R for Research

2024-09-11

Table of contents

Welcome

Welcome to **R for Research**, a book written to give you a strong introduction to R programming language and its uses in research workflow. The primary goal of **R for Research** is to provide a clear and practical guide to using R for your entire research workflow. While R is practically seen as a programming language for statistical computing, it should be noted that since the development of Sweave (Leisch 2002), and R Markdown, Xie, Allaire, and Grolemund (2018), R use has gone beyond statistical analysis. Using these package within RStudio IDE, gives you a powerful word-processor and statistical program.

This book will be under development for a while; expected completion should be around early 2027. Do not be surprised if a lot of the chapters do not cohere. You are welcome to provide suggestions and give corrections through github issues. If you find anything that could improve the work and would also like to collaborate do not hesitate to reach me via email.

Cover image, a Red-legged Patridge walking in Snowy Van, in Turkiye is by ezvedat on the pexels platform. All pexels images are free to use.

Preface

In today's research landscape, the ability to analyze data effectively is crucial. Whether in academia, industry, or governmental research, data-driven decisions shape outcomes. R, a powerful programming language for statistical analysis and data visualization, has emerged as a cornerstone in this field.

This book, **R for Research**, is designed for researchers, data scientists, and students looking to leverage R for a variety of analytical tasks. From data manipulation to visualization, and advanced modeling, this guide provides the foundational knowledge to transform raw data into actionable insights.

The journey of mastering R may feel daunting at first, but with the right approach and persistence, it quickly becomes intuitive and rewarding. Through hands-on examples, practical exercises, and detailed explanations of core concepts, this book will take you from a beginner to an adept user, ready to tackle complex research problems with confidence.

Whether you're aiming to publish a research, gain insights from large datasets, or deepen your understanding of the data you are working on, this book equips you with the tools and skills to succeed in your endeavors. The book is beginner friendly, so you can follow along with no coding experience. Since R is statistics software, basic knowledge of statistics is required to follow along. Efforts will be made to explain some statistical concepts, but this will not be in all cases.

By the end of this book, readers will have:

- a solid understanding of R's core functionalities for data analysis.
- proficiency in using key R packages for data manipulation, visualization, and statistical modeling.
- practical experience through real-world examples and exercises that are relevant to various research fields.
- the ability to integrate R into your research workflow, allowing for reproducible and transparent analyses.

Ultimately, R for Research is a beginner oriented book designed to introduce you to data analysis with R, helping you transform raw data into meaningful insights that drive impactful research.

Introduction

R has become one of the cornerstone in the data science community and its use in academia is increasing daily. This is largely due to its flexibility, open-source nature, and extensive statistical capabilities. Originally developed in the early 1990s by statisticians Ross Ihaka and Robert Gentleman, R was designed as a language specifically for data analysis (Ihaka and Gentleman 1996). Over time, it evolved into one of the most popular tools for researchers, data scientists, and statisticians alike.

R and Academia

In academia, R holds a unique position due to its roots in statistical analysis. Many academic researchers favor R for its comprehensive set of statistical tools, making it ideal for disciplines like economics, psychology, biostatistics, pharmacology, life sciences, nature management, environmental sciences, and courses where complex data analysis is routine. The fact that R is free and open-source has contributed to its popularity in academic settings, where budgets can be tight, and access to proprietary software may be limited, even without budget limits, its capabilities rivals a lot of proprietary software.

R is particularly favored in research , with increased popularity in finance, healthcare and life sciences. Universities often integrate R into their curricula in courses such as bioinformatics, silviculture, biometrics, and even social sciences, making it a key part of training the next generation of data scientists and researchers across discipline. While there's a common misconception that not all fields require sophisticated analytical capabilities, this is demonstrably incorrect. The bedrock of robust research is rigorous data analysis and interpretation, regardless of the specific domain.

In academic research, reproducibility is a crucial aspect, and R excels in this area. In addition to that, R through the RStudio IDE allows users create dynamic reports that integrate code, data, and narrative in a single document. This allows researchers to ensure that their analyses are transparent and reproducible by others, a key requirement for peer-reviewed publications. Reports, research papers, presentations, and even books can be generated directly from RStudio (R's associated and most popular integrated development environment - IDE), providing a streamlined workflow for sharing results.

The R Community and Ecosystem

One of R's most significant strengths is its vibrant and growing community. This is made through the efforts of the R Development Core Team responsible for the ongoing development and maintenance of the R language and environment, and Posit PBC, the creator of RStudio and Positron which sponsors and develops open-source technologies. The Comprehensive R Archive Network (CRAN), a central repository for R packages, contains over 21000 packages as at the time of this book was written, with contributions from thousands of developers worldwide. These packages extend R's functionality to virtually every field in the world. This ecosystem makes R adaptable to a broad range of industry, enabling users to apply cutting-edge techniques to their data problem.

The vibrant R community, often identified with the hashtag **#rstats** is also known for its strong support culture. Forums like Stack Overflow, R-bloggers, Linkedin, X (formerly Twitter) and Posit Community provide a wealth of knowledge and resources, ensuring that newcomers and experienced users alike can find help when they encounter challenges. This highly collaborative environment is a significant driver of R's enduring popularity, enabling users to efficiently overcome obstacles and continuously learn from the collective expertise of their peers.

Why Learn R

- Data Analysis and Statistics: R was developed specifically for statistical computing, making it ideal for data analysis, modeling, and visualization.
- Rich Ecosystem: With thousands of packages available via CRAN, , R offers tools for everything from machine learning to bioinformatics.
- Reproducible Research: Packages like `RMarkdown` help in creating reproducible reports, crucial for academia and industry.
- Visualization: R excels in producing publication ready visuals.
- Growing Popularity in Data Science: Many industries, including finance, healthcare, and tech, use R for data-driven decision-making.

Why Not R?

Despite R's strengths, there are scenarios where it may not be the best fit:

- **Performance Limitations**: R is not the fastest of programming language as it was deliberately designed to prioritize ease of use for humans instead of computational efficiency (Wickham 2019). This design choice reflects R's fundamental philosophy of making data analysis and statistics easier for users, even if it means the computer has to work harder . To increase the performance of R, using packages like `data.table`, `parallel`, `duckdb`, `arrow` and `future` can be helpful.
- **Learning Curve**: While R's syntax is intuitive for statistical tasks, its learning curve can be steep for those unfamiliar with its unique paradigms. Tasks beyond basic data analysis may require more in-depth coding skills.
- **Less Versatility**: R is heavily focused on data analysis and statistics, making it less suited for general-purpose programming. Its capabilities outside of data science, machine learning, and statistical computing are continually be addressed with packages.
- **Package Dependency**: Although R has a vast library of packages, their quality can vary. Some packages might be poorly maintained or have compatibility issues with newer R versions.
- **Integration**: R is not always the first choice for web development, app development, or integration with production systems. While solutions exist (such as Shiny for web apps), these use cases are generally more efficient in other languages like Python or JavaScript.

Alternatives to R

- **Python**: a popular alternative, known for its versatility beyond data science. It has extensive libraries for machine learning (e.g., scikit-learn, TensorFlow), data manipulation (pandas), and visualization (matplotlib, seaborn). Python's general-purpose nature makes it suitable for both data analysis and broader applications like web development, app development and automation.
- **Julia**: Julia is emerging as a high-performance language designed for numerical and scientific computing. It offers speed advantages over R and Python, particularly in tasks that involve heavy computation.
- **SAS**: Commonly used in industries such as healthcare, SAS is a robust tool for statistical analysis with a long history in academia and corporate sectors. It provides a stable environment but comes with high licensing costs compared to R's open-source nature.
- **SPSS**: SPSS is another statistical tool widely used in academic research and business analytics. Like SAS, it's user-friendly for statistical analysis but is expensive and less flexible than R or Python.

- **Excel / Googlesheets**: Spreadsheets are one of the fundamental tools in data analysis, but they come with significant limitations. Regardless of this, they can perform some tasks well, and are easy to use.

Each of these alternatives has their strengths, and the choice depends on the project requirements, performance needs, and user expertise. I have often found myself falling back to R during data analysis.

Note

While R was originally designed with a focus on statistical analysis, it has evolved into a full-fledged programming language with comprehensive capabilities beyond statistics. Its extensive package ecosystem ensures robust problem-solving, improved workflow, and enhanced communication of findings, including computing dynamic results.

Scope, Limitations, and Expectations

Scope

R for Research covers essential techniques for data analysis using R, focusing on the needs of researchers across various disciplines. It introduces readers to R programming, data manipulation, statistical analysis, and visualization, with practical examples and case studies from academic research. The book is structured to help readers at any experience level, from beginner to intermediate, make effective use of R in their research.

Limitation

While the book provides a comprehensive introduction to R, it does not cover advanced topics such as machine learning, deep learning, books like Hands-On Machine Learning with R by Bradley Boehmke and Deep Learning with R by François Chollet and Joseph J Allaire would provide sufficient knowledge on these topics. Field specific use of R like its application for genomics or financial modeling are not covered here, Computational genomics with R by Altuna Alkalin is a good resource for people interested in genomics. The focus of this book is on R's use for general research, so readers looking for highly specialized content may need to supplement their learning with some of the advanced resources suggested.

What should you expect to learn from reading this books?

By the end of this book, readers are should be able to:

- understand R's core functions for data manipulation and visualization.
- apply R to basic and complex statistical analyses and data preparation.
- integrate R into their research workflow for reproducible and transparent analysis.
- have confidence in handling real-world data sets for various research contexts, but be aware that mastering complex, domain-specific analyses may require further study.

How This Book is Organized

Readers are firstly introduced to the basics of R covering the foundational aspect of R as a programming language. Next is the story mode section, that will set the stage for using R in a Research setting. These aspect will cover concepts we will mostly likely encounter in everyday routine, such as, importing data, visualization, conducting analysis, and cleaning data among others.s

! Important

still under development.

Part I

Part I: The Basics

Tip

This section is under review.

The first part of this book introduces R as a language to you. It focuses on the fundamental concepts needed to follow through the other parts of the books. If R is your first programming language or are unfamiliar with the R Syntax, this chapter will be a gentle introduction to R, else you can skip to Part II.

Chapter 1 introduces R, RStudio IDE, Posit Cloud, and the Positron IDE in . After that, the syntax and some common operations are introduced in Chapter 2. The goal here is to get you familiar with R as fast as possible. Next is the data types in Chapter 3 and this is one of the key building block in any programming language. Chapter 4 gets you acquainted to performing some operation on the different data-types, and Chapter 5 shows how the different data types can be organized for data analysis. This continues with Chapter 6, which introduces you to conditionals and loops which provides tools to control our output based on criteria which we set. To round up Part I, we aggregate the knowledge of this part and get a guide on how to initiate a project in Chapter 8.

Each chapter builds upon previous concepts, ensuring a solid foundation for more advanced topics in later parts of the book. The progression is designed to get you comfortable with R as quickly as possible while ensuring you understand the underlying concepts that are crucial for effective data analysis.

1 R, RStudio, and Posit Cloud

ℹ Note

You can skip all the whole of 1.1 if you already have R and RStudio installed. If not, proceed to the operating system loaded in your machine at 1.1.1.

1.1 Installing R and RStudio

R and RStudio are free to use and install software. The guide below provides the steps to install both software.

1.1.1 Installing R

Windows OS (10/ 11): Download R-4.4.2 for Windows and run the installer. This is the latest version at the time of writing this chapter.

Mac OS: There are two versions of R for Mac users based on the processor in your computer.

- For M1, M2, and recent M__ processors: Download R-4.4.2 for Mac M-Processor and run the installer
- For Intel processors: Download R-4.4.2 for Mac Intel processor and run the installer.

Linux Users R installation on Linux depends on your distribution. Open terminal (`Ctrl` + `Alt` + `T`) and enter the command below based on your distribution.

- Debian/Ubuntu distribution:

```
#| eval: false
$ sudo apt update
$ sudo apt install r-base r-base-dev
```

- Redhat/Fedora distribution:

```
#| eval: false
$ sudo dnf install R
```

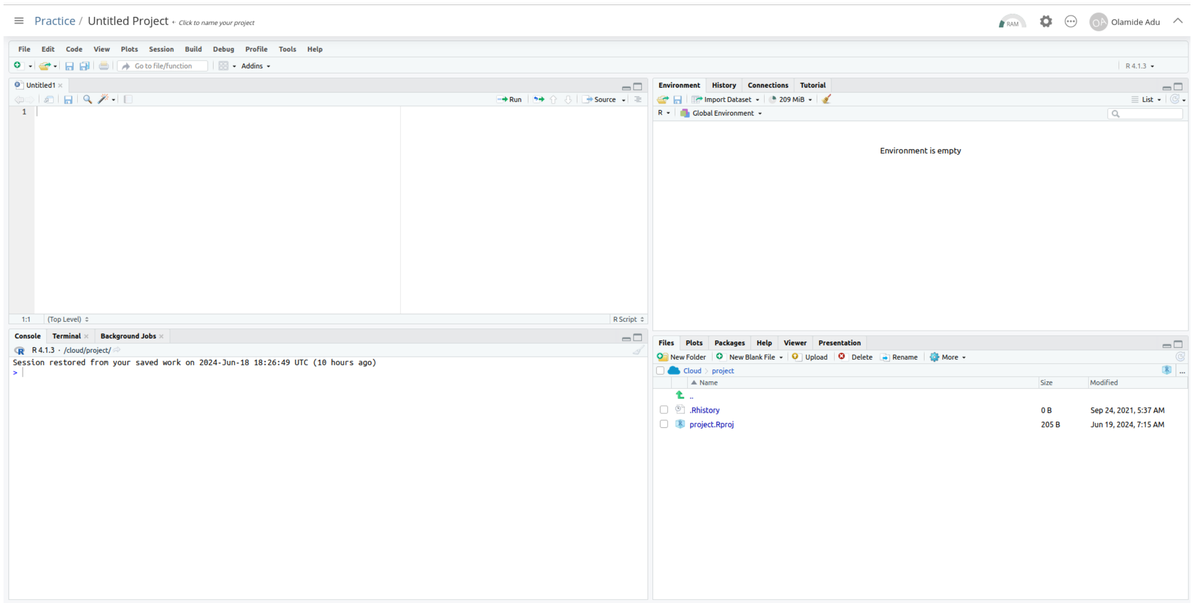
1.1.2 Installing RStudio

To install RStudio, visit the link https://posit.co/download/rstudio-desktop/ scroll down and download the package for your OS, and install. Installation of downloaded file is direct. If you prefer a video guide, choose the link based on your OS:

- window
- Mac
- linux-ubuntu

1.2 The RStudio Interface

When you open RStudio newly, you are presented with three panes by default. To get a fourth pane like what is shown in Figure 1.1, click the + button at the top-left and select R Script, or use the keyboard shortcut `Ctrl + Shift + N` on Windows/Linux and `Cmd + Shift + N` on Mac.


Practice / Untitled Project · Click to name your project
Olamide Adu
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
R 4.1.3
Untitled1
Run
Source
(Top Level)
R Script
Console Terminal Background Jobs
Environment History Connections Tutorial
Import Dataset
209 MiB
List
Global Environment
Files Plots Packages Help Viewer Presentation
New Folder New Blank File Upload Delete Rename More
Cloud > project

```
R 4.1.3 · /cloud/project/
Session restored from your saved work on 2024-Jun-18 18:26:49 UTC (10 hours ago)
>
```

Environment is empty

Name	Size	Modified
..		
.Rhistory	0 B	Sep 24, 2021, 5:37 AM
project.Rproj	205 B	Jun 19, 2024, 7:15 AM

Figure 1.1: R Studio Environment

You can change the arrangement of the panes by clicking `Tools` on the menu bar then navigate to `Global Options` and `Pane Layout` in the popup box.

1.2.1 RStudio Panes

The four components/panes shown in Figure 1.1 are:

1. **Source Pane**: Located in the top-left. This pane shows after adding a R script. The R script is where you will do most of your data analysis task. It is technically a plain text file with a .R extension containing a series of R commands that are executed in sequence. It also serves as an editor for other programming languages. The source pane displays the dataset in spreadsheet-like format.

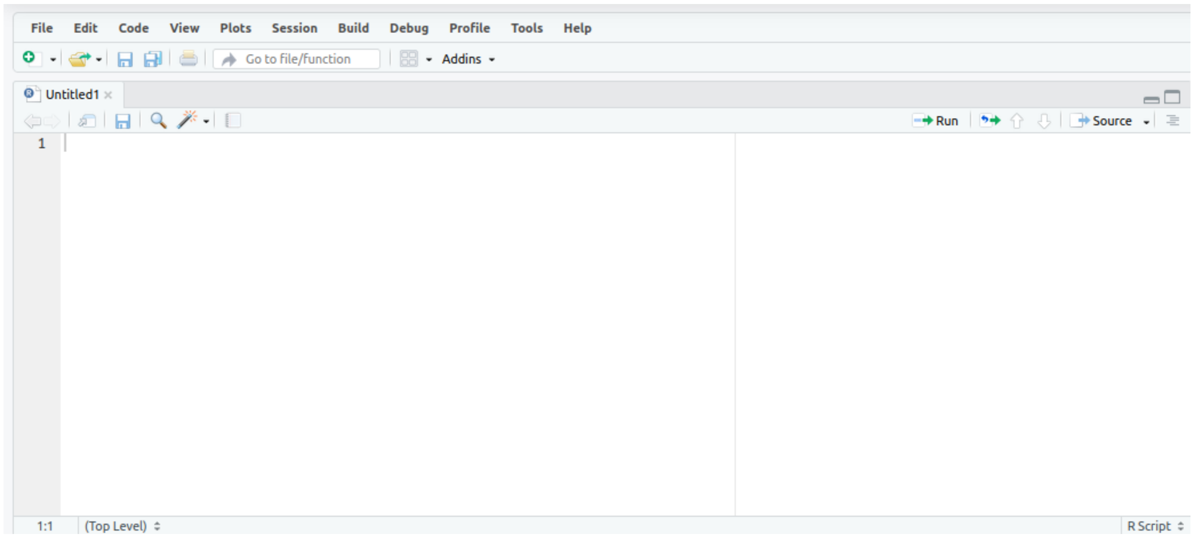
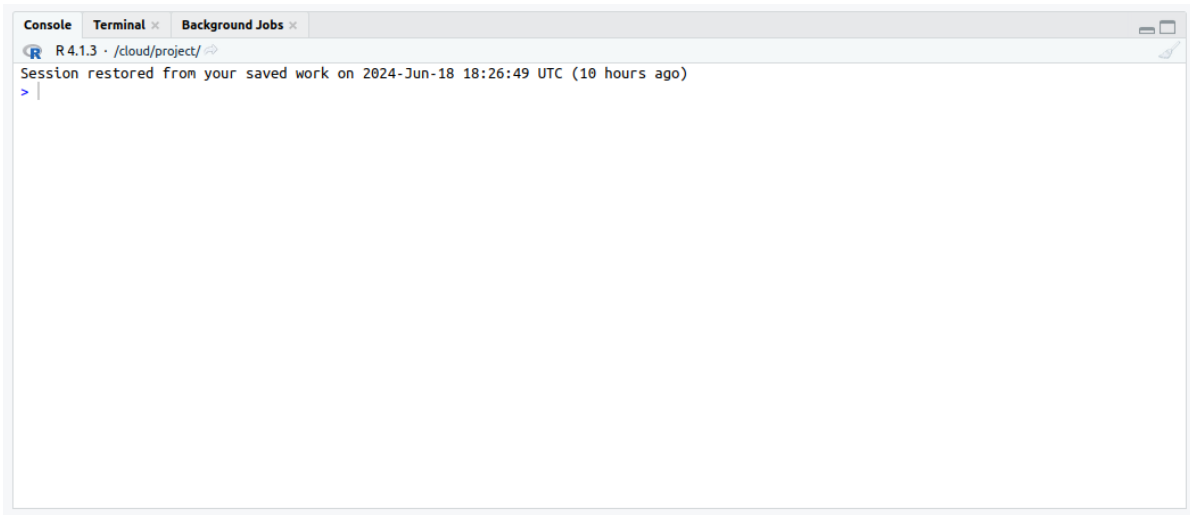


Figure 1.2: Source window

2. **Console/Terminal/Background Jobs pane**: Located in the bottom-left. It include the **Console** tab where outputs and results of your analysis are displayed. The console is also an interactive section where you can write code and carry out some analysis. All the codes written in the script are usually displayed and executed on the console. This pane also include the **Terminal** tab which provides direct access to your computer through a command line interface. Lastly, we have the **Background Jobs** tab which allows you to run tasks in the background while you continue working on other tasks in the main R session. This pane is especially useful for long-running operations that might otherwise block your R session, such as data processing, simulations, or model fitting. You can read more on background jobs on RStudio User Guide.



```
Session restored from your saved work on 2024-Jun-18 18:26:49 UTC (10 hours ago)
>
```

Figure 1.3: Console window

3. **Environment/History/Connection/Tutorial Pane**: Located in the top-right. It includes the **Environment** tab which displays all objects created during a session, such as data and variables. It also includes some important features to import data, load and save a workspace and clear objects in an active workspace. We also have the **History** tab which stores the history of commands executed. This tab allows you to select previous commands and rerun them or add them back to console if you want. Located closely to **History** tab is the **Connections** tab which helps you interact with and manage connections to external data sources, such as databases, cloud services, or remote data systems. It displays the connections you have made to data sources, and lets you know which of them are active. After the **Connections** tab is the **Tutorial** tab which provides additional learning materials for R and RStudio. It is designed to be an interactive R tutorial and it runs on shiny and learnr package (You will learn about packages later).

An R session refers to the instance of R that is currently running, which includes everything that is active in memory and the state of your environment in RStudio or any R interface.

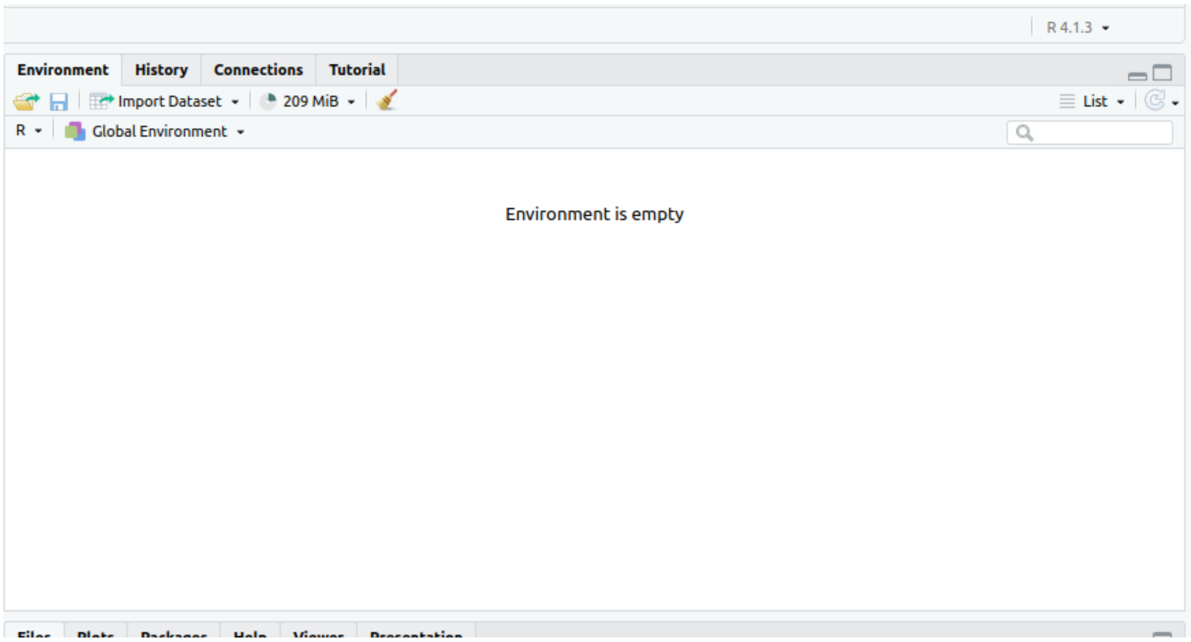
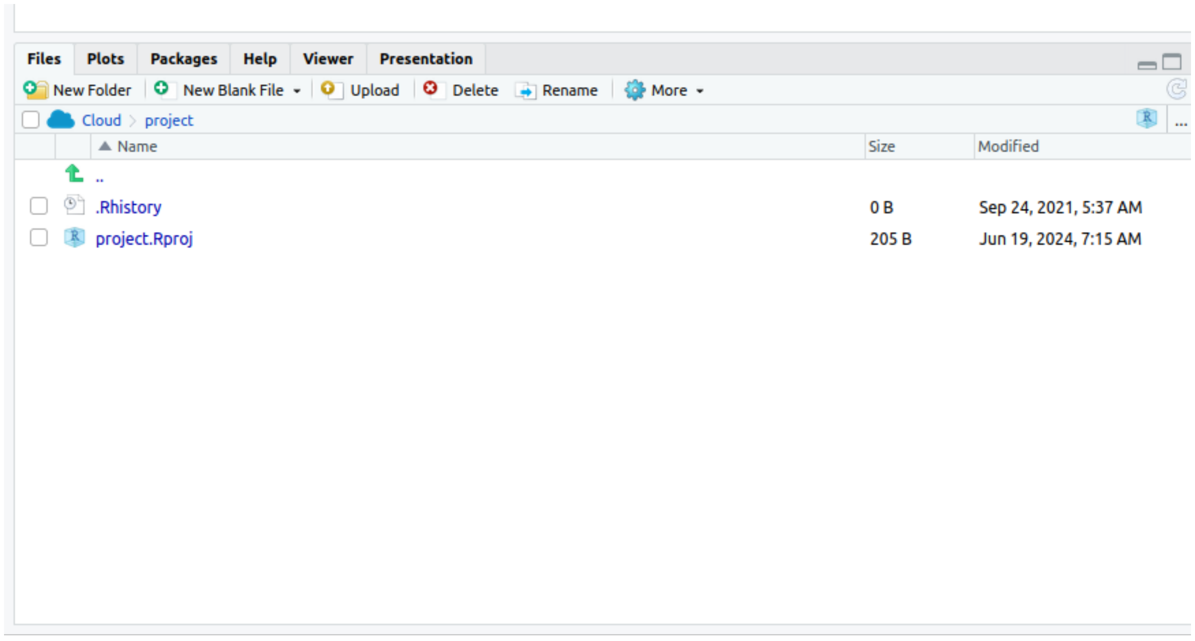


Figure 1.4: Environment window

4. **Files / Plots / Packages / Help / Viewer Window**: This is located in the bottom-right. The **Files** tab shows files and folders in your working directory. It also include buttons for renaming, deleting, creating, copying and moving files and folders just like your native file manager. The **Plots** tab is next and it displays visualizations, providing the option to zoom in or out, and copy to clipboard graphs created. The **Packages** tab follows closely and it helps to manage packages you have installed. It provides the option to update, install and remove packages through a GUI. Next is the **Help** tab, and it houses the documentation of functions, and vignettes for different packagesred. Next is the **Viewer** tab which is used to display a variety of visual outputs including interactive web content, plots, maps, rendered websites, and web-based applications. Lastly is the presentation tab which is used to view slides created through the R Markdown ecosystem.



Name	Size	Modified
..		
.Rhistory	0 B	Sep 24, 2021, 5:37 AM
project.Rproj	205 B	Jun 19, 2024, 7:15 AM

Figure 1.5: Files window

Script vs Console

It is advised to use a script to ensure reproducibility. It ensures you can edit your work when you make errors, share with others your analysis, and also reproduce all steps in your analysis instantly.

1.3 Posit Cloud

There's an option to use R and RStudio online from your browser. This is through Posit Cloud (formerly RStudio Cloud). Posit cloud is owned by the same organization that made RStudio. To use posit cloud, visit https://posit.cloud/ and register. After registering, you get a page similar to Figure 1.6, then click on **new space** on the left sidebar then **new project** on the right side to start a new R session. Options to start from a git repository and a template file is also provided.

The video series in this link is a good resource for starting with Posit Cloud.

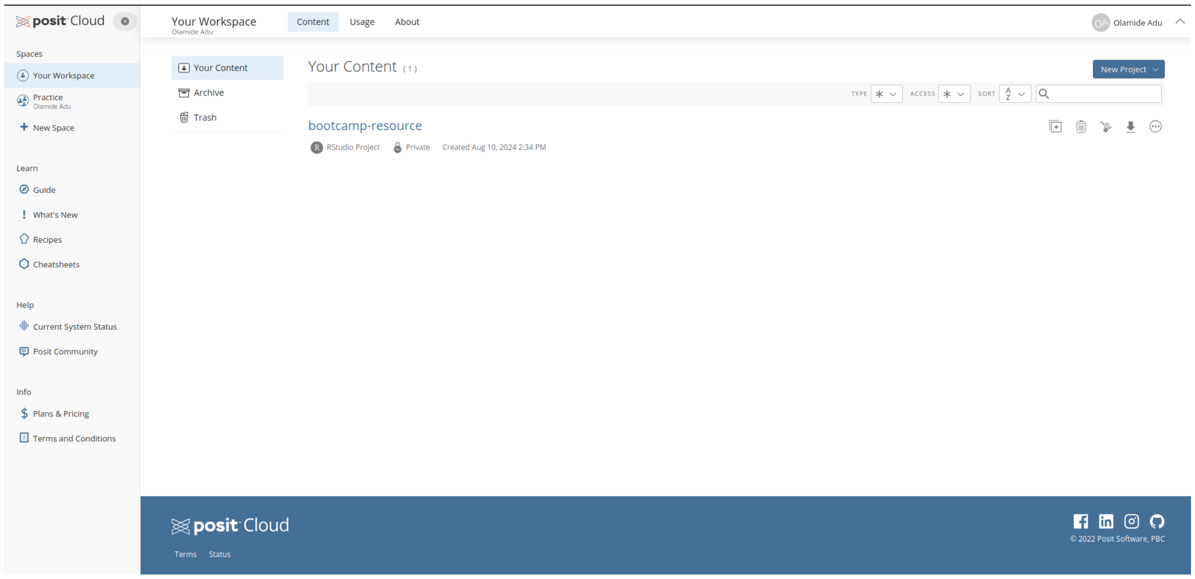
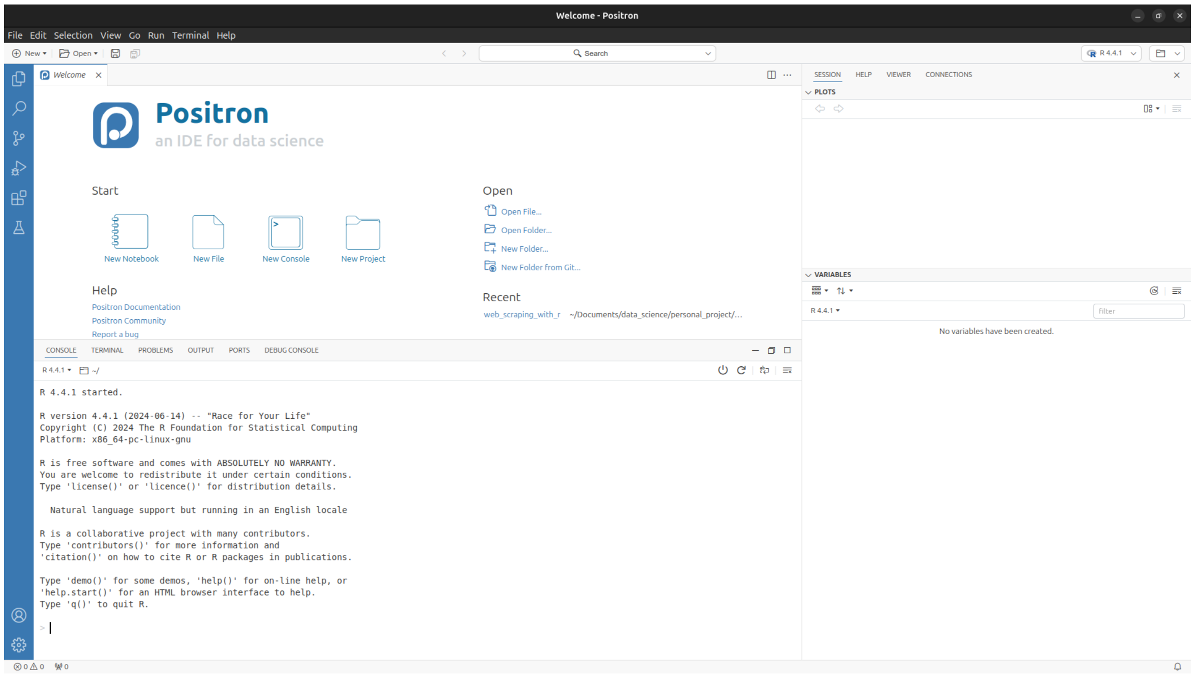


Figure 1.6: posit cloud

1.4 Positron

There's a new IDE that supports Python and R that is still under development by the company Posit PBC. This is called Positron. Positron is still in its early stage of development. Positron is a blend of the popular IDE VS Code and RStudio, with the major inspiration from VS Code, (Radečić 2024). What makes Positron different is the ease of using python, which is one of the most common programming language of choice for data professionals. To get more details about the development, click here and here for detailed introduction on Positron.



```
R 4.4.1 started.

R version 4.4.1 (2024-06-14) -- "Race for Your Life"
Copyright (C) 2024 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

Figure 1.7: Positron IDE

Summary

In this chapter we covered how to install R, and RStudio. We also introduced the different panes/windows of RStudio IDE which includes, the source, environment, console and files panes. Lastly, we introduced the Posit cloud platform, a cloud-based RStudio platform as well as Positron, which is the next iteration of IDE from Posit.co

2 Operators in R

This chapter introduces some of the common operations you can do with R. To achieve this, we will use R as a simple calculator using its operators and getting a view of how its statements are structured, i.e., its syntax. Understanding the basic syntax is essential for using R for solving your pressing problem be it a simple task such as adding two numbers or performing more complex operations. This codes in this chapter are simplified one-liners and are easy to follow. With the goal being to give you a feel of the language, it will cover basic operations including arithmetic operations such as, addition, subtraction, division, and the likes, comparison operations which involves comparing two values.

2.1 Arithmetic Operators

Operations such as addition, subtraction, multiplication and so on, are easy to execute in R. The arithmetic operators are given below, they are the same as symbols used in elementary mathematics with only few changes:

Table 2.1: Arithmetic operators in R

Operator	Definition
+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Exponent or Power
**	Exponent or Power
%%	Remainder division
%/%	Integer division

Run the following commands in your console:

```
2 + 1
```

```
[1] 3
```

```r
3 - 1
```

```
[1] 2
```

```r
4 * 2
```

```
[1] 8
```

```r
15 / 5
```

```
[1] 3
```

```r
3 ^ 2
```

```
[1] 9
```

```r
3 ** 2
```

```
[1] 9
```

The above operations are straightforward. `%%` operator returns the remainder when two numbers are divided. While `%/%` returns the whole number or integer and discard the remainder.

```r
5 %% 3
```

```
[1] 2
```

```r
5 %/% 2
```

```
[1] 2
```

R also follows the order of operations.

```r
5 * 15 + 14 / 7
```

```
[1] 77
```

The order of operation here is division, `14 / 7 = 2`, followed by multiplication, `5 * 15 = 75`, then addition of the values from the former operations `75 + 2` which is evaluated to give a total of 77. To avoid confusion, a convenient way to go about how you want your numbers to be evaluated is by using parenthesis () to block codes.

```
3 + (5 - 2)
```

```
[1] 6
```

```
(5 * 4) / (7 - (2 + 4))
```

```
[1] 20
```

Tip

Can you notice the space between numbers and the operators. While not required, such spacing makes your code readable. Don't be surprised that a lot of times you will read your codes. Also, if you work with in a team or want to share your work, people will be reading your codes.

2.2 Comparison Operators

Comparison operators are as their names imply, they are used to compare values. The result from their operation returns either TRUE or FALSE. You will get more information on this when we get to Chapter 3 in the next chapter.

Comparison operator in R include

Table 2.2: Comparison Operators in R

Symbol	Meaning
>	Greater than
<	Less than
==	Equal to
>=	Greater than or equal to
<=	Less than or equal to
!=	Not equal to

Try these commands in R's console:

```
4 > 5
```

```
[1] FALSE
```

```
(5^2) < (5 * 2)
```

```
[1] FALSE
```

```
10 >= 7
```

```
[1] TRUE
```

```
10 == 12
```

```
[1] FALSE
```

Multiple comparisons can be combined using operator `&` (and), `|` (or), and `!` (not).

2.3 AND (&), | (OR), and !(NOT) Operators.

- AND (`&`) operator returns TRUE when **all** conditions on both its sides of are TRUE. All other conditions returns FALSE

Table 2.3: Combination of operations with `&` (And) operator

Conditions	result
TRUE & TRUE	TRUE
TRUE & FALSE	FALSE
FALSE & TRUE	FALSE
FALSE & FALSE	FALSE

```
28 > 10 & 50 > 40
```

```
[1] TRUE
```

```r
20 < 10 & 50 < 40
```

```
[1] FALSE
```

```r
20 > 10 & 50 < 40
```

```
[1] FALSE
```

- OR (`|`) returns TRUE when **one** of the conditions on both sides of it are TRUE.

Table 2.4: Combination of operations with `|` (OR) operator

Conditions	result
TRUE | FALSE	TRUE
FALSE | TRUE	TRUE
TRUE | TRUE	TRUE
FALSE | FALSE	FALSE

```r
20 > 15 | 30 > 10
```

```
[1] TRUE
```

```r
20 < 15 | 30 < 10
```

```
[1] FALSE
```

You can also use parentheses to enclose conditions when testing multiple conditions.

```r
(20 < 15) | (30 > 10)
```

```
[1] TRUE
```

- `!` is used to negate, and returns the opposite of a result. This is an operator that can be very useful in complex operations. You will see this being used as we continue.

Table 2.5: Result from negating logicals using `!` operator

Conditions	result
!TRUE	FALSE
!FALSE	!TRUE

```r
!TRUE
```

```
[1] FALSE
```

```r
!((20 < 15) | (30 > 10))
```

```
[1] FALSE
```

2.4 Assignment Operators

The assignment operator is used to bind values to names/labels called *variables*. These names, variables, are used to reference values.

Table 2.6: Assignment Operators in R

Symbol	Meaning
<-	Assignment / bind to value
=	Assignment / bind to value
->	Rightward assignment / bind left value to the right variable

For example, we can assign the value `17` to the variable `c`. Whenever `c` is called, the value 17 is returned.

```r
c <- 17
c
```

```
[1] 17
```

This does not mean that the value(s) assigned to a particular variable cannot be changed. They can be changed easily by assigning a new value to the name.

```r
c <- "man"
```

Now the value `man` is assigned to `c` using `<-` operator. This is the convention used for assigning objects to variables in R. We can assign values to variables using other operators too, but they have specific use. For example while we can assign the value `23` to the variable weight using `=` as shown below. The `=` operator is best used within functions.

```r
weight = 23
weight
```

```
[1] 23
```

::: Functions are collections of codes that perform certain actions, more on that in Chapter 4. :::

For `->` operator, the variable is at the right side of the operator.

```r
20 * 20 -> plot_size
plot_size
```

```
[1] 400
```

2.5 Making Comments

There are times when we do not want R to executes some lines or text when writing our analysis code. We can instruct the program to skip these lines by placing the `#` symbol followed by these lines or texts. These lines are called *comments*. Comments are essential to in-code documentation. They make our code easier to understand, as we can explain why a code is written in a certain way. This is very important when we collaborate with others or to help the understanding of our codes when we review/read them in the future. For example

```r
x <- 15 # 15 here is in kg instead of g
# the line here is a comment
x
```

```
[1] 15
```

2.6 Bringing it all together

We can perform simple operations with the operators covered in this chapter. You recently got a job at the forestry research institute and you've been assigned to a department where you work with indigenous trees. In your department, the local/common names and not scientific names are used to identify indigenous trees. You've been instructed by your supervisor to go collect data on one of the experiment your team has been working on, and its data collection time. The research your team is working on needs data on the dry weights of the stem sections of four different species. You go to the lab, measure the dry weight of the tree stems, recorded them on your recording sheet and head back to your office to fire up your computer. As a R beginner, you quickly setup R and you want to see if it'll improves your research workflow overtime. You load R up and input the data with the knowledge you just gained in your most recent lesson (Operators in R). You assign the values of the dry weight of each species to their respective name:

Note

Trees used in this example is native to Africa.

```
# The dry weight in kg/m3
ekki <- 900
moabi <- 700
obeche <- 240
iroko <- 450
```

Your supervisor moves in and ask you to get the difference between the dry weight of `ekki` and `obeche`, and the difference between the dry weight of `iroko` and `moabi`.

```
ekki - obeche
```

```
[1] 660
```

```
iroko - moabi
```

```
[1] -250
```

Within seconds you said the difference between the dry weight of Ekki and Obeche is `660kg` and that of Iroko and Moabi is `-250`. This is quite easy to follow for you since you understand that the command you gave above have the variables holding values you assigned to it earlier, thus representing `900 - 240` and `450 - 700`.

Surprised to hear a negative value, your supervisor ask if `iroko` is greater than `obeche`. You executed the command quickly, making use of the greater than comparison operator

```
iroko > obeche
```

```
[1] TRUE
```

You said yes, and continued in your response stating that the result you gave earlier was the difference between Iroko and Moabi. Realizing the error, your supervisor said he intended the difference between Iroko and Obeche, and the difference between Ekki and Moabi. You executed the new command and gave the result swiftly using R as a basic calculator.

Summary

In this chapter, you've been introduced to some basic R syntax and the operators. The arithmetic operators are used for mathematical operations, such as addition, multiplication, and division to mention a few. We covered the comparison operator which include `<`, `<=`, `==`, `>=` and `>` for comparing values as well as their combination with logical operators like `&` and `|`.

Lastly, we covered the assignment operator used to reference or bind values to names called variable. When we assign a different value to the same variable name in R, it gets replaced by the new value, thus a variable can only hold one value at a time.

Now that we've got a good understanding of some of the basic R syntax and the operators, we will move on to Chapter 3 to learn about the data types in R.

3 Data Types in R

Now that we are familiar with the basic syntax of R and its syntax. We move on to different data types, and that's because these operators works on one or more of these data types. Understanding the concept of the data types, reduces the likelihood of an error occurring, or in some case ease the debugging process when an error does occur. This is because one of the most common source of error when using R, or performing analysis in general regardless of the software used is wrong, inappropriate or inconsistent data types, read this blog post for more details.

Researchers regardless of the field are analysts working with various forms of data, which can be survey responses, secondary data from institutions, and field or lab measurements among others. These data are usually any of numbers, response in form of alphabets, ranking, rating or choice response such as yes/no. It is crucial to understand the types of data you are working with before any analysis is done. In the research process Figure 3.1, data collection and analysis are key stages, and one that frustrates a lot of researcher. While data collection is often tedious, this should not be the same with data analysis. With a proper framework and workflow, analyzing your data should be less daunting than collecting it.

There are six different data types in R, five which we will come across frequently in this book. In R, the function `typeof()` and `class()` are used to check the data type of a variable, these functions will be useful as we move along. The data types are in R are:

- character
- factor
- double

- integer
- logical
- complex

3.1 Character

Characters represent strings/texts and are written in R by enclosing contents in quotation marks. The quotation mark can either be the single `'` or double `"` and anything can be a character as long its contents are wrapped in quotation mark.

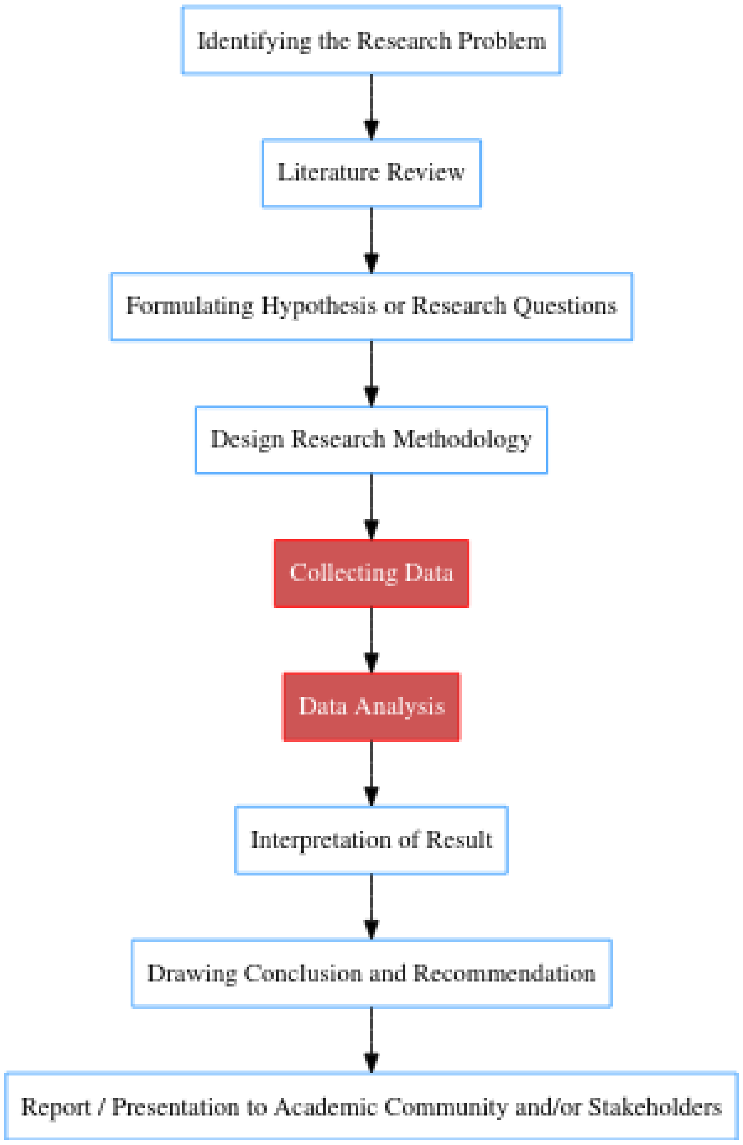

Identifying the Research Problem
Literature Review
Formulating Hypothesis or Research Questions
Design Research Methodology
Collecting Data
Data Analysis
Interpretation of Result
Drawing Conclusion and Recommendation
Report / Presentation to Academic Community and/or Stakeholders

Figure 3.1: Stages of Research

```r
"tree"
```

```
[1] "tree"
```

I'd advise to use the double " over the single ' as it allows you to use the apostrophe without worry, and it just also looks better.

```r
'My boss said - "what makes a good boss is not knowing all, that's why I have employees"'
```

```
Error in parse(text = input): <text>:1:66: unexpected symbol
1: 'My boss said - "what makes a good boss is not knowing all, that's
                                                                    ^
```

The above code is wrong because of the apostrophe as it is mistaken as closing the character statement.

```r
"My boss said - 'what makes a good boss is not knowing all, that's why I have employees'"
```

```
[1] "My boss said - 'what makes a good boss is not knowing all, that's why I have employees'
```

```r
"The man's children" # Definitely looks better
```

```
[1] "The man's children"
```

```r
'The man"s children'
```

```
[1] "The man\"s children"
```

We can check the data type of an object with either `class()` or `typeof()`.

```r
typeof("tree")
```

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

```r
class("tree")
```

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

```r
class("2")
```

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

```r
class("+")
```

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

When we assign characters to a variable, the content of the variable will be evaluated and not the variable in its self. As explained in Chapter 2.4 variables are references and not a value in themselves.

```r
tree <- "Quercus robur"
class(tree)
```

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

Let's look at the example below, we have error because `oak` is an object with no values assigned to it. So `oak` and `"oak"` are different. The latter is a character while the former is an object.

```r
class(oak)
```

```
Error: object 'oak' not found
```

There is a way to check if an objects is a character or not different from using `class()` and `typeof()`. The `is.character()` function returns a `TRUE` or `FALSE` used. `TRUE`, its a character data type and vice-versa.

```r
is.character("moabi")
```

```
[1] TRUE
```

```r
is.character(2)
```

```
[1] FALSE
```

```
is.character("2")
```

```
[1] TRUE
```

3.2 Factors

Factors are used to represent categorical data in R. They are usually leveled, and can only contain predefined values. The levels are usually distinct values. Data that can be represented as factors relates to responses like educational status, income level/class, satisfaction rating and so on. For example, in a farm of 5000 livestock, 2000 are goats, 390 cattles, 1500 sheeps, and 1100 pigs. The levels, i.e., the distinct animals in the farm include goat, sheep, pig, and cattle.

3.2.1 Types of Factors

There are two types of factors or categorical data. We have ordered (ordinal) categorical data or unordered (nominal) categorical data. The function `factor()` is used to create factors in R.

3.2.1.1 Nominal Factor

These are unordered categorical data with levels. There's no degree of distance or ranking in nominal factor variables. Example are the name of states in a country, name of trees, names of treatments, and so on. Below are example nominal data.

Note

Treatment is a technical term used in research, if the word treatment seem unfamiliar to you at the moment, we will discuss it in the next part of this book.

```
livestock <- factor(c("pig", "cattle", "goat", "sheep", "sheep",
                       "sheep", "pig", "sheep","sheep",
                       "sheep", "pig", "pig", "goat", "goat", "cattle")
                     )
tree_names <- factor(c("Lophira alata", "Triplochiton scleroxylon",
                       "Mansonia altissima", "Celtis africana",
                       "Borassus aethiopum")
                     )
```

```r
poultry_birds <-factor(c("breeders", "broilers", "layers"))
fertilizer <- factor(c("N", "P", "K"))
```

```r
livestock
```

```
 [1] pig    cattle goat   sheep  sheep  sheep  pig    sheep  sheep  sheep 
[11] pig    pig    goat   goat   cattle
Levels: cattle goat pig sheep
```

```r
tree_names
```

```
[1] Lophira alata           Triplochiton scleroxylon Mansonia altissima      
[4] Celtis africana         Borassus aethiopum      
5 Levels: Borassus aethiopum Celtis africana ... Triplochiton scleroxylon
```

```r
poultry_birds
```

```
[1] breeders broilers layers  
Levels: breeders broilers layers
```

```r
fertilizer
```

```
[1] N P K
Levels: K N P
```

Tip

The function, `c()`, is used to combine values. More about `c()` will be discussed in Chapter 4

When we print this variables, notice the output, especially that of `livestock`, we see that `Levels` is included as part of the result with the categories arranged in alphabetical order. At a closer look, you see that `livestock` Levels returns only the distinct animals even while they appear more than once in the data.

Using the function `levels()` we can get this distinct values of a factor returned as character.

```r
levels(tree_names)
```

```
[1] "Borassus aethiopum"       "Celtis africana"
[3] "Lophira alata"            "Mansonia altissima"
[5] "Triplochiton scleroxylon"
```

Using `class()` we can confirm the data type of these objects.

```r
class(poultry_birds)
```

```
[1] "factor"
```

If we use the `typeof()` function instead we get `integer`,soon to be discussed in 3.3.1. This is because `class()` checks the object class, while `typeof()` storage mode or internal type of objects and the factors are built on top of integers.

```r
typeof(tree_names)
```

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

3.2.1.2 Ordinal Factors

Ordinal factors are different from nominal factors, as they indicate a degree of difference, hierarchy, rank or order. Before we proceed to creating ordinal factors, let's get more understanding into the levels attribute of factors. When we created the nominal factors, we did not include the level argument, as R automatically does that for us. If we specify the level argument we take more control of how factors are organized dictating a particular print order. For example, let's check the difference in how the levels are arranged in the gender object below:

```r
gender_1 <- factor(c("m", "f"), levels = c("f", "m"))
gender_2 <- factor(c("m", "f"), levels = c("m", "f"))
```

```r
gender_1
```

```
[1] m f
Levels: f m
```

```r
gender_2
```

```
[1] m f
Levels: m f
```

The levels of `gender_1` and `gender_2` follows the arrangement specified in their `levels` argument. The output of `gender_1` is also similar to running the code without the `levels` argument specified, i.e., arranged in alphabetical order. This knowledge will be important when we want to create ordinal factors. There are two ways to create ordinal factors. The first is using the `ordered()` function, and the second is passing `TRUE` to the `ordered` argument in `factor`. Let's take a quick example of a response that include educational level. The survey question would be typically asked such that only one response is ticked - " What's your highest educational level attained." The responses when recorded could be encoded using numbers such 1, 2, 3, and so on, or alphabets such as A, B, C, etc., to represent the levels. For this example, the 2011 International Standard Classification of education which categorize education level into 9 distinct classes will be useful (ISCED 2012) .

```r
education_level <- factor(
  c(7, 8, 4, 2, 3, 3, 4, 1, 5, 4, 4, 7, 2, 6, 5,
    1, 2, 7, 7, 6, 0, 3, 3, 7, 8, 1, 0, 5, 3, 7, 7,
    0, 5, 8, 0, 5, 6, 2, 2, 0, 2, 2, 6, 1, 5, 0, 3,
    8, 8, 5),
  levels = c(0:8),
  ordered = TRUE
)

education_level
```

```
 [1] 7 8 4 2 3 3 4 1 5 4 4 7 2 6 5 1 2 7 7 6 0 3 3 7 8 1 0 5 3 7 7 0 5 8 0 5 6 2
[39] 2 0 2 2 6 1 5 0 3 8 8 5
Levels: 0 < 1 < 2 < 3 < 4 < 5 < 6 < 7 < 8
```

The data might not make sense to some people, and it should not. With a good data documentation, understanding what the data means becomes easier. Some could argue that it would be better to write the label outright without the stress of encoding them with numbers, and that's a solid argument, but such data entry task could be daunting depending on the size of the project. There's an easy way to go about this would be that makes the stress of retyping reduce with the `labels` argument. The `labels` argument replaces the numbers with texts/labels so they are easy to understand.

```r
education_level_labelled <- factor(
  c(7, 8, 4, 2, 3, 3, 4, 1, 5, 4, 4, 7, 2, 6, 5,
    1, 2, 7, 7, 6, 0, 3, 3, 7, 8, 1, 0, 5, 3, 7, 7,
    0, 5, 8, 0, 5, 6, 2, 2, 0, 2, 2, 6, 1, 5, 0, 3,
    8, 8, 5),
  levels = c(0:8),
  labels = c("Early childhood", "Primary", "Lower secondary",
             "Upper secondary", "Post secondary non-tertiary",
             "Short cycle tertiary", "Bachelors", "Masters",
             "Doctoral"),
  ordered = TRUE
)

education_level_labelled
```

```
 [1] Masters                     Doctoral
 [3] Post secondary non-tertiary Lower secondary
 [5] Upper secondary             Upper secondary
 [7] Post secondary non-tertiary Primary
 [9] Short cycle tertiary        Post secondary non-tertiary
[11] Post secondary non-tertiary Masters
[13] Lower secondary             Bachelors
[15] Short cycle tertiary        Primary
[17] Lower secondary             Masters
[19] Masters                     Bachelors
[21] Early childhood             Upper secondary
[23] Upper secondary             Masters
[25] Doctoral                    Primary
[27] Early childhood             Short cycle tertiary
[29] Upper secondary             Masters
[31] Masters                     Early childhood
[33] Short cycle tertiary        Doctoral
[35] Early childhood             Short cycle tertiary
[37] Bachelors                   Lower secondary
[39] Lower secondary             Early childhood
[41] Lower secondary             Lower secondary
[43] Bachelors                   Primary
[45] Short cycle tertiary        Early childhood
[47] Upper secondary             Doctoral
[49] Doctoral                    Short cycle tertiary
9 Levels: Early childhood < Primary < Lower secondary < ... < Doctoral
```

Now, this is easier to understand.

Let's switch our attention back to ordinal factors. First, we set the `ordered` argument within factor to `TRUE` and a closer look at the Levels shows a different output. We have the lesser than symbol, which shows that `Early childhood` is lesser than `Primary`, `Primary` lesser than `Lower secondary` and so on. This represents a rank. Here, `Early childhood` is the lowest rank and `Doctoral` the highest rank. Using the function `table()` we can take a frequency of the response.

```
table(education_level_labelled)
```

```
education_level_labelled
            Early childhood                     Primary 
                          6                           4 
            Lower secondary             Upper secondary 
                          7                           6 
Post secondary non-tertiary        Short cycle tertiary 
                          4                           7 
                  Bachelors                     Masters 
                          4                           7 
                   Doctoral 
                          5 
```

Specifying the levels is important as it also helps us detect if a level is missing.

```
edu_level <- factor(
  c(1, 2, 3, 4, 1, 1, 2, 3, 7, 8),
  levels = c(0:8),
  labels = c("Early childhood", "Primary", "Lower secondary",
             "Upper secondary", "Post secondary non-tertiary",
             "Short cycle tertiary", "Bachelors", "Masters",
             "Doctoral"),
  ordered = TRUE
)

edu_level
```

```
 [1] Primary                     Lower secondary            
 [3] Upper secondary             Post secondary non-tertiary
 [5] Primary                     Primary                    
 [7] Lower secondary             Upper secondary            
 [9] Masters                     Doctoral                   
9 Levels: Early childhood < Primary < Lower secondary < ... < Doctoral
```

Even when missing some levels, the Levels are still shown. In the current version of education level, `edu_level`, `Early childhood`, `Bachelors`, and `Short cycle tertiary` education levels are missing. Using `table()` shows this better.

```r
table(edu_level)
```

```
edu_level
             Early childhood                      Primary
                           0                            3
             Lower secondary              Upper secondary
                           2                            2
Post secondary non-tertiary         Short cycle tertiary
                           1                            0
                   Bachelors                      Masters
                           0                            1
                    Doctoral
                           1
```

> **Note**
>
> We have been using functions like `c()`, `table()`, `class()` and so on for a while now without properly introducing functions properly. Just keep in mind that functions are variables with parenthesis in front of them. They are block of codes that perform specified actions. More will be discussion on that in Chapter 4.

To confirm if a variable is a factor data type use the `is.factor()` . Let's create a new survey response, but this time it won't be wrapped in the factor variable.

```r
edu_level_chr <- c("Early childhood", "Primary", "Lower secondary",
                            "Upper secondary", "Post secondary non-tertiary",
                            "Short cycle tertiary", "Bachelors", "Masters",
                            "Doctoral")
```

```r
is.factor(edu_level_chr)
```

```
[1] FALSE
```

```r
is.factor(livestock)
```

```
[1] TRUE
```

We can also confirm if a factor is ordered or not using the function `is.ordered()`

```r
is.ordered(edu_level)
```

```
[1] TRUE
```

> ❗ Important
>
> It is important to take a closer look at the output of your codes it shows if the output is what you expect. Factors have no quotation marks when returned, but characters do. This is a good distinction to keep in mind.

3.3 Numeric Data type

Numeric data are digits. There are two types of numeric data; integer and double.

3.3.1 Integer

`Integer` are whole numbers and are written by writing a digit followed by a `L`.

```r
5L
```

```
[1] 5
```

```r
class(5L)
```

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

When divided by another integer or another numeric data type the result is usually a `double`.

```r
25L/5L
```

```
[1] 5
```

```r
typeof(25L / 5L)
```

```
[1] "double"
```

Just like characters and factors, we can also check if a particular object is an integer using `is.integer()`.

```r
is.integer(5)
```

```
[1] FALSE
```

```r
is.integer(5L)
```

```
[1] TRUE
```

3.3.2 Doubles

Doubles are also referred to as floats. They are numbers with decimal point.

```r
5.1
```

```
[1] 5.1
```

```r
typeof(5.1)
```

```
[1] "double"
```

When a mathematical operation is performed on a double, a double is return

```r
typeof(1.5/3)
```

```
[1] "double"
```

To check if an object is a double we use its `is._` variant, in this case `is.double()`

```r
is.double(2.9)
```

```
[1] TRUE
```

A sequence of integers or doubles can be created using `:` with the start on the left and the end at the right of the symbol.

```r
1:15
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
```

As `is.integer()` returns true for all integers and `is.doubl()e` returns true for doubles, `is.numeric()` returns true for all both types of data.

```r
is.numeric(5L)
```

```
[1] TRUE
```

```r
is.numeric(3.5)
```

```
[1] TRUE
```

3.4 Logical

Logical data type are also referred to as Boolean values. Logical includes `TRUE`, `FALSE` and `NA` which means *Not Available*.

```r
class(TRUE)
```

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

```r
class(NA)
```

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

```r
class(FALSE)
```

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

In Chapter 2.2, we saw that comparison operators always returns a logical operator as their result. We can also check the type as it is evaluated.

```r
class(5 > 2)
```

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

`5 > 2` is first evaluated which returns `TRUE`, then the `class()` of the result is checked which returns `TRUE`. In R, the deepest of the nested expressions are evaluated first and evaluated outwardly to the umbrella expression. We can also check for a logical value using `is.logical()`

```r
is.logical(20 < 3)
```

```
[1] TRUE
```

```r
is.logical(NA)
```

```
[1] TRUE
```

3.5 Complex

We will not use the complex data type and its unlikely to analyze these type of data. Complex data types are numbers with an imaginary term `i` added to them

```r
5i
```

```
[1] 0+5i
```

```r
1 + 5i
```

```
[1] 1+5i
```

```r
typeof(3i)
```

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

```r
typeof(5 + 2i)
```

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

Similar to other data types, we can confirm if a data is a complex by using `is.complex()`

```r
is.complex(5i)
```

```
[1] TRUE
```

3.6 Changing Data Types

In R we can change data types using `as.data_type` similar to `is.*` which is used to check a data type.

```r
tree_height <- 20:40
tree_height
```

```
 [1] 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
```

```r
typeof(tree_height)
```

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

The above can be changed to a double using `as.double()` and back to integer using `as.integer()`

```r
tree_height_2 <- as.double(tree_height)
typeof(tree_height_2)
```

```
[1] "double"
```

We can also change characters to factors and factors to characters in a similar way

```r
tree_names <- c("terminalia", "eucalyptus", "iroko", "oak")
class(tree_names)
```

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

To change the above to factor data type we'll use the `as.factor()` function.

```r
tree_names_fct <- as.factor(tree_names)

class(tree_names_fct)
```

```
[1] "factor"
```

We can change this back to character by using `as.character()`.

```r
tree_names_chr <- as.character(tree_names_fct)
class(tree_names_chr)
```

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

`as.character()` can be used to coerce any data type to a character

```r
tree_height
```

```
 [1] 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
```

```r
class(tree_height)
```

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

```r
tree_height_chr <- as.character(tree_height)
class(tree_height_chr)
```

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

Characters can be changed back to numeric or double data type using the `as.double()`, `as.integer()` or `as.numeric()` function. The functions will return a number only if what is been converted contains only digits, else, `NA` is returned as the result. we get a warning when there's not a digit in the object being converted.

```r
tree_height <- c(10, 15, 20, "25m")
tree_height
```

```
[1] "10"  "15"  "20"  "25m"
```

```r
class(tree_height)
```

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

```r
tree_height <- as.double(tree_height)
```

```
Warning: NAs introduced by coercion
```

```r
tree_height
```

```
[1] 10 15 20 NA
```

```r
class(tree_height)
```

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

we can also convert factors to integers. As said in 3.2, factors are built on integers.

```r
fruits <- factor(c("clementine", "kiwi", "avocado"))
class(fruits)
```

```
[1] "factor"
```

```r
as.integer(fruits)
```

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

We can also convert other data types to logical data type. All other numbers evaluate to TRUE either if they are positive or negative except zero which evaluates to zero.

```r
as.logical(c(1, 0, -1, 3, 0, 0 , 32))
```

```
[1]  TRUE FALSE  TRUE  TRUE FALSE FALSE  TRUE
```

For characters, every other thing converts to NA, except **T**, **TRUE**, **True**, and **true** which converts to `TRUE` and **F**, **FALSE**, **False** and **false** which converts to `FALSE`

```
as.logical(c("TRUE", 1, 0, "FALSE", "F", "T", 3, "False", "man", "true", "tRUE", "True", "t"
```

```
 [1]  TRUE    NA    NA FALSE FALSE  TRUE    NA FALSE    NA  TRUE    NA  TRUE
[13]    NA
```

Note

Data coercion/conversion can either be implicit or explicit. Coercion is explicit when we intentionally change the variable type. Implicit coercion is when R changes the variable type for us automatically.

3.7 Other Data Types

There are other data types not covered such as:

- Date
- POSIXct
- Raw

3.8 Bringing it all together

Just like you, your friend is a beginner of R and a new recruit at the National Bureau of Statistics.On his very first day, a dataset in 852 data sheet collected by 80 field staff was placed on his desk. The data includes responses from a recent survey on selected food prices across different regions of the country.

Feeling a bit overwhelmed, he paused and thought: ***"How can I make sense of all this data?"*** Then he remembered his last R training session especially the part about data types.

He picked up the survey sheets. Since the questions were well-structured, it became easier to identify what types of data he was dealing with. He decided to begin by categorizing the responses by region. For simplicity, we'll focus on four main regions: North, East, West, and South. For the regions, he recorded them as characters.

```
regions <- c("north", "east", "west", "south")
```

Next, he sorted the data according to the food items, recording them also as characters also.

```r
food_item <- c("rice", "beans", "yam", "flour", "cassava")
```

After, he recorded the prices of each item, their quantity sold and their availability in the market.

```r
rice_price <- 200
beans_price <- 300
yam_price <- 700
flour_price <- 120
cassava_price <- 90


rice_quantity <- 3000L
beans_quantity <- 1500L
yam_quantity <- 1800L
flour_price <- 4500L
cassava_quantity <- 10000L
```

Recommended Reading

Chapter 3 of Advanced R by Hadley Wickham, 2019 gives in-depth understanding of the data types in R.

Hands-On Programming with R by Garret Grolemund explain the concepts of R in a game-like approach using deck of cards.

Summary

Data types in R include character, factor, double, integer, and logical. This data type can be checked using `class()`, `typeof()` and their adjoining `is.data_type` function. These data types can be converted from one type to another using `as.data_type` variant.

Short Exercise

Take a moment to reflect on your friend's approach to organizing the survey data. Based on what you've just learned:

- What data types did your friend use to record the responses?

- Do you think he chose the right data types for each variable? If not, what changes would you suggest and why?
- Given the task assigned to your friend, could there be a better way to organize or structure the data for easier analysis?

4 Functions

Tip

This section is under review.

In the previous chapters, functions such as `c()`, `as.factors()`, and `class()` were used. These objects with parentheses in front of them are referred to as **functions**. Functions are self-contained blocks of code designed to perform specific tasks.

We use functions because they simplify our work and make analyses more efficient. If we make mistakes, functions allow for easy corrections and the interesting thing is they are reusable. You don't throw away a knife once you are done using it, you clean it up and reuse it whenever you need it again. Functions are similar to that reusable chunks of codes.

In R, function names are usually verbs, representing actions to be taken. Since functions are named objects followed by parentheses, their parentheses serve a purpose – to hold arguments. **Arguments** are the values passed to the function to perform a task on, think of them as onions, beef, or whatever you use a knife to cut, but this time around they are our a part or the whole of a data

One of the most basic and very important function is `c()` which combines values. Let's take a hypothetical tomato experiment as an example. In this experiment, we are interested in seeing how three fertilizers influences the weight of tomato. So we go to our greenhouse and get some tomato fruit measuring their weight in `g`. Using `c()`, we can combine the values of all the tomato weights collected.

```r
tomato_weight <- c(43.68, 70.23, 29.31, 83.08, 27.42, 53.50, 30.95, 10.51, 41.41, 68.06)
```

To see the weights that have been recorded simply call the variable as is and run it.

```r
tomato_weight
```

```
 [1] 43.68 70.23 29.31 83.08 27.42 53.50 30.95 10.51 41.41 68.06
```

Alternatively, you can use either the `print()` function

```
print(x = tomato_weight)
```

```
 [1] 43.68 70.23 29.31 83.08 27.42 53.50 30.95 10.51 41.41 68.06
```

Let's breakdown the `print()` function:

- The name of the variable that will hold the value, `x`, is the **parameter**.
- The name-value pair `x = tomato_weight` is the argument
- `tomato_weight` is the value

4.1 Arguments

Arguments are the input you pass to a function when you call it. It provides a value for a parameter, and it can be passed by position or by name, i.e., keyword.

4.1.1 Types of Arguments

- **Keyword or named arguments**: Here, you explicitly name the parameter in the function call. This makes your code clearer and order-independent. The previous print statement used the keyword argument. Another example is below with the `round()` function which round-up values

```
round(x = tomato_weight, digits = 1)
```

```
 [1] 43.7 70.2 29.3 83.1 27.4 53.5 31.0 10.5 41.4 68.1
```

The `tomato_weight` values are rounded up to 1 decimal point. This could be written with a different order, and the result will still be the same because the keywords are used.

```
round(digits = 1, x = tomato_weight)
```

```
 [1] 43.7 70.2 29.3 83.1 27.4 53.5 31.0 10.5 41.4 68.1
```

- **Positional argument**: Here, you provide the value in order without including the parameter name. The function matches the value to parameters by their position in the function definition. This is quite a common practice as it reduces typing. Most functions usually takes the object/data and with that understanding positional arguments are used.

```r
round(tomato_weight, 1)
```

```
 [1] 43.7 70.2 29.3 83.1 27.4 53.5 31.0 10.5 41.4 68.1
```

`tomato_weight` is matched to the first parameter `x` in the `round()` function and `1` matched to `digits`. If this is reordered without the keyword, we'd either get an unexpected result or an error.

```r
round(1, tomato_weight)
```

```
 [1] 1 1 1 1 1 1 1 1 1 1
```

To know more about any function use `help()` and pass the function in between the parentheses or type a question mark `?` followed by the function name.

```r
?round()
help("round")
```

The result should be something like in Figure 4.1

4.2 Some Built-in Functions

There are many functions readily available in R to assist with your analysis tasks. For tasks that R's built-in functions can't handle, you can always install packages (covered in Chapter 7) or develop your custom functions, a concept we will cover in Section 4.3.

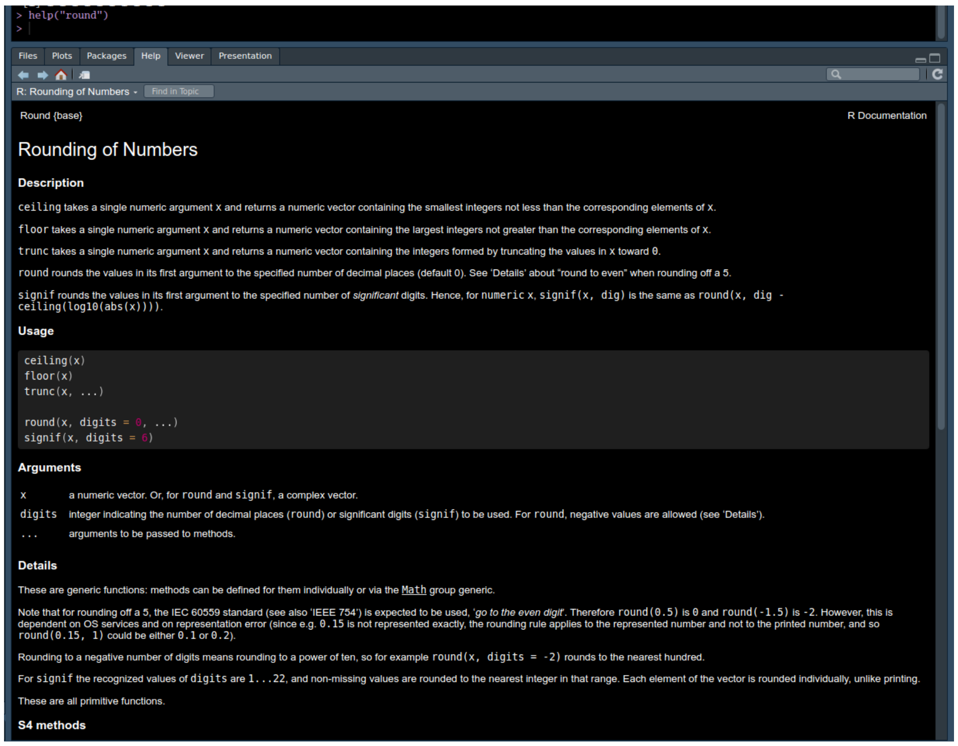
For examples, to know the total number of observations we have in `tomato_weight` we can use `length()`.

```r
length(tomato_weight)
```

```
[1] 10
```

To find the average of `tomato_weight`, we use `mean()`.

```r
mean(x = tomato_weight)
```



```
> help("round")
>
```

Round {base}   R Documentation

Rounding of Numbers

Description

`ceiling` takes a single numeric argument `x` and returns a numeric vector containing the smallest integers not less than the corresponding elements of `x`.

`floor` takes a single numeric argument `x` and returns a numeric vector containing the largest integers not greater than the corresponding elements of `x`.

`trunc` takes a single numeric argument `x` and returns a numeric vector containing the integers formed by truncating the values in `x` toward `0`.

`round` rounds the values in its first argument to the specified number of decimal places (default 0). See 'Details' about "round to even" when rounding off a 5.

`signif` rounds the values in its first argument to the specified number of *significant* digits. Hence, for `numeric` `x`, `signif(x, dig)` is the same as `round(x, dig - ceiling(log10(abs(x))))`.

Usage

```
ceiling(x)
floor(x)
trunc(x, ...)

round(x, digits = 0, ...)
signif(x, digits = 6)
```

Arguments

`x`	a numeric vector. Or, for `round` and `signif`, a complex vector.
`digits`	integer indicating the number of decimal places (`round`) or significant digits (`signif`) to be used. For `round`, negative values are allowed (see 'Details').
`...`	arguments to be passed to methods.

Details

These are generic functions: methods can be defined for them individually or via the `Math` group generic.

Note that for rounding off a 5, the IEC 60559 standard (see also 'IEEE 754') is expected to be used, '*go to the even digit*'. Therefore `round(0.5)` is `0` and `round(-1.5)` is `-2`. However, this is dependent on OS services and on representation error (since e.g. `0.15` is not represented exactly, the rounding rule applies to the represented number and not to the printed number, and so `round(0.15, 1)` could be either `0.1` or `0.2`).

Rounding to a negative number of digits means rounding to a power of ten, so for example `round(x, digits = -2)` rounds to the nearest hundred.

For `signif` the recognized values of `digits` are `1...22`, and non-missing values are rounded to the nearest integer in that range. Each element of the vector is rounded individually, unlike printing.

These are all primitive functions.

S4 methods

Figure 4.1: Documentation of the `round()` function. Preview of functions documentations looks like

```
[1] 45.815
```

To check the median of the data use (you guessed it right) `median()`. We also have `sd()`, `var()`, and `cor()` for estimating the standard deviation, variance and correlation respectively.

```r
median(tomato_weight)
```

```
[1] 42.545
```

```r
sd(tomato_weight)
```

```
[1] 22.70789
```

```r
var(tomato_weight)
```

```
[1] 515.6485
```

```r
cor(tomato_weight, 1:10)
```

```
[1] -0.1669699
```

We've seen `round()` in action before. However `round()` is having two siblings . The `ceiling()` and `floor()` function to round up and down respectively.

```r
floor(3.544)
```

```
[1] 3
```

```r
ceiling(3.544)
```

```
[1] 4
```

To create a sequence of number, we can use colon (`:`).

```r
20:30
```

```
 [1] 20 21 22 23 24 25 26 27 28 29 30
```

```r
30:20
```

```
 [1] 30 29 28 27 26 25 24 23 22 21 20
```

This gives a sequence of numbers increased or decreased by 1. To get more control over the sequence you want to create use the `seq()` function. Important arguments of `seq()` to remember are `from`, `to`, and `by`.

```r
seq(from = 1, to = 100, by = 1)
```

```
  [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
 [19]  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
 [37]  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
 [55]  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
 [73]  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
 [91]  91  92  93  94  95  96  97  98  99 100
```

With positional argument, it could be written as shown below

```r
seq(1, 100, 1)
```

R also comes with built-in constants such as `pi` , `letters`, `LETTERS`, `month.abb` and `month.name`

```r
pi
```

```
[1] 3.141593
```

```r
letters
```

```
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
[20] "t" "u" "v" "w" "x" "y" "z"
```

```r
month.abb
```

```
 [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"
```

4.2.1 Simulating Numbers

Aside from generating sequence of numbers, we can simulate numbers using R to meet certain distributions. An important part of research is data simulation and R provides a robust set of functions for generating random numbers from different distributions. For example, the `runif()` function generates random numbers from a uniform distribution, `rnorm()` for the normal distribution, `rpois()` for the Poisson distribution, and `rbinom()` for the binomial distribution. These functions belong to a family that also includes functions for calculating density (**d**), distribution functions (**p**), quantiles (**q**), and random deviates (**r**) for various statistical distributions. A comprehensive list of these functions can be found in Table 4.1.

For example, to simulate the height of 100 *Tectona grandis* in a plantation with mean diameter of 35 cm and a standard deviation of 2.3 cm we use `rnorm()`. `rnorm()` generates random numbers following the normal distribution.

```r
teak_diameter <- rnorm(100, mean = 35, sd = 2.3)
teak_diameter
```

```
  [1] 35.30383 30.64409 35.72010 31.91685 31.78326 32.22395 30.53345 34.79336
  [9] 35.52737 34.07964 32.94878 39.00874 34.42119 37.87564 34.68062 36.08479
 [17] 31.42119 34.44189 33.39934 31.84838 33.48843 33.30848 35.56403 36.30976
 [25] 35.16969 37.98228 34.60775 36.25893 40.14792 39.04321 35.09635 32.75637
 [33] 31.56914 36.94288 37.70538 36.25425 36.10210 34.72734 34.11126 38.96444
 [41] 37.54399 33.88946 32.42084 36.02367 33.53863 34.40285 35.89482 35.33568
 [49] 34.28463 35.85705 38.85629 30.62109 30.96638 31.24979 35.24205 35.16335
 [57] 34.94007 36.00580 32.63617 30.10547 35.88956 33.33359 32.36988 35.51088
 [65] 35.68430 34.64499 32.35206 30.13996 37.81297 36.79717 37.15572 32.88475
 [73] 31.26843 37.40919 33.68244 33.61746 33.60983 32.50431 36.48373 37.94503
 [81] 35.21156 34.35563 36.00109 34.58097 36.27919 34.60772 35.79629 30.16265
 [89] 32.98596 42.17230 35.41974 31.74654 37.63961 35.19506 33.26862 38.70970
 [97] 36.11023 36.00204 34.15717 35.04978
```

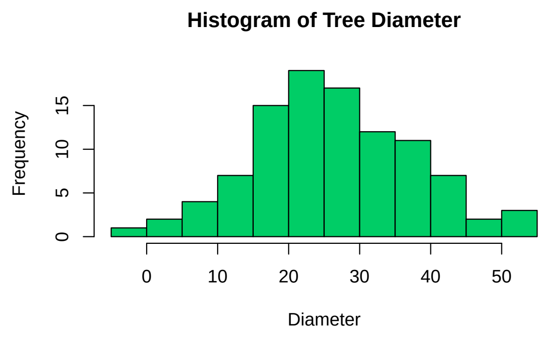
The numbers generated above will be different from that which you will produce. This is because they are random numbers. The beauty of R is its reproducibility, and what use is reproducibility if others can't get the same result as us if they follow exactly the same steps as us. That's why there's the `set.seed()` function which captures or get a snapshot of specific numbers randomly generated. Given that the seed number is the same, whatever pseudorandom numbers are produce can be replicated by another person.

```r
set.seed(123)
tree_diameter <- rnorm(100, mean = 25, sd = 12.3)
tree_diameter
```

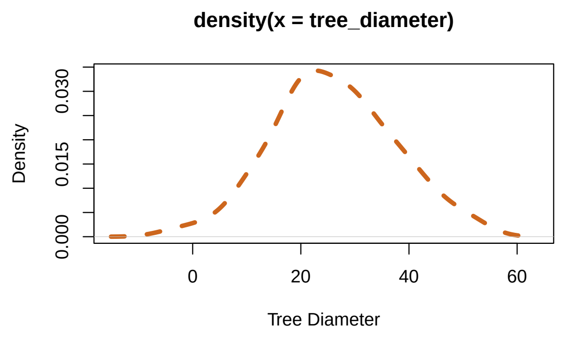
```
  [1] 18.106150 22.168817 44.172112 25.867253 26.590239 46.095299 30.669269
  [8]  9.439747 16.551710 19.518358 40.056206 29.425710 29.929489 26.361397
 [15] 18.163154 46.979032 31.123561  0.810609 33.626678 19.184666 11.865768
 [22] 22.318909 12.380145 16.034638 17.312017  4.253672 35.304781 26.886489
 [29] 11.000916 40.421924 30.245510 21.370621 36.010046 35.801042 35.105447
 [36] 33.470275 31.813187 24.238486 21.236659 20.320207 16.455104 22.442617
 [43]  9.435625 51.678158 39.857933 11.185764 20.044517 19.260139 34.593571
 [50] 23.974560 28.115818 24.648875 24.472693 41.833808 22.223017 43.652588
 [57]  5.950341 32.190749 26.523407 27.656081 29.669566 18.821422 20.901549
 [64] 12.471523 11.816968 28.733402 30.512980 25.651952 36.343890 50.216042
 [71] 18.960317 -3.402777 37.370584 16.276831 16.537494 37.614528 21.497292
 [78]  9.985172 27.230033 23.291636 25.070899 29.738949 20.440882 32.925832
 [85] 22.288015 29.080918 38.491120 30.352732 20.991041 39.130334 37.220097
 [92] 31.745283 27.936400 17.276755 41.736025 17.616807 51.904196 43.851111
 [99] 22.100886 12.375023
```

With the seed set already, you should have exactly the same result as what is produced here.

We can understand the data which we just produced if it is visualized. With the `plot()` function, we can produce a graph. Not to worry we will cover more on visualizations Chapter 12.



(a)



(b)

Figure 4.2: Histogram of simulated tree diameter distribution

In forestry, the distribution of trees diameter in a plantation is usually a bell-shaped curve. Figure 4.2b shows the distribution of the diameter, while the distribution of the diameters represented as a histogram can be seen in Figure 4.2a.

While `rnorm()` generates numbers that follow a normal distribution `runif()` generates numbers that follow a uniform distribution, Figure 4.3.

```r
set.seed(123)
dt <- runif(100, min = 10, max = 20)
dt
```

```
  [1] 12.87578 17.88305 14.08977 18.83017 19.40467 10.45556 15.28105 18.92419
  [9] 15.51435 14.56615 19.56833 14.53334 16.77571 15.72633 11.02925 18.99825
 [17] 12.46088 10.42060 13.27921 19.54504 18.89539 16.92803 16.40507 19.94270
 [25] 16.55706 17.08530 15.44066 15.94142 12.89160 11.47114 19.63024 19.02299
 [33] 16.90705 17.95467 10.24614 14.77796 17.58460 12.16408 13.18181 12.31626
 [41] 11.42800 14.14546 14.13724 13.68845 11.52445 11.38806 12.33034 14.65962
 [49] 12.65973 18.57828 10.45831 14.42200 17.98925 11.21899 15.60948 12.06531
 [57] 11.27532 17.53308 18.95045 13.74463 16.65115 10.94841 13.83970 12.74384
 [65] 18.14640 14.48516 18.10064 18.12390 17.94342 14.39832 17.54475 16.29221
 [73] 17.10182 10.00625 14.75317 12.20119 13.79817 16.12771 13.51798 11.11135
 [81] 12.43619 16.68056 14.17647 17.88196 11.02865 14.34893 19.84957 18.93051
 [89] 18.86469 11.75053 11.30696 16.53102 13.43516 16.56758 13.20373 11.87691
 [97] 17.82294 10.93595 14.66779 15.11505
```

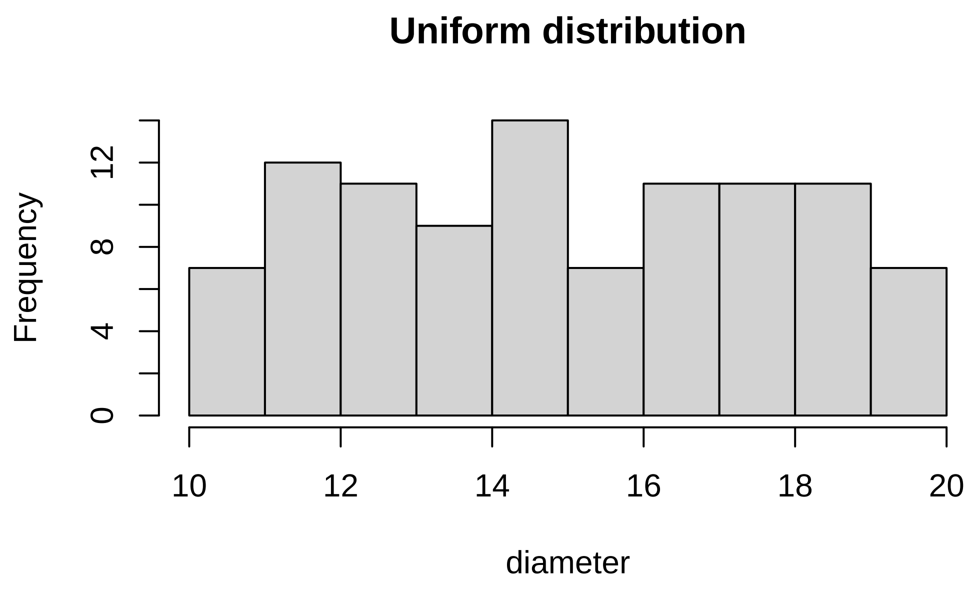


Figure 4.3: Uniformly distributed randomly generated numbers. The higher the number of observations the more uniform the distribution.

Other functions for generating random numbers according to distribution are:

Table 4.1: Functions for generating random numbers of data in R

Function	Distribution	Description
runif	Uniform distribution	Generates random numbers from a uniform distribution
rnorm	Normal distribution	Generates random numbers from a normal distribution
rpois	Poisson distribution	Generates random numbers from a poisson distribution
rbinom	Binomial distribution	Generates random numbers from a binomial distribution
dunif	Uniform distribution	Computes the density of a uniform distribution
dnorm	Normal distribution	Computes the density of a normal distribution
dpois	Poisson distribution	Computes the density of a poisson distribution
dbinom	Binomial distribution	Computes the density of a binomial distribution
punif	Uniform distribution	Computes the cumulative distribution function (CDF) of a uniform distribution
pnorm	Normal distribution	Computes the cumulative distribution function (CDF) of a normal distribution
ppois	Poisson distribution	Computes the cumulative distribution function (CDF) of a poisson distribution
pbinom	Binomial distribution	Computes the cumulative distribution function (CDF) of a binomial distribution
qunif	Uniform distribution	Computes the quantiles of a uniform distribution
qnorm	Normal distribution	Computes the quantiles of a normal distribution
qpois	Poisson distribution	Computes the quantiles of a poisson distribution
qbinom	Binomial distribution	Computes the quantiles of a binomial distribution

4.2.2 Sampling with sample()

Another important function for randomization is using `sample()`. `sample()` is used for selecting values at random from a set of data. Let's take the example below:

```r
new_dt <- seq(1, 20, 2)
new_dt
```

```
 [1]  1  3  5  7  9 11 13 15 17 19
```

We can randomly select five values from `new_dt` using `sample()`

```r
set.seed(10)
sample(new_dt, 5)
```

```
[1] 17 13 15 11  5
```

The `sample()` function is used for simulating die roll, coin toss, and even draws from a stack of card. For example, we can simulate the number we get from a die with the following:

```r
die <- 1:6
```

```r
sample(die, 1)
```

```
[1] 2
```

If you continue to run the code above, you will continue to get different values. A good challenge for you would be to simulate two die.

4.2.2.1 Weighted Samples and Sampling with Replacement

More than selecting values at random, with `sample()`, we can also select with replacement by setting the value of the argument `replace` to `TRUE`.

```r
sample(new_dt, 20, replace = TRUE)
```

```
 [1] 13 19  3 15 15 13 11 13 11  3  9 17  3 19  9 19  1 13 19  3
```

The probabilities of the values we want to randomly select can also be determined by the `prob` argument. This is perfect for weighted probability. Example is a die of unequal prob.

```r
die <- 1:6
die_prob <- c(0.23, 0.23, 0.23, 0.21, 0.07, 0.03)
```

```r
set.seed(123)
die_selected <- sample(x = die, size = 100, replace = TRUE, prob = die_prob)
die_selected
```

```
  [1] 1 4 1 4 5 2 3 4 3 1 5 1 3 3 2 4 1 2 1 5 4 4 3 6 3 4 3 3 1 2 5 5 4 4 2 3 4
 [38] 2 1 1 2 1 1 1 2 2 1 3 1 4 2 1 4 2 3 2 2 4 4 1 3 2 1 1 4 1 4 4 4 1 4 3 4 2
 [75] 3 2 1 3 1 2 1 3 1 4 2 1 6 4 4 2 2 3 1 3 1 2 4 2 3 3
```

To get the number of occurrence of each die face we can use the `table()` function. and notice that values with lower probability of occurring such as 5 and 6 were sampled less.

```r
table(die_selected)
```

```
die_selected
 1  2  3  4  5  6 
27 22 20 24  5  2 
```

4.2.3 Character Functions

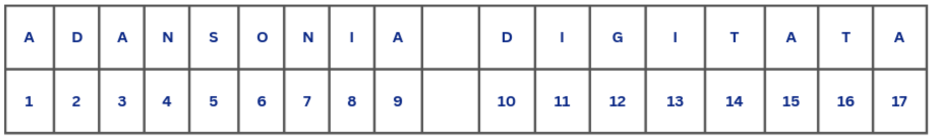
We've been working with numbers for a while, now lets see some functions that we can use for character data types. To begin, let's create a vector of character data type.

```r
tree_1 <- "Adansonia digitata"
tree_1
```

```
[1] "Adansonia digitata"
```

4.2.3.1 Counting Characters

Counting character is done using `nchar()`. Whenever, we count characters, we usually neglect the space and as such when we count the `tree_1` object we should expect 17 characters as what we have in Figure 4.4.



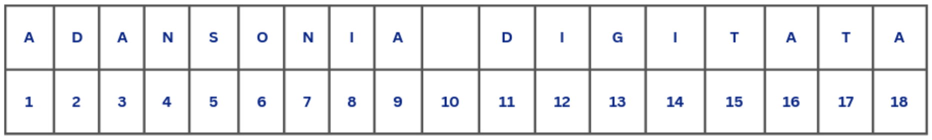
A	D	A	N	S	O	N	I	A		D	I	G	I	T	A	T	A
1	2	3	4	5	6	7	8	9		10	11	12	13	14	15	16	17

Figure 4.4: Number of characters in the object if counted without the space

Using `nchar()` we get a different result.

```r
nchar(tree_1)
```

```
[1] 18
```



A	D	A	N	S	O	N	I	A		D	I	G	I	T	A	T	A
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Figure 4.5: Number of characters in the object with the spaces included

This might come as a surprise. This is because spaces are also counted as characters, not only in R, but also in word processors like Microsoft Word or LibreOffice Writer and other programming languages. Given that, Figure 4.5 shows the actual number of characters with spaces included.

Characters can also be changed to lower cases by using `tolower()`, and changed to upper cases by using `toupper()`. Unfortunately, Base R does not provide a function to changed strings to title or sentence cases, but there are open source packages available to do this . You can explore stringr by Hadley Wickham and stringi by Marek Gagolewski which contains functions for such operation and other complex string manipulation operations in R.

```
tolower(tree_1)
```

```
[1] "adansonia digitata"
```

```
toupper(tree_1)
```

```
[1] "ADANSONIA DIGITATA"
```

4.2.3.2 Trimming Characters

There are some data where part of the variable are filled by respondents. This can include records of names, for example, clinic require records of physicians and the name of their patients. A particular physician could attend to multiple patient in a day and at different times in a day. Below is an example of physicians name in a clinic working different times. e could have the names of five clinic physicians repeated multiple times, according to the number of times they have shifts, or attend to patients.

```
physician <- c("Clara Reeves", "Motolani Eniola", "Praaval Duval", " Praaval Duval ", "Sunmi

physician
```

```
 [1] "Clara Reeves"           "Motolani Eniola"        "Praaval Duval"
 [4] " Praaval Duval "        "Sunmisola Aderibigbe " " Sunmisola Aderibigbe"
 [7] " Clara Reeves"          "Clara Reeves "          " Motolani Eniola "
[10] "Praaval Duval "
```

The example above is a classic example of having white spaces. We can check the data to see the number of distinct physicians we have.

```
length(unique(physician))
```

```
[1] 10
```

This returns 10 but we can see clearly that some names just has extra white spaces. With the `trimws()` function we could remove excess white spaces in the data. Before using the function lets count the number of characters we have for each observation. We will do the same after trimming to see the difference.

```
nchar(physician)
```

```
 [1] 12 15 13 15 21 21 13 13 17 14
```

```
physician_trimmed <- trimws(physician)
physician_trimmed
```

```
 [1] "Clara Reeves"         "Motolani Eniola"      "Praaval Duval"
 [4] "Praaval Duval"        "Sunmisola Aderibigbe" "Sunmisola Aderibigbe"
 [7] "Clara Reeves"         "Clara Reeves"         "Motolani Eniola"
[10] "Praaval Duval"
```

```
nchar(physician_trimmed)
```

```
 [1] 12 15 13 13 20 20 12 12 15 13
```

Now we can tell the number of characters reduced for most of the observations. We can now confirm the number of physicians we have using `length()` and `unique()`.

```
length(unique(physician_trimmed))
```

```
[1] 4
```

The four physicians are:

```r
unique(physician_trimmed)
```

```
[1] "Clara Reeves"          "Motolani Eniola"      "Praaval Duval"
[4] "Sunmisola Aderibigbe"
```

> **!** Important
>
> The technique taught here is very useful in the data cleaning process.

4.2.3.3 Extracting and Replacing Part(s) of a Chaaracter

To extract certain aspect of your text, use `substring()`, or `substr()`. `substr()` takes the argument `x` which is your data, `start` which is the index position for the first value of the text to be extracted or replaced, and `stop` which is the index position for the last value of the text to be extracted or replaced. For example, we can extract `Adansonia` from the text with the following code.

```r
substr(tree_1, start = 1, stop = 9)
```

```
[1] "Adansonia"
```

To replace a certain part of the text we can also use `substr()`.

```r
substr(tree_1, start = 11, nchar(tree_1)) <- "gregorii"
tree_1
```

```
[1] "Adansonia gregorii"
```

There are times, we would like to split text data, using `strsplit()`. For example we can split the `tree_1` to genus and species with the spaces between them.

```r
strsplit(tree_1, split = " ")
```

```
[[1]]
[1] "Adansonia" "gregorii"
```

Notice the output, both words now have their inverted comma.

There are still more functions for string manipulation in R, not all will be covered, as this will be a large book volume.

4.3 Custom Functions

The functions that are preloaded in R are a lot, but that does not mean they will meet all our statistical or operational needs. For some tasks you want to do, you would need to write your own function.

4.3.1 Creating Custom Functions

For example we can estimate the z-score of the `tomato_weight` data. The z-score tells you how many standard deviations an element is from the mean of the vector. It transforms our data into a common scale. The formula for z-score is given as:

$Z = \frac{x-\mu}{\sigma}$

Where:

- x is the value in the vector,
- μ is the mean of the vector,
- σ is the standard deviation of the vector.

Without a custom function we will do the following:

```
average_tomato <- mean(tomato_weight)
sd_tomato <- sd(tomato_weight)
z_tomato <- (tomato_weight - average_tomato) / sd_tomato

average_tree_diameter <- mean(tree_diameter)
sd_tree_diameter <- sd(tree_diameter)
z_tree_diameter <- (tree_diameter - average_tree_diameter) / sd_tree_diameter

average_dt <-mean(dt)
sd_tomato <- sd(dt)
z_dt <- (dt - average_dt) / sd_tomato
```

We got the following results

```
z_tomato
```

```
 [1] -0.09402017  1.07517674 -0.72683973  1.64105924 -0.81007070  0.33842856
 [7] -0.65461816 -1.55474564 -0.19398540  0.97961526
```

```
z_tree_diameter
```

```
  [1] -0.71304802 -0.35120270  1.60854170 -0.02179795  0.04259548  1.77983218
  [7]  0.40589817 -1.48492941 -0.85149566 -0.58726835  1.24195461  0.29513939
 [13]  0.34000892  0.02221347 -0.70797086  1.85854263  0.44636008 -2.25349176
 [19]  0.66930255 -0.61698896 -1.26885349 -0.33783464 -1.22304003 -0.89754917
 [25] -0.78377819 -1.94683206  0.81876439  0.06898128 -1.34588242  1.27452758
 [31]  0.36815564 -0.42229479  0.88157948  0.86296437  0.80101058  0.65537241
 [37]  0.50778230 -0.16686565 -0.43422620 -0.51585092 -0.86009994 -0.32681639
 [43] -1.48529653  2.27707482  1.22429519 -1.32941869 -0.54040552 -0.61026684
 [49]  0.75541982 -0.19037243  0.17847258 -0.13031397 -0.14600575  1.40027842
 [55] -0.34637532  1.56226982 -1.79571669  0.54141021  0.03664303  0.13752572
 [61]  0.31685861 -0.64934164 -0.46407310 -1.21490140 -1.27319995  0.23347834
 [67]  0.39197814 -0.04097396  0.91131364  2.14685001 -0.63697081 -2.62876100
 [73]  1.00275711 -0.87597805 -0.85276181  1.02448422 -0.41101270 -1.43635058
 [79]  0.09957931 -0.25119772 -0.09272595  0.32303830 -0.50510289  0.60688103
 [85] -0.34058618  0.26443017  1.10255872  0.37770550 -0.45610238  1.15949090
 [91]  0.98935390  0.50173432  0.16249260 -0.78691881  1.39156928 -0.75663177
 [97]  2.29720706  1.57995139 -0.35725306 -1.22349625
```

```
z_dt
```

```
  [1] -0.74029816  1.01667000 -0.31432828  1.34899927  1.55058114 -1.58950882
  [7]  0.10367363  1.38198803  0.18553298 -0.14717528  1.60800687 -0.15868628
 [13]  0.62812145  0.25991452 -1.38821361  1.40797416 -0.88587877 -1.60177908
 [19] -0.59874073  1.59983236  1.37188355  0.68157065  0.49807079  1.73936495
 [25]  0.55140145  0.73675424  0.15967644  0.33538461 -0.73474643 -1.23316204
 [31]  1.62972964  1.41665527  0.67420868  1.04180123 -1.66299360 -0.07285392
 [37]  0.91194687 -0.99002015 -0.63291572 -0.93662330 -1.24829781 -0.29478615
 [43] -0.29767044 -0.45514279 -1.21445611 -1.26231194 -0.93168176 -0.11437574
 [49] -0.81610602  1.26061293 -1.58854506 -0.19775388  1.05393276 -1.32163504
 [55]  0.21891237 -1.02467527 -1.30187194  0.89387052  1.39120332 -0.43543256
 [61]  0.58441742 -1.41657907 -0.40207459 -0.78659323  1.10907471 -0.17559117
 [67]  1.09301940  1.10117798  1.03785346 -0.20606415  0.89796636  0.45847125
```

```
[73]  0.74255060 -1.74716662 -0.08155371 -0.97699906 -0.41664711  0.40075054
[79] -0.51495972 -1.35940351 -0.89453948  0.59473476 -0.28390722  1.01628648
[85] -1.38842427 -0.22339407  1.70668793  1.38420585  1.36111055 -1.13512882
[91] -1.29076986  0.54226490 -0.54401788  0.55509389 -0.62522354 -1.09078258
[97]  0.99557901 -1.42094993 -0.11151046  0.04542695
```

If we look closely, we can spot an error while computing `z_dt`. The following steps can be done easily with a custom function. To create a custom function we need to solidify our understanding of functions. In R, every functions have three basic parts; a name, a body, and set of arguments. To create functions in R we call `function()` function followed by `{}`.

```r
new_function <- function() {}
```

The name of the arguments is passed into the parenthesis of `function()` and the body, or expresions are passed into the curly brackets.

```r
z_score <- function(x) {
  average_x <- mean(x, na.rm = TRUE)
  sd_x <- sd(x, na.rm = TRUE)
  z_value = (x - average_x)/sd_x
  return(z_value)
}
```

Next is using the custom function. We should note that, custom functions are called in a similar way that R's built-in functions are called, the object name followed by parenthesis.

```r
z_tomato_2 <- z_score(tomato_weight)
z_tomato_2
```

```
[1] -0.09402017  1.07517674 -0.72683973  1.64105924 -0.81007070  0.33842856
[7] -0.65461816 -1.55474564 -0.19398540  0.97961526
```

```r
z_tree_diameter_2 <- z_score(tree_diameter)
z_tree_diameter_2
```

```
 [1] -0.71304802 -0.35120270  1.60854170 -0.02179795  0.04259548  1.77983218
 [7]  0.40589817 -1.48492941 -0.85149566 -0.58726835  1.24195461  0.29513939
[13]  0.34000892  0.02221347 -0.70797086  1.85854263  0.44636008 -2.25349176
[19]  0.66930255 -0.61698896 -1.26885349 -0.33783464 -1.22304003 -0.89754917
[25] -0.78377819 -1.94683206  0.81876439  0.06898128 -1.34588242  1.27452758
```

```
[31]  0.36815564 -0.42229479  0.88157948  0.86296437  0.80101058  0.65537241
[37]  0.50778230 -0.16686565 -0.43422620 -0.51585092 -0.86009994 -0.32681639
[43] -1.48529653  2.27707482  1.22429519 -1.32941869 -0.54040552 -0.61026684
[49]  0.75541982 -0.19037243  0.17847258 -0.13031397 -0.14600575  1.40027842
[55] -0.34637532  1.56226982 -1.79571669  0.54141021  0.03664303  0.13752572
[61]  0.31685861 -0.64934164 -0.46407310 -1.21490140 -1.27319995  0.23347834
[67]  0.39197814 -0.04097396  0.91131364  2.14685001 -0.63697081 -2.62876100
[73]  1.00275711 -0.87597805 -0.85276181  1.02448422 -0.41101270 -1.43635058
[79]  0.09957931 -0.25119772 -0.09272595  0.32303830 -0.50510289  0.60688103
[85] -0.34058618  0.26443017  1.10255872  0.37770550 -0.45610238  1.15949090
[91]  0.98935390  0.50173432  0.16249260 -0.78691881  1.39156928 -0.75663177
[97]  2.29720706  1.57995139 -0.35725306 -1.22349625
```

```r
z_dt_2 <- z_score(dt)
z_dt_2
```

```
  [1] -0.74029816  1.01667000 -0.31432828  1.34899927  1.55058114 -1.58950882
  [7]  0.10367363  1.38198803  0.18553298 -0.14717528  1.60800687 -0.15868628
 [13]  0.62812145  0.25991452 -1.38821361  1.40797416 -0.88587877 -1.60177908
 [19] -0.59874073  1.59983236  1.37188355  0.68157065  0.49807079  1.73936495
 [25]  0.55140145  0.73675424  0.15967644  0.33538461 -0.73474643 -1.23316204
 [31]  1.62972964  1.41665527  0.67420868  1.04180123 -1.66299360 -0.07285392
 [37]  0.91194687 -0.99002015 -0.63291572 -0.93662330 -1.24829781 -0.29478615
 [43] -0.29767044 -0.45514279 -1.21445611 -1.26231194 -0.93168176 -0.11437574
 [49] -0.81610602  1.26061293 -1.58854506 -0.19775388  1.05393276 -1.32163504
 [55]  0.21891237 -1.02467527 -1.30187194  0.89387052  1.39120332 -0.43543256
 [61]  0.58441742 -1.41657907 -0.40207459 -0.78659323  1.10907471 -0.17559117
 [67]  1.09301940  1.10117798  1.03785346 -0.20606415  0.89796636  0.45847125
 [73]  0.74255060 -1.74716662 -0.08155371 -0.97699906 -0.41664711  0.40075054
 [79] -0.51495972 -1.35940351 -0.89453948  0.59473476 -0.28390722  1.01628648
 [85] -1.38842427 -0.22339407  1.70668793  1.38420585  1.36111055 -1.13512882
 [91] -1.29076986  0.54226490 -0.54401788  0.55509389 -0.62522354 -1.09078258
 [97]  0.99557901 -1.42094993 -0.11151046  0.04542695
```

Using a custom function does not only make our code more readable, it prevents errors from copying and pasting, and if we need to make a change in out formula, we need to do it in only a place–the function definition.

4.3.2 When to Write a Custom Functiom

When do you need to write a function?

- When you find yourself copying, pasting and adapting blocks of code. Copying, pasting and adapting codes to new data or objects should not be more than three times.
- When the code is clunky and not readable, writing custom functions improves the readability of your code
- For individuals that regular submit report to different people, stakeholders, and groups, parametizing different inputs in your report ensure you produce multiple focused report at once.
- Avoiding code duplication

! Getting Help

Use the `help()` function and `?` to get help. Also, use `args()` to see the arguments of functions. Just to let you know, there are more than 2300 functions loaded when an R session starts and that's a lot. You will remember some and forget some, but as keep using R they become a part of you.
You are not expected to remember these functions, use `help()`, `?` and `args()` and check online for resources when you need help.

4.4 Summary

Functions are little codes that performs specific functions. Armed with them we can fly on eagles wing. Functions such as `mean()`, `length()`, `plot()` and so on, makes it easier to perform tasks. However, this does not imply that there are functions for all tasks we want to accomplish. In such case, we need to develop our own functions called, custom functions. There are also functions in other packages which extends the capabilities of R.

Exercise

1. Using `seq()`, create a sequence of odd numbers between 1 and 100.
2. Read the documentation of `mean()`, `median()`, `sd()`, and `var()`. What does the `is.na()` argument does?
3. What is the difference between `substring()` and `substr()`?
4. What is the difference between `paste()` and `paste0()`?

5 Data Structure

Caution

Still under construction.

In the Chapter 3, we learned about different data types in R such as double, integer, character, and logical. Just as we organize physical items, and organize our personal space - our data needs special containers to organize and store different types of data. These containers are called **data structures**.

Each data structure in R has specific characteristics:

- What types of data can it hold? ***single or multiple types***
- How are the data organized? ***one-dimensional, i.e., having a single dimension, two-dimensional, i.e., having x or y axis or multidimensional, having more than two axis.***
- How can we interact with the data inside the object? ***using `[]` or`$`***

Before exploring the different data structures, we have three functions that can help us understand the data structure. `length()` tells us the number of elements we have in one dimensional data structures, `dim()` tells us the number of rows and columns in a multidimensional data structure. Lastly, we have `str()` which gives information about the number of rows and columns, and the underlying data types for the columns with preview.

5.1 Atomic Vectors

5.1.1 What is an Atomic Vector?

We've actually worked with atomic vectors, and we have been working with them since the start of this book. An atomic vector is the simplest data structure in R (Grolemund 2014). Think of it as a one-dimensional data that can hold only one type of data. Honestly, the different data types are actually atomic vectors. The simplest for of atomic vectors is the scalar vector which contains only one element.

```
# Single Vector
fruit <- "papaya"

fruit
```

```
[1] "papaya"
```

We can make our vectors longer using `c()`.

```
# Multivalue vector
fruits <- c("papaya", "orange", "apple", "pineapple", "grape",
            "strawberries", "avocado")

fruits
```

```
[1] "papaya"       "orange"       "apple"        "pineapple"    "grape"
[6] "strawberries" "avocado"
```

Since both `fruit` and `fruits` have one dimension we can use `length()` to get their number of elements.

```
length(fruit) # shows the number of element(s) in fruit
```

```
[1] 1
```

```
length(fruits) # shows the number of element(s) in fruits
```

```
[1] 7
```

Atomic vectors can only hold one data type at a time. In the example below, we created a new object of double data type, then combine it with the `fruits` object which we created earlier. Checking the data type of the new object `new_data`, only a the character data type is returned.

```
set.seed(123)
diameter <- round(rnorm(10, mean = 23.5, sd = 0.6), 2)
typeof(diameter)
```

```
[1] "double"
```

```r
new_data <- c(fruits, diameter)
new_data
```

```
 [1] "papaya"       "orange"       "apple"        "pineapple"    "grape"
 [6] "strawberries" "avocado"      "23.16"        "23.36"        "24.44"
[11] "23.54"        "23.58"        "24.53"        "23.78"        "22.74"
[16] "23.09"        "23.23"
```

```r
typeof(new_data)
```

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

5.1.2 Vector Operations

Simple operations like addition, subtraction, multiplication, division, and so on, are the operations that can be performed on vectors. We can also use functions on them as shown in Chapter 4.

```r
# Arithmetic Operation
x <- 15
y <- 20
y + x
```

```
[1] 35
```

```r
y %% x
```

```
[1] 5
```

```r
y / x
```

```
[1] 1.333333
```

Working with different data types that doesn't mix well could be the only stumbling blocks in these simple operations.

```r
"man" * 5
```

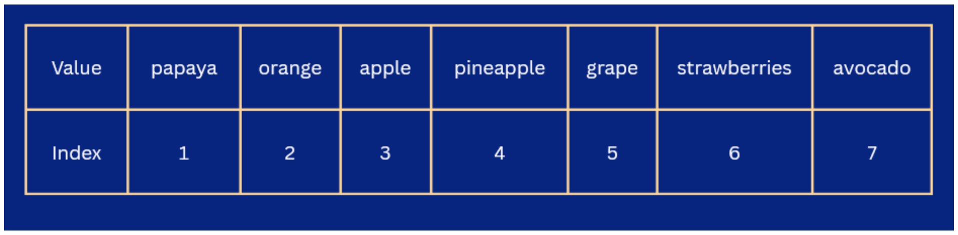
```
Error in "man" * 5: non-numeric argument to binary operator
```

Note

In general character and numeric data types don't mix, logical mixes with all data types.

5.1.3 Accessing Vector Elements

To access a value in a vector we can get it with its index number. Indexing allows us access, slice, subset or modify specific elements in different data structures. We use the square brackets [] to specify the index position of elements we are interested in. The index position starts with 1, the index position of elements in `fruits` is show in Figure 5.1.



Value	papaya	orange	apple	pineapple	grape	strawberries	avocado
Index	1	2	3	4	5	6	7

Figure 5.1: Index number of `fruits` vector

To get the first element *papaya* in the `fruits` vector, we specify its index position. As seen in Figure 5.1, the index of *papaya* is 1 in this case.

```r
fruits[1]
```

```
[1] "papaya"
```

To return multiple values at the same time, we combine those values with `c()` and pass it to the square brackets. To return *avocado*, *grape*, and *orange*, we select them by index position.

```r
# select elements by specific position
fruits[c(7, 5, 2)]
```

```
[1] "avocado" "grape"   "orange"
```

We can also select a range of elements using `:`. To select elements from *orange* to *grape*, we do the following:

```
# select the range of elements that fall within that number.
fruits[2:5]
```

```
[1] "orange"    "apple"     "pineapple" "grape"
```

5.1.4 Modifying Vectors

There are two ways to modify new elements in a vector:

- `append()` function
- The square bracket `[]`

5.1.4.0.1 The `append()` Function

We can use `append()` to add elements to a vector. The elements are usually added to the end of the vector. `append()` takes three arguments, **x**, the vector that you want to add a value(s) to, **values**, the new value(s) you want to add to the vector, and **after**, which determines the index position where you want the new value(s) slotted into.

```
append(x = fruits, values = "banana", after = 5) # adding new value in index position 5
```

```
[1] "papaya"       "orange"       "apple"        "pineapple"    "grape"
[6] "banana"       "strawberries" "avocado"
```

Multiple values can be added when they are combined with `c()`. If the `after` argument is left empty, the new values are automatically placed at the end of the vector.

```
new_fruits <- c("kiwi", "grape", "cherry", "mango", "peach")
append(fruits, new_fruits)
```

```
 [1] "papaya"       "orange"       "apple"        "pineapple"    "grape"
 [6] "strawberries" "avocado"      "kiwi"         "grape"        "cherry"
[11] "mango"        "peach"
```

5.1.4.0.2 Using the Square Bracket []

To add a new value using the square bracket you have to assign the value to the index position you want the new value to occupy.

```
fruit[2] <- "mango" # add new element to a new index number
fruit
```

```
[1] "papaya" "mango"
```

If you insert an index that is more than the very next index number, you get the index positions filled with `NA` until the position you specified.

```
fruit[5] <- "Clementine"
fruit
```

```
[1] "papaya"     "mango"      NA           NA           "Clementine"
```

To prevent this, you can use a simple trick. Simply pass the length of the fruit + 1 into the square bracket when you want to add a new value to the last position.

```
fruit[length(fruit)+1] <- "Guava"
fruit
```

```
[1] "papaya"     "mango"      NA           NA           "Clementine"
[6] "Guava"
```

You can also remove elements of a vector using the square bracket. You do this by passing a negative value into the index number

```
fruit[-4] # Show values left after removing first value.
```

```
[1] "papaya"     "mango"      NA           "Clementine" "Guava"
```

```
fruit <- fruit[-1] # Reassign to variable to confirm the change.
fruit
```

```
[1] "mango"      NA           NA           "Clementine" "Guava"
```

The square brackets is also used to change the values of object using their index position.

```r
fruit[3] # Position to be changed
```

```
[1] NA
```

```r
fruit[3] <- "tomato" # Replace avocado with tomato
fruit # new elements
```

```
[1] "mango"      NA           "tomato"     "Clementine" "Guava"
```

Multiple positions can also be changed by using `c()` or using `:` for a sequence of index position.

```r
fruit[2:3] <- c("cayenne", "pomegranate")
fruit
```

```
[1] "mango"       "cayenne"     "pomegranate" "Clementine"  "Guava"
```

Note

Using `[]` offers more flexibility than `append()`, as it can be used to add, change and remove the elements of a vector.

5.1.4.1 Recycling

There are instances where we perform operations such as converting from one unit to the other. In the case where our recorded values are more than one, which is usually the case, we would like to convert all of them with a single value. For example: An experiment is designed to evaluate the impact of various feed formulations on the weight gain and overall health of pigs, with the objective of selecting the optimal diet for a new litter to maximize growth and well-being. The weights of 40 pigs that have been administered `feed_x` for 3 months is given below.

```r
set.seed(123)
feed_x <- round(rnorm(n = 40, mean = 87000, sd = 10000), 2)
feed_x
```

```
 [1]  81395.24  84698.23 102587.08  87705.08  88292.88 104150.65  91609.16
 [8]  74349.39  80131.47  82543.38  99240.82  90598.14  91007.71  88106.83
[15]  81441.59 104869.13  91978.50  67333.83  94013.56  82272.09  76321.76
[22]  84820.25  76739.96  79711.09  80749.61  70133.07  95377.87  88533.73
[29]  75618.63  99538.15  91264.64  84049.29  95951.26  95781.33  95215.81
[36]  93886.40  92539.18  86380.88  83940.37  83195.29
```

The value gotten was recorded in grams and we prefer the kilogram. To do this, we simply divide by a 1000

```
feed_x_kg <- round(feed_x / 1000, 2)

feed_x_kg
```

```
 [1]  81.40  84.70 102.59  87.71  88.29 104.15  91.61  74.35  80.13  82.54
[11]  99.24  90.60  91.01  88.11  81.44 104.87  91.98  67.33  94.01  82.27
[21]  76.32  84.82  76.74  79.71  80.75  70.13  95.38  88.53  75.62  99.54
[31]  91.26  84.05  95.95  95.78  95.22  93.89  92.54  86.38  83.94  83.20
```

This is possible because the operation is carried out element-wise, and a phenomenon called **vector recycling** is happening. In vector recycling, vectors of shorter length are repeated multiple times to match up with longer vectors. As long as the longer vector is a multiple of the short vector, we get no warning message as we have below.

```
feed_x_kg * c(0.1, 0.5)
```

```
 [1]  8.140 42.350 10.259 43.855  8.829 52.075  9.161 37.175  8.013 41.270
[11]  9.924 45.300  9.101 44.055  8.144 52.435  9.198 33.665  9.401 41.135
[21]  7.632 42.410  7.674 39.855  8.075 35.065  9.538 44.265  7.562 49.770
[31]  9.126 42.025  9.595 47.890  9.522 46.945  9.254 43.190  8.394 41.600
```

If the longer vector is not a multiple, we get a warning message and the result gets computed regardless, but there will be a cut off in operation.

```
feed_x_kg / c(0.3, 0.3, 0.4)
```

```
Warning in feed_x_kg/c(0.3, 0.3, 0.4): longer object length is not a multiple
of shorter object length
```

```
 [1] 271.3333 282.3333 256.4750 292.3667 294.3000 260.3750 305.3667 247.8333
 [9] 200.3250 275.1333 330.8000 226.5000 303.3667 293.7000 203.6000 349.5667
[17] 306.6000 168.3250 313.3667 274.2333 190.8000 282.7333 255.8000 199.2750
[25] 269.1667 233.7667 238.4500 295.1000 252.0667 248.8500 304.2000 280.1667
[33] 239.8750 319.2667 317.4000 234.7250 308.4667 287.9333 209.8500 277.3333
```

Note

While recycling can be useful, it can also leads to wrong results, so ensure you check if there are warnings and your results are as expected.

5.2 Matrices

5.2.1 What is a Matrix?

A matrix is a two-dimensional data structure that holds elements of the same data type arranged in rows and columns. Think of it as a calendar with with just numbers. To create a matrix, use the `matrix()` function. Since it is two dimensional, you have to specify the number of rows and columns you want. You can specify the number of rows with `nrow` and the number of columns with `ncol`.

```r
mat <- matrix(1:6, nrow = 2, ncol = 3)
mat
```

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

If you use only one of `nrow` or `ncol`, the other gets filled automatically. For example we can create a simple calendar month.

```r
my_calendar <-  matrix(1:30, ncol = 7)
```

```
Warning in matrix(1:30, ncol = 7): data length [30] is not a sub-multiple or
multiple of the number of columns [7]
```

```r
my_calendar
```

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]    1    6   11   16   21   26    1
[2,]    2    7   12   17   22   27    2
[3,]    3    8   13   18   23   28    3
[4,]    4    9   14   19   24   29    4
[5,]    5   10   15   20   25   30    5
```

The calendar looks weird and that's because the numbers are usually filled by columns. To change this arrangement, set the `byrow` to TRUE

```r
my_calendar <- matrix(1:30, ncol = 7, byrow = TRUE)
```

```
Warning in matrix(1:30, ncol = 7, byrow = TRUE): data length [30] is not a
sub-multiple or multiple of the number of columns [7]
```

```r
my_calendar
```

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]    1    2    3    4    5    6    7
[2,]    8    9   10   11   12   13   14
[3,]   15   16   17   18   19   20   21
[4,]   22   23   24   25   26   27   28
[5,]   29   30    1    2    3    4    5
```

Now this looks more like a calendar. But an incomplete calendar, we need the names of the day, and also the week numbers. We will get to this in the next section, but let's address something else which you will come across a lot. That is creating matrix from vector objects. We make a matrix from a vector by passing it to the `matrix()` function and giving a value to either of `nrow` or `ncol`.

```r
fruit_mat <- matrix(fruits, nrow = 4) # make matrix from fruits vector
```

```
Warning in matrix(fruits, nrow = 4): data length [7] is not a sub-multiple or
multiple of the number of rows [4]
```

```r
fruit_mat
```

```
     [,1]        [,2]          
[1,] "papaya"    "grape"       
[2,] "orange"    "strawberries"
[3,] "apple"     "avocado"     
[4,] "pineapple" "papaya"      
```

To get the dimension of matrix use `dim()` function.

```
dim(fruit_mat)
```

```
[1] 4 2
```

The result `[1] 4 2` is interpreted 4 by 2, i.e. four rows and two column. That matrix is thereby called a 4 by 2. The object `my_calendar` is having 5 rows by 7 columns, that makes it a 5 by 7 matrix

5.2.1.1 Naming Rows and Columns

A property of matrix is being two dimensional. From the current result of the calendar, we have [1,] and [,1] signifying rows and columns respectively. These numbers are the column and row indices of a matrix and they can replaced with names of our choosing using the `colnames()` and `rownames()` functions.

```
colnames(my_calendar) <- c("Mon", "Tues", "Wed", "Thurs", "Fri", "Sat", "Sun")

rownames(my_calendar) <- paste("Week", 1:5)

my_calendar
```

```
       Mon Tues Wed Thurs Fri Sat Sun
Week 1   1    2   3     4   5   6   7
Week 2   8    9  10    11  12  13  14
Week 3  15   16  17    18  19  20  21
Week 4  22   23  24    25  26  27  28
Week 5  29   30   1     2   3   4   5
```

Now `my_calendar` looks more like a calendar.

5.2.2 Matrix Operations

Similar to vectors, matrix can perform arithmetic operations. For arithmetic with scalar vector, the operation is carried on each element of the matrix.

```
my_mat <- matrix(1:6, nrow = 3)
my_mat
```

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

```
my_mat * 5
```

```
     [,1] [,2]
[1,]    5   20
[2,]   10   25
[3,]   15   30
```

```
my_mat - 15
```

```
     [,1] [,2]
[1,]  -14  -11
[2,]  -13  -10
[3,]  -12   -9
```

```
my_mat2 <- matrix(2:7, nrow = 3)
my_mat2 / my_mat
```

```
         [,1]     [,2]
[1,] 2.000000 1.250000
[2,] 1.500000 1.200000
[3,] 1.333333 1.166667
```

Matrix with unequal dimension cannot be added together

```r
my_mat3 <- matrix(7:15, nrow = 3)
my_mat3
```

```
     [,1] [,2] [,3]
[1,]    7   10   13
[2,]    8   11   14
[3,]    9   12   15
```

```r
my_mat3 + my_mat
```

```
Error in my_mat3 + my_mat: non-conformable arrays
```

5.2.3 Accessing Elements in a Matrix

Like with vectors, you can access any particular element in a matrix using the square brackets, `[]`. Since matrix is two dimensional, there's a little adjustment in how we access elements as we have to specify rows and columns. The syntax is [**row_index, column_index**]. For example, let's access the third row and second column element of `fruit_mat`.

```r
fruit_mat
```

```
     [,1]        [,2]          
[1,] "papaya"    "grape"       
[2,] "orange"    "strawberries"
[3,] "apple"     "avocado"     
[4,] "pineapple" "papaya"      
```

```r
fruit_mat[3, 2]
```

```
[1] "avocado"
```

We can also access more than one element of a particular axis (row and column) by passing in a vector.

```r
fruit_mat[c(1, 3, 2), 1] # This returns row 1, 3, and 2 elements of column 1
```

```
[1] "papaya" "apple"  "orange"
```

```
fruit_mat[2, c(1, 2)]
```

```
[1] "orange"       "strawberries"
```

You can also use : within an axis to access elements within a range

```
fruit_mat[2:4, c(1, 2)] # returns row 2, 3, and 4 element of column 1 and 2.
```

```
     [,1]        [,2]
[1,] "orange"    "strawberries"
[2,] "apple"     "avocado"
[3,] "pineapple" "papaya"
```

To return all the rows of a particular column, leave the row space within the square bracket empty, and specify the index of the column you want to return

```
fruit_mat[,1]
```

```
[1] "papaya"    "orange"    "apple"     "pineapple"
```

```
fruit_mat[,2]
```

```
[1] "grape"        "strawberries" "avocado"      "papaya"
```

To return all the columns of a particular row, the column space is left empty.

```
fruit_mat[3:4, ] # returns element of row 3 and 4 for all the columns.
```

```
     [,1]        [,2]
[1,] "apple"     "avocado"
[2,] "pineapple" "papaya"
```

You can also access an element using the row and column names.

```
my_calendar["Week 2", ]
```

```
  Mon  Tues   Wed Thurs   Fri   Sat   Sun
    8     9    10    11    12    13    14
```

5.2.4 Adding New Element to a Matrix

To add new columns to a matrix use `cbind()` and to add rows use `rbind()`. We'll create a new matrix of fruits to show this .

```
new_fruit_list <- matrix(c("raspberry", "blue berries",
                           "kiwi", "clementine"),
       nrow = 4)

new_fruit_list
```

```
     [,1]
[1,] "raspberry"
[2,] "blue berries"
[3,] "kiwi"
[4,] "clementine"
```

```
fruit_list <- cbind(fruit_mat, new_fruit_list)
fruit_list
```

```
     [,1]        [,2]           [,3]
[1,] "papaya"    "grape"        "raspberry"
[2,] "orange"    "strawberries" "blue berries"
[3,] "apple"     "avocado"      "kiwi"
[4,] "pineapple" "papaya"       "clementine"
```

To prevent error when combining column-wise ensure the number of rows of the matrices to be joined are equal and the number of columns are equal when combining row-wise.

```
rbind(fruit_list, c("avocado", "pear", "lemon"))
```

```
     [,1]        [,2]           [,3]
[1,] "papaya"    "grape"        "raspberry"
[2,] "orange"    "strawberries" "blue berries"
[3,] "apple"     "avocado"      "kiwi"
[4,] "pineapple" "papaya"       "clementine"
[5,] "avocado"   "pear"         "lemon"
```

5.2.5 Transposing Matrix

Transposing matrix is done using `t()`. This flips row elements to columns and column elements to rows.

```r
fruit_mat_transposed <- t(fruit_mat)
fruit_mat_transposed
```

```
     [,1]     [,2]           [,3]      [,4]       
[1,] "papaya" "orange"       "apple"   "pineapple"
[2,] "grape"  "strawberries" "avocado" "papaya"   
```

```r
dim(fruit_mat_transposed)
```

```
[1] 2 4
```

5.3 Data frames

5.3.1 What is a Data Frame

Data frames are two-dimensional like matrix and are one of the most common way to store data. They are similar to Excel spreadsheets in how they look. What makes data frames different from matrix is their ability to store different data types. To create a data frame, we use the `data.frame()` function.

```r
data.frame(
  x = 1:10,
  y = letters[1:10],
  z = month.abb[1:10]
)
```

```
    x y   z
1   1 a Jan
2   2 b Feb
3   3 c Mar
4   4 d Apr
5   5 e May
6   6 f Jun
7   7 g Jul
```

```
8    8 h Aug
9    9 i Sep
10 10 j Oct
```

Data frames can combine vectors into a table where each vector becomes a column. The vectors to be used have to be of the same length to avoid error.

```
fruit <- c("orange", "mango", "apple")
stock <- c(5, 3, 0)
available <- c(TRUE, TRUE, FALSE)

fruit_tbl <- data.frame(fruit, stock, available)
fruit_tbl
```

```
   fruit stock available
1 orange     5      TRUE
2  mango     3      TRUE
3  apple     0     FALSE
```

5.3.2 Accessing and Modifying Elements in a Data Frame

Elements in data.frame are accessed in a similar fashion as matrix with an extra tweak of their columns being accessible using the dollar sign `$`. Using the dollar sign returns the columns as a vector.

```
fruit_tbl$stock
```

```
[1] 5 3 0
```

When accessing a column with its name and the square bracket, it's not necessary to include the comma when the name is in quotation marks. The column with the specified name will be returned. This is also the same when you use index number, once a comma is not included, the column is automatically returned

```
fruit_tbl["stock"]
fruit_tbl[2]
```

You can still use the commas to return specific rows and columns if you choose to, or return a particular observation as shown below:

```
  stock
1     5
2     3
3     0
```

```
  stock
1     5
2     3
3     0
```

```r
fruit_tbl[2, 3]
```

```
[1] TRUE
```

Using the square brackets, and index number of the column we want to remove, we can remove old columns or add new columns to a data frame.

```r
# Removing a Column
fruit_tbl[-1]
```

```
  stock available
1     5      TRUE
2     3      TRUE
3     0     FALSE
```

```r
# Adding a column
fruit_tbl["location"] <- c("online", "on site", "online")
fruit_tbl
```

```
   fruit stock available location
1 orange     5      TRUE   online
2  mango     3      TRUE  on site
3  apple     0     FALSE   online
```

5.4 List

5.4.1 What is a List

A list is a versatile data structure that can store elements of different types and sizes - including other data structures like vectors, matrices, and even other lists. Think of it like a container that can hold different kinds of boxes.

```r
# Basic list creation
my_list <- list(
  fruits,
  fruit_mat,
  c(TRUE, FALSE),
  FALSE,
  fruit_mat_transposed,
  fruit_tbl,
  my_calendar
)

my_list
```

```
[[1]]
[1] "papaya"       "orange"       "apple"        "pineapple"    "grape"
[6] "strawberries" "avocado"

[[2]]
     [,1]        [,2]
[1,] "papaya"    "grape"
[2,] "orange"    "strawberries"
[3,] "apple"     "avocado"
[4,] "pineapple" "papaya"

[[3]]
[1]  TRUE FALSE

[[4]]
[1] FALSE

[[5]]
     [,1]     [,2]           [,3]      [,4]
[1,] "papaya" "orange"       "apple"   "pineapple"
[2,] "grape"  "strawberries" "avocado" "papaya"

[[6]]
   fruit stock available location
1 orange     5      TRUE   online
2  mango     3      TRUE  on site
3  apple     0     FALSE   online

[[7]]
```

```
       Mon Tues Wed Thurs Fri Sat Sun
Week 1   1    2   3     4   5   6   7
Week 2   8    9  10    11  12  13  14
Week 3  15   16  17    18  19  20  21
Week 4  22   23  24    25  26  27  28
Week 5  29   30   1     2   3   4   5
```

5.4.2 Naming Objects in a List

The objects in a list can be given a name using `names()` function.

```
names(my_list) <- c("fruits", "fruit_mat", "logical_1", "logical_2",
                    "fruit_transposed_matrix", "fruit_dataframe", "my_calendar_matrix")
```

On printing list now, we'll see each object named.

```
my_list
```

```
$fruits
[1] "papaya"       "orange"       "apple"        "pineapple"    "grape"
[6] "strawberries" "avocado"

$fruit_mat
     [,1]        [,2]
[1,] "papaya"    "grape"
[2,] "orange"    "strawberries"
[3,] "apple"     "avocado"
[4,] "pineapple" "papaya"

$logical_1
[1]  TRUE FALSE

$logical_2
[1] FALSE

$fruit_transposed_matrix
     [,1]     [,2]           [,3]      [,4]
[1,] "papaya" "orange"       "apple"   "pineapple"
[2,] "grape"  "strawberries" "avocado" "papaya"

$fruit_dataframe
```

```
   fruit stock available location
1 orange     5      TRUE   online
2  mango     3      TRUE  on site
3  apple     0     FALSE   online

$my_calendar_matrix
       Mon Tues Wed Thurs Fri Sat Sun
Week 1   1    2   3     4   5   6   7
Week 2   8    9  10    11  12  13  14
Week 3  15   16  17    18  19  20  21
Week 4  22   23  24    25  26  27  28
Week 5  29   30   1     2   3   4   5
```

5.4.3 Accessing Elements in a List

The square bracket, [] is used to select elements in a list but to print the items in the object you use [[]]. Like data.frame you can also use the dollar sign, **$**,

```
my_list[1]
```

```
$fruits
[1] "papaya"       "orange"       "apple"        "pineapple"    "grape"
[6] "strawberries" "avocado"
```

Notice the difference when we use [[]]. The result is similar to using $ to access the objects within the list

```
my_list[[1]]
```

```
[1] "papaya"       "orange"       "apple"        "pineapple"    "grape"
[6] "strawberries" "avocado"
```

```
my_list$fruits # similar to [1]
```

```
[1] "papaya"       "orange"       "apple"        "pineapple"    "grape"
[6] "strawberries" "avocado"
```

To access the item in each object add a square bracket in front.

```r
my_list[[1]][3]
```

```
[1] "apple"
```

Tip

To confirm an object data structure, use their `is.*()` variant. For vectors, `is.vector()`, for matrix, `is.matrix()` and so on. To convert from one structure to another, use their `as.*()` variant. For vector `as.vector()`, for matrix, `as.matrix()` and so on.

```r
is.vector(my_calendar)
```

```
[1] FALSE
```

```r
is.vector(fruits)
```

```
[1] TRUE
```

```r
is.matrix(my_calendar)
```

```
[1] TRUE
```

```r
as.data.frame(my_calendar)
```

```
       Mon Tues Wed Thurs Fri Sat Sun
Week 1   1    2   3     4   5   6   7
Week 2   8    9  10    11  12  13  14
Week 3  15   16  17    18  19  20  21
Week 4  22   23  24    25  26  27  28
Week 5  29   30   1     2   3   4   5
```

5.5 Bringing it all together

5.6 Summary

In this chapter you learned about data structures in R. You saw how to create each data structure, how to access the elements of each structure. Also, you learned how to add and remove items from each data structure. You got introduced to checking the properties of each data structure such as their length and dimensions. Next you will learn about packages in R and how to install them, after you will bring together the knowledge you've gathered from chapter one into making R Scripts and R projects.

6 Control Structure

::: Chapter is still under construction :::

Control structures are components of programming that determines the order in which instructions are executed within a program. It allows us control the flow of execution of R expressions (Peng 2016). Control structure allows us to execute different expressions based on conditions.

Basically, they introduce some level of strictness and flexibility at the same time which is determined by the programmer. In addition to getting results based on conditions, they prevents unnecessary manual repetition of the same instruction through loops. Some of the control structure functions in R are:

- `if()`, `else()`, `ifelse()` and `case_when()`
- `for()`
- `while()`
- `switch()`
- `repeat`
- `break` and `next`

While we won't cover the whole list provided here, we will cover enough to help you in your research workflow. For example, case_when is not a part of base R but of a package, a concept which we will cover Chapter 7. To get more info on control structure, read Chapter 13 of R Programming for Data Science by Roger D Peng.

6.1 `if`, `else`, and `ifelse`

The `if` statement is exactly as it sounds:

> if …. then do ……

The syntax of `if` statement is written as: `if()` with the condition or test in the bracket then `{}`, the expression of what should happen if the condition in the parentheses is passed is written in the curly braces. In simple terms we can relate it as *if a certain condition is met then do the following:*

```
if (condition) {
  expression
}
```

For example let's multiply a number by 2 if its greater than 5.

```
x <- 10

if (x > 5) {
  x * 2
}
```

```
[1] 20
```

The `if` statement only does something when one result of the condition, `TRUE`, is met, and does nothing when the result equates to `FALSE`. If `x` is lesser than 5 we won't get a result because `x` did not pass the test.

```
x <- 3
if (x > 5) {
  x * 2
}
```

This is when the `else` clause comes in handy to give results when the test in the `if` are `FALSE`. It is important to note that the if clause operates only on single element and the `else` clause is used in combination with the `if` clause. The syntax is now updated to look like the following:

```
if (condition) {
  expression_1
} else {
  expression_2
}
```

Let's build on the previous last example to see how `else` works

```
x <- 15
if (x < 5)  {
  print(paste("The number", x, "is less than 5"))
} else {
  print(paste("The number", x, "is greater than 5"))
}
```

```
[1] "The number 15 is greater than 5"
```

As seen above, the else does not hold any condition, it simply handles all other conditions not handled by the `if` statement. If you want to introduce more structure to your code, testing more than two conditions you can use the `else if` clause. The `else if` comes after the `if` clause, the syntax is shown below:

```
if (condition) {
  expression_1
} else if (condition) {
  expression_2
} else {
  expression_3
}
```

```
x <- 10

if (x > 10) {
  print(paste("The number", x, "is greater than 10"))
} else if (x == 10) {
  print("The number is exactly 10")
} else {
  print(paste("The number", x, "is less than 10"))
}
```

```
[1] "The number is exactly 10"
```

There's a show form of the `if` and `else` statement, the `ifelse()`, and it takes the following syntax

```
ifelse(test_or_condition, result_if_true, result_if_false)
```

For example:

```
x <- 11
ifelse(x == 10, x - 2, x + 2)
```

```
[1] 13
```

Since x is not equal to 10, the expression when false is executed. while all the different methods of the if-else statement works, they can lead to a long chain of command, which can be difficult to follow. An alternative method is to use `switch` or `case_when()` of `dplyr` package. `case_when()` is an inspiration from SQL and won't be covered now but used as we progress. However, the syntax is given below:

```
case_when(
  condition_1 ~ result/expression,
  condition_2 ~ result_2/expression,
  .default = default_result/default_expression
)
```

6.2 `for` Loops

`for` loops are used for iterating over elements of objects such as list, vector, matrices, and so on (Peng 2016). The syntax of the `for` is straightforward. It takes the `for` clause, an iterator and an object covered in parenthesis and a curly bracket which takes the expression of what you want to do on the iteration of elements of the object.

```
for (iterator in object) {
  expression
}
```

Let's deconstruct the parts of the `for` loop:

- for: The keyword that initiates the loop structure.
- iterator: A variable that takes on the value of each element in the object sequentially during each iteration of the loop. You can choose any valid variable name for the iterator.
- object: This is the sequence (like a vector, list, or other iterable structure) over which the loop will iterate. The loop will run once for each element contained within this object.
- expression: This is the block of code enclosed in curly braces {} that will be executed for each value assigned to the iterator. This is where you put the actions you want to perform on each element (or based on each element) of the object.

In simple terms, the for loop goes through each item in the object, assigns that item's value to the iterator variable, and then runs the expression using that iterator value, repeating until all items in the object have been processed. Let's take the example below:

```
num <- 1:10

for (i in num){
  print(i + 3)
}
```

```
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
[1] 11
[1] 12
[1] 13
```

`i` here is the iterator and takes each element, and in each iteration of the loop it given the values 1, 2, 3, …, 10. For the first iteration where `i` is equal to 1, the expression `print(i + 3)` is executed and result given as 4 (1 + 3). The code is started again for the next value, 2, and now `i` is equal to 2. The expression is repeated again `print(i + 3)` and result is given as 5 (2 + 3). This repeated continuously until the end of the loop.

It is not uncommon to use some functions such `seq_along()`, `length()` in the `for` loop in order to generate a sequence based on the object.

```
set.seed(123)

my_num <- runif(10, min = -1, max = 1)

for (i in seq_along(my_num))  {
  print(my_num[i])
}
```

```
[1] -0.424845
[1] 0.5766103
[1] -0.1820462
[1] 0.7660348
[1] 0.8809346
[1] -0.908887
[1] 0.05621098
[1] 0.7848381
[1] 0.10287
[1] -0.08677053
```

```r
for (i in 1:length(my_num)) {
  print(my_num[i])
}
```

```
[1] -0.424845
[1] 0.5766103
[1] -0.1820462
[1] 0.7660348
[1] 0.8809346
[1] -0.908887
[1] 0.05621098
[1] 0.7848381
[1] 0.10287
[1] -0.08677053
```

Tip

The iterator can be given any name as long as it follows variable naming conventions, but it is advised to use the name that reflects the objects in the collection. For example:

Object	Iterator
fruits	fruit
names	name
cities	city
products	product

They are common stand in iterator variables popular in the programming world such as `i`, `j` or `k`.

6.2.1 Nesting the `for` Loop

You can nest `for` loops inside each other. In this case indent your code, so you can understand the flow of your code. An example is given below:

```r
x <- 20
y <- 1:10
for (i in seq_len(x)) {
  z <- ""
  for (j in seq_along(y)) {
    z <- j * 2
  }
  a <- z + i
  print(a)
}
```

```
[1] 21
[1] 22
[1] 23
[1] 24
[1] 25
[1] 26
[1] 27
[1] 28
[1] 29
[1] 30
[1] 31
[1] 32
[1] 33
[1] 34
[1] 35
[1] 36
[1] 37
[1] 38
[1] 39
[1] 40
```

6.3 `while` Loop

`while()` loops test a condition and keeps iterating until they are FALSE and then exits the loop.

```
x <- 0

while (x < 10) {
  print(paste("The value of x was", x))
  x = x + 1
  print(paste("The value of x is now", x))
}
```

```
[1] "The value of x was 0"
[1] "The value of x is now 1"
[1] "The value of x was 1"
[1] "The value of x is now 2"
[1] "The value of x was 2"
[1] "The value of x is now 3"
[1] "The value of x was 3"
[1] "The value of x is now 4"
[1] "The value of x was 4"
[1] "The value of x is now 5"
[1] "The value of x was 5"
[1] "The value of x is now 6"
[1] "The value of x was 6"
[1] "The value of x is now 7"
[1] "The value of x was 7"
[1] "The value of x is now 8"
[1] "The value of x was 8"
[1] "The value of x is now 9"
[1] "The value of x was 9"
[1] "The value of x is now 10"
```

When using the while loop, the object value should always be updated to ensure there's an end to the loop. In the example above, x keeps updating until it reaches the condition where FALSE is the result. Without the object updating, the loop will continue forever.

Use the `while` loop with care, if not used with care, the code will keep running till infinity. You can break this infinite loop with `CTRL/CMD+C`

6.4 `switch`

`switch` is used to select to select one of several alternatives based on the value of an expression. It's like an `ifelse()` statement but more concise. `switch` are used in two ways depending on their first argument.

- a character strung to match, or
- an integer index position to match

The syntax of `switch` is given below:

```
switch (object,
  case = action
)
```

The `case = action` argument can be repeated multiple times.

```
x <- "a"

switch (x,
  a = "Man",
  b = "Woman",
  c = "boy",
  d = "girl"
)
```

```
[1] "Man"
```

Let's take an example using the integer index position

```
y <- 3

switch(y,
   "Apple",
   "Mango",
   "Clementine"
)
```

```
[1] "Clementine"
```

If there's no match for the object, NULL is returned. we can add a default option to return to by adding a value at the end that doesn't match any case name

```
x <- "dog"

switch(x,
  man = "tade",
  woman = "sarah",
  boy = "zee",
  "bruno"

)
```

```
[1] "bruno"
```

6.5 Bringing it all together

6.6 Summary

Control structures in R are essential tools that allow you to direct the flow of execution in your code based on conditions or repetitions. They help make your code dynamic, responsive, and efficient by avoiding redundancy and enabling decision-making. In this chapter, you explored:

- Conditional statements like `if`, `else`, and `ifelse()` that let you run different code depending on whether a condition is true or false.
- The `switch()` function, which provides a cleaner way to select one of many alternatives based on a matching value or position.
- Loops such as `for` and `while`, which repeat actions across multiple elements or until a condition is no longer met.
- How to nest loops and use helper functions like `seq_along()` and `length()` to control iteration.
- The importance of updating conditions in loops to avoid infinite execution.

Together, these structures form the building blocks for writing programs that can automate tasks, process data, and respond intelligently to varying inputs in your research workflow.

7 Packages

This section is still under development

In Chapter 4, we saw that R comes with many built-in functions. These functions are pre-installed in what is called the **Base packages**. They include `base`, `compiler`, `datasets`, `grDevices`, `graphics`, `grid`, `methods`, `parallel`, `splines`, `stats`, `stats4`, `tcltk`, `tools`, `translations`, and `utils` (R Core Team 2024). These packages can handle a plethora amount of tasks, as your new mobile phones handle a lot of tasks. Notwithstanding, one of the first few things you do would be to go to playstore and appstore to download apps that your native device app However, the true power of R lies in its extensive ecosystem of additional packages created by the R community. These packages extend R's capabilities, similar to how third-party apps like WhatsApp, Telegram or Signal extend a phone's basic messaging capabilities with features like video calls, read receipts, status updates and so on.

7.1 Package Repositories

7.1.1 CRAN (Primary Repository)

Like Apple store and Play store, most R packages are hosted on CRAN (Comprehensive R Archive Network), R's official package repository. CRAN provides rigorous quality control, ensuring packages meet strict guidelines for reliability and documentation. The tidyverse, for example, is a popular meta-package from CRAN that includes several essential packages for data science like dplyr, ggplot2, and others, which we'll use extensively in this training.

7.1.1.1 Installing Packages from CRAN

To install packages from CRAN, use `install.packages()`:

```r
# Install the tidyverse package

install.packages("tidyverse")
```

Package installation requires an internet connection and only needs to be done once per package.

7.1.2 Alternative Repositories

7.1.2.1 Bioconductor

Bioconductor is a specialized repository focusing on packages for bioinformatics and genomic data analysis. It maintains its own installation system:

```r
# Install Bioconductor's package manager
if (!require("BiocManager", quietly = TRUE))
    install.packages("BiocManager")

# Install Bioconductor packages
BiocManager::install("DESeq2")  # Example package for RNA-seq analysis
```

7.1.2.2 GitHub and Other Development Platforms

Many R package developers share their work on platforms like GitHub, GitLab, or Bitbucket before official CRAN submission or for experimental features. To install packages from GitHub, you'll need the devtools or remotes package:

```r
# Install devtools first if you haven't
install.packages("devtools")
```

Using `devtools`

```r
# Install a package from GitHub
# Format: devtools::install_github("username/repository")
devtools::install_github("clauswilke/ggtextures")  # Latest the ggtexture package

# Specify a specific branch, tag, or commit
devtools::install_github("username/repository@branch")
devtools::install_github("username/repository@v1.0.0")  # Install a specific version
```

Note

For package download statistics, visit R Package Stats.

While alternative repositories offer cutting-edge or specialized packages, CRAN remains the primary and most reliable source for R packages. Always prefer CRAN versions for production environments unless you specifically need features from other sources.

7.2 Using Installed Packages

There are two ways to use installed packages:

1. **Loading Entire Packages** Use the `library()` function to load all functions from a package:

```r
library(tidyverse)
```

If you see this error:

> Error in library(tidyverse) : there is no package called 'tidyverse'

It means the package needs to be installed first using `install.packages("tidyverse")`.

2. **Declarative Use of Functions** To use individual functions without loading the entire package, use the package name followed by `::` and the function name:

```r
stats::lag()     # Use lag() from stats package
dplyr::lag()     # Use lag() from dplyr package
```

This method is particularly useful when different packages have functions with the same name (like `lag()` and `filter()` in dplyr and stats).

Remember: While package installation is a one-time process, you must reload packages with `library()` in each new R session.

7.3 Common Package Installation Errors

When installing R packages, you might encounter various errors. Here's how to identify and resolve the most common ones:

1. **Forgetting the quotation marks**. This is common, especially amongst beginners

```r
install.packages(tidyverse)
```

```
Error: object 'tidyverse' not found
```

Fix:

- Wait for the auto-complete option and select the package, it automatically inserts the quotation marks.
- Manually enclose the package name in quotation marks.

2. **Permission Errors** This typically occurs when R doesn't have write permissions to the installation directory.

Fix:

- Windows: Run RStudio as administrator
- Linux/Mac:

```r
install.packages("tidyverse", lib = "~/R/library")
```

Or use `sudo R` in terminal (not recommended)

Summary

R packages extend base R's functionality, providing specialized tools for various analyses. They're primarily hosted on CRAN and can be installed using `install.packages()`. After installation, packages can be used either by loading them entirely with `library()` or by accessing specific functions using the `package::function` syntax. While installation is permanent, packages must be reloaded in each new R session.

8 Workflow in R: Best Practices for Efficient R Programming

This section is still under development

Early when learning to program, it is easy to focus on getting our code to work as expected. While this is good, we pay less attention on organizing and styling our code, things which is something that can be learned alongside getting our codes to work. Adopting good programming practices makes our work efficient, easy to follow and understand our logic when we revisit the code, easy for others to reproducible and collaborate with us.

For a comprehensive exposition on good coding practices read the Tidyverse Style Guide.

8.1 Names

Naming is an important part of the coding process, as you will name your scripts, name your variables and name your custom functions. This might be tedious at first but with time you get better giving names. There a few caveat when it comes to naming.

8.1.1 Files

- For scripts, ensure the names are meaningful and end in `.R` extension.

```
chicken.R
import.R
thesis.R
```

- For names that have more than one words, the words should be separated by `-` or `_`.

```
# Good
wrangle-data.R
wrangle_data.R
joburg.R
alaves.R
lagos.R
```

- File names should start with names or numbers. Use numbers if your files are modular or organized in a pipeline.

```
# Good
01_dependcies.R
02_import_data.R
03_clean_data.R
```

- For the internal structure of your file ensure you break your script into sections. A shortcut to do this in RStudio is `CTRL/CMD` + `SHIFT` + `R`. This command opens up a dialog box to input the section title.

```
# Load packages ----------------------------------------------------------
library(tidyverse)
library(ggthemes)

# Load data --------------------------------------------------------------

# Wrangle data -----------------------------------------------------------
```

8.2 The R Script

The R script is a file containing R commands. The commands are run sequentially from the top to the bottom of the script. This allows you to break your scripts into sections which can be useful when you share your script to other people as it make them understand what a part of your code do.

8.2.1 Creating an R Script

- **Opening a New Script**: Click the **+** button in the top-left corner of RStudio and select **R Script**, or use the shortcut `CTRL` + `SHIFT` + `N` (Windows/Linux) or `CMD` + `SHIFT` + `N` (Mac).

- **Writing Code**: Enter your R commands in the script editor. This might include simple arithmetic, importing your data, cleaning, analysis, and visualization code.
- **Saving the Script**: Save your script by clicking **File** > **Save**, or using the shortcut `CTRL` + `S` (Windows/Linux) or `CMD` + `S` (Mac).
- **Running the Script**: Execute the entire script or selected lines using the **Run** button or the `CTRL` + `ENTER` (Windows/Linux) or `CMD` + `ENTER` (Mac) shortcut.

8.2.2 Benefits of Using R Scripts

- With R scripts instead of console, you can share your work with others
- Scripts allow you to document your entire workflow and reproduce it easily
- Once written, you can reuse your script for similar tasks in the future.

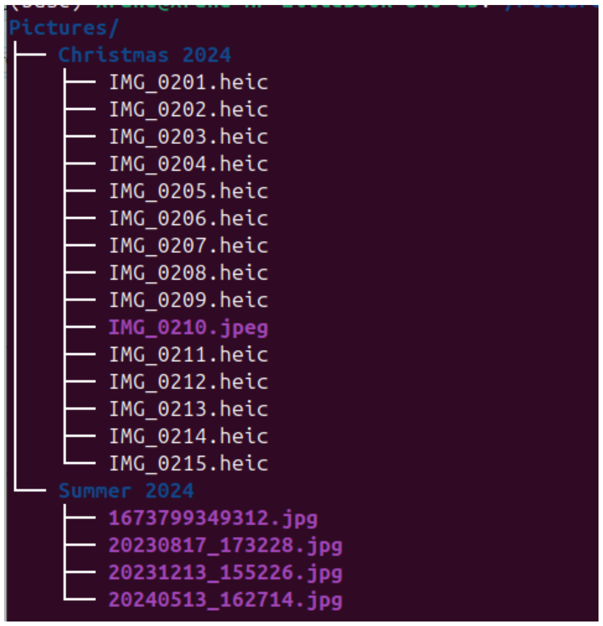
8.3 The R Project

As we continue to progress in our career and increase our use of R in handling different projects, it becomes imperative that we shift focus from solving problems alone to organizing how we solve our problems. Since a lot of research life cycle is almost the same, and the data analysis stage of research almost follows the same pattern, carefully locating each projects which we are working on is key to a stressfree workflow. This is where R projects becomes super important, as it helps us stay organize.

When should you use an R project? For all new projects . This is the logical thing to do. R projects makes us switch from one project to another without worrying about where the project-specific files are. To see more on why you should work more with projects, read chapter 3 What They Forgot to Teach You About R.

8.4 Creating an R Project

Before we create any project, we need to understand file organizations. Preinstalled with many operating system is the file manager. In file managers, we have some folders created readily, such as Music, Pictures, Videos, Documents, Downloads, and Desktop or Home. These folders ensure we organize our files based on their types and/or purposes. The memorable images we took last christmas is sure to be saved in a pictures folder. If we want to be more organized we can ensure the christmas pictures have it owns dedicated folder within the Pictures folder. A simple diagram of such file organization is shown in Figure 8.1



```
Pictures/
├── Christmas 2024
│   ├── IMG_0201.heic
│   ├── IMG_0202.heic
│   ├── IMG_0203.heic
│   ├── IMG_0204.heic
│   ├── IMG_0205.heic
│   ├── IMG_0206.heic
│   ├── IMG_0207.heic
│   ├── IMG_0208.heic
│   ├── IMG_0209.heic
│   ├── IMG_0210.jpeg
│   ├── IMG_0211.heic
│   ├── IMG_0212.heic
│   ├── IMG_0213.heic
│   ├── IMG_0214.heic
│   └── IMG_0215.heic
└── Summer 2024
    ├── 1673799349312.jpg
    ├── 20230817_173228.jpg
    ├── 20231213_155226.jpg
    └── 20240513_162714.jpg
```

Figure 8.1: Example of a file tree showing the contents of Pictures folder and the contents of its subfolders Christmas 2024 and Summer 2024

A similar level of organization is needed when you create R Projects, you can have all your data in a folder, images in another folder, and have another folder housing your R scripts or manuscript.

To create a project click on *File > New Project* on the top-left edge of RStudio. You can create a new directory, use an existing directory, or clone a project from a version control system repository. Alternatively, on the top-right edge of RStudio you will see a drop down button (horizontal arrow) that is like what is shown in Figure 8.2, you can create a new project from there, as well as recent projects (rectangular bar). Using that button also gives you the option to open multiple projects at once by clicking the icon on the edge (second arrow). I tend to use this button a lot, and I think getting used to it saves you a lot of trouble.

Summary

We have cleared the first step to using R effectively and that's getting the basics down. This chapter shows the first approach towards using R for our research. These are the steps:

- First make use of projects for each project. Using the project button at the right-top edge of RStudio could be useful for this.
- Secondly make use of R scripts. R Scripts ensures your work are usable.

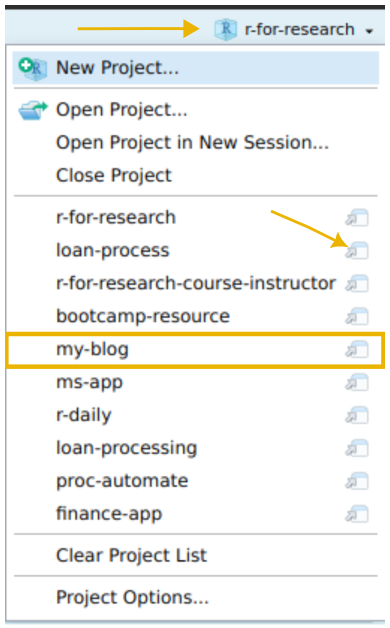


Figure 8.2: The R Project button on the top-right edge of RStudio. This drop down gives you options to create new project, switch to a different project, and open multiple projects at once.

More advanced features like using version control would be covered in part II of this book.

Part II

Wrangle

! Section will be written after whole content of this part is finished

9

10 Getting External Data into RStudio

This section is still under development

When working in the data field, you are more likely to get the data from external sources than record it manually yourself, and being honest, recording your data in R is a big task, especially when it is large. While it's difficult to record data in R, it is possible using `edit()` which opens an empty box for you to input data. Good thing is that lab equipment have features that makes them record data as they are measured. These recorded data can later be exported as spreadsheet file or plain-text files.

10.1 Important Concepts

Before we go into data importation, it is important we understand some concepts that will make this process easy and less error prone.

10.1.1 Data Sources

When we want to import data into R, we need to know where to get it from. ***A data source is the place where the data we want to use originates from. This can be a physical or digital place*** A data source could be any of a live measurements from physical devices, a database, a flat file or plain-text file, scraped web data, or any of the innumerable static and streaming data services which abound across the internet. An example of a data source is data.gov, and wikipedia and these are web data. Another example that is actually close to you is your mobile phone. It's a mobile database holding contact information, music data, games, and so on.

10.1.2 Data Format

Data formats *represent the form, structure and organization of data.* It defines how information is stored, accessed, and interpreted. It's a standardized way to represent data, whether in files or databases, and is crucial for efficient data management and processing. This brings us to another aspect that is important to knowing how to handle data, and that's **file extension**.

If you've taken a closer look at the music/audio file on any of your device, you usually see a dot which separates the name of the file and a text which is usually fairly consistent depending on your file organization. This text is regarded to as the **file extension**. This is also true for your video, and picture files. For example, you could see the following extensions:

Table 10.1: Some of the common multimedia file format including audio, picture and video files.

Audio	Pictures	Video
.mp3	.png	.mp4
.aac	.jpeg	.mov
.flac	.gif	.avi
.wav	.webp	.web,
.ogg	.tiff	.mov

For data set, there are some common file format such as:

- flat file : `.csv`, `.tsv`, `.txt`, `.rtf`
- some web data format excluding those above: `.html`, `.json`, `xml`
- spreadsheet: `.xlsx`, `.xls`, `.xlsm`, `.ods`, `.gsheet`
- others: `.shp`, `.hdsf`, `.sav`, and many more.

10.2 Importing Data {sec-import-section}

Data is imported into R based on the file format you are dealing with. We will import some of the file types you would come across when working with data in R.

10.3 Flat file data

Flat files are one of the most common forms in which data are stored. They are basically text files with a consistent structure. To import text files, we can use either of base R `utils`, `data.table` or `readr`. There are still other packages that we can use to import flat files, but, the ones provided here should be sufficient for any flat file. Firstly let's import `pacman` and use it's `p_load()` function to import the packages we need. If you do not have `pacman` installed, you can use `install.packages( "< package name >" )` to do so.

Note

`pacman` makes data importation and management very easy. It makes download easy without the need for quotation marks as required when using `install.packages()`. It also has the `p_load()` function which is used to download, install, and load packages at the same time, performing the function of `install.packages`, and `library` at once.

```r
library(pacman)
p_load(readr, data.table)
```

Without `pacman`, the steps would be:

```r
install.packages(c("data.table", "readr")) # this can be further broken down
library(data.table)
library(readr)
```

This is what makes `pacman` shines to me. You could also run the below, if you do not want to load `pacman` itself but just make use of its function.

```r
pacman::p_load(readr, data.table)
```

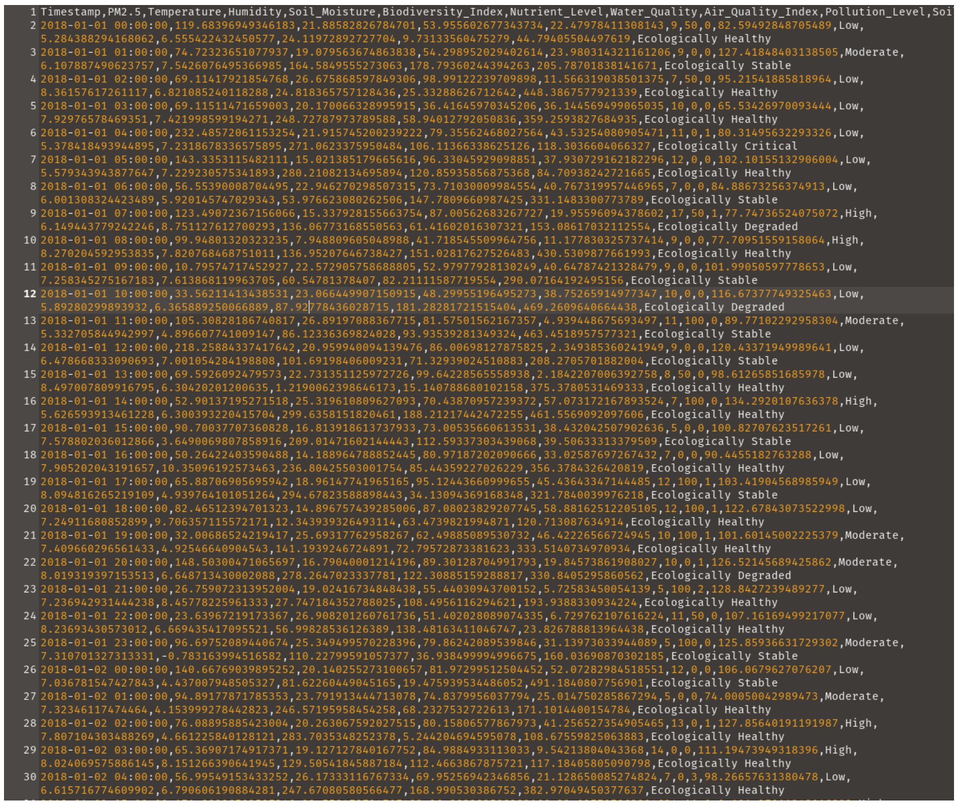
Now that we are done with installing and loading our packages, let's take a closer look at the different flat files. There are two things we should consider when dealing with flat files:

- The extension of the file.
- The structure of file when opened.

To some degree, the extension of your file could give you a hint on how to read it into R. .csv, would need a csv file reading function, .txt, would need its own also. Notwithstanding, the internal structure is what makes reading a file easier. In flat files data, the data presented in a way that mimics standard spreadsheet table, with the difference being in how their recorded are separated. While a spreadsheet is having its rows and column clearly defined, flat files have its own separated using a consistent sign or symbol in the document. These signs and symbols are regarded to as **delimiters**. For example, Figure 10.1 have it's columns separated by commas, so the delimiter is a comma, thus the name comma separated values.

If it is separated by semi-colon, the delimiter is a semi colon, but the name remains the same. The interesting thing is that comma separated files can still have their name saved in `.txt` format, that's why checking the file internal structure is important. Although almost anything can be used as a delimiter. Some common ones includes:

- comma - ,



```
Timestamp,PM2.5,Temperature,Humidity,Soil_Moisture,Biodiversity_Index,Nutrient_Level,Water_Quality,Air_Quality_Index,Pollution_Level,Soi
2018-01-01 00:00:00,119.68396949346183,21.88582826784701,53.955602677343734,22.47978411308143,9,50,0,82.59492848705489,Low,
5.284388294168062,6.555422432450577,24.11972892727704,9.73133560475279,44.79405504497619,Ecologically Healthy
2018-01-01 01:00:00,74.72323651077937,19.079563674863838,54.298952029402614,23.980314321161206,9,0,0,127.41848403138505,Moderate,
6.107887490623757,7.5426076495366985,164.5849555273063,178.79360244394263,205.78701838141671,Ecologically Stable
2018-01-01 02:00:00,69.11417921854768,26.675868597849306,98.99122239709898,11.566319038501375,7,50,0,95.21541885818964,Low,
8.36157617261117,6.821085240118288,24.818365757128436,25.33288626712642,448.3867577921339,Ecologically Healthy
2018-01-01 03:00:00,69.11511471659003,20.170066328995915,36.41645970345206,36.144569499065035,10,0,0,65.53426970093444,Low,
7.92976578469351,7.421998599194271,248.72787973789588,58.94012792050836,359.2593827684935,Ecologically Healthy
2018-01-01 04:00:00,232.48572061153254,21.915745200239222,79.35562468027564,43.53254080905471,11,0,1,80.31495632293326,Low,
5.378418493944895,7.2318678336575895,271.0623375950484,106.11366338625126,118.3036604066327,Ecologically Critical
2018-01-01 05:00:00,143.3353115482111,15.021385179665616,96.33045929098851,37.930729162182296,12,0,0,102.10155132906004,Low,
5.579343943877647,7.229230575341893,280.21082134695894,120.85935856875368,84.70938242721665,Ecologically Healthy
2018-01-01 06:00:00,56.55390008704495,22.946270298507315,73.71030009984554,40.767319957446965,7,0,0,84.88673256374913,Low,
6.001308324423489,5.920145747029343,53.976623080262506,147.7809660987425,331.1483300773789,Ecologically Stable
2018-01-01 07:00:00,123.49072367156066,15.337928155663754,87.00562683267727,19.95596094378602,17,50,1,77.74736524075072,High,
6.149443779242246,8.751127612700293,136.06773168550563,61.41602016307321,153.08617032112554,Ecologically Degraded
2018-01-01 08:00:00,99.94801320323235,7.948809605048988,41.718545509964756,11.177830325737414,9,0,0,77.70951559158064,High,
8.270204592953835,7.820768468751011,136.95207646738427,151.02817627526483,430.5309877661993,Ecologically Healthy
2018-01-01 09:00:00,10.79574717452927,22.572905758688805,52.97977928130249,40.64787421328479,9,0,0,101.99050597778653,Low,
7.258345275167183,7.613868119963705,60.54781378407,82.21111587719554,290.07164192495156,Ecologically Stable
2018-01-01 10:00:00,33.56211413438531,23.066449907150915,48.29955196495273,38.75265914977347,10,0,0,116.67377749325463,Low,
5.89280299893932,6.365889250066889,87.92778436028715,181.28281721515404,469.2609640664438,Ecologically Degraded
2018-01-01 11:00:00,105.30828186740817,26.89197088367715,81.57501562167357,4.939448675693497,11,100,0,89.77102292958304,Moderate,
5.332705844942997,4.896607741009147,86.12336369824028,93.93539281349324,463.4518957577321,Ecologically Stable
2018-01-01 12:00:00,218.25884337417642,20.959940094139476,86.00698127875825,2.349385360241949,9,0,0,120.43371949989641,Low,
6.478668333090693,7.001054284198808,101.69198406009231,71.32939024510883,208.2705701882004,Ecologically Stable
2018-01-01 13:00:00,69.5926092479573,22.731351125972726,99.64228565558938,2.1842207006392758,8,50,0,98.61265851685978,Low,
8.497007809916795,6.30420201200635,1.2190062398646173,15.140788680102158,375.3780531469333,Ecologically Healthy
2018-01-01 14:00:00,52.90137195271518,25.319610809627093,70.43870957239372,57.073172167893524,7,100,0,134.2920107636378,High,
5.626593913461228,6.300393220415704,299.6358151820461,188.21217442472255,461.5569092097606,Ecologically Healthy
2018-01-01 15:00:00,90.70037707360828,16.813918613737933,73.00535660613531,38.432042507902636,5,0,0,100.82707623517261,Low,
7.578802036012866,3.6490069807858916,209.01471602144443,112.59337303439068,39.50633313379509,Ecologically Stable
2018-01-01 16:00:00,50.26422403590488,14.188964788852445,80.97187202090666,33.02587697267432,7,0,0,90.4455182763288,Low,
7.905202043191657,10.35096192573463,236.80425503001754,85.44359227026229,356.3784326420819,Ecologically Healthy
2018-01-01 17:00:00,65.88706905695942,18.96147741965165,95.12443660999655,45.43643347144485,12,100,1,103.41904568985949,Low,
8.094816265219109,4.939764101051264,294.67823588898443,34.13094369168348,321.7840039976218,Ecologically Stable
2018-01-01 18:00:00,82.46512394701323,14.896757439285006,87.08023829207745,58.88162512205105,12,100,1,122.67843073522998,Low,
7.24911680852899,9.706357115572171,12.343939326493114,63.4739821994871,120.713087634914,Ecologically Healthy
2018-01-01 19:00:00,32.00686524219417,25.69317762958267,62.49885089530732,46.42226566724945,10,100,1,101.60145002225379,Moderate,
7.409660296561433,4.92546640904543,141.1939246724891,72.79572873381623,333.5140734970934,Ecologically Healthy
2018-01-01 20:00:00,148.50300471065697,16.79040001214196,89.30128704991793,19.84573861908027,10,0,1,126.52145689425862,Moderate,
8.019319397153513,6.648713430002088,278.2647023337781,122.30885159288817,330.8405295860562,Ecologically Degraded
2018-01-01 21:00:00,26.759072313952004,19.02416734848438,55.44030943700152,5.72583450054139,5,100,2,128.8427239489277,Low,
7.236942931444238,8.45778225961333,27.747184352788025,108.4956116294621,193.9388330934224,Ecologically Healthy
2018-01-01 22:00:00,23.63967219173367,26.908201260761736,51.402028089074335,6.729762107616224,11,50,0,107.16169499217077,Low,
8.23693430573012,6.669435417095521,56.99828536126389,138.48163411046747,23.826788813964438,Ecologically Healthy
2018-01-01 23:00:00,96.69752089440674,25.349499570228396,79.86242089539846,31.13973033944089,5,100,0,125.85936631729302,Moderate,
7.310701327313331,-0.783163994516582,110.22799591057377,36.938499994996675,160.03690870302185,Ecologically Stable
2018-01-02 00:00:00,140.66769039895252,20.140255273100657,81.97299512504452,52.07282984518551,12,0,0,106.0679627076207,Low,
7.036781547427843,4.437007948505327,81.62260449045165,19.475939534486052,491.1840807756901,Ecologically Stable
2018-01-02 01:00:00,94.89177871785353,23.791913444713078,74.8379956037794,25.014750285867294,5,0,0,74.00050042989473,Moderate,
7.32346117474464,4.153999278442823,246.57195958454258,68.2327532722613,171.1014400154784,Ecologically Healthy
2018-01-02 02:00:00,76.08895885423004,20.263067592027515,80.15806577867973,41.256527354905465,13,0,1,127.85640191191987,High,
7.807104303488269,4.661225840128121,283.7035348252378,5.244204694595078,108.67559825063883,Ecologically Healthy
2018-01-02 03:00:00,65.36907174917371,19.127127840167752,84.9884933113033,9.54213804043368,14,0,0,111.19473949318396,High,
8.024069575886145,8.151266390641945,129.50541845887184,112.4663867875721,117.18405805090798,Ecologically Healthy
2018-01-02 04:00:00,56.99549153433252,26.17333116767334,69.95256942346856,21.128650085274824,7,0,3,98.26657631380478,Low,
6.615716774609902,6.790606190884281,247.67080580566477,168.990530386752,382.97049450377637,Ecologically Healthy
```

Figure 10.1: Example of the structure of a CSV file. the file is having its columns separated by a comma delimiter

- tab - `\t`
- semicolon - `;`
- pipe - `|`,
- whitespace

To read a csv into R, you can run any of the following. They all have a first common argument which is the file path:

```r
# Base R implementation

my_data <- read.csv(file = "data/ecological_health_dataset.csv")

head(my_data)
```

```
            Timestamp     PM2.5 Temperature Humidity Soil_Moisture
1 2018-01-01 00:00:00 119.68397    21.88583 53.95560      22.47978
2 2018-01-01 01:00:00  74.72324    19.07956 54.29895      23.98031
3 2018-01-01 02:00:00  69.11418    26.67587 98.99122      11.56632
4 2018-01-01 03:00:00  69.11511    20.17007 36.41646      36.14457
5 2018-01-01 04:00:00 232.48572    21.91575 79.35562      43.53254
6 2018-01-01 05:00:00 143.33531    15.02139 96.33046      37.93073
  Biodiversity_Index Nutrient_Level Water_Quality Air_Quality_Index
1                  9             50             0          82.59493
2                  9              0             0         127.41848
3                  7             50             0          95.21542
4                 10              0             0          65.53427
5                 11              0             1          80.31496
6                 12              0             0         102.10155
  Pollution_Level  Soil_pH Dissolved_Oxygen Chemical_Oxygen_Demand
1             Low 5.284388         6.555422               24.11973
2        Moderate 6.107887         7.542608              164.58496
3             Low 8.361576         6.821085               24.81837
4             Low 7.929766         7.421999              248.72788
5             Low 5.378418         7.231868              271.06234
6             Low 5.579344         7.229231              280.21082
  Biochemical_Oxygen_Demand Total_Dissolved_Solids Ecological_Health_Label
1                  9.731336               44.79406     Ecologically Healthy
2                178.793602              205.78702      Ecologically Stable
3                 25.332886              448.38676     Ecologically Healthy
4                 58.940128              359.25938     Ecologically Healthy
5                106.113663              118.30366    Ecologically Critical
6                120.859359               84.70938     Ecologically Healthy
```

`head()` returns the firsts 6 records of the dataset.

The readr implementation is also quite straight forward, instead of `read.csv()` it uses `read_csv()`.

Note

This will be the first time we are seeing the operator `|>`. It is called the **pipe operator** and takes the result from its left hand side to the function/operation on the right hand side for evaluation. It is a nice way to chain operations without the need to break down your code. The above could be written as:

```r
# Written as this
my_data <- read_csv("data/ecological_health_dataset.csv")
head(my_data)

# Or
head(read_csv("data/ecological_health_dataset.csv"))
```

The pipe will be used alot moving forward so we get used to it.

The result of `read_csv()`, and `read.csv()` seems different, and that's because one is a tibble–often regarded as the modern and clean version of data.frame–and the other is a data.frame. They show the same data with difference in presentation. The tibble is more detail and displays information about the dimension of the data, its column specification, then the data itself with each data type displayed under the column name. To learn more about tibble visit the Tibbles chapter in the R for Data Science 2nd Edition Book.

Next is `data.table` implementation of reading files. It is the easiest, and fastest of all three when it comes to data reading, and we will see that later on in time. It uses the function `fread()`.

```r
fread("data/ecological_health_dataset.csv") |>
  head()
```

```
             Timestamp     PM2.5 Temperature Humidity Soil_Moisture
                <POSc>     <num>       <num>    <num>         <num>
1: 2018-01-01 00:00:00 119.68397    21.88583 53.95560      22.47978
2: 2018-01-01 01:00:00  74.72324    19.07956 54.29895      23.98031
3: 2018-01-01 02:00:00  69.11418    26.67587 98.99122      11.56632
4: 2018-01-01 03:00:00  69.11511    20.17007 36.41646      36.14457
5: 2018-01-01 04:00:00 232.48572    21.91575 79.35562      43.53254
6: 2018-01-01 05:00:00 143.33531    15.02139 96.33046      37.93073
```

Table 10.2: Reading a CSV file with `readr's` read_csv. Produces a tibble instead of a data.frame.

```r
read_csv("data/ecological_health_dataset.csv") |> head()
```

```
Rows: 61345 Columns: 16
-- Column specification --------------------------------------------------------
Delimiter: ","
chr   (2): Pollution_Level, Ecological_Health_Label
dbl  (13): PM2.5, Temperature, Humidity, Soil_Moisture, Biodiversity_Index, ...
dttm  (1): Timestamp

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.

# A tibble: 6 x 16
  Timestamp           PM2.5 Temperature Humidity Soil_Moisture
  <dttm>              <dbl>       <dbl>    <dbl>         <dbl>
1 2018-01-01 00:00:00 120.         21.9     54.0          22.5
2 2018-01-01 01:00:00  74.7        19.1     54.3          24.0
3 2018-01-01 02:00:00  69.1        26.7     99.0          11.6
4 2018-01-01 03:00:00  69.1        20.2     36.4          36.1
5 2018-01-01 04:00:00 232.         21.9     79.4          43.5
6 2018-01-01 05:00:00 143.         15.0     96.3          37.9
# i 11 more variables: Biodiversity_Index <dbl>, Nutrient_Level <dbl>,
#   Water_Quality <dbl>, Air_Quality_Index <dbl>, Pollution_Level <chr>,
#   Soil_pH <dbl>, Dissolved_Oxygen <dbl>, Chemical_Oxygen_Demand <dbl>,
#   Biochemical_Oxygen_Demand <dbl>, Total_Dissolved_Solids <dbl>,
#   Ecological_Health_Label <chr>
```

```
   Biodiversity_Index Nutrient_Level Water_Quality Air_Quality_Index
                <int>          <int>         <int>             <num>
1:                  9             50             0          82.59493
2:                  9              0             0         127.41848
3:                  7             50             0          95.21542
4:                 10              0             0          65.53427
5:                 11              0             1          80.31496
6:                 12              0             0         102.10155
   Pollution_Level  Soil_pH Dissolved_Oxygen Chemical_Oxygen_Demand
            <char>    <num>            <num>                  <num>
1:             Low 5.284388         6.555422               24.11973
2:        Moderate 6.107887         7.542608              164.58496
3:             Low 8.361576         6.821085               24.81837
4:             Low 7.929766         7.421999              248.72788
5:             Low 5.378418         7.231868              271.06234
6:             Low 5.579344         7.229231              280.21082
   Biochemical_Oxygen_Demand Total_Dissolved_Solids Ecological_Health_Label
                       <num>                  <num>                  <char>
1:                  9.731336               44.79406    Ecologically Healthy
2:                178.793602              205.78702     Ecologically Stable
3:                 25.332886              448.38676    Ecologically Healthy
4:                 58.940128              359.25938    Ecologically Healthy
5:                106.113663              118.30366   Ecologically Critical
6:                120.859359               84.70938    Ecologically Healthy
```

Of the three implementations, only the tibbles limits the column display to only what the screen/document can contain at a point in time, while the others have no limits.

Sometimes we do have files that we want to import that are available online. These functions read online files without trouble, just ensure you know the file extension, so your data gets imported as expected. The import speed now depends on your internet speed and the file size.

```r
read_csv("https://raw.githubusercontent.com/EU-Study-Assist/data-for-r4r/refs/heads/main/r4r
  head()
```

```
`curl` package not installed, falling back to using `url()`
Rows: 61345 Columns: 16
-- Column specification --------------------------------------------------------
Delimiter: ","
chr   (2): Pollution_Level, Ecological_Health_Label
dbl  (13): PM2.5, Temperature, Humidity, Soil_Moisture, Biodiversity_Index, ...
```

```
dttm  (1): Timestamp

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.

# A tibble: 6 x 16
  Timestamp           PM2.5 Temperature Humidity Soil_Moisture
  <dttm>              <dbl>       <dbl>    <dbl>         <dbl>
1 2018-01-01 00:00:00 120.         21.9     54.0          22.5
2 2018-01-01 01:00:00  74.7        19.1     54.3          24.0
3 2018-01-01 02:00:00  69.1        26.7     99.0          11.6
4 2018-01-01 03:00:00  69.1        20.2     36.4          36.1
5 2018-01-01 04:00:00 232.         21.9     79.4          43.5
6 2018-01-01 05:00:00 143.         15.0     96.3          37.9
# i 11 more variables: Biodiversity_Index <dbl>, Nutrient_Level <dbl>,
#   Water_Quality <dbl>, Air_Quality_Index <dbl>, Pollution_Level <chr>,
#   Soil_pH <dbl>, Dissolved_Oxygen <dbl>, Chemical_Oxygen_Demand <dbl>,
#   Biochemical_Oxygen_Demand <dbl>, Total_Dissolved_Solids <dbl>,
#   Ecological_Health_Label <chr>
```

In addition to `read.csv` or `read_csv`, we have other `read` functions that are named according to their file extensions. A list of some read_* functions are give below. There's a super function that reads a lot of flat file, that's the `read.delim` or `read_delim`, you only have to specify your delimiter in the right argument–`sep` for `read.delim()` and `delim` for `read_delim`.

file extension	base R	readr	data.table
txt	read.table	read.table	fread
tsv	read.delim	read_tsv	fread
dat	read.table	read_table	fread
log	readLines	read_lines	fread
tab	read.delim	read_delim	fread
psv	read.table	read_delim	fread
fixed-width	read.fwf	read_fwf	fread

Note

The good thing about data.table fread is that it guesses delimiter for its user, making it easier to import files. This in addition to how fast the package is, make it a useful package to learn.

There are other useful arguments you should take note off when you are importing data. Reading the documentation by using `help()` would expose you to these arguments. This include arguments like, header/col_names, sep/delim, na, skip, etc.

10.4 SpreadSheet

Base R, readr, and data.table cannot import spreadsheet data. Instead, packages are used. When we hear spreadsheet, MS Excel is the first thing that comes to mind–well maybe that's my mind–but, spreadsheet are also different, and we can tell this by their file extensions. We have ods, xlsx, xls, gsheet. Except from gsheet which is usually a link in its non-native format, the rest can be seen as you tradition file extension on your personal computer.

10.5 CSV

10.6 Base R Approach

10.7 Tidyverse Approach

10.8 Excel Files

10.9 Writing Excel Files

10.10 Google Sheets

11 Data Manipulation

This section is still under development

11.1 Reorder your data with arrange()

11.2 Pick variables with select()

11.3 Subset data with filter()

11.4 Add or change the content of a variable with mutate()

11.5 Collapse columns using summarize()

11.6 Reshaping your data

Part III

Exploratory Data Analysis

❗ Section will be written after whole content of this part is finished

12 Introduction to Data Visualization with ggplot2

This section is still under development

12.0.1 Making Graphs with the `plot()` Function

To see a graph of the *tomato_weight* object, pass it to `plot()` function as the first argument `x`. `x` stands for x-axis, and a scattered plot graph is produced as shown in **?@fig-td**.

```
#plot(x = tomato_weight)
```

The graph can be customized by passing values into other arguments within the `plot()` function. Arguments such as `col` changes the color while `pch` changes the shape of each point, **?@fig-plot-cust**. You can also use `hist()` to see the distribution of the data, **?@fig-td-hist**.

```
#| label: fig-plot-cust
#| eval: false
#| fig-cap: |
#|    Simple plot customization: point shape changed to cross and color changed to red


#plot(tomato_weight, col = "red", pch = 3)
```

```
#hist(tomato_weight, col = "coral3", xlab = "Tomato Diameter", main = "Distribution of Tomato
```

The `xlab` argument is used to customize the label of the x-axis, while the y-axis is customized with `ylab`. The argument `main`, is used to change the title of the graph. More on how to make visualization with R will be discussed in Chapter 12 and Chapter 13.

The code used to produce the graph is clear, but as you write more complex codes, readability reduces. The code could improve by writing one or two arguments in a line rather than all in a line. The code above can also be written as:

```r
# hist(
#   tomato_weight,
#   col = "coral3",
#   xlab = "Tomato Diameter",
#   main = "Distribution of Tomato Plant Diameter"
# )
```

12.1 Introduction to the grammar of graphics

12.2 When to use a plot type

13 Exploratory Data Analysis

This section is still under development

13.1 Understanding your data

13.1.1 Data preview

13.1.2 Data structure

13.1.3 Data summary / Descriptive Statistics

13.2 Missing Values

13.2.1 Detecting missing values

13.2.2 Removing missing values

13.3 Variable analysis

13.3.1 Univariate analysis

13.3.2 Character / Factor variables

13.3.3 Numerical variables

13.4 Bivariate Analysis

###Two categorical variable analysis ### Two numerical variable analysis ### One numerical and one categorical variable analysis

13.5 Multivariate analysis

14 Inferential Statistics

This section is still under development

14.1 Hypothesis testing with t-test

14.1.1 One-sample t-test

14.1.2 Two Sample t-test

###Limitations and Assumptions of t-test ## ANOVA and Linear regression ### ANOVA ### Linear regression ### Making predictions with linear regression models

Part IV

Communicate and Collaborate

❗ Area will be written after whole content of this part is finished

References

Grolemund, Garrett. 2014. *Hands-on Programming with r: Write Your Own Functions and Simulations.* " O'Reilly Media, Inc.".

Ihaka, R, and R Gentleman. 1996. "R: A Language for Data Analysis and Graphics. Journal of Computational and Graphical Statistics 5: 299." *Doi: 10.2307/1390807.*

ISCED, UNESCO. 2012. "International Standard Classification of Education 2011." *UNESCO Institute for Statistics.*

Leisch, Friedrich. 2002. "Sweave: Dynamic Generation of Statistical Reports Using Literate Data Analysis." In *COMPSTAT: Proceedings in Computational Statistics*, 575–80. Springer.

Peng, Roger D. 2016. *R Programming for Data Science.* Leanpub Victoria, BC, Canada.

R Core Team. 2024. "R FAQ: R Add-on Packages." https://cran.r-project.org/doc/FAQ/R-FAQ.html#R-Add_002dOn-Packages. https://cran.r-project.org/doc/FAQ/R-FAQ.html#R-Add_002dOn-Packages.

Radečić, Dario. 2024. "Introducing Positron: A New, yet Familiar IDE for r and Python." *RSS.* https://www.appsilon.com/post/introducing-positron.

Wickham, Hadley. 2019. *Advanced r.* chapman; hall/CRC.

Xie, Yihui, Joseph J Allaire, and Garrett Grolemund. 2018. *R Markdown: The Definitive Guide.* Chapman; Hall/CRC.