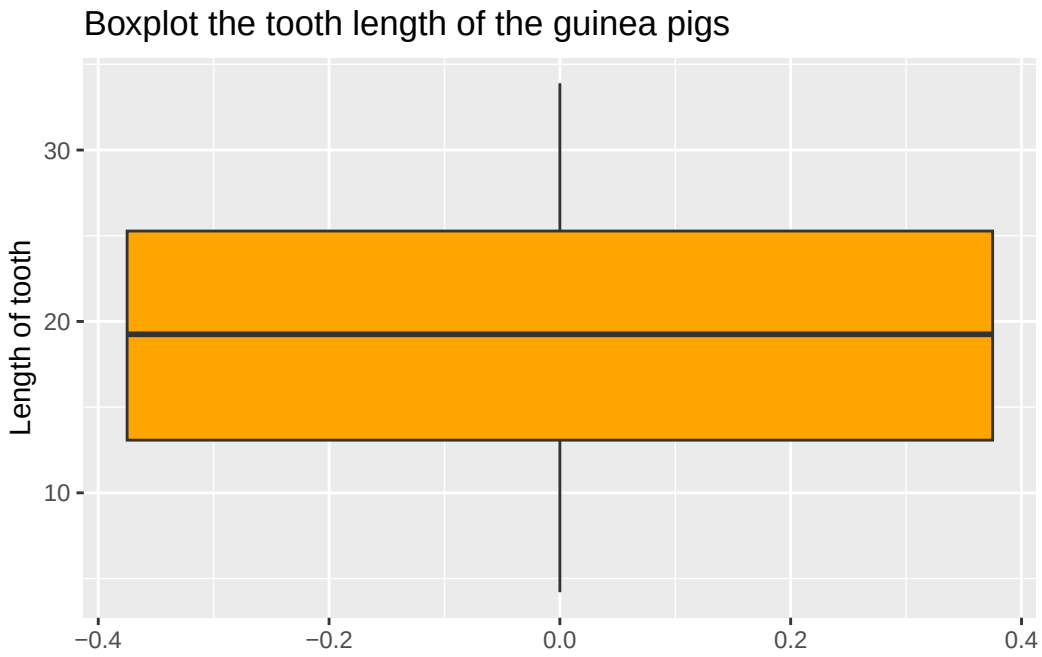
Let's do a similar boxplot using `ggplot2`.

```
tooth_data %>%
  ggplot(aes(y = len)) +
  geom_boxplot()
```



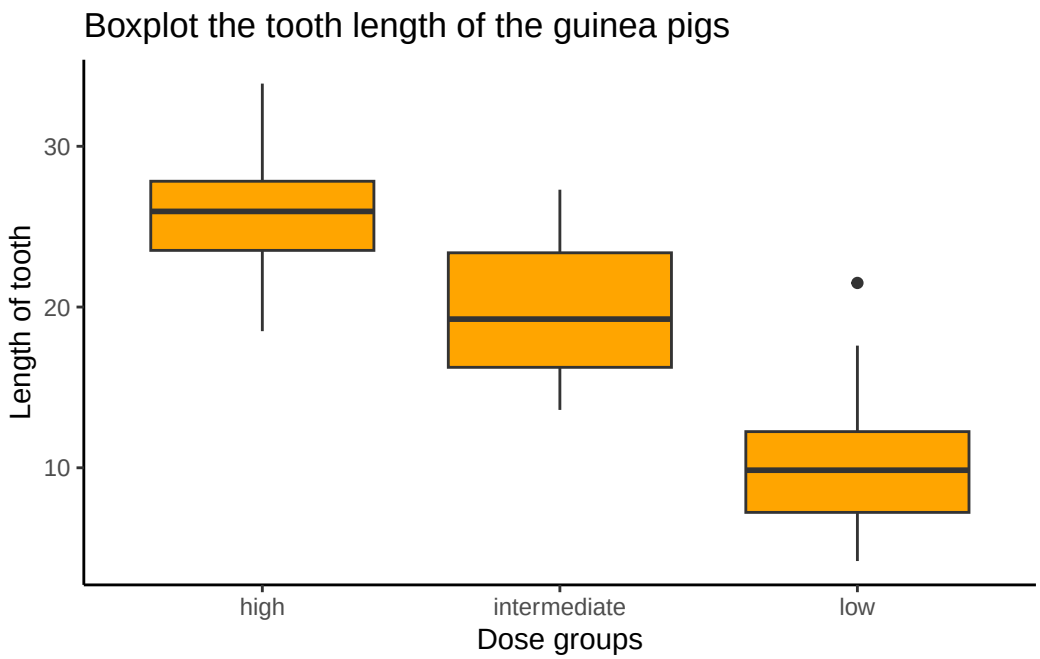
To beautify the plot.

```
tooth_data %>%
  ggplot(aes(y = len)) +
  geom_boxplot(fill = "orange") +
  labs(title = "Boxplot the tooth length of the guinea pigs",
       y = "Length of tooth")
```



We can further divide the boxplots by `dose_group`.

```
tooth_data %>%
  ggplot(aes(x = dose_group, y = len)) +
  geom_boxplot(fill = "orange") +
  labs(title = "Boxplot the tooth length of the guinea pigs",
       y = "Length of tooth",
       x = "Dose groups") +
  theme_classic()
```



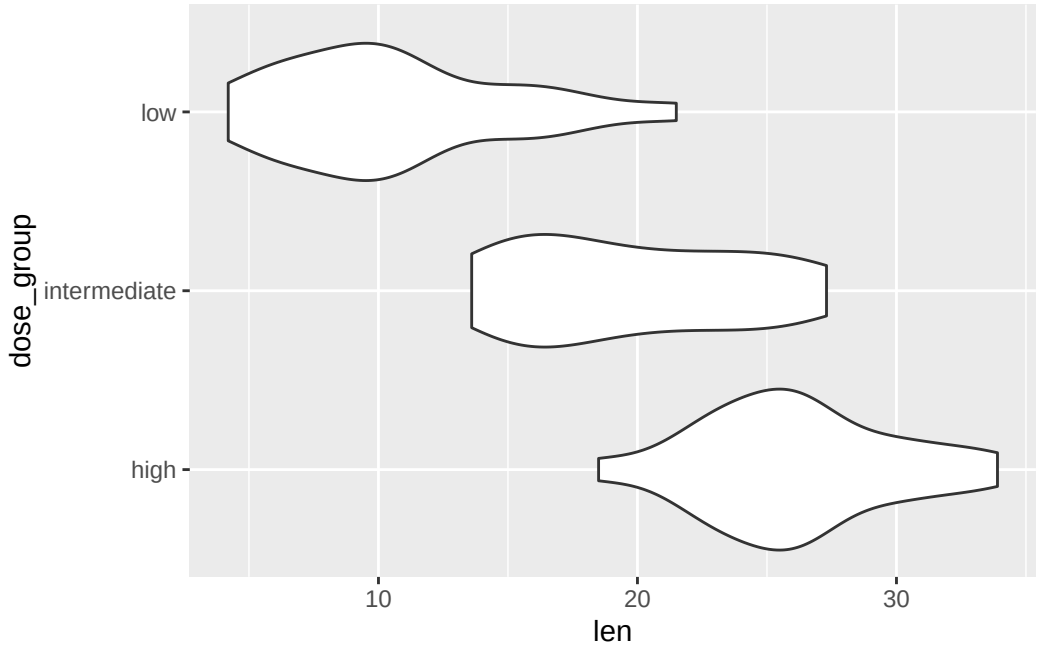
We can see that we may have an outlier in a low-dose group.

6.6 Violin plot

A violin plot allows us to visualise the distribution of the numeric variable according to a factor variable. It is a combination of a boxplot and a density plot. In contrast to the boxplot, the violin plot needs two variables.

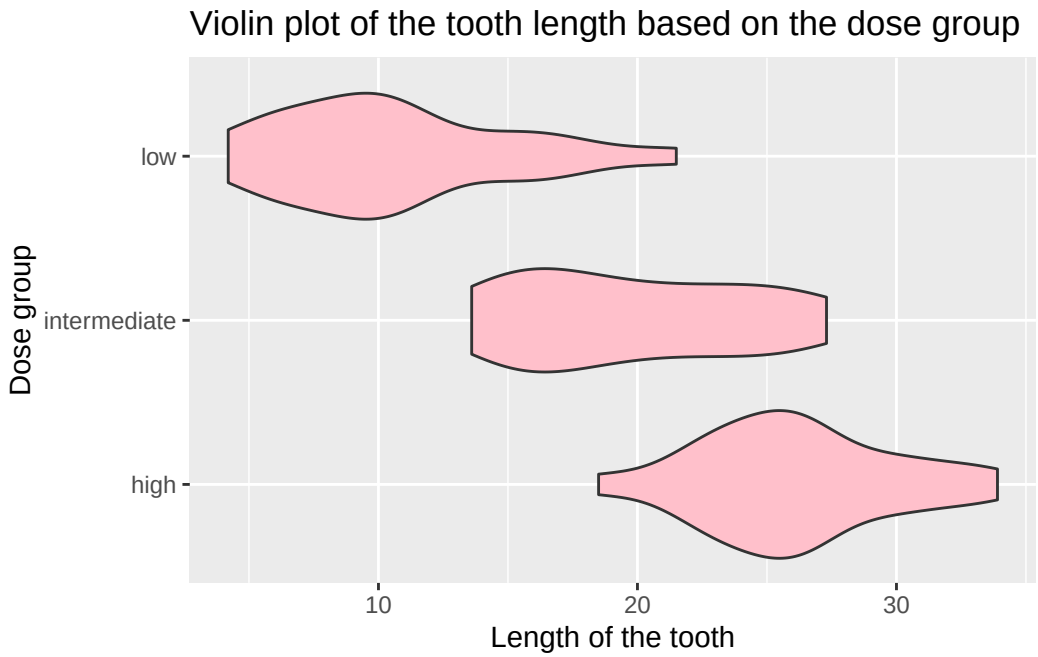
There is no function in base R to plot the violin plot. However, we can use `ggplot2` to plot the violin plot.

```
tooth_data %>%
  ggplot(aes(x = len, y = dose_group)) +
  geom_violin()
```



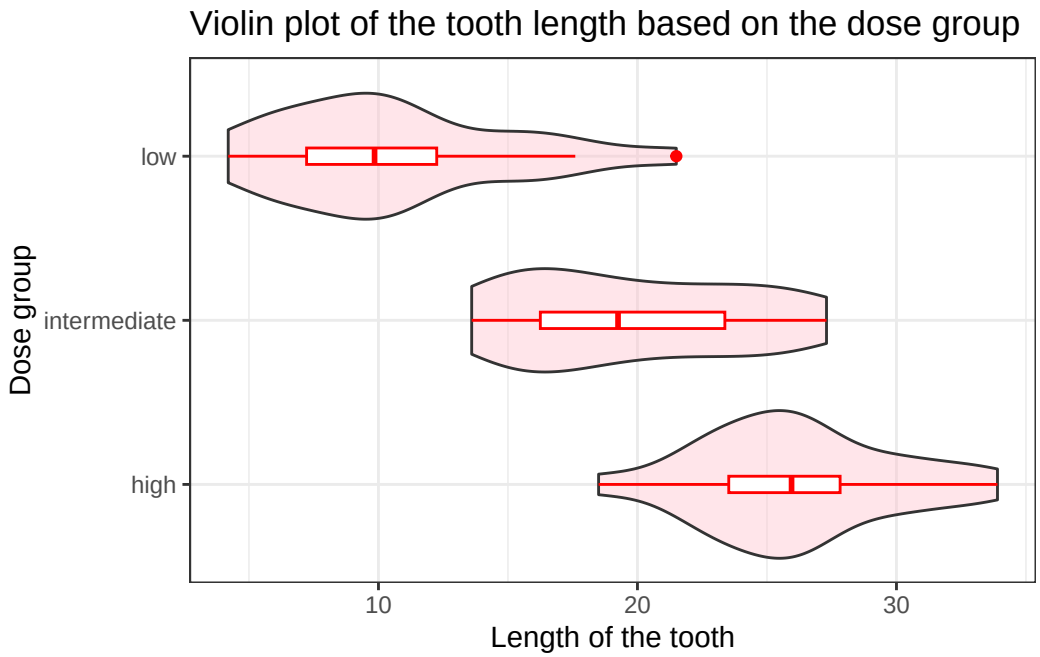
We can further beautify the plot.

```
tooth_data %>%
  ggplot(aes(x = len, y = dose_group)) +
  geom_violin(fill = "pink") +
  labs(title = "Violin plot of the tooth length based on the dose group",
       y = "Dose group",
       x = "Length of the tooth")
```



We can further add the boxplot on top of the violin plot.

```
tooth_data %>%
  ggplot(aes(x = len, y = dose_group)) +
  geom_violin(fill = "pink", alpha = 0.4) +
  geom_boxplot(width = 0.1, color = "red") +
  labs(title = "Violin plot of the tooth length based on the dose group",
       y = "Dose group",
       x = "Length of the tooth") +
  theme_bw()
```



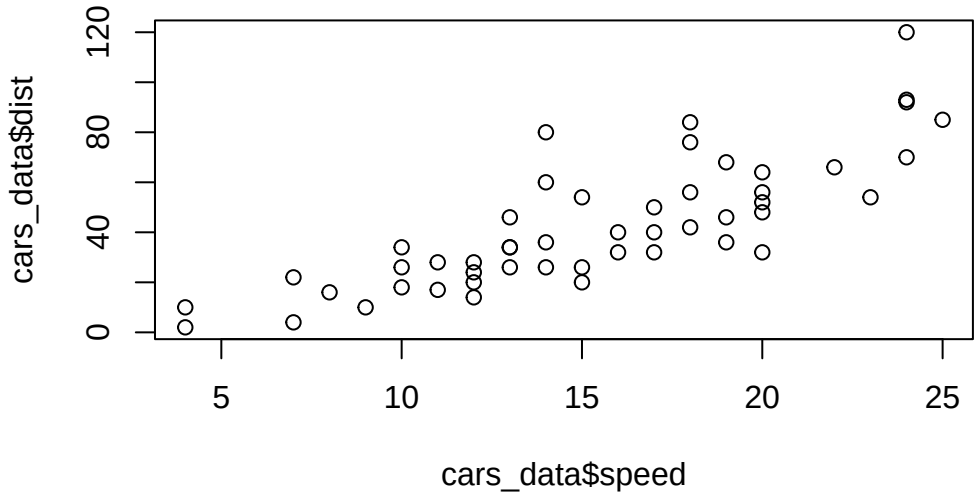
The argument `width` specifies the width of the boxplot and `alpha` specifies the colour density.

6.7 Scatter plot

Scatter plots allow us to plot two continuous variables.

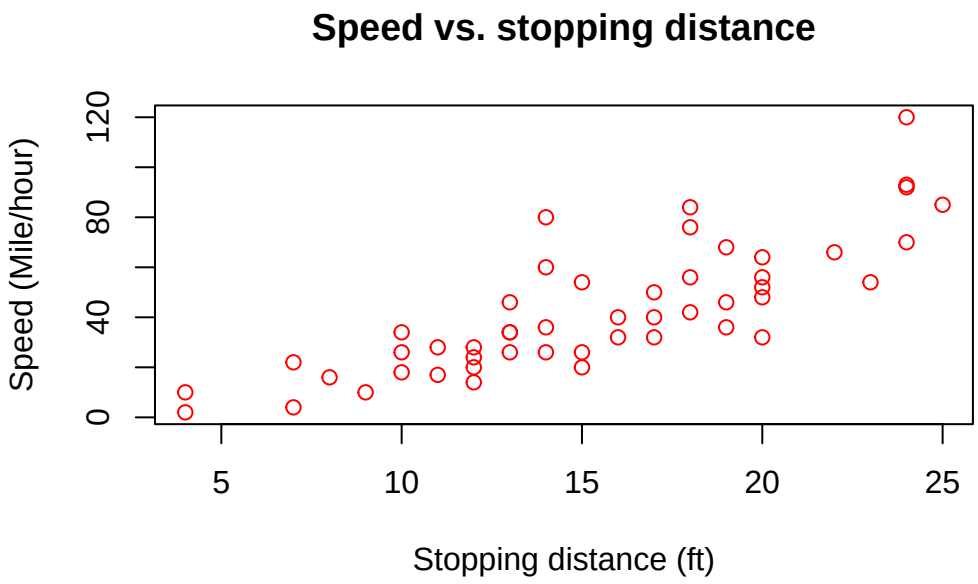
Let's plot a scatter plot using base R. We going to use the last dataset, `cars_data`.

```
plot(x = cars_data$speed, y = cars_data$dist)
```



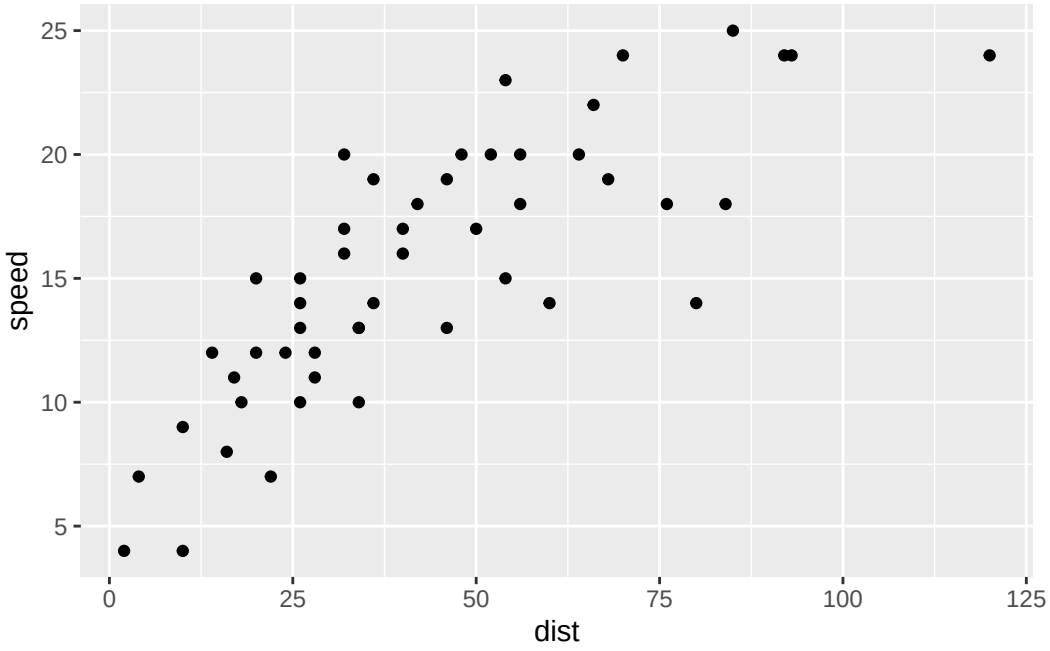
We can see that as the speed increases, the distance taken to stop also increases. Next, to beautify the plot, we name the axis and title, and specify a colour.

```
plot(x = cars_data$speed,
     y = cars_data$dist,
     col = "red",
     main = "Speed vs. stopping distance",
     ylab = "Speed (Mile/hour)",
     xlab = "Stopping distance (ft)")
```



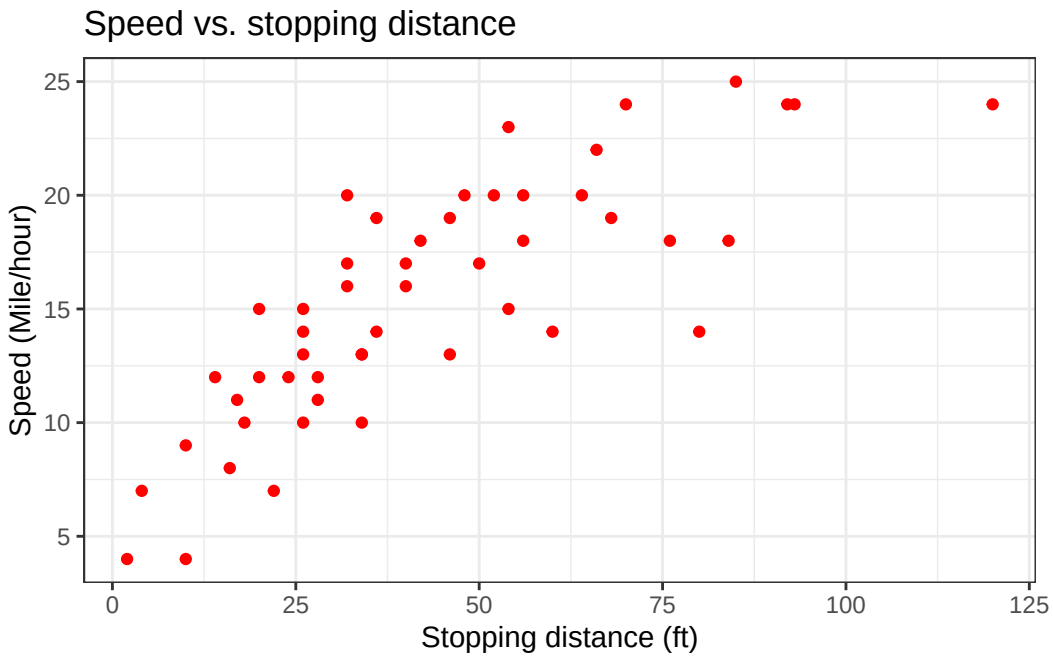
Now, let's do the plot using `ggplot2`.

```
cars_data %>%
  ggplot(aes(x = dist, y = speed)) +
  geom_point()
```



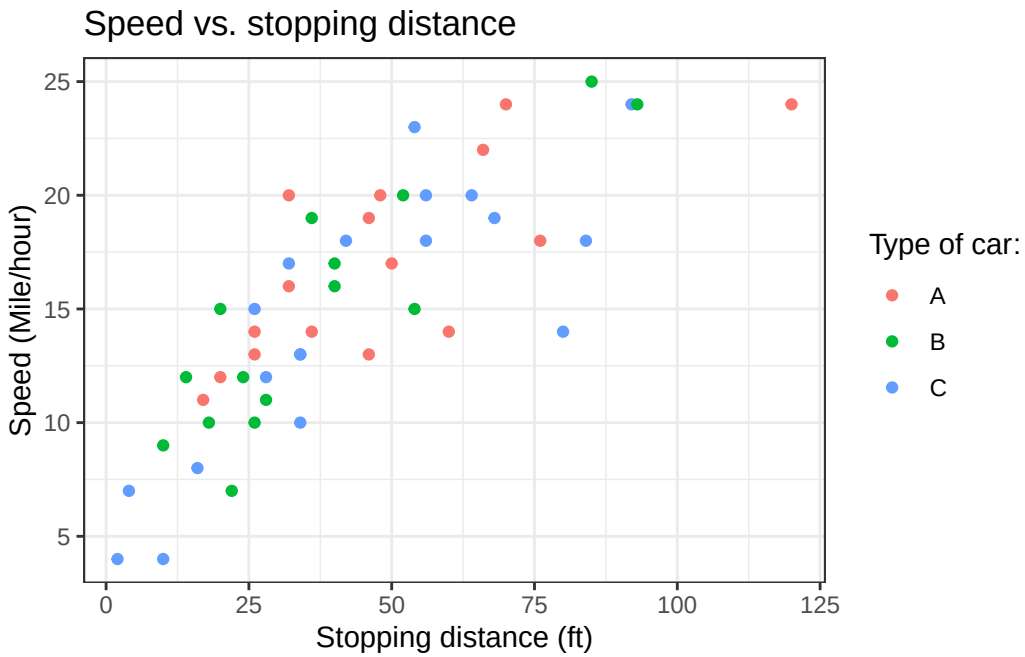
Similarly to the plot in base R, we can further beautify the scatter plot.

```
cars_data %>%
  ggplot(aes(x = dist, y = speed)) +
  geom_point(colour = "red") +
  labs(title = "Speed vs. stopping distance",
       y = "Speed (Mile/hour)",
       x = "Stopping distance (ft)") +
  theme_bw()
```



Lastly, we further add another variable.

```
cars_data %>%
  ggplot(aes(x = dist, y = speed, colour = car)) +
  geom_point() +
  labs(title = "Speed vs. stopping distance",
       y = "Speed (Mile/hour)",
       x = "Stopping distance (ft)",
       colour = "Type of car:") +
  theme_bw()
```



6.8 Chapter summary

In this chapter, we have learned about five different plots.

Table 6.1: Commonly used plots in data analysis.

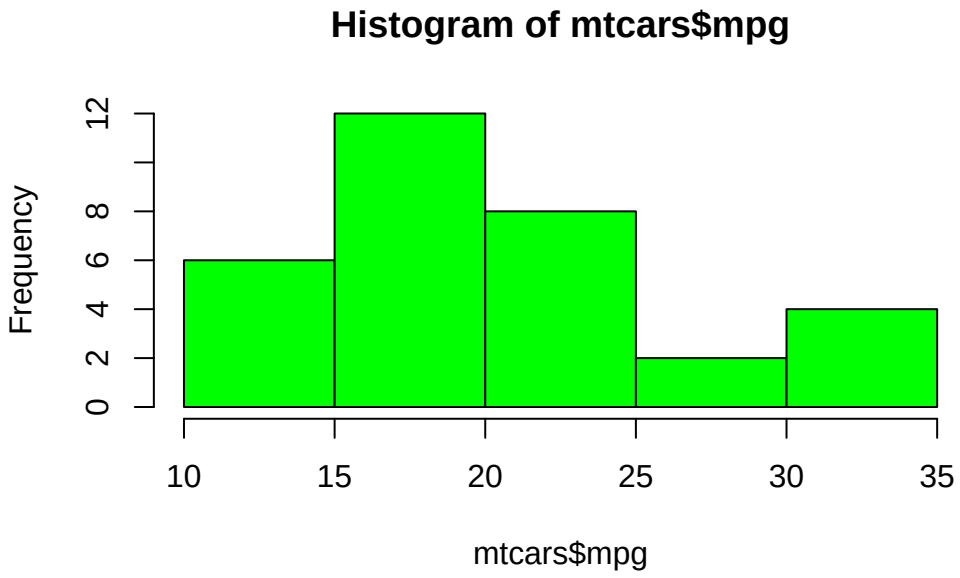
Plots	Variables	Application
Barplot	1 factor variable	Display the proportion of a factor variable
Histogram	1 numerical variable	• Display the frequency distribution of a numerical variable • Identify the pattern of the data (skewed or normally distributed)
Boxplot	1 numerical variable	• Display the spread of distribution of a numerical variable • Identify extreme values and outliers
Violin plot	1 numerical variable vs. 1 factor variable	Display the frequency distribution of a numerical variable based on another factor variable
Scatter plot	1 numerical variable vs. 1 numerical variable	Display a relationship between two numerical variables

6.9 Revision

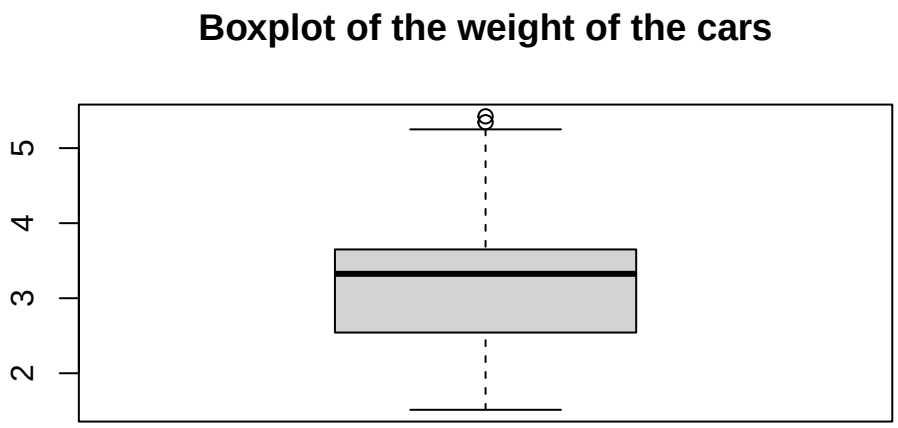
1. Load and read about `mtcars` dataset from base R.

```
# Load the data
data("mtcars")

# Read about the data
?mtcars
```

   a. Create a histogram using base R for the `mpg` variable using base R. The plot should look like the one below.

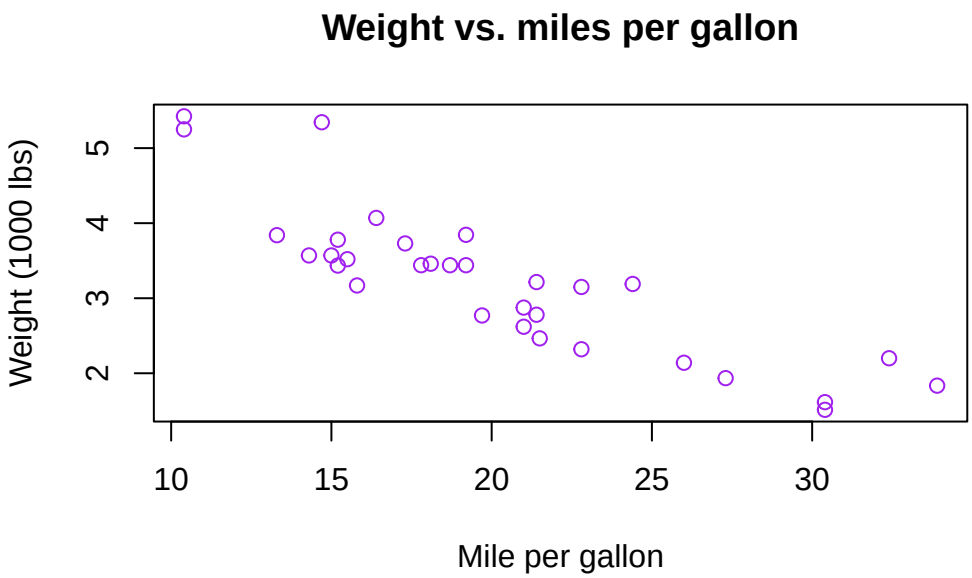


   b. Create a boxplot for the `wt` variable using base R. The plot should look like the one below.



c. Create a scatter plot for `mpg` vs. `wt` variables using base R. The plot should look like the one below.

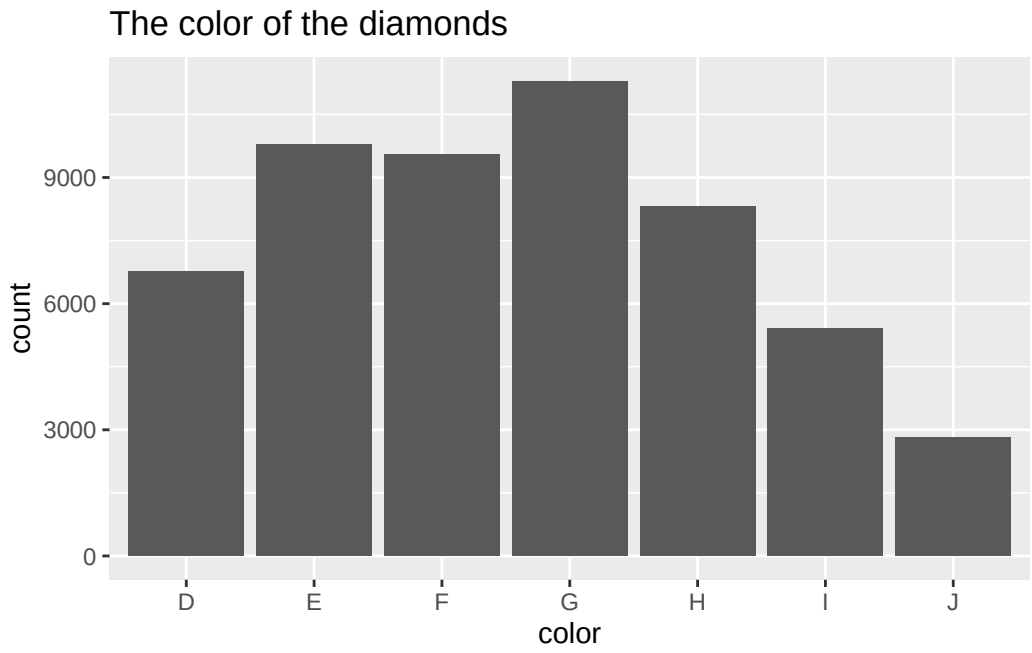
```
plot(mtcars$mpg, mtcars$wt,
     col = "purple",
     main = "Weight vs. miles per gallon",
     xlab = "Mile per gallon",
     ylab = "Weight (1000 lbs)")
```



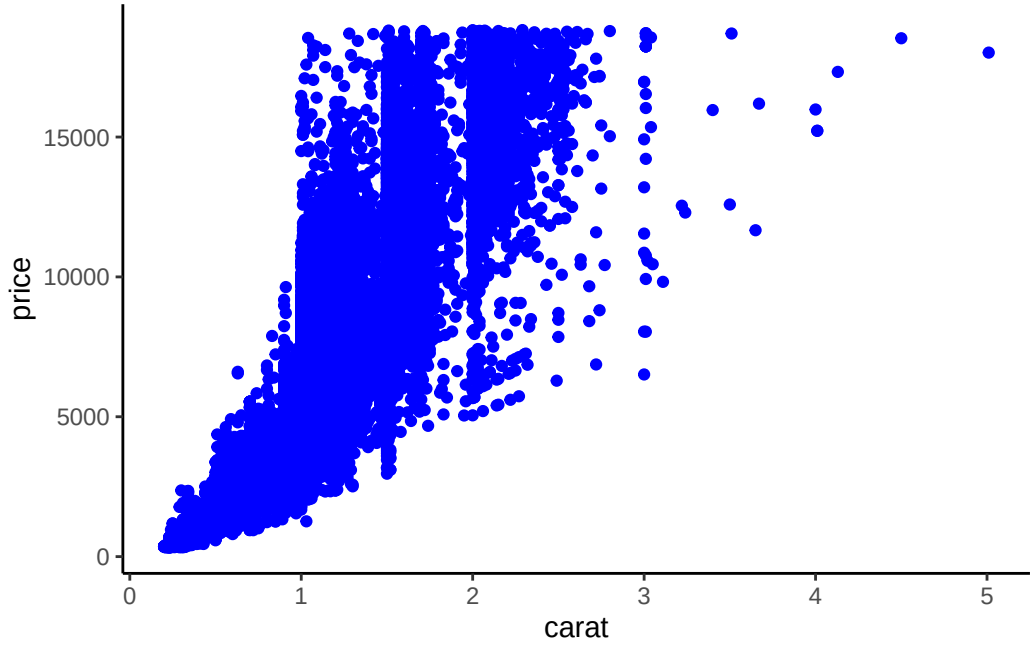
2. Load and read about `diamonds` data from the `ggplot2` package.

```
# Load the packages
library(tidyverse)

# Load the data
data("diamonds")

# Read about the data
?diamonds
```

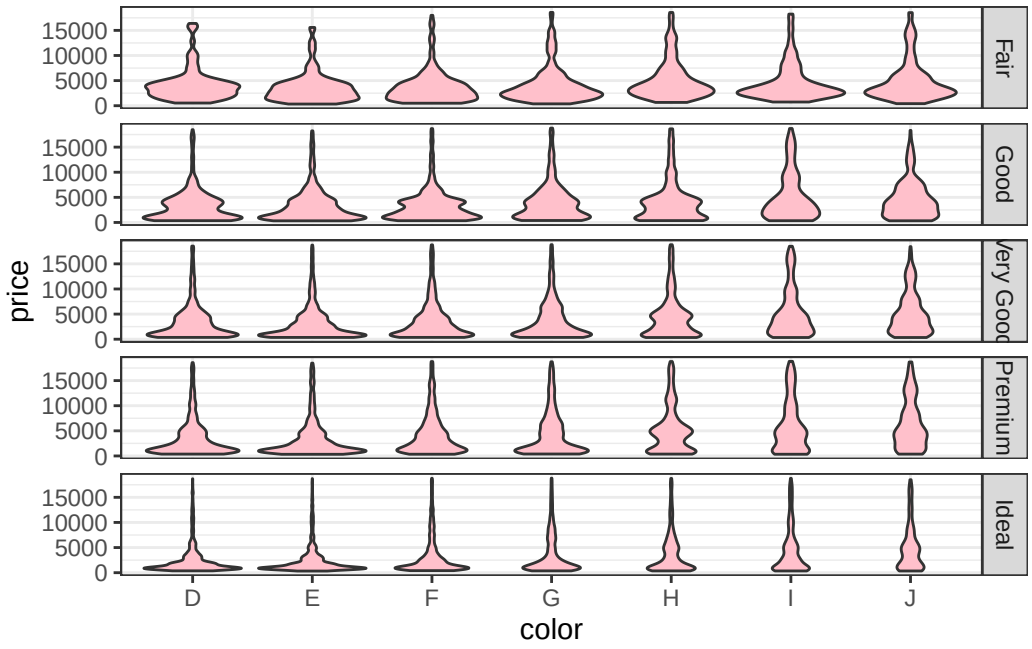
   a. Create a barplot of the `color` variable using `ggplot2`. The plot should look like the one below.



b. Create a scatter plot between `carat` and `price` using `ggplot2`. The plot should look like the one below.



c. Create a violin plot between `price` and `color` and further separate it by `cut` The plot should look like this.


price
15000
10000
5000
0
Fair
Good
Very Good
Premium
Ideal
D
E
F
G
H
I
J
color

7 Efficient coding

> "I think you can have a ridiculously enormous and complex data set, but if you have the right tools and methodology, then it's not a problem."
>
> – Aaron Koblin

This chapter covers how to write a clear, concise, and optimised code to handle data and computations effectively in R. We going to cover what is a loop, how to make a function, and how to use an apply family function. For beginners, this chapter will be quite challenging. So, it is completely understandable if you need to read this chapter several times. In fact, you are free to skip this chapter for the time being, and come back again once you become more familiar with R.

Mastering the topics such as a loop, apply family and functions is crucial especially once you become more advanced in using R. More often than not, you will have a large data and you need to repeat the analysis several times, thus, knowing these topics, makes the R codes more efficient and concise (though not completely readable to beginners).

This chapter does not aim to cover everything on these topics. However, this chapter intends to provide a brief introduction to these topics for beginners and novices in R.

7.1 Load packages

Please load a `tidyverse` package before moving to the next section.

```
library(tidyverse)
```

7.2 Loop

A loop is utilised in the case of an iterative process. Generally, there are two types of loops:

1. For loop
2. While loop

A for loop iterate iterates over a sequence. The general structure of the for loop is:

```
for (variable in sequence) {
    # Code to execute
}
```

For example, if we want to add 1 over a sequence of numbers:

```
# Number sequence 1 to 10
num_seq <- 1:10
num_seq
```

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

```
# Make a for loop that adds 1 to each number in the sequence
for (i in num_seq) {
  i = i + 1
  print(i)
}
```

```
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
[1] 11
```

`i` reflects each number in the number sequence, while `print()` will print each number after 1 is added.

Now, let's try something more related to the data analysis. We going to use the `iris` dataset for this example. For those who might not be familiar, the `iris` dataset, is a built-in dataset in R. Further details can be read on the `Help` pane by typing `?iris`.

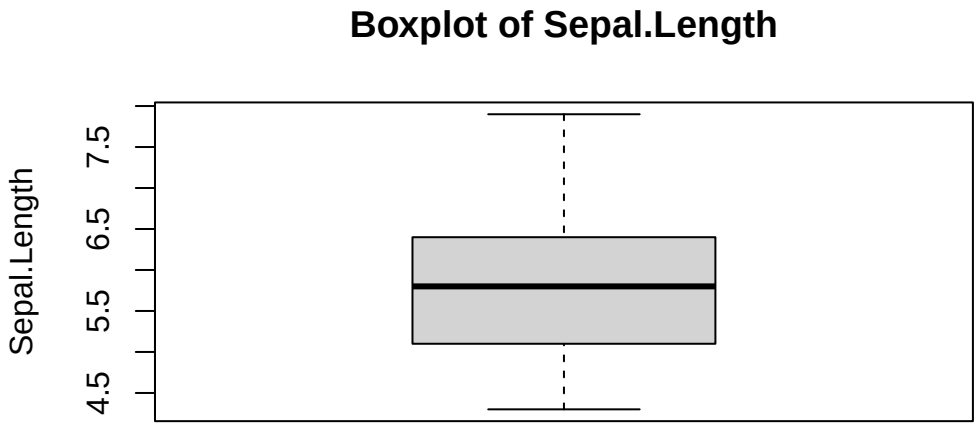
We going to calculate the mean for each numeric column in the `iris` dataset.

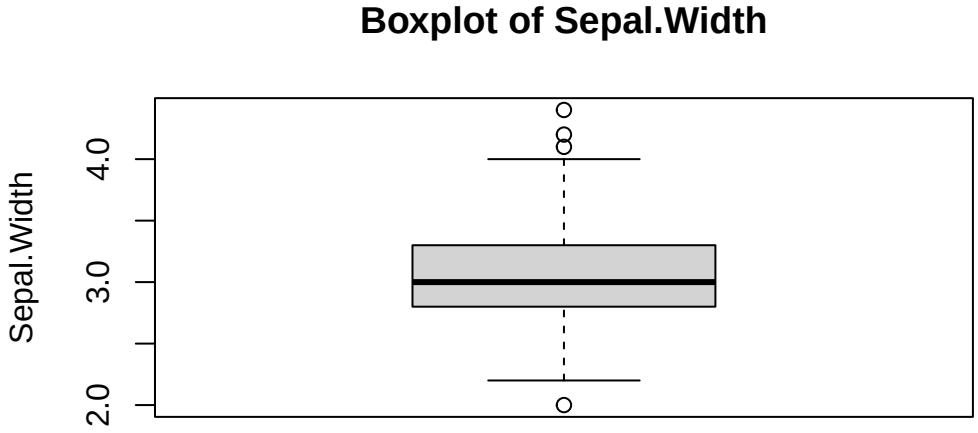
```r
# Loop through numeric columns of the iris dataset
for (col in names(iris)[1:4]) {

    # Calculate the mean of the current column
    column_mean <- mean(iris[[col]])

    # Round the mean to 2 decimal points
    column_mean <- round(column_mean, digits = 2)

    # Print the column name and its mean
    print(paste("Mean of", col, "is", column_mean))
}
```

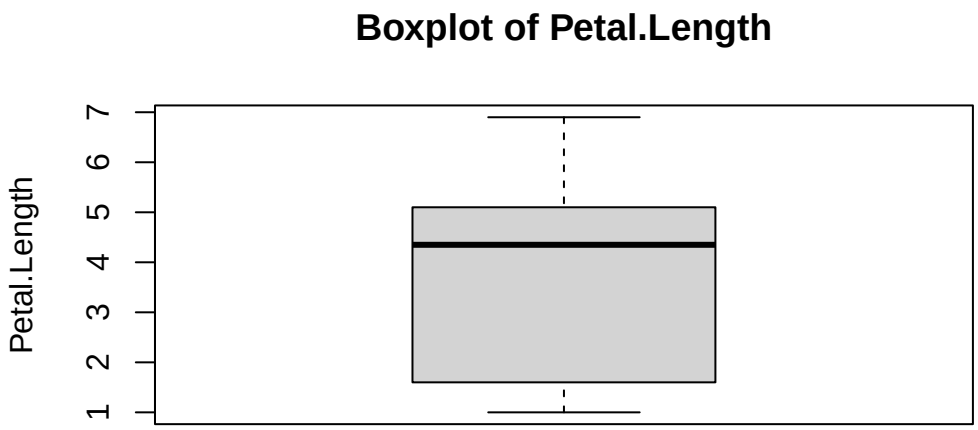
```
[1] "Mean of Sepal.Length is 5.84"
[1] "Mean of Sepal.Width is 3.06"
[1] "Mean of Petal.Length is 3.76"
[1] "Mean of Petal.Width is 1.2"
```

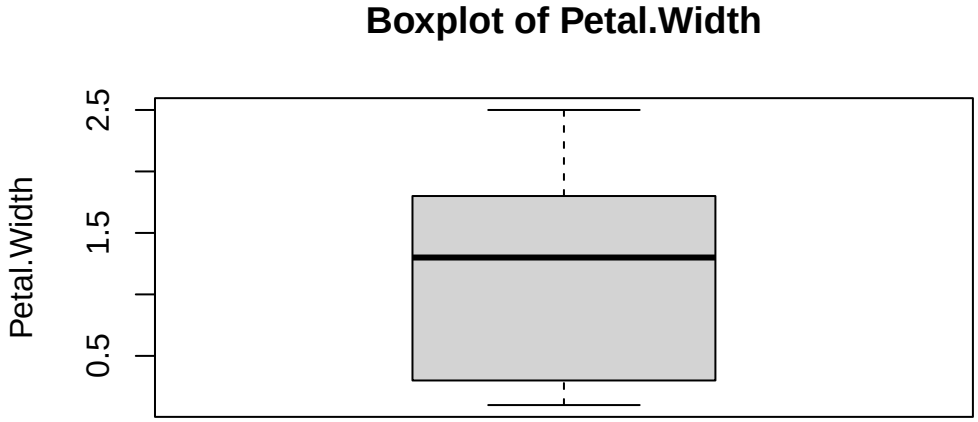
Additionally, we can use for loop to do several plots. For example, we can plot a boxplot for each numeric column in the `iris` dataset.

```r
# Select numeric columns only from iris
df <- iris[, -5]

# Loop through numeric columns of the iris dataset
for (col_name in names(df)) {

  # Create a boxplot
  boxplot(df[[col_name]],
          main = paste("Boxplot of", col_name),
          ylab = col_name)
}
```


Boxplot of Sepal.Length
Sepal.Length
4.5
5.5
6.5
7.5


Boxplot of Sepal.Width
Sepal.Width
2.0
3.0
4.0





Thus, integrating a for loop in the data analysis code will make it more efficient. Next, let's see what is a while loop.

The while loop executes a block of code as long as a specified condition remains true. A general structure of the while loop is:

```
while (condition) {
    # Code to execute
}
```

As a basic example of the while loop, we can add 1 to the sequence of numbers as long as the number is below 10.

```
# Initialise the number
count <- 0

# A while loop that adds 1 to each number as long as the number is less than 10
while (count < 10) {
  count <- count + 1
  print(count)
}
```

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

So, as long as the `count` is not equal to 10, the while loop will keep adding 1 to the `count`. Next, let's try the while loop which is more related to the data analysis. By using the `iris` dataset, let's say we want to calculate the cumulative sum of `Sepal.Length` until the sum exceeds 60.

```
# Initialize variables
index <- 1 #index for looping
cumulative_sum <- 0 #variable to hold the cumulative sum

# While loop to calculate the cumulative sum
while (cumulative_sum <= 60 && index <= nrow(iris)) {

  # Calculate the cumulative sum starting with the 1st row
  cumulative_sum <- cumulative_sum + iris$Sepal.Length[index]
```

```r
  # Move to the next row
  index <- index + 1

  # Display the result
  print(paste0("Row ", index - 1, ", Cumulative sum: ", cumulative_sum))
}
```

```
[1] "Row 1, Cumulative sum: 5.1"
[1] "Row 2, Cumulative sum: 10"
[1] "Row 3, Cumulative sum: 14.7"
[1] "Row 4, Cumulative sum: 19.3"
[1] "Row 5, Cumulative sum: 24.3"
[1] "Row 6, Cumulative sum: 29.7"
[1] "Row 7, Cumulative sum: 34.3"
[1] "Row 8, Cumulative sum: 39.3"
[1] "Row 9, Cumulative sum: 43.7"
[1] "Row 10, Cumulative sum: 48.6"
[1] "Row 11, Cumulative sum: 54"
[1] "Row 12, Cumulative sum: 58.8"
[1] "Row 13, Cumulative sum: 63.6"
```

The conditions in the while loop above are:

1. The loop will run until the cumulative sum exceeds 60
2. Or the last row of the dataset is reached.

The last line of the codes is just to print the result. For now, you do not actually need to understand it.

Generally, the for loop is more popular and commonly used compared to the while loop. However, knowing both loops, at least at the basic level will definitely benefit you in future. Additionally, the loop functions are not only available in R but in other programming software such as Python, Julia, MATLAB and Stata.

7.3 Apply

The apply family function in R is utilised to apply a specific operation over elements of data structures such as vectors, matrices, and data frames. This function is relatively similar to the loop function. However, in R, generally, the apply family functions are faster and more efficient compared to the loop functions.

There are 7 types of apply family functions:

1. `apply()`: applies a function over rows or columns of a matrix or array.
2. `lapply()`: applies a function to each element of a vector, data frame or list and returns a list.
3. `sapply()`: similar to `lapply()` but tries to simplify the result (e.g., into a vector or matrix).
4. `vapply()`: similar to `sapply()`, but requires specifying the output type.
5. `mapply()`: multivariate version of `sapply()`, applying a function to multiple arguments.
6. `tapply()`: applies a function over subsets of a vector grouped by a factor.
7. `rapply()`: recursive version of `lapply()` for nested lists.

We are not going to cover all the apply family functions in this book, but we going to cover the top three apply family functions (`apply()`, `lapply()`, `sapply()`).

`apply()` function

The `apply()` is used to apply the function over rows or columns. The basic syntax is:

```
apply(X, MARGIN, FUN, ...)
```

The basic arguments that we need to supply:

1. `X`: an array.
2. `MARGIN`: where the function should be applied, 1 is at row, and 2 is at column.
3. `FUN`: the function.

Let's try applying this function to our iris dataset. Let's say we want to calculate the mean for each numeric column in the dataset.

```
apply(iris %>% select(-Species), MARGIN = 2 , FUN = mean)
```

```
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
    5.843333     3.057333     3.758000     1.199333
```

`MARGIN = 2` indicates we want the mean of columns. If we set `MARGIN = 1` means that we want the mean of each row.

`lapply()` function

The `lapply()` applies a function to each element of a list and returns a list. The basic syntax is:

```
lapply(X, FUN)
```

Where:

1. `X`: a vector, data frame or a list.
2. `FUN`: the function.

As an example of `lapply()`, we can find the mean of each column and see the result is returned in a list format.

```
lapply(iris %>% select(-Species), FUN = mean)
```

```
$Sepal.Length
[1] 5.843333

$Sepal.Width
[1] 3.057333

$Petal.Length
[1] 3.758

$Petal.Width
[1] 1.199333
```

`sapply()` function

The `sapply()` function is similar to the `lapply()`, but it simplifies the result. The syntax is similar to the `lapply()`.

Let's use the same example as in the `lapply()` section and see how is the output is formatted.

```
sapply(iris %>% select(-Species), FUN = mean)
```

```
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
    5.843333     3.057333     3.758000     1.199333
```

The output is formatted in a more simplified, and in this case, the output format is similar to the one in the `apply()` function section.

So, we have seen three functions from the apply family functions. As the example in the for loop, we can also use the apply family function to plot several plots.

7.3.1 purrr

purrr package is part of tidyverse, which contains many functions that are equivalent to base R apply family functions.

The equivalent of lapply() function is map().

```
map(iris %>% select(-Species), .f = mean)
```

```
$Sepal.Length
[1] 5.843333

$Sepal.Width
[1] 3.057333

$Petal.Length
[1] 3.758

$Petal.Width
[1] 1.199333
```

There are many more functions in the purrr package that are highly beneficial to learn. However, most of these functions are more advanced and require a deeper understanding of R, which goes beyond what is covered in this book. As such, they may not be suitable for beginners and novices at this stage.

7.4 Function

One of the flexibility in R is we can make our own function. Let's make a basic function that adds two numbers.

```
add_num <- function(number1, number2) {
  result <- number1 + number2
  return(result)
}
```

Next, let's try the function.

```
add_num(number1 = 100, number2 = 200)
```

```
[1] 300
```

Let's upgrade our function, instead of adding the number, we going to calculate the mean of the numbers.

```
avg_num <- function(x_range) {

  # Sum of all elements in the vector
  sum_x <- sum(x_range)

  # Determine the number of elements in the vector
  n <- length(x_range)

  # Calculate mean
  mean_value <- sum_x / n

  # Return the mean
  return(mean_value)
}
```

Let's try our function using the `iris` dataset, and compare it with the `mean()` in the base R.

```
# Our function
avg_num(iris$Sepal.Length)
```

```
[1] 5.843333
```

```
# Mean function in R
mean(iris$Sepal.Length)
```

```
[1] 5.843333
```

Additionally, we can also integrate the R functions in our own function. Let's say we want to build a function that can return the value of mean, standard deviation (SD), median, and interquartile range (IQR). Instead of typing the function one by one every time we need it, we can create a function that gives us the four statistical measures. So, it is more efficient to create this function if we need to run it more than two times. For those who are not familiar with these statistical measures, you can just ignore them for now as we will cover it in the later chapter.

```r
summary_func <- function(x_range) {

  # Calculate mean and standard deviation, then round the decimal points to 2
  mean_val <- mean(x_range) |> round(digits = 2)
  std_val <- sd(x_range) |> round(digits = 2)

  # Calculate median and interquartile range, then round the decimal points to 2
  med_val <- median(x_range) |> round(digits = 2)
  iqr_val <- IQR(x_range) |> round(digits = 2)

  # Display the result
  return(c(
    `Mean (SD)` = paste0(mean_val, " (", std_val, ")"),
    `Median (IQR)` = paste0(med_val, " (", iqr_val, ")")
  ))
}
```

Now, we have the function ready. Let's test out on the iris dataset.

```r
summary_func(iris$Sepal.Length)
```

```
    Mean (SD)  Median (IQR)
"5.84 (0.83)"   "5.8 (1.3)"
```

Perfect! Now, every time we want to calculate the mean, SD, median, and IQR for any column, we can just call our function.

To be more efficient we can combine our function with loop or apply family functions that we have learnt in the previous sections. Instead of running `summary_func()` one by one for each column in the `iris` dataset, we integrate it in the for loop.

```r
# Loop through numeric columns of the iris dataset
for (col in names(iris)[1:4]) {

    # Calculate mean of the current column
    res <- summary_func(iris[[col]])

    # Print the result
    print(res)
}
```

```
    Mean (SD)  Median (IQR)
"5.84 (0.83)"   "5.8 (1.3)"
    Mean (SD)  Median (IQR)
"3.06 (0.44)"     "3 (0.5)"
    Mean (SD)  Median (IQR)
"3.76 (1.77)"  "4.35 (3.5)"
   Mean (SD) Median (IQR)
"1.2 (0.76)"  "1.3 (1.5)"
```

In R, it is more efficient to use the apply family functions. Let's use `sapply()` for this.

```
sapply(iris |> select(-Species), FUN = summary_func)
```

```
             Sepal.Length  Sepal.Width   Petal.Length  Petal.Width
Mean (SD)    "5.84 (0.83)" "3.06 (0.44)" "3.76 (1.77)" "1.2 (0.76)"
Median (IQR) "5.8 (1.3)"   "3 (0.5)"     "4.35 (3.5)"  "1.3 (1.5)"
```

What we have learned just now, is known as named function. It is probably the most commonly used function.

7.4.1 Anonymous function

There is another type of function in R, known as the anonymous function. It is defined without a name and is often used temporarily.

For example, let's create an anonymous function that squares a number.

```
function(x) { x^2 }
```

So, to use this function, we need to type the whole function again. In contrast, the named function can be called by the name of the function to be reused.

```
function(x) { x^2 }(3)
```

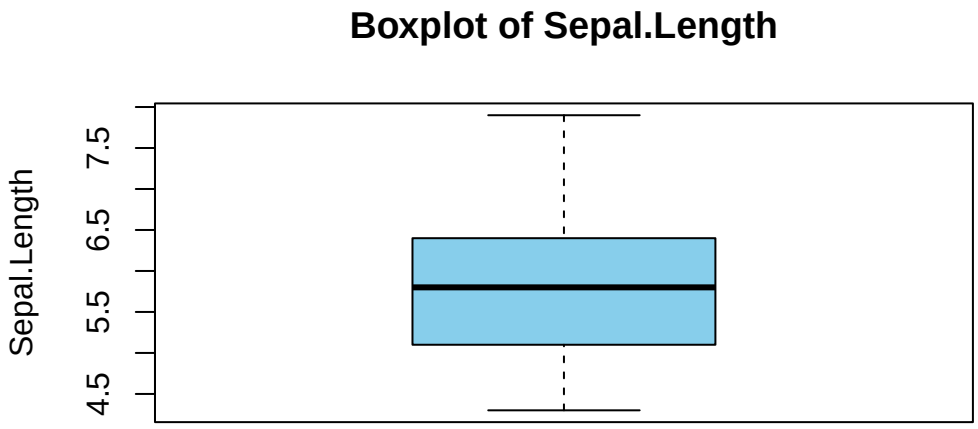
An example of a practical use of the anonymous function is to be used in line with apply family functions. For example, if we want to square the range of numeric values.

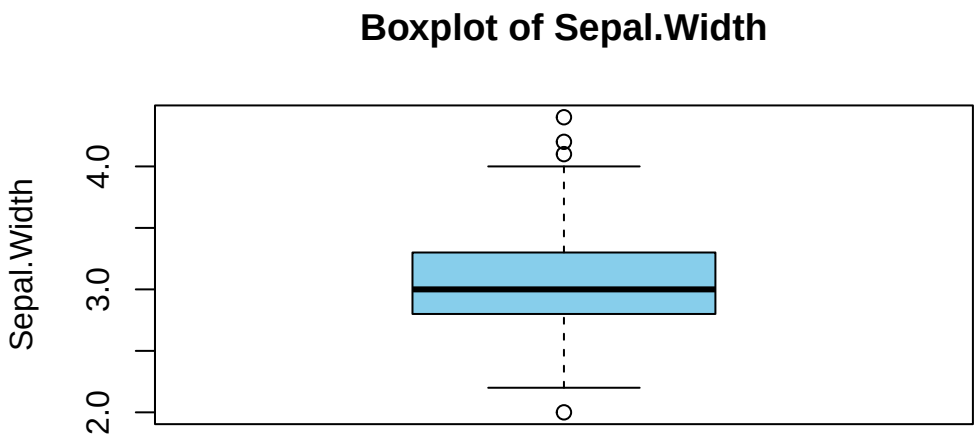
```
sapply(1:10, function(x) x^2)
```

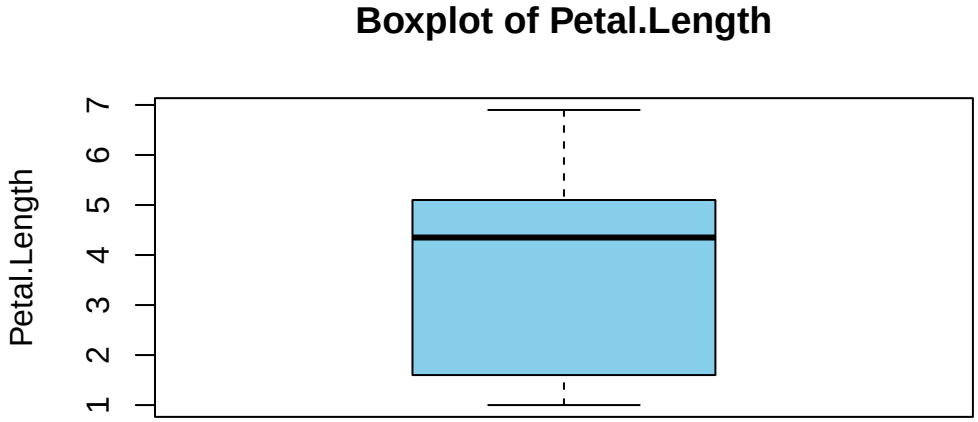
```
 [1]   1   4   9  16  25  36  49  64  81 100
```

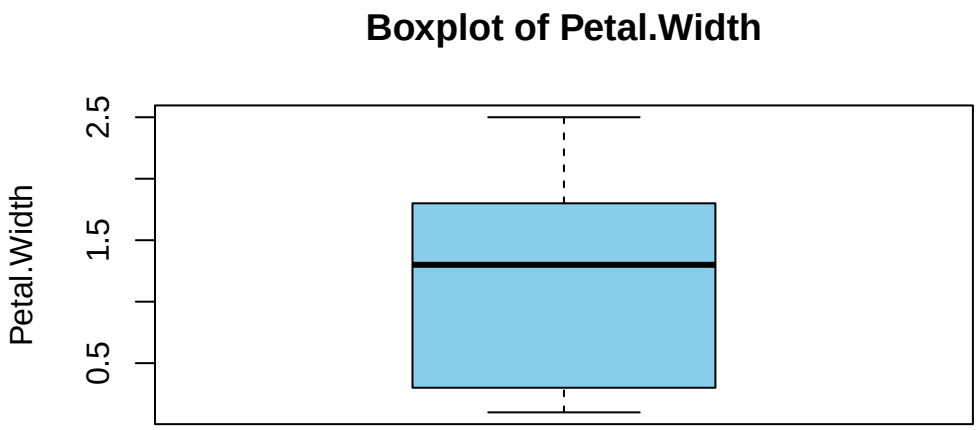
Another example of using anonymous functions with the apply family is to generate multiple plots, similar to the example in the for loop section. In this case, we will use `walk()`, which is the equivalent of the apply family function available in `purrr` package. Let's create a boxplot for each numeric column in the `iris` dataset using `walk()` and anonymous function.

```
# Select numeric columns only from iris
df <- iris[, -5]

# Create boxplots for each numeric column using walk
walk(names(df), function(col_name) {
  boxplot(df[[col_name]],
          main = paste("Boxplot of", col_name),
          ylab = col_name,
          col = "skyblue")
})
```




Boxplot of Sepal.Width
Sepal.Width
2.0
3.0
4.0


Boxplot of Petal.Length
Petal.Length
1
2
3
4
5
6
7



The function `function(col_name)` defines an anonymous function to process each column name in `names(df)`. Using `walk()` allows us to generate multiple boxplots, as it executes the function for its side effects without returning any output. In contrast, using `lapply()` for this task would return additional information, specifically the statistics for each boxplot, alongside creating the plots.

To observe this difference, try running the code below on your machine and compare the behaviour of `walk()` and `lapply()` in this context.

```
# Select numeric columns only from iris
df <- iris[, -5]

# Create boxplots for each numeric column using lapply
lapply(names(df), function(col_name) {
  boxplot(df[[col_name]],
          main = paste("Boxplot of", col_name),
          ylab = col_name,
          col = "skyblue")
})
```

7.5 Chapter summary

We have covered adequately about the loop, apply family, and function.

Table 7.1: Brief summary of loop, apply family, and function.

Concept	Brief Description	When to Use
Loop	Iterative control structures (e.g., `for`, `while`) used to repeat a block of code for a set number of iterations or conditions.	When flexibility and customization are needed for tasks that may not easily fit into vectorized operations.
Apply Family	A group of vectorized functions (`apply`, `lapply`, `sapply`, `mapply`, etc.) that simplify applying functions to data structures.	When performing repetitive operations over data (e.g., rows, columns, or lists) without writing explicit loops.
Function	A reusable block of code that performs a specific task. Functions can be named or anonymous (e.g., `function(x) x^2`).	Use when you need modular, repeatable tasks or computations that are applied across datasets.

For readers seeking a greater challenge, I recommend reading the second edition of *R for Data Science* book, particularly Chapter 25 on functions and Chapter 27 on loops and the apply family (Wickham, Çetinkaya-Rundel, and Grolemund 2023). Additionally, I suggest exploring Chapter 21 of the first edition of *R for Data Science*, as it covers the loops and apply family more extensively (Wickham and Grolemund 2017).

7.6 Revision

1. Using `USJudgeRatings` dataset, a built-in dataset in R, write a for loop that calculates the median for the first two columns and prints the result. Ensure the medians are rounded to two decimal places.

```
data("USJudgeRatings")
```

2. Write a for loop that generates histograms for each numeric column in the `iris` dataset. Customize each histogram to include a title and an appropriate x-axis label.

3. Analyse the following code and explain why it does not produce the expected output. Correct the code to ensure it calculates the sum of the first 5 rows of the `Sepal.Width` column in the `iris` dataset:

```
count <- 0
for (i in 1:5) {
    count = count + iris$Sepal.Width
}
print(count)
```

4. Review the following code and identify the issue:

```
apply(iris, MARGIN = 1, FUN = mean)
```

   Why does this produce an error or unexpected output? Correct the code so it calculates the mean of all numeric columns for each row in the `iris` dataset.

5. Rewrite the following base R code using the equivalent `purrr` function:

```
# Load the library
library(dplyr)

# Change the codes below to the equivalent purrr function
lapply(iris %>% select(-Species), mean)
```

6. What are the difference between `lapply()` and `sapply()`.

7. Using the `summary_func()` function in Section 7.4, calculate the statistical summaries (mean, SD, median, IQR) for all numeric columns in the `USJudgeRatings` dataset, a built-in dataset in R. Use the `sapply()` function and for loop implementation.

```
data("USJudgeRatings")
```

8. Create a function called `multiply_num` that takes two arguments and returns their product. Use this function to multiply 10 and 20. The function should result an output as below.

```
multiply_num(10, 20)
```

```
[1] 200
```

9. Modify the `avg_num()` function in Section 7.4 to include an optional argument `round()` that rounds the mean to a specified number of decimal places. Test it with the `iris$Sepal.Length` column, rounding to 3 decimal places. The function should result an output as below.

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

```
[1] 5.843
```

10. Write an anonymous function that takes a numeric vector and returns a vector of squared values only for numbers greater than 5. Test this using `sapply()` with a sequence from 1 to 10.

8 Data exploration

> "Data is like garbage. You'd better know what you are going to do with it before you collect it."
>
> – Mark Twain

Data exploration, as the name suggests involves the process of exploring the data to understand its structures, identifying underlying patterns, and discovering its characteristics. This process is the first stage in analysing the data and gaining insight from it. Doing data analysis without data exploration is akin to going to war without a proper strategy.

R provides numerous functions and packages that can help to explore the data efficiently and systematically. Thus, this chapter intends to introduce those functions and packages to readers adequately and further equip the readers to do any data analysis.

8.1 Missing data

Missing data are recognised as `NA` in R. Let's use `airquality` data, a built-in dataset in R to see how R recognised missing values.

```
# Data with missing data
data("airquality")
```

By using `summary()`, we can see that the first two columns have missing values, recognised as `NA`.

```
summary(airquality)
```

```
     Ozone           Solar.R          Wind             Temp      
 Min.   :  1.00   Min.   :  7.0   Min.   : 1.700   Min.   :56.00  
 1st Qu.: 18.00   1st Qu.:115.8   1st Qu.: 7.400   1st Qu.:72.00  
 Median : 31.50   Median :205.0   Median : 9.700   Median :79.00  
 Mean   : 42.13   Mean   :185.9   Mean   : 9.958   Mean   :77.88  
 3rd Qu.: 63.25   3rd Qu.:258.8   3rd Qu.:11.500   3rd Qu.:85.00  
 Max.   :168.00   Max.   :334.0   Max.   :20.700   Max.   :97.00  
```

```
 NA's   :37       NA's   :7
     Month            Day
 Min.   :5.000   Min.   : 1.0
 1st Qu.:6.000   1st Qu.: 8.0
 Median :7.000   Median :16.0
 Mean   :6.993   Mean   :15.8
 3rd Qu.:8.000   3rd Qu.:23.0
 Max.   :9.000   Max.   :31.0
```

Another function to give a quick answer to whether we have available data or not is `anyNA()`.

```
anyNA(airquality)
```

```
[1] TRUE
```

To get the count of missing values, we can use `is.na()`:

```
is.na(airquality) |> table()
```

```
FALSE  TRUE
  874    44
```

`TRUE` is 44, which means that we have 44 missing values across individual cells in our dataset. To drop the missing values, we can use `complete.cases()`:

```
# Make an index
na_index <- complete.cases(airquality)

# Apply the index to drop the NA's
airquality_noNA <- airquality[na_index, ]
```

Alternatively, it is easier to use `na.omit()`:

```
airquality_noNA2 <- na.omit(airquality)
```

Both, `complete.cases()` and `na.omit()` return an identical output.

```
# Drop NA's using complete.cases()
dim(airquality_noNA)
```

```
[1] 111   6
```

```
# Drop NA's using na.omit()
dim(airquality_noNA2)
```

```
[1] 111   6
```

We may want to explore the data with missing values. Thus, we can use `complete.cases()` plus `!` in our codes to isolate the data with missing values.

```
# Make an index
na_index <- complete.cases(airquality)

# Apply the index to drop the NA's
airquality_NA_only <- airquality[!na_index, ]

# Data with missing values
dim(airquality_NA_only)
```

```
[1] 42  6
```

Notice that the `airquality_NA_only` returns 44 rows of data with missing values, while using `is.na()` earlier we get 44 missing values. `airquality_NA_only` returns rows of data with missing values. The missing values may exist in more than a single column in the dataset. The `is.na()` function counts all the missing values regardless of the position.

8.2 Outliers

An outlier is a data point that lies significantly outside the range of most other observations in a dataset. It is an extreme value, either unusually high or low, that differs markedly from other data points. Outliers can occur due to variability in the data, errors in measurement, or experimental anomalies.

Since outliers may not be representative of the data and may distort statistical measures such as mean and standard deviation, it is important to identify them early during the data exploration stage.

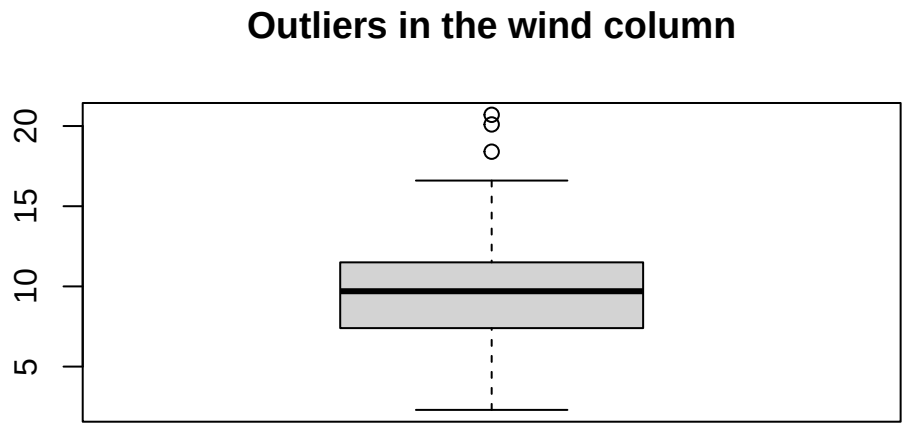
There are two methods to identify outliers:

1. Interquartile range (IQR) method:

   The IQR method identifies outliers as data points lying below Q1 $-$ 1.5 $\times$ IQR or above Q3 + 1.5 $\times$ IQR , where Q1 and Q3 are the first and third quartiles, respectively. This is the method that applies in the boxplot.

   Let's use `airquality_noNA` data to demonstrate:

```
out_bp <- boxplot(airquality_noNA$Wind, main = "Outliers in the wind column")
```



   To access the values of the outliers:

```
out_bp$out
```

```
[1] 20.1 18.4 20.7
```

   To see the rows with the outliers:

```
airquality_noNA |>
  dplyr::filter(Wind == c(20.1, 18.4, 20.7))
```

```
  Ozone Solar.R Wind Temp Month Day
1     8      19 20.1   61     5   9
2     6      78 18.4   57     5  18
3    37     284 20.7   72     6  17
```

2. Z-score method:

   This method standardizes data and identifies outliers as those with a z-score greater than a threshold (commonly 3 or -3). The z-score (also called a standard score) is a statistical measure that indicates how many standard deviations a data point is from the mean of the dataset.

   Basically, the z-score is a distribution which reflects our dataset and it shows how far each value is from the average. So, in this context, outliers are the values that are significantly far from the average.

   Now, let's focus on understanding how we can detect the outliers using this method.

```
# Create z-score for the data
z_scores <- scale(airquality_noNA$Wind)

# How many outliers
table(abs(z_scores) > 3)
```

```
FALSE  TRUE
  110     1
```

   `TRUE` is 1, which means that we have a single outlier in our data. Let's see which value is the outlier.

```
airquality_noNA$Wind[abs(z_scores) > 3]
```

```
[1] 20.7
```

   To see the row with the outlier:

```
airquality_noNA |>
  dplyr::filter(Wind == 20.7)
```

```
  Ozone Solar.R Wind Temp Month Day
1    37     284 20.7   72     6  17
```

There are other advanced methods to detect outliers. However, the basics to understand those methods are beyond the scope of this book.

8.3 Useful packages

In this section, we going to see several useful R packages to explore the data. We are only going to cover the main functions of each package.

8.3.1 skimr

The `skimr` package provides various helpful functions to do data exploration. Firstly, we need to install `skimr`, and then load the packages. We going to use the `dplyr` package together with the `skimr` package.

```
# Install the package
install.packages("skimr")

# Load the necessary packages
library(skimr)
library(dplyr)
```

To explore the whole dataset, we can use `skim`.

```
# Let's use the iris dataset
data("iris")

# Use skim
skim(iris)
```

Table 8.1: Data summary

Name	iris
Number of rows	150
Number of columns	5
_____	
Column type frequency:	
factor	1
numeric	4
_____	
Group variables	None

Variable type: factor

skim_variable	n_missing	complete_rate	ordered	n_unique	top_counts
Species	0	1	FALSE	3	set: 50, ver: 50, vir: 50

Variable type: numeric

skim_variable	n_missing	complete_rate	mean	sd	p0	p25	p50	p75	p100	hist
Sepal.Length	0	1	5.84	0.83	4.3	5.1	5.80	6.4	7.9	
Sepal.Width	0	1	3.06	0.44	2.0	2.8	3.00	3.3	4.4	
Petal.Length	0	1	3.76	1.77	1.0	1.6	4.35	5.1	6.9	
Petal.Width	0	1	1.20	0.76	0.1	0.3	1.30	1.8	2.5	

The `skim` function will return the basic statistics for our data including the information on the missing values (`n_missing` and `complete_rate`) and a histogram for the numerical variables. Additionally, we can get the basic statistics based on a certain group. For example, here we use `group_by()`, then, we apply `skim()`.

```
iris %>%
  group_by(Species) %>%
  skim()
```

Table 8.4: Data summary

Name	Piped data
Number of rows	150
Number of columns	5
_____	
Column type frequency:	
numeric	4
_____	
Group variables	Species

Variable type: numeric

skim_variable	Species	n_missing	complete_rate	mean	sd	p0	p25	p50	p75	p100	hist
Sepal.Length	setosa	0	1	5.01	0.35	4.3	4.80	5.00	5.20	5.8	
Sepal.Length	versicolor	0	1	5.94	0.52	4.9	5.60	5.90	6.30	7.0	
Sepal.Length	virginica	0	1	6.59	0.64	4.9	6.23	6.50	6.90	7.9	
Sepal.Width	setosa	0	1	3.43	0.38	2.3	3.20	3.40	3.68	4.4	
Sepal.Width	versicolor	0	1	2.77	0.31	2.0	2.52	2.80	3.00	3.4	

skim_variable	Species	n_missing	complete_rate	mean	sd	p0	p25	p50	p75	p100	hist
Sepal.Width	virginica	0	1	2.97	0.32	2.2	2.80	3.00	3.18	3.8	
Petal.Length	setosa	0	1	1.46	0.17	1.0	1.40	1.50	1.58	1.9	
Petal.Length	versicolor	0	1	4.26	0.47	3.0	4.00	4.35	4.60	5.1	
Petal.Length	virginica	0	1	5.55	0.55	4.5	5.10	5.55	5.88	6.9	
Petal.Width	setosa	0	1	0.25	0.11	0.1	0.20	0.20	0.30	0.6	
Petal.Width	versicolor	0	1	1.33	0.20	1.0	1.20	1.30	1.50	1.8	
Petal.Width	virginica	0	1	2.03	0.27	1.4	1.80	2.00	2.30	2.5	

Readers interested in learning more about the `skimr` package can explore its comprehensive documentation for more details and examples.

8.3.2 naniar

`naniar` package provides tidy ways to summarise, visualise, and manipulate missing data. First, let's install and load the necessary packages.

```
# Install the package
install.packages("naniar")

# Load the necessary packages
library(naniar)
library(dplyr)
```

Let's use the `oceanbuoys` data, a dataset from the `naniar` package. First, let's summarise the missing data according to a variable. Further detail on the dataset can be found by running `?oceanbuoys` in the `Console`.

```
# Load the data
data("oceanbuoys")

# Missing values summary based on variables
miss_var_summary(oceanbuoys)
```

```
# A tibble: 8 x 3
  variable   n_miss pct_miss
  <chr>       <int>    <num>
1 humidity       93   12.6
2 air_temp_c     81   11.0
3 sea_temp_c      3    0.408
4 year            0    0
5 latitude        0    0
6 longitude       0    0
7 wind_ew         0    0
8 wind_ns         0    0
```

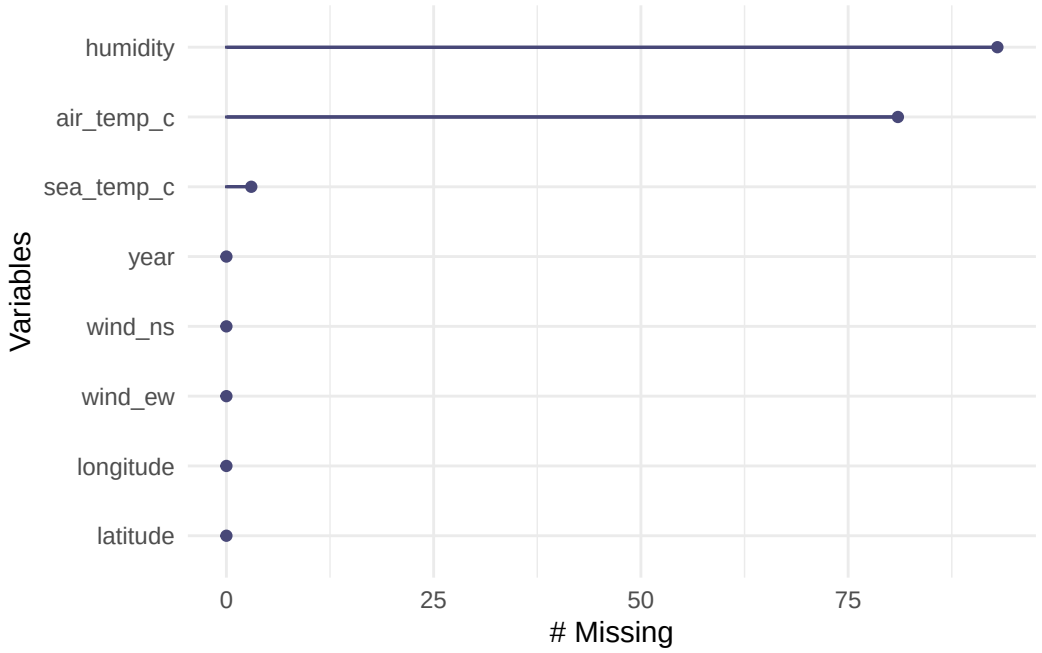
`n_miss` is the number of missing values and `pct_miss` is the percentage of missing values. We can further group the missing values according to a certain variable.

```
oceanbuoys %>%
  group_by(year) %>%
  miss_var_summary()
```

```
# A tibble: 14 x 4
# Groups:   year [2]
    year variable   n_miss pct_miss
   <dbl> <chr>       <int>    <num>
 1  1997 air_temp_c     77   20.9
 2  1997 latitude        0    0
 3  1997 longitude       0    0
 4  1997 sea_temp_c      0    0
 5  1997 humidity        0    0
 6  1997 wind_ew         0    0
 7  1997 wind_ns         0    0
 8  1993 humidity       93   25.3
 9  1993 air_temp_c      4    1.09
10  1993 sea_temp_c      3    0.815
11  1993 latitude        0    0
12  1993 longitude       0    0
13  1993 wind_ew         0    0
14  1993 wind_ns         0    0
```

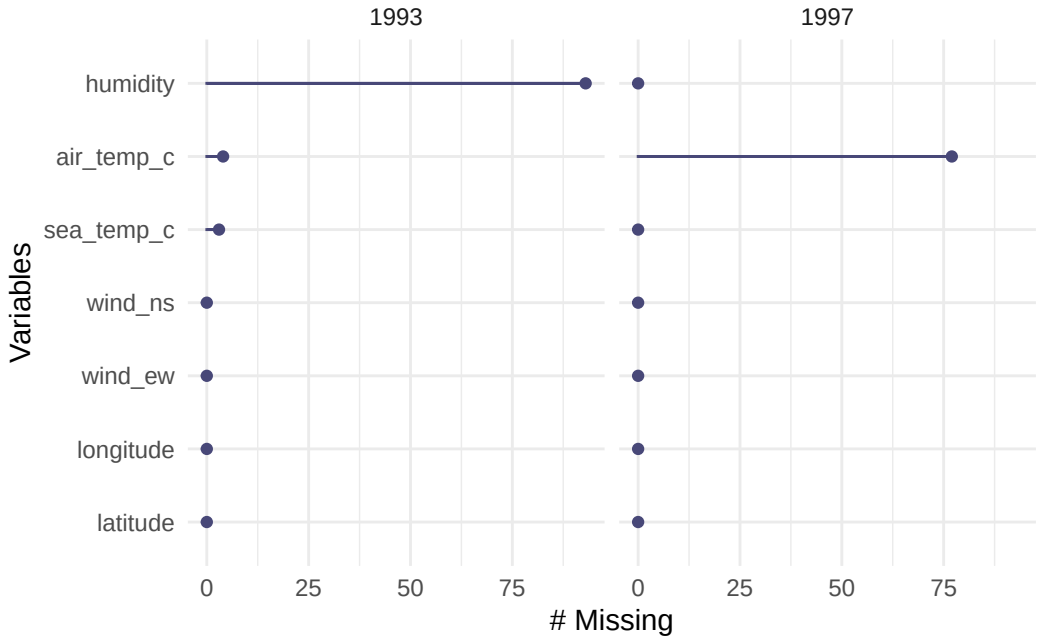
`naniar` also provides a visual summary for missing values.

```
gg_miss_var(oceanbuoys)
```



We can further group by a variable using the `facet` argument.

```
gg_miss_var(oceanbuoys, facet = year)
```



The content we just covered provides only a glimpse of the powerful features and capabilities of the `naniar` package. For a deeper understanding and comprehensive insights, readers are encouraged to explore the `naniar` documentation website.

8.3.3 DataExplorer

`DataExplorer` provides various helpful functions for data exploration. First, let's install and load the necessary packages.

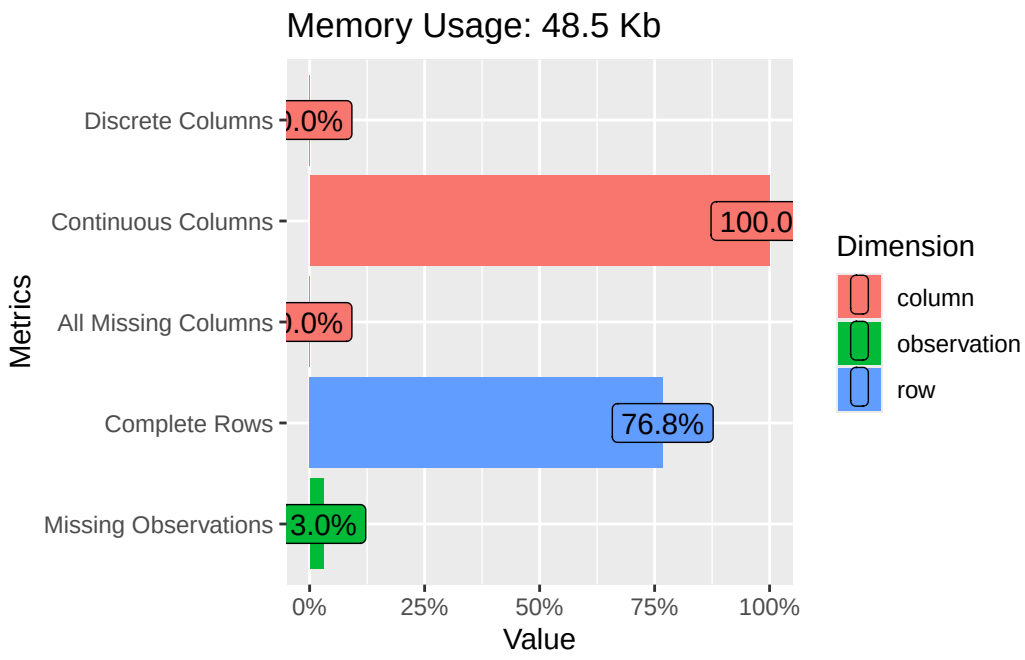
```
# Install package
install.packages("DataExplorer")

# Load the necessary packages
library(DataExplorer)
library(dplyr)
```

Let's use `oceanbuoys` data from the `naniar` package previously.

```
# Load the data
data("oceanbuoys", package = "naniar")
```

`DataExplorer` provides a general function to explore the data.

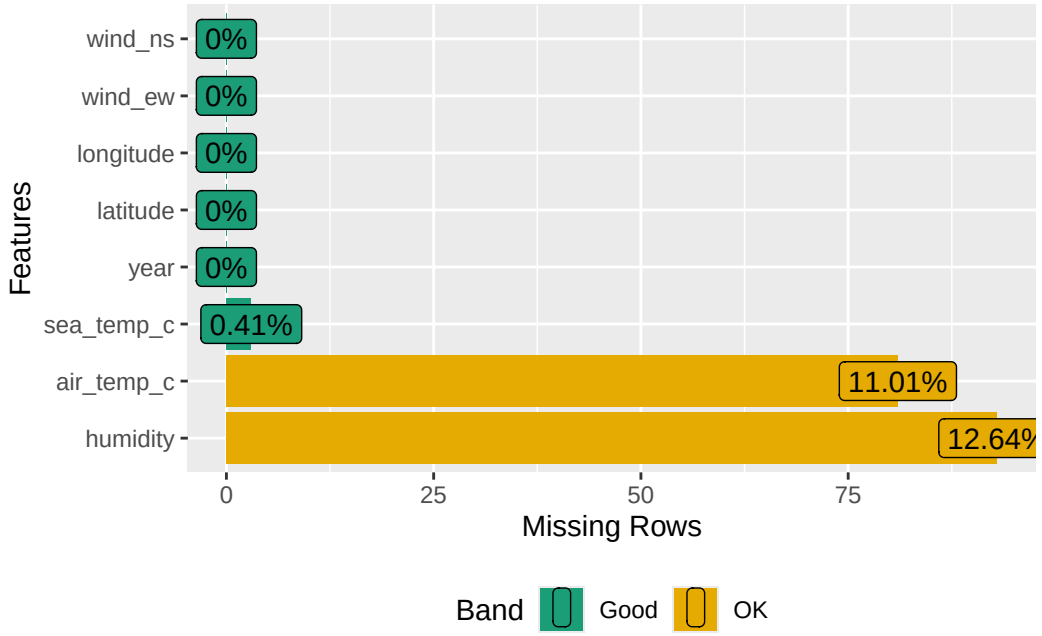
```
plot_intro(oceanbuoys)
```



From the plot, we understand that our data consists of all continuous (numeric) columns and no discrete (categorical) columns. No missing columns but we have 3% missing observations.

To investigate the missing observations, we can use `plot_missing()`:

```
plot_missing(oceanbuoys)
```



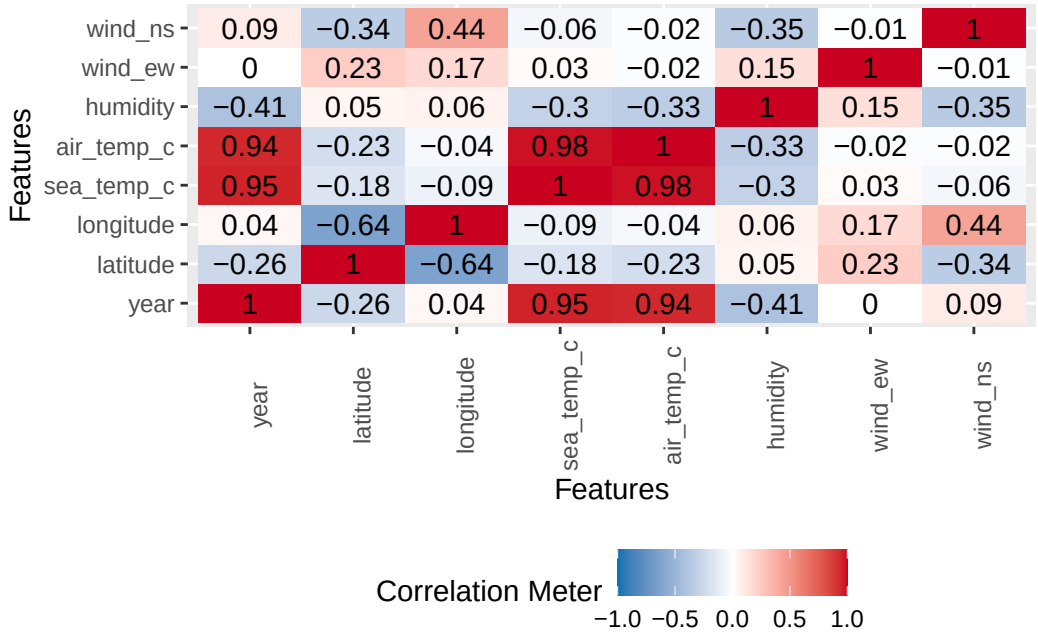
Alternatively, we can get a summary instead of a plot.

```
profile_missing(oceanbuoys)
```

```
# A tibble: 8 x 3
  feature    num_missing pct_missing
  <fct>            <int>       <dbl>
1 year                 0     0
2 latitude             0     0
3 longitude            0     0
4 sea_temp_c           3     0.00408
5 air_temp_c          81     0.110
6 humidity            93     0.126
7 wind_ew              0     0
8 wind_ns              0     0
```

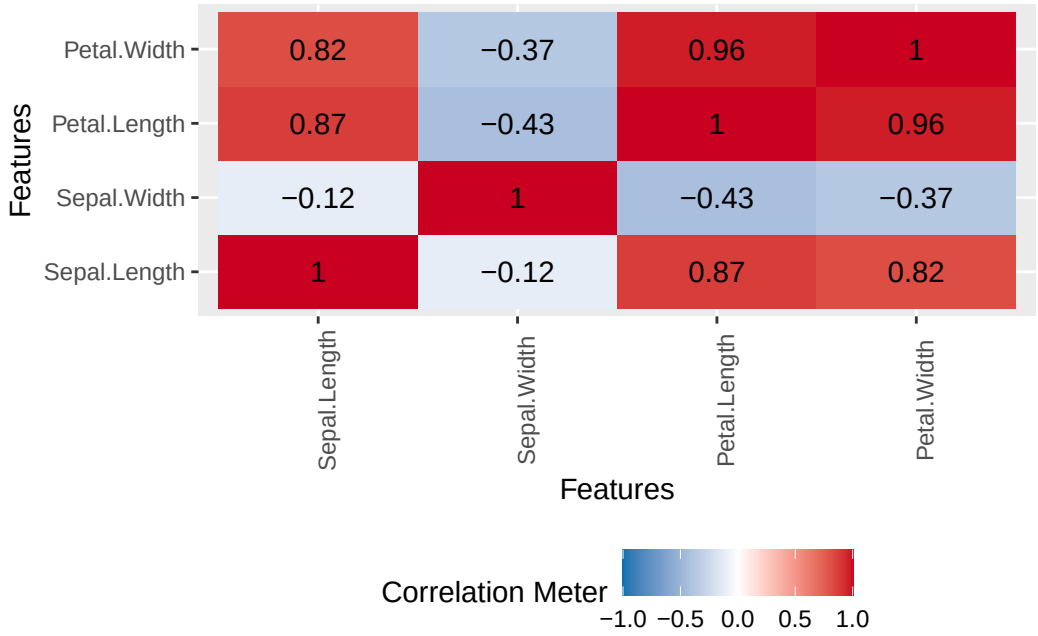
Additionally, `DataExplorer` provides a function to plot a correlation matrix. Correlation is a measure of association between two numerical variables. It ranges between -1 and 1. Values close to 1 indicate a high positive correlation between the two variables, while values close to -1 indicate a high negative correlation between the two numerical variables. Values close to 0 indicate a low correlation between the two values.

```
oceanbuoys %>%
  na.omit() %>%
  plot_correlation()
```



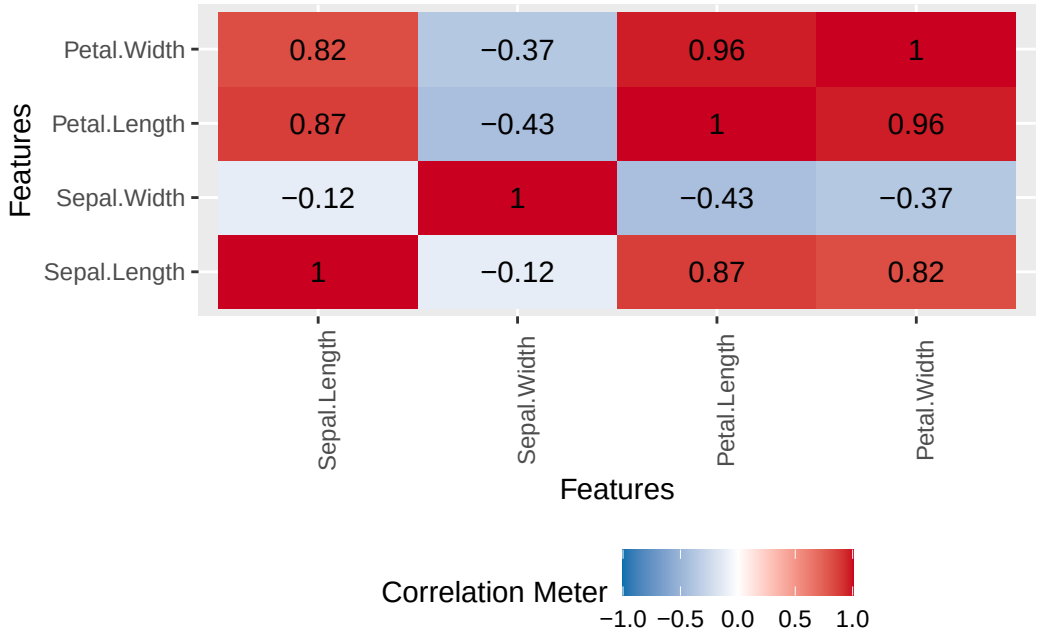
To do a correlation or in this case correlation plot, the variables should be numerical and have no missing values (hence, the use of `na.omit()` in the code). If for example, we want to apply this function to the `iris` dataset, in which we know that one of the variables is categorical, we need to exclude the variable first.

```
iris %>%
  select(-Species) %>%
  plot_correlation()
```



Alternatively, the more efficient code to exclude the non-numerical variable is using the `select_if()`.

```
iris %>%
  select_if(is.numeric) %>%
  plot_correlation()
```



`DataExplorer` provides more useful functions, which can not be extensively covered in this section. Interested readers can further study its documentation for more.

8.3.4 VIM

The `VIM` package contains tools for the visualisation of missing and/or imputed values. Imputation of the missing values is beyond the scope of this book. However, we going to see how functions from the `VIM` package can be utilised to explore the pattern of missingness.

First, make sure to install the `VIM` package and load the necessary packages.

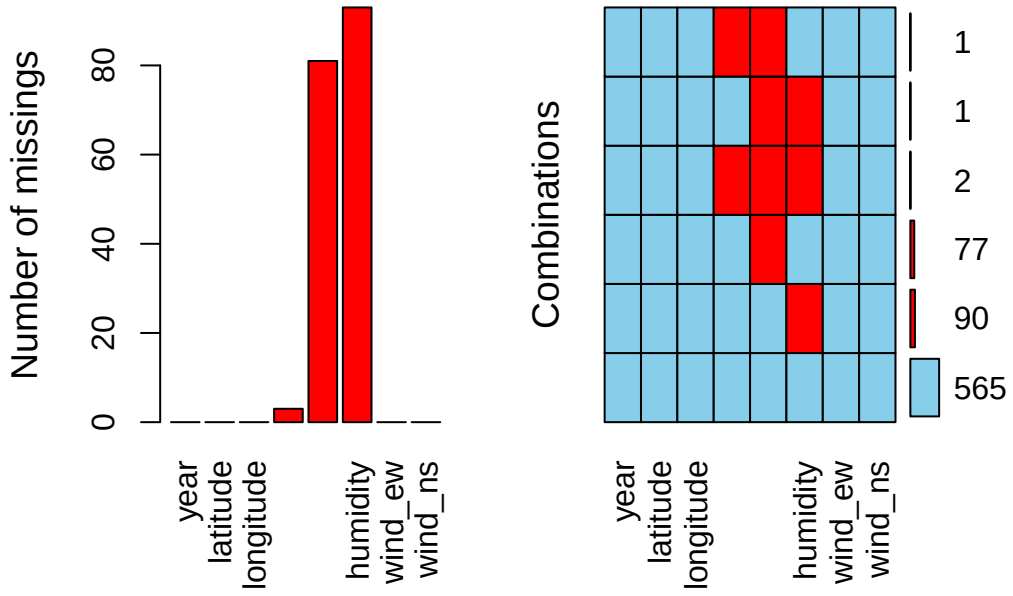
```
# Install package
install.packages("VIM")

# Load the necessary packages
library(VIM)
library(dplyr)
```

Let's use `oceanbuoys` data from the `naniar` package previously.

```
# Load the data
data("oceanbuoys", package = "naniar")
```

The `aggr()` function will plot our data and visualise the pattern of missing values. `numbers = TRUE` and `prop = FALSE` to make sure the number missing values in a number not a proportion.

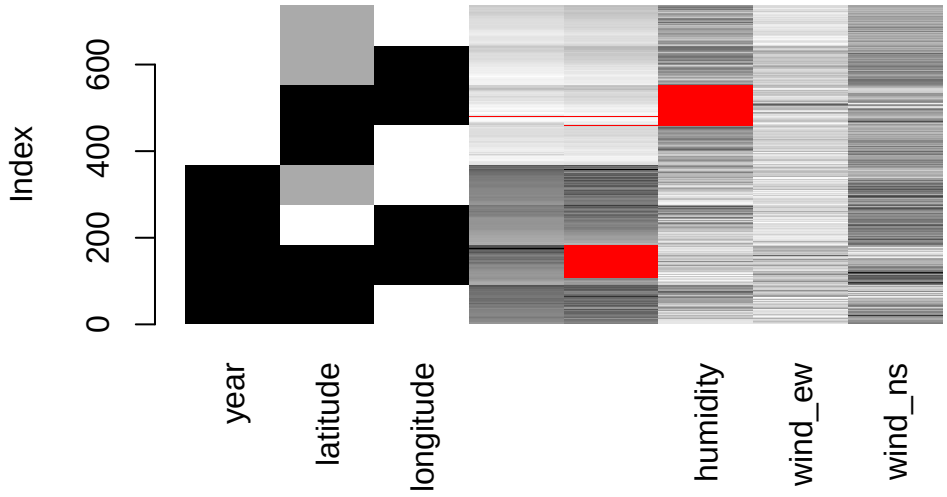
```
aggr(oceanbuoys, numbers = TRUE, prop = FALSE)
```



The red colour represents the missing values, and the blue colour represents the observed values. The first plot on the left presents the proportion of missing values according to variables. The second plot on the right presents the combination of missing values. Notice that if we have too many variables, the variable names will not be fully displayed. For example in the second plot, for a combination of `sea_temp_c`, `air_temp_c`, and `humidity`, we have 2 values missing.

The `aggr()` function is suitable to be utilised when we have a small to intermediate number of variables. For data with a large number of variables, `matrixplot()` may be more appropriate.

```
matrixplot(oceanbuoys)
```



`matrixplot()` scales the data into 0 (white) and 1 (black). The higher values will become close to 1, and the lower values will become close to 0. The red colour represents the missing values.

The `VIM` package contains more useful functions, however, the majority of them are for exploring the imputation methods and values. Interested readers can further study its documentation for more details.

8.3.5 dlookr

The `dlookr` package provides various helpful functions especially related to outliers. First, we need to install the `dlookr` package and load the necessary packages.

```
# Install package
install.packages("dlookr")

# Load the packages
library(dlookr)
library(dplyr)
```

Let's use the `Carseats` dataset, a dataset from `dlookr` package. The `diagnose_numeric()` function from the `dlookr` package is particularly useful for diagnosing numeric variables.

```
# Load the data
data("Carseats")

# Diagnose function
diagnose_numeric(Carseats)
```

```
    variables min     Q1       mean median     Q3    max zero minus outlier
1       Sales   0   5.39   7.496325   7.49   9.32  16.27    1     0       2
2   CompPrice  77 115.00 124.975000 125.00 135.00 175.00    0     0       2
3      Income  21  42.75  68.657500  69.00  91.00 120.00    0     0       0
4 Advertising   0   0.00   6.635000   5.00  12.00  29.00  144     0       0
5  Population  10 139.00 264.840000 272.00 398.50 509.00    0     0       0
6       Price  24 100.00 115.795000 117.00 131.00 191.00    0     0       5
7         Age  25  39.75  53.322500  54.50  66.00  80.00    0     0       0
8   Education  10  12.00  13.900000  14.00  16.00  18.00    0     0       0
```

Among its results, it provides the following insights:

- `zero`: the number of zero values in the data.
- `minus`: the count of negative values.
- `outlier`: the number of potential outliers detected in the dataset.

Additionally, we have `diagnose_category()` which summarises categorical variables. This function returns several outputs:

- `levels`: level for each categorical variable.
- `N`: number of observations.
- `freq`: number of observations at the levels.
- `ratio`: percentage of observations at the levels
- `rank`: rank of occupancy ratio of levels

```
diagnose_category(Carseats)
```

```
  variables levels   N freq ratio rank
1 ShelveLoc Medium 400  219 54.75    1
2 ShelveLoc    Bad 400   96 24.00    2
3 ShelveLoc   Good 400   85 21.25    3
4     Urban    Yes 400  282 70.50    1
5     Urban     No 400  118 29.50    2
6        US    Yes 400  258 64.50    1
7        US     No 400  142 35.50    2
```

We can further explore the outliers identified by `diagnose_numeric()` using `diagnose_outlier()`.

```
diagnose_outlier(Carseats)
```

```
    variables outliers_cnt outliers_ratio outliers_mean  with_mean without_mean
1       Sales            2           0.50         15.95   7.496325     7.453844
2   CompPrice            2           0.50        126.00 124.975000   124.969849
3      Income            0           0.00           NaN  68.657500    68.657500
4 Advertising            0           0.00           NaN   6.635000     6.635000
5  Population            0           0.00           NaN 264.840000   264.840000
6       Price            5           1.25        100.40 115.795000   115.989873
7         Age            0           0.00           NaN  53.322500    53.322500
8   Education            0           0.00           NaN  13.900000    13.900000
```

This function returns:

- `outliers_cnt`: number of outliers.
- `outliers_ratio`: percent of outliers.
- `outliers_mean`: mean of outliers.
- `with_mean`: mean of values with outliers.
- `without_mean`: mean of values of without outliers.

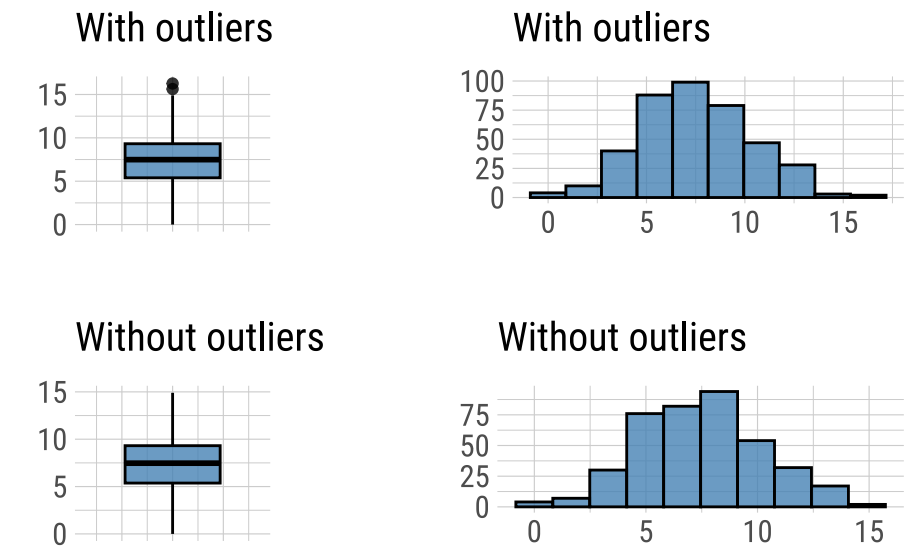
In fact, we can further simplify the result using the `filter()` from the `dplyr` package.

```
diagnose_outlier(Carseats) %>%
  filter(outliers_cnt > 0)
```

```
  variables outliers_cnt outliers_ratio outliers_mean  with_mean without_mean
1     Sales            2           0.50         15.95   7.496325     7.453844
2 CompPrice            2           0.50        126.00 124.975000   124.969849
3     Price            5           1.25        100.40 115.795000   115.989873
```

Furthermore, `dlookr` provides another useful function to visualise the outliers. However, for this example, we going to select a single variable.

```
Carseats %>%
  select(Sales) %>%
  plot_outlier()
```



This function returns a boxplot and histogram for values with and without outliers. Thus, we can see how much the outliers in the variables change the distribution of the data. `dlookr` actually provides more functions than the ones we covered in this section. Readers are suggested to go through its documentation to further learn this package.

8.4 Chapter summary

In this chapter, we have covered how missing data and outliers were identified in R using base R functions. Moreover, we have covered several useful functions from five R packages:

1. skimr
2. naniar
3. DataExplorer
4. VIM
5. dlookr

`skimr` provides general functions for data exploration (Waring et al. 2024). `naniar`, `DataExplorer`, and `VIM` provide additional functions to explore and investigate missing data (Tierney and Cook 2023; Cui 2024; Kowarik and Templ 2016). Lastly, `dlookr` provides more functions for investigating outliers in the dataset (Ryu 2024).

While this chapter highlights the capabilities of these five packages, it is important to note that R offers many additional packages that can further enhance your analysis. Readers are encouraged to explore beyond these tools to find packages best suited to their specific needs.

8.5 Revision

1. Why is data exploration considered a crucial first step in data analysis, and what are some R functions and packages that can help in this process?
2. Write an R code to check for missing values in each column of the `riskfactors` dataset, a dataset from the `naniar` package. Then, count how many `NA` values exist in total.

```
# Load the data
data("riskfactors", package = "naniar")
```

3. Using the `riskfactors` dataset from question 2, write R code to remove all missing values in the dataset and determine how many rows are left.
4. Besides dropping the missing values, what are the possible solutions to missing values?
5. What are the possible solutions for outliers?
6. What are the pros and cons of dropping:
    a. Missing values
    b. Outliers
7. Name three packages for data exploration besides the five packages covered in this chapter.

9 Descriptive statistics

> "Descriptive statistics are not just numbers; they tell stories about data."
>
> – Anonymous

Descriptive statistics, as the name suggests, involves summarizing and describing the main features of a dataset. It allows us to distil large or complex datasets into manageable summaries, making it easier to interpret and understand the data. While some details are inevitably lost during this process, we gain valuable insights and a clearer direction for further analysis.

This statistical approach is crucial in data analysis because it provides a foundation for understanding the overall structure of the data. Descriptive statistics use measures such as central tendency (mean, median, and mode), variability (range, variance, and standard deviation), and data visualization tools (like histograms and pie charts) to offer a clear overview of the data's characteristics.

By simplifying the data, descriptive statistics not only highlight trends and patterns but also serve as a stepping stone for more advanced statistical methods. In this chapter, we will cover the fundamentals of descriptive statistics, helping you understand its key concepts, techniques, and practical applications.

9.1 Load packages

Please load these packages before proceeding to the next section.

```
# Install the packages if necessary
install.packages("tidyverse")
install.packages("DescTools")
install.packages("summarytools")
install.packages("gtsummary")

# Load the packages
library(tidyverse)
```

9.2 Measures of central tendency

Measures of central tendency aim to identify the centre or typical value within a dataset. It provides a summary of the data by describing the point around which most values cluster.

9.2.1 Mean

The mean represents the average of a dataset. It is calculated by summing all the values in the dataset and dividing the total by the number of values.

$$Mean(\overline{x}) = \frac{\sum x}{n}$$

For example, to calculate the mean manually for the numbers:

$$5, 6, 7, 8, 13, 2$$

1. Sum up all the values (Σx).

$$5 + 6 + 7 + 8 + 13 + 2 = 41$$

2. Divide the sum of all the values by the count of the values (n).

$$\frac{41}{6} = 6.83$$

Alternatively, R provides a `mean()` function to calculate the mean of values.

```
# The values
x_mean = c(5, 6, 7, 8, 13, 2)

# Calculate mean
mean(x_mean)
```

```
[1] 6.833333
```

9.2.2 Median

The median represents the middle value in a sorted dataset. It divides the dataset into two halves: 50% of the data values are smaller than the median, and 50% are larger.

For example, to determine the median for the below values:

$$5, 6, 7, 8, 13, 2$$

1. Sort the values in order from the smallest to the largest.

$$2, 5, 6, 7, 8, 13$$

2. Determine the middle values (if there are two middle values, calculate the average of the two values).

$$\frac{6+7}{2} = 6.5$$

Alternatively, to calculate the median in R, we can use the `median()` function.

```
# The values
x_med = c(5, 6, 7, 8, 13, 2)

# Calculate the median
median(x_med)
```

```
[1] 6.5
```

9.2.3 Mode

The mode represents the most frequently occurring value in a dataset. It is the value that appears the most times. Unlike the mean or median, the mode can be used for both numerical and categorical data.

For example, to determine the mode for the numbers:

$$5, 6, 7, 8, 13, 2, 8, 1$$

The mode is the most frequent value which is 8.

In R, there is no function in base R to determine the value of mode. However, we can manually determine the its value in R.

```r
# Range of values
x_mode_num = c(5, 6, 7, 8, 13, 2, 8, 1)

# Calculate the count for each value
x_mode_num_table <-
  x_mode_num %>%
  table() %>%
  as.data.frame()
x_mode_num_table
```

```
   . Freq
1  1    1
2  2    1
3  5    1
4  6    1
5  7    1
6  8    2
7 13    1
```

```r
# Determine the frequently appearing value
x_mode_num_table %>%
  filter(Freq == max(Freq))
```

```
  . Freq
1 8    2
```

Additionally, we can determine the mode for the categorical data using the same approach:

```r
# Range of values
x_mode_cat <- c("yes", "no", "unsure", "yes", "yes", "no")

# Calculate the count for each value
x_mode_cat_table <-
  x_mode_cat %>%
  table() %>%
  as.data.frame()
x_mode_cat_table
```

```
       . Freq
1     no    2
2 unsure    1
3    yes    3
```

```
# Determine the frequently appearing value
x_mode_cat_table %>%
  filter(Freq == max(Freq))
```

```
    . Freq
1 yes    3
```

Alternatively, there are several packages available to determine the mode. For example, we can use the `DescTools` package.

```
# Determine the mode for numerical values
DescTools::Mode(x_mode_num)
```

```
[1] 8
attr(,"freq")
[1] 2
```

```
# Determine the mode for categorical values
DescTools::Mode(x_mode_cat)
```

```
[1] "yes"
attr(,"freq")
[1] 3
```

The `Mode()` function from `DescTools` returns the mode value and the frequency of the values. For example, `yes` is the mode and it appears in the data 3 times.

9.2.4 Comparison of mean, median, and mode

9.2.4.1 Mean vs. median

The mean is sensitive to outliers compared to the median and mode. In the presence of outliers, the median is more robust and thus, preferred to describe the central tendency of the data. Let's demonstrate the change of mean and median in the presence of the outliers.

Firstly, let's create the data with and without the outlier.

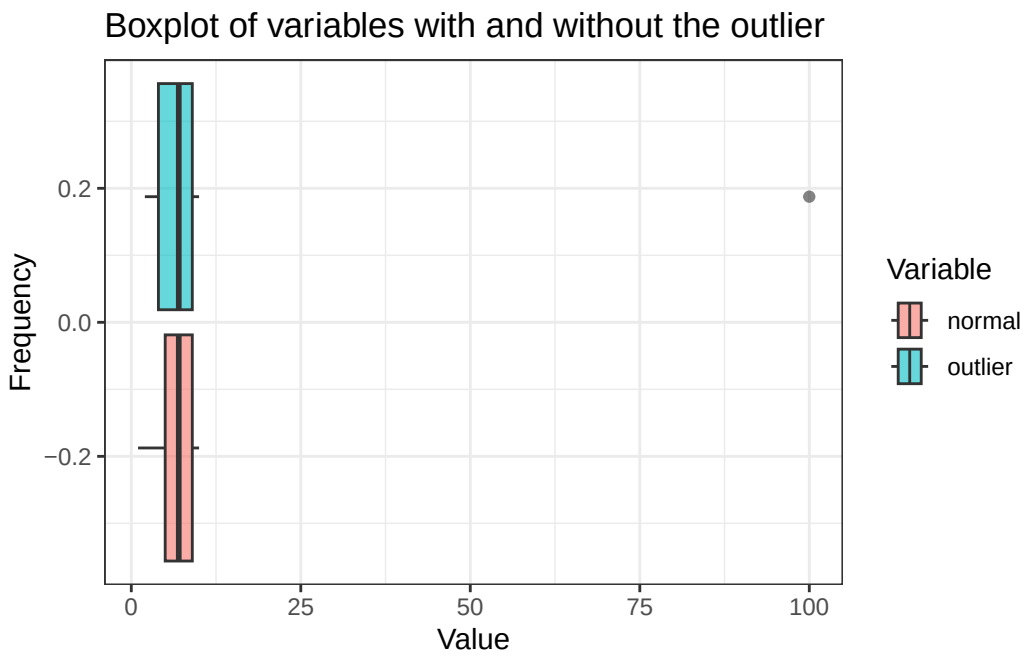
```r
# Set seed for reproducibility
set.seed(123)

# Create data
outlier_data <- data.frame(
  outlier = c(sample(x = c(1:10), size = 20, replace = TRUE), 100),
  normal = sample(x = c(1:10), size = 21, replace = TRUE)
)
```

Before we demonstrate further, let's change the data structure, so, that it is easier to do the plots.

```r
outlier_data_long <-
  outlier_data %>%
  # Change data structure to long data
  pivot_longer(cols = 1:2, names_to = "Variable", values_to = "Value")
```

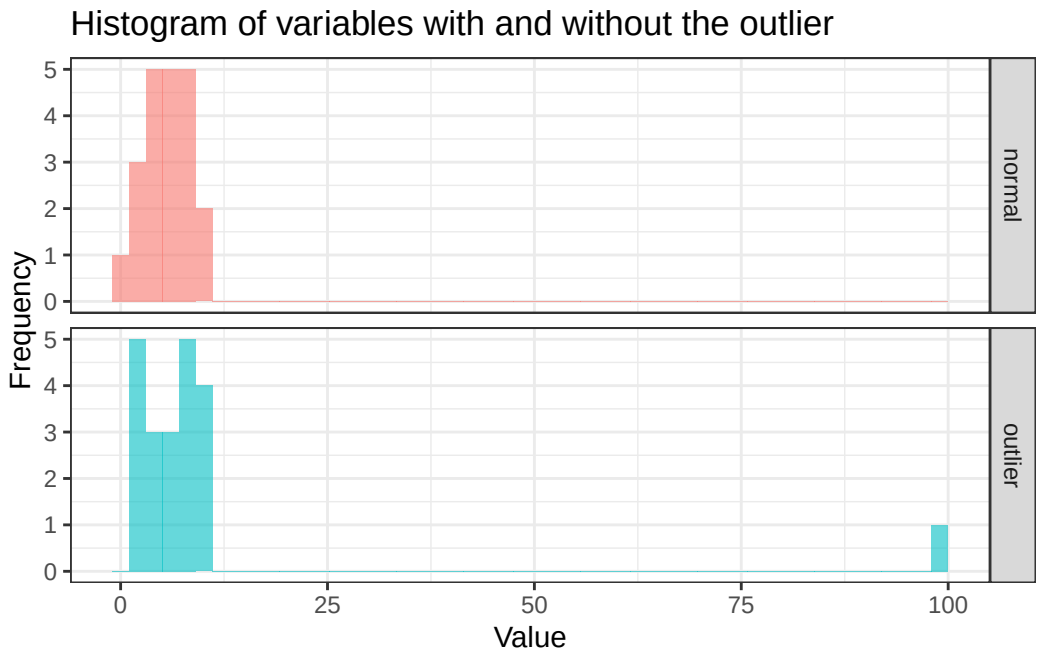
Let's plot the boxplot for both data to visualise the outlier.

```r
outlier_data_long %>%
  ggplot(aes(x = Value, fill = Variable)) +
  geom_boxplot(alpha = 0.6) +
  theme_bw() +
  labs(
    title = "Boxplot of variables with and without the outlier",
    x = "Value",
    y = "Frequency"
  )
```



We can see the outlier in the `var_outlier`. If we plot the histogram for both variables, we can further visualise the distribution of the data.

```
outlier_data_long %>%
  ggplot(aes(x = Value, fill = Variable)) +
  geom_histogram(alpha = 0.6, bins = 50) +
  facet_grid(rows = vars(Variable)) +
  theme_bw() +
  labs(
    title = "Histogram of variables with and without the outlier",
    x = "Value",
    y = "Frequency"
  ) +
  theme(legend.position = "none")
```



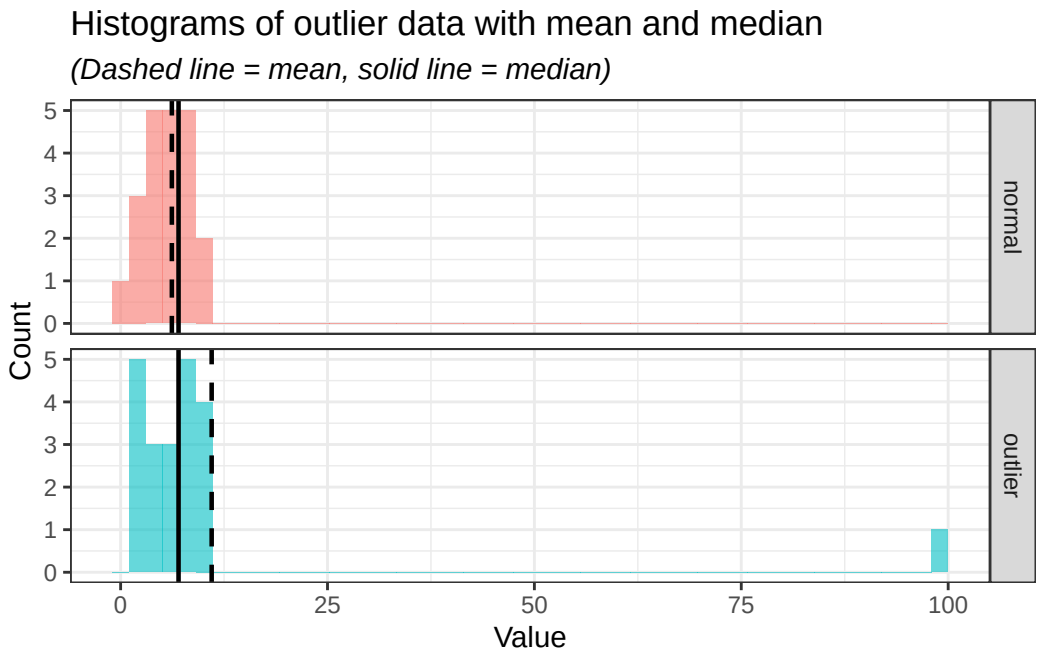
In the histogram, it is evident that the majority of values are concentrated between 0 and 25. When comparing the mean and median for both variables, with and without the outlier, the median demonstrates greater stability irrespective of the outlier's presence. The medians for both variables remain relatively similar, whereas the means show significant variation, reflecting the mean's sensitivity to outliers.

```
summary_stats <-
  outlier_data_long %>%
  group_by(Variable) %>%
  summarise(
    mean = mean(Value),
    median = median(Value)
  )
summary_stats
```

```
# A tibble: 2 x 3
  Variable  mean median
  <chr>    <dbl>  <dbl>
1 normal    6.19      7
2 outlier  11         7
```

Let's put the values of the mean and median onto a histogram so that we can see how these values change in the presence of the outlier.

```r
outlier_data_long %>%
  ggplot(aes(x = Value, fill = Variable)) +
  geom_histogram(alpha = 0.6, bins = 50) +
  geom_vline(
    data = summary_stats,
    aes(xintercept = mean, color = Variable),
    linetype = "dashed",
    linewidth = 0.8
    ) +
  geom_vline(
    data = summary_stats,
    aes(xintercept = median, color = Variable),
    linetype = "solid",
    linewidth = 0.8
    ) +
  scale_color_manual(values = c("black", "black")) +
  labs(
    title = "Histograms of outlier data with mean and median",
    x = "Value",
    y = "Count",
    subtitle = "(Dashed line = mean, solid line = median)"
  ) +
  facet_grid(rows = vars(Variable)) +
  theme_bw() +
  theme(
    legend.position = "none",
    plot.subtitle = element_text(face = "italic")
    )
```



From the histograms, we can see the median (solid line) is relatively at the same position while the mean (the dashed line) moves according to the outlier.

Besides the sensitivity of the mean to the outliers, the mean is the best measure of central tendency for normally distributed data because it incorporates all data points and accurately reflects the centre of a symmetric distribution. In a normal distribution, data points are symmetrically distributed around the mean, and thus, the mean provides a reliable measure of the "typical" value. However, it is to be noted despite the mean being more preferred in normal distribution, the values of median and mode are very close to the mean.

On the other hand, for the skewed distribution, the median is preferred because it is not influenced by extreme values (outliers) that can distort the mean. In a skewed distribution, where data points are unevenly spread, the median represents the middle value, providing a better reflection of the "typical" data point compared to the mean. This makes the median more robust when dealing with outliers and skewed data, as the mean may be pulled toward the skewed tail, giving a misleading representation of central tendency.

Again, let's demonstrate this point in R. First, we create the data consisting of two variables of normal and skewed distributions.

```
# Set seed for reproducibility
set.seed(123)

# Create data
distribution_data <-
```

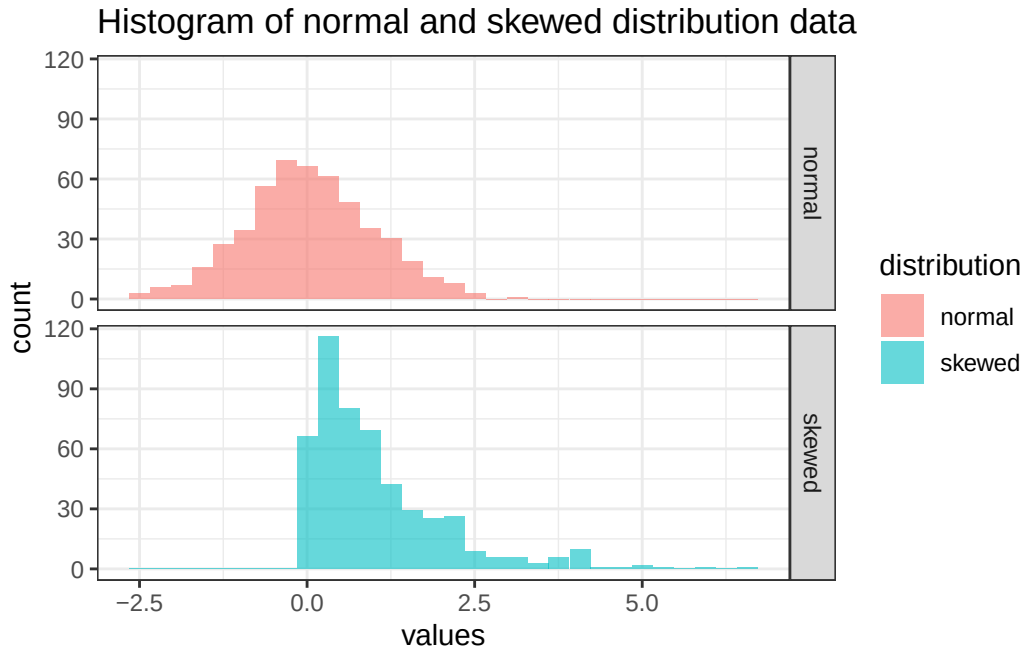
```
  data.frame(
    normal = rnorm(500, mean = 0, sd = 1),
    # Create a skewed distribution variable (an exponential distribution with rate = 1)
    skewed = rexp(500, rate = 1))
```

Next, we need to change the data structure to a long data type to fully explore the data.

```
distribution_data_long <-
  distribution_data %>%
  pivot_longer(cols = 1:2, names_to = "distribution", values_to = "values")
```

We going to plot the histogram to visualise the distributions.

```
distribution_data_long %>%
  ggplot(aes(x = values, fill = distribution)) +
  geom_histogram(bins = 30, alpha = 0.6) +
  facet_grid(rows = vars(distribution)) +
  labs(title = "Histogram of normal and skewed distribution data") +
  theme_bw()
```



Next, let's calculate the mean and median for each of the distributions.

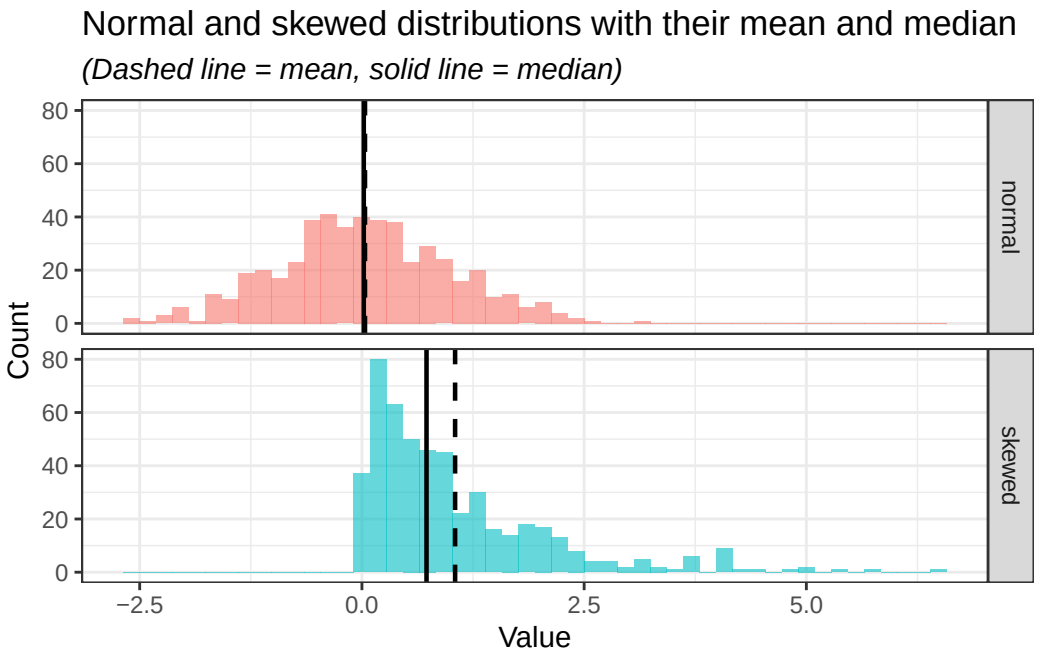
```r
summary_stat_dist <-
  distribution_data_long %>%
  group_by(distribution) %>%
  summarise(
    mean = mean(values),
    median = median(values)
    )
summary_stat_dist
```

```
# A tibble: 2 x 3
  distribution   mean median
  <chr>         <dbl>  <dbl>
1 normal       0.0346 0.0207
2 skewed       1.05   0.727
```

Notice that the mean and median for normal distribution are close while skewed distribution is otherwise. Let's visualise the mean and median in the histogram.

```r
distribution_data_long %>%
  ggplot(aes(x = values, fill = distribution)) +
  geom_histogram(alpha = 0.6, bins = 50) +
  geom_vline(
    data = summary_stat_dist,
    aes(xintercept = mean, color = distribution),
    linetype = "dashed",
    linewidth = 0.8
    ) +
  geom_vline(
    data = summary_stat_dist,
    aes(xintercept = median, color = distribution),
    linetype = "solid",
    linewidth = 0.8
    ) +
  scale_color_manual(values = c("black", "black")) +
  labs(
    title = "Normal and skewed distributions with their mean and median",
    x = "Value",
    y = "Count",
    subtitle = "(Dashed line = mean, solid line = median)"
    ) +
  facet_grid(rows = vars(distribution)) +
```

```
  theme_bw() +
  theme(
    legend.position = "none",
    plot.subtitle = element_text(face = "italic")
    )
```



In the normal distribution, despite the values of mean and median being close, the mean is preferred as it takes into account the whole range of values mathematically. However, in the skewed distribution, the median is preferred as it is less affected by the extreme values and sparsity of the values as compared to the mean.

9.2.4.2 Mode vs. mean and median

Lastly, the mode is less useful for the data analysis compared to the mean and median. Moreover, the mode may not be unique or exist in certain datasets. Let's demonstrate these cases:

1. Mode values are not unique.

```
# Values
two_mode_data <- c("yes", "no", "unsure", "yes", "yes", "no", "no")

# Determine the mode
DescTools::Mode(two_mode_data)
```

```
[1] "no"  "yes"
attr(,"freq")
[1] 3
```

2. Mode value does not exist.

```r
# Values
mode_not_exist <- c("yes", "unsure", "no")

# Determine the mode
DescTools::Mode(mode_not_exist)
```

```
[1] NA
attr(,"freq")
[1] NA
```

9.3 Measures of variability

Measures of variability describe the spread or dispersion of data points in a dataset. These measures indicate how much the data values differ from each other and from the central tendency (e.g., mean or median).

9.3.1 Range

The range is the difference between maximum and minimum values in a dataset.

$$Range = Maximum - Minimum$$

For example, the range for this range of values:

$$1, 2, 5, 8, 10, 5$$

1. Identify the maximum and minimum values.

$$Maximum = 10, Minimum = 1$$

2. Calculate the range.

$$10 - 1 = 9$$

In R, we have a `range()` function. However, instead of giving a range, the `range()` function returns a minimum and maximum value.

```
# The numeric values
x_range <- c(1, 2, 5, 8, 10, 5)

# Range function
range(x_range)
```

```
[1]  1 10
```

To calculate the range in R, we can utilise `max()` and `min()` functions, which identify the maximum and minimum values, respectively.

```
max(x_range) - min(x_range)
```

```
[1] 9
```

9.3.2 Variance

Variance measures the average squared difference of each data point from the mean. It shows how data points spread around the mean in squared units.

$$Variance(s^2) = \frac{\sum(x_i - \overline{x})^2}{n-1}$$

For example, to calculate the variance for these data points:

$$3, 5, 6, 8, 2, 9$$

1. Calculate the mean ($\overline{x}$).

$$\frac{3+5+6+8+2+9}{6} = 5.5$$

2. Calculate the difference between each data point and mean, square it, and sum up all the squared differences ($\sum(x_i - \overline{x})^2$).

$$(3-5.5)^2 + (5-5.5)^2 + (6-5.5)^2 + (8-5.5)^2 + (2-5.5)^2 + (9-5.5)^2 = 37.5$$

3. Divide the value by the count of the numbers $(n-1)$.

$$\frac{37.5}{6-1} = 7.5$$

In R, we can calculate the variance using the `var()` function.

```
# The values
x_variance <- c(3, 5, 6, 8, 2, 9)

# Calculate variance
var(x_variance)
```

```
[1] 7.5
```

The equation that we used to calculate the variance manually and the one in R is known as sample variance. There is another equation known as population variance. The sample variance is used to describe the spread out of the values in a sample, while population variance describes the spread out of the values in a population.

$$\text{Sample variance}(s^2) = \frac{\sum(x_i - \overline{x})^2}{n-1}$$
$$\text{Population variance}(\sigma^2) = \frac{\sum(x_i - \mu)^2}{N}$$

Notice the slight differences in the denominators of the two equations. Sample variance is calculated for a subset of data and includes an adjustment for smaller sample sizes by dividing by $n-1$ instead of n. This correction (known as Bessel's correction) ensures that the sample variance is an unbiased estimate of the population variance. In contrast, population variance is calculated for the entire dataset and does not require this adjustment. Additionally, instead of using sample mean $(\overline{x})$, population variance is using the population mean (μ).

In practical data analysis, we are almost always working with a sample rather than the entire population. As a result, the formula for sample variance is typically used to estimate the variability of the dataset.

9.3.3 Standard deviation

Standard deviation is the square root of the variance, providing a measure of dispersion in the same units as the data. A smaller value indicates less variability and a larger value indicates greater variability.

$$\text{Standard deviation}(SD) = \sqrt{Variance}$$

For example, to calculate the standard deviation for these numbers:

$$3, 5, 6, 8, 2, 9$$

1. Calculate the sample variance (s^2).

$$\frac{(3-5.5)^2 + (5-5.5)^2 + (6-5.5)^2 + (8-5.5)^2 + (2-5.5)^2 + (9-5.5)^2}{6-1} = 7.5$$

2. Square root the sample variance (s^2).

$$\sqrt{7.5} = 2.74$$

In R, we can use the `sd()` function to calculate the standard deviation, This function uses sample variance to calculate the standard deviation.

```
# The values
x_sd <- c(3, 5, 6, 8, 2, 9)

# Calculate variance
sd(x_sd)
```

```
[1] 2.738613
```

9.3.4 Interquartile range

The interquartile range describes the range of the middle 50% of the data (the difference between the third quartile and first quartile).

$$\text{Interquartile range}(IQR) = Q_3 - Q_1$$

To calculate the interquartile range for these numbers:

$$3, 5, 6, 8, 2, 9$$

1. Divide the data into lower half and upper half

$$\text{Lower half} = 2, 3, 5$$
$$\text{Upper half} = 6, 8, 9$$

2. Identify the median of the lower half (first quartile, Q_1) and the median of the upper half (third quartile, Q_3).

$$\text{Median of lower half}(Q_1) = 3$$
$$\text{Median of the upper half}(Q_3) = 8$$

3. Calculate the interquartile range.

$$8 - 3 = 5$$

In R, we can use `IQR()` to calculate the interquartile range.

```
# The values
x_iqr <- c(3, 5, 6, 8, 2, 9)

# Calculate interquartile range
IQR(x_iqr, type = 2)
```

```
[1] 5
```

`IQR()` actually uses the `quantile()` function to calculate the quartiles, subsequently calculating the interquartile range. `quantile()` with `type = 2` reflects the exact first and the third quartile that we manually calculate. `quantile()` and `IQR()` have 9 types of algorithms or formulas to calculate the quartile in R. The default, both functions use `type = 7`. However, Hyndman & Fan proposed to use type 8 (1996).

```
# Quantile type 2 reflects our manual calculation
quantile(x_iqr, type = 2)
```

```
  0%  25%  50%  75% 100%
 2.0  3.0  5.5  8.0  9.0
```

```
# IQR type 2 reflects our manual calculation
IQR(x_iqr, type = 2)
```

```
[1] 5
```

```
# Quantile type 7 is the default
# We don't actually need to specify the type since it is the default
quantile(x_iqr, type = 7)
```

```
  0%  25%  50%  75% 100%
 2.0  3.5  5.5  7.5  9.0
```

```
# IQR type 7 is the default
IQR(x_iqr, type = 7)
```

```
[1] 4
```

```
# Quantile type 8
quantile(x_iqr, type = 8)
```

```
      0%      25%      50%      75%     100%
2.000000 2.916667 5.500000 8.083333 9.000000
```

```
# IQR type 8
IQR(x_iqr, type = 8)
```

```
[1] 5.166667
```

Briefly, I suggest using `type = 7` or `type = 8` for calculating the quartile and interquartile range. For accuracy, we can use `type = 8` as proposed by Hyndman & Fan(1996). For simplicity, we can use `type = 7` as it is the default in R and widely used in other statistical software as well.

9.4 Packages for descriptive statistics

We have covered in Section 8.3 several general packages for data exploration which includes descriptive statistics. In this, section we going to explore two useful packages for descriptive statistics: `summarytools` and `gtsummary`. It is to be noted that there are several other R packages available for descriptive statistics, to name a few: `psych`, `pastecs`, and `Hmisc`.

9.4.1 summarytools

`summarytools` provides an easy-to-use function for descriptive statistics. For numerical variables, we can use `descr()`. We going to use the `iris` dataset to demonstrate the capabilities of `summarytools`. More details on the `iris` dataset can be read by typing `?iris` in the

```
# Load packages
library(summarytools)

# Descriptive statistics for numerical data
descr(iris %>% select(-Species))
```

```
Descriptive Statistics
iris
N: 150

                   Petal.Length   Petal.Width   Sepal.Length   Sepal.Width
---------------- -------------- ------------- -------------- -------------
            Mean           3.76          1.20           5.84          3.06
         Std.Dev           1.77          0.76           0.83          0.44
             Min           1.00          0.10           4.30          2.00
              Q1           1.60          0.30           5.10          2.80
          Median           4.35          1.30           5.80          3.00
              Q3           5.10          1.80           6.40          3.30
             Max           6.90          2.50           7.90          4.40
             MAD           1.85          1.04           1.04          0.44
```

IQR	3.50	1.50	1.30	0.50
CV	0.47	0.64	0.14	0.14
Skewness	-0.27	-0.10	0.31	0.31
SE.Skewness	0.20	0.20	0.20	0.20
Kurtosis	-1.42	-1.36	-0.61	0.14
N.Valid	150.00	150.00	150.00	150.00
N	150.00	150.00	150.00	150.00
Pct.Valid	100.00	100.00	100.00	100.00

We can see that `descr()` returns several familiar measures for descriptive statistics such as mean, standard deviation (`Std.Dev`), minimum value, maximum value, median, first and third quartile. There are several other measures that we have not covered previously:

1. `MAD`:
    - Describes the median of the absolute deviations of the data points from the median of the dataset.
    - A small MAD indicates that most data points are close to the median, reflecting low variability and a large MAD suggests greater spread, meaning data points are more dispersed from the median.
    - Unlike the variance and standard deviation is less sensitive to outliers, thus, making it more suitable for a skewed distribution.
2. `CV`:
    - CV or coefficient of variation is a relative measure of variability. It reflects the ratio of standard deviation (s^2) to the mean ($\overline{x}$).
    - $$CV = \frac{s^2}{\overline{x}}$$
      $$0.47 = \frac{1.77}{3.76}$$
    - Low CV indicates less variability relative to the mean, suggesting consistency and uniformity in the dataset, while high CV indicates greater variability relative to the mean, which could imply inconsistency or a wide spread of data values.
    - There is no exact value of CV that we can rate the variability of the data, however, as a rule of thumb:
        - CV < 0.3 indicates a low variability and spread which can be considered as good.
        - CV > 0.3 indicates a high variability, thus, requiring us to inspect the data distribution further.

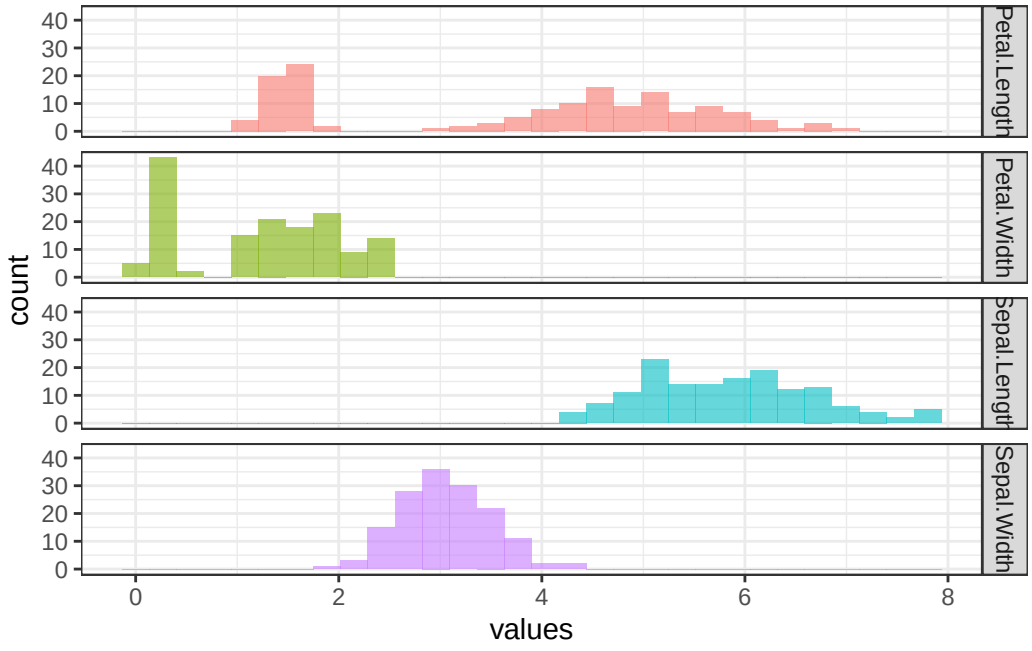
- We can recall the CV of each of the numeric variables in the `iris` dataset.

```
descr(iris %>% select(-Species),
      stats = c("CV", "mean", "sd"),
      transpose = TRUE,
      headings = FALSE)
```

```
                      CV    Mean   Std.Dev
------------------ ------ ------ ---------
      Petal.Length   0.47   3.76      1.77
       Petal.Width   0.64   1.20      0.76
      Sepal.Length   0.14   5.84      0.83
       Sepal.Width   0.14   3.06      0.44
```

- Subsequently, we can reflect the values of CV for each numeric variable in the `iris` dataset to their respective histogram.

```
iris %>%
  pivot_longer(1:4, names_to = "variable", values_to = "values") %>%
  ggplot(aes(x = values, fill = variable)) +
  geom_histogram(alpha = 0.6, bins = 30) +
  facet_grid(rows = vars(variable)) +
  theme_bw() +
  theme(legend.position = "none")
```



3. `Skewness` and `SE.skewness`

   - Skewness describes the symmetry of the distribution of the data.
   - As a rule of thumb (Hair et al. 2022):
     - Skewness ≈ 0: indicates the data is nearly symmetry.
     - -1 < Skewness < 1: indicates a low skewness, which is excellent.
     - -2 < Skewness < 2: indicates a moderate skewness, which is acceptable.
     - Skewness > 2 or Skewness < -2: indicates a substantial skewness, in which data distribution is not normal.
   - A positive value indicates the tail of the histogram is longer on the right, indicating more extreme high values, while the negative skewness indicates the tail of the histogram is longer on the left, indicating more extreme low values.
   - Comparing skewness to the standard error (SE) of the skewness helps decide whether the asymmetry is meaningful or due to random variation in the sample.
   - $$\frac{Skewness}{SE.skewness} > \pm 1.96 = \text{significant skewness}$$

4. `Kurtosis`

   - Kurtosis is a statistical measure that evaluates the extent to which data points cluster in the tails or the peak of the distribution compared to a normal distribution.
   - In simpler words, kurtosis indicates whether the data distribution is too "thin" or "broad" compared to a normal distribution.
   - Kurtosis and skewness are typically interpreted together to assess the shape of a data distribution. When both values are close to 0, it suggests that the data approximates a normal distribution.
   - As a rule of thumb (Hair et al. 2022):
     - Kurtosis ≈ 0: data approximate a normal distribution.
     - Kurtosis > 2: data has a peaked distribution.
     - Kurtosis < -2: data has a flat distribution.

5. `N.Valid`: number of observations that is not a missing value.

6. `Pct.Valid`: percentage of the observations that is not a missing value.

Additionally, we can get descriptive statistics based on certain variables. For example, we can get the descriptive statistics of the numerical variables in the `iris` dataset according to the `species`.

```
iris %>%
  group_by(Species) %>%
  descr(Petal.Length)
```

```
Descriptive Statistics
Petal.Length by Species
Data Frame: iris
N: 150
```

	setosa	versicolor	virginica
Mean	1.46	4.26	5.55
Std.Dev	0.17	0.47	0.55
Min	1.00	3.00	4.50
Q1	1.40	4.00	5.10
Median	1.50	4.35	5.55
Q3	1.60	4.60	5.90
Max	1.90	5.10	6.90
MAD	0.15	0.52	0.67
IQR	0.18	0.60	0.78
CV	0.12	0.11	0.10
Skewness	0.10	-0.57	0.52
SE.Skewness	0.34	0.34	0.34
Kurtosis	0.65	-0.19	-0.37
N.Valid	50.00	50.00	50.00
N	50.00	50.00	50.00
Pct.Valid	100.00	100.00	100.00

For the descriptive statistics for the categorical variables, we can use the `freq()` function which calculates count and percentage.

```
# Descriptive statistics for categorical data
freq(iris$Species)
```

```
Frequencies
iris$Species
Type: Factor
```

	Freq	% Valid	% Valid Cum.	% Total	% Total Cum.
setosa	50	33.33	33.33	33.33	33.33
versicolor	50	33.33	66.67	33.33	66.67
virginica	50	33.33	100.00	33.33	100.00
<NA>	0			0.00	100.00
Total	150	100.00	100.00	100.00	100.00

Lastly, we can do a cross-tabulation between two categorical variables. To demonstrate this function, we need to create an additional categorical variable for the `iris` dataset.

```
# Create a new categorical variable
dat <-
  iris %>%
  mutate(Sepal_Length_Cat = ifelse(Sepal.Length > 5, "large", "small"))

# Cross-tabulation
ctable(dat$Sepal_Length_Cat, dat$Species)
```

```
Cross-Tabulation, Row Proportions
Sepal_Length_Cat * Species
Data Frame: dat

------------------ --------- ------------ ------------ ------------ --------------
                     Species       setosa   versicolor    virginica          Total
  Sepal_Length_Cat
             large             22 (18.6%)   47 (39.8%)   49 (41.5%)   118 (100.0%)
             small             28 (87.5%)    3 ( 9.4%)    1 ( 3.1%)    32 (100.0%)
             Total             50 (33.3%)   50 (33.3%)   50 (33.3%)   150 (100.0%)
------------------ --------- ------------ ------------ ------------ --------------
```

The first variable will appear as row names and the second variable will appear as column names. In our code, `Sepal_Length_Cat` will appear as a row, and `Species` will appear as a column.

9.4.2 gtsummary

The `gtsummary` package provides an easy way to create a publication-ready summary table. We going to use the `iris` dataset to demonstrate the capabilities of `gtsummary`.

```
# Load packages
library(gtsummary)

# Create a summary table
tbl_summary(data = iris)
```

We can further adjust this summary table. For example, we can:

1. Change the median (Q1, Q3) to the mean and standard deviation.

Characteristic	**N = 150**[1]
Sepal.Length	5.80 (5.10, 6.40)
Sepal.Width	3.00 (2.80, 3.30)
Petal.Length	4.35 (1.60, 5.10)
Petal.Width	1.30 (0.30, 1.80)
Species	
setosa	50 (33%)
versicolor	50 (33%)
virginica	50 (33%)

[1]Median (Q1, Q3); n (%)

Characteristic	**N = 150**[1]
Sepal.Length	5.8 (0.8)
Sepal.Width	3.1 (0.4)
Petal.Length	3.8 (1.8)
Petal.Width	1.2 (0.8)
Species	
setosa	50.0 (33.3%)
versicolor	50.0 (33.3%)
virginica	50.0 (33.3%)

[1]Mean (SD); n (%)

2. Change all values to 1 decimal point.

```
tbl_summary(
  data = iris,
  statistic = all_continuous() ~ "{mean} ({sd})", # Use mean and standard deviation
  digits = list(all_continuous() ~ 1, # numerical variables to 1 decimal place
                all_categorical() ~ 1) # categorical variable to 1 decimal place
)
```

9.5 Chapter summary

In this chapter, we have covered the basic descriptive statistics, which is consist of:

1. Measures of central tendency.

2. Measures of variability.

We have learned how to interpret and implement each measure in R. Additionally, we have covered two useful R packages for descriptive statistics:

1. `summarytools`
2. `gtsummary`

For more information on `summarytools` (Comtois 2022), refer to its official documentation. Similarly, detailed guidance on `gtsummary` (Sjoberg et al. 2021) is available on its documentation page.

9.6 Revision

1. Explain measures of central tendency.
2. Explain measures of variability.
3. What is the relationship of outliers with mean, median, and mode?
4. Explain the differences between sample variance and population variance.
5. Read about `airquality` data by typing `?airquality` in the `Console`. Next, using the data:

```
data("airquality")
```

   a. Find the median for each numerical variable in the dataset.
   b. Using `summarytools`, get descriptive statistics for all variables.
   c. Create a summary table using the `gtsummary` package.

6. Read about `CO2` data by typing `?CO2` in the `Console`. Next, using the data:

```
data("CO2")
```

   a. Get descriptive statistics for `uptake` stratified by `Type`.
   b. Create cross-tabulation between `Type` and `Treatment` with the former as a row and the latter as a column.

10 A way forward

This book has provided a strong foundation for mastering R in the context of data analysis. However, R is an expansive tool with applications in numerous fields, and there are many avenues for further exploration once you have completed this book.

Generally, the field of data analysis is diverse and continually evolving. Here are three major areas that I can suggest to consider as part of your journey forward:

1. Statistical Analysis and Modeling
   Statistical analysis forms the backbone of data-driven decision-making. It involves descriptive techniques to summarize data and inferential methods to draw conclusions and test hypotheses. With R, you can perform regression analysis, ANOVA, time series forecasting, and multivariate analysis, among others.

2. Machine Learning
   Machine learning builds on statistical methods to create algorithms capable of learning from data. R offers robust tools for machine learning, including packages like `caret`, `tidymodels`, and `mlr3`. These tools support tasks such as classification, clustering, and predictive modeling. R's integration with deep learning libraries, like `tensorflow`, `keras`, and `torch`, also enables work in neural networks and reinforcement learning.

3. Data visualisation and communication
   Effective communication of results is key in data analysis. R's visualization capabilities, through tools like `ggplot2`, `plotly`, and `shiny`, allow analysts to create dynamic, interactive, and publication-ready visualizations. Mastering these tools helps ensure that insights are clearly conveyed to both technical and non-technical audiences.

Wrapping up, mastering R isn't just about knowing the basics—it's your ticket to exploring a ton of exciting opportunities in data analysis and beyond. Whether you're diving into statistical analysis to uncover trends or exploring the cutting-edge world of machine learning, R has got you covered. Plus, its powerful tools for creating stunning visualizations and tackling big data make it a must-have for any data enthusiast.

As you build on the foundation you've gained, think about where you want to specialize. Maybe it's predictive modeling, visual storytelling through data, or even solving big, messy real-world problems. The possibilities are endless—so keep learning, experimenting, and pushing boundaries.

References

Comtois, Dominic. 2022. *Summarytools: Tools to Quickly and Neatly Summarize Data*. https://CRAN.R-project.org/package=summarytools.

Cui, Boxuan. 2024. *DataExplorer: Automate Data Exploration and Treatment*. http://boxuancui.github.io/DataExplorer/.

Hair, Joseph F., G. Tomas M. Hult, Christian M. Ringle, and Marko Sarstedt. 2022. *A Primer on Partial Least Squares Structural Equation Modeling (PLS-SEM)*. SAGE Publications, Inc.

Hyndman, Rob J., and Yanan Fan. 1996. "Sample Quantiles in Statistical Packages." *The American Statistician* 50 (4): 361–65. https://doi.org/10.1080/00031305.1996.10473566.

Kowarik, Alexander, and Matthias Templ. 2016. "Imputation with the r Package VIM." *Journal of Statistical Software* 74 (7): 1–16. https://doi.org/10.18637/jss.v074.i07.

Ryu, Choonghyun. 2024. *Dlookr: Tools for Data Diagnosis, Exploration, Transformation*. https://CRAN.R-project.org/package=dlookr.

Sjoberg, Daniel D., Karissa Whiting, Michael Curry, Jessica A. Lavery, and Joseph Larmarange. 2021. "Reproducible Summary Tables with the Gtsummary Package." *The R Journal* 13: 570–80. https://doi.org/10.32614/RJ-2021-053.

Tierney, Nicholas, and Dianne Cook. 2023. "Expanding Tidy Data Principles to Facilitate Missing Data Exploration, Visualization and Assessment of Imputations." *Journal of Statistical Software* 105 (7): 1–31. https://doi.org/10.18637/jss.v105.i07.

Waring, Elin, Michael Quinn, Amelia McNamara, Eduardo Arino de la Rubia, Hao Zhu, and Shannon Ellis. 2024. *Skimr: Compact and Flexible Summaries of Data*. https://docs.ropensci.org/skimr/.

Wickham, Hadley. 2016. *Ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York. https://ggplot2.tidyverse.org.

Wickham, Hadley, Mara Averick, Jennifer Bryan, Winston Chang, Lucy D'Agostino McGowan, Romain François, Garrett Grolemund, et al. 2019. "Welcome to the tidyverse." *Journal of Open Source Software* 4 (43): 1686. https://doi.org/10.21105/joss.01686.

Wickham, Hadley, Mine Çetinkaya-Rundel, and Garrett Grolemund. 2023. *R for Data Science*. O'Reilly Media, Inc.

Wickham, Hadley, and Garrett Grolemund. 2017. *R for Data Science*. O'Reilly Media, Inc.