R for Data Analysis

Trevor French

11/13/22

Table of contents

Part I

Introduction

> "There is synthesis when, in combining therein judgments that are made known to us from simpler relations, one deduces judgments from them relative to more complicated relations. There is analysis when from a complicated truth one deduces more simple truths." -André-Marie Ampère (Hofmann 1996)

Everyone is a data analyst. The purpose of this book is to inspire and enable anyone who reads it to reconsider the methods they currently employ to analyse data. This is not to suggest that the methodologies outlined will be useful or sufficient for everyone who reads it. Some analyses can be performed quickly without the need for additional computation while others will require advanced analytics techniques not outlined in this book; however, the aspiration is that all will be equipped with novel tools and ideas for approaching data analysis.

Prerequisites

No prior knowledge is required to begin this book. The content will start at the very beginning by showing you how to set up your R environment and the basics of programming in R. By the end of the book, you will be able to perform intermediate analytics techniques such as linear regression and automatic report generation.

You will need an environment which you use to run your code. It is recommended that you download R and R Studio locally for this requirement. This book will walk you through how to do that as well as offer alternatives if that is not an option for you.

Structure of the Book

- **Part I (Fundamentals)** will introduce you to the basics of programming in the context of R.
- **Part II (Data Acquisition)** will teach you how to create, import, and access data.
- **Part III (Data Preparation)** will show you how to begin preparing your data for analysis.
- **Part IV (Developing Insights)** goes through the process of searching for and extracting insights from your data.
- **Part V (Reporting)** demonstrates how to wrap your analysis up by developing and automating reports.

Each part will contain several chapters which cover specific ideas related to the overarching topic. At the end of each of these chapters you will find additional resources for you to use to dive deeper into the ideas. Each part will be concluded with practical exercises for you to test your skills.

While sections of this book could be used to supplement formal education programs, it was initially designed to be used for independent study.

License

This work is free to use, and is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

About Me

I have an M.S. in Data Analytics, a B.S. in Business Analytics, and currently work in industry as an Analytics Manager for a software company. I began my journey into analytics by working as a Data Analyst for the university I was attending. This role allowed me to automate processes, build dashboards, deliver reports to executive stakeholders, and provide insight on how operations might be improved. I performed this role until I was promoted to lead the team. Later, I worked for a major CPG company driving pricing and promotion strategy for a large piece of the business.

Despite my education, most of my basic analytics knowledge was hard-won through self-study. I created this resource to be what I wish I had when I started my journey into the analytics domain. Additionally, I don't believe that one must be a domain expert to be effective at analyzing data. In fact, I think most people can quickly learn the skills necessary to be very effective at it.

Physical copies of this book are not currently available; however, you can download a pdf in the top left corner of this site. Feel free to contribute by reporting a typo or leaving a pull request at https://github.com/TrevorFrench/R-for-Data-Analysis.

Chapter 1

What is R?

> R was a programming language that was designed specifically for the needs of statistics and data analysis. -Hadley Wickham (Hermans 2021)

R is a statistical programming language used commonly for data analysis across a wide array of disciplines and industries. It's often preferred over similar languages for it's robust support of statistical analysis, the ease in which one is able to create beautiful graphics, and its open source nature among other reasons.

1.1 History

R was built by Ross Ihaka and Robert Gentleman at the University of Auckland and was first released in 1993.

Robert Gentleman and Ross Ihaka "both had an interest in statistical computing and saw a common need for a better software environment in [their] Macintosh teaching laboratory. [They] saw no suitable commercial environment and [they] began to experiment to see what might be involved in developing one [them]selves." (Ihaka 1998)

While R was officially first released in 1993, it wasn't until 1995 that Ross Ihaka and Robert Gentlemann were convinced by Martin Mächler to release the source code freely (Ihaka 1998).

1.2 Resources

- You can learn more about R here: https://www.r-project.org/
- Read Ross Ihaka's account of R's origination: https://www.stat.auckland.ac.nz/~ihaka/downloads/Interface98.pdf

- "What is R?"" by Microsoft: https://mran.microsoft.com/documents/what-is-r
- R manuals by the R Development Core Team: https://cran.r-project.org/manuals.html
- R-bloggers: https://www.r-bloggers.com/
- R User Groups: https://www.meetup.com/pro/r-user-groups/
- R Studio Community: https://community.rstudio.com/
- The R Journal: https://journal.r-project.org/

Chapter 2

What is Data Analysis?

> I mean my definition is data science is like data analysis by programming. Which of course begs the question of what data analysis is, and so I think of data analysis as really any activity where the input is data and the output is understanding or knowledge or insights. So I think of that pretty broadly. And then to do data science you're not doing it by pointing and clicking. You're doing it by writing some code in a programming language. -Hadley Wickham (Eremenko 2020)

Data analysis at its most simple form is the process of searching for meaning in data with the ultimate goal to draw insight from that meaning.

2.1 The Process of Data Analysis

The process of data analysis can be generally described in five steps:

1. **Gathering Requirements -** Before one embarks on an analysis, it's important to make sure the requirements are understood. Requirements include the questions your stakeholders are hoping to answer as well as the technical requirements of how you are going to perform your analysis.

2. **Data Acquisition -** As you might imagine, you must acquire your data before conducting an analysis. This may be done through methods such as manual creation of datasets, importing pre-constructed data, or leveraging APIs.

3. **Data Preparation -** Most data will not be received in the precise format you need to begin your analysis. The process of data preparation involves structuring and adding features to your data.

4. **Developing Insights** - Once your data is prepared, you can begin to make sense of it and develop insights about its meaning.

5. **Reporting** - Finally, it's important to report on your data in such a way that the information can be digested by the people who need to see it when they need to see it.

Other sources may include additional steps such as "acting on the analysis". While this is a critical step for organizations to capture the full value of their data, I would argue that it occurs outside of the *analysis* process.

This book will focus on the technical skills required to conduct an analysis. Because of this, we will be covering steps two through five and omitting step one.

2.2 Resources

- "Data Science & Big Data Analytics: Discovering, Analyzing, Visualizing and Presenting Data" by EMC Education Services: https://onlinelibrary.wiley.com/doi/book/10.1002/9781119183686
- "Managing the Analytics Life Cycle for Decisions at Scale" by SAS: https://www.sas.com/content/dam/SAS/en_us/doc/whitepaper1/manage-analytical-life-cycle-continuous-innovation-106179.pdf

Chapter 3

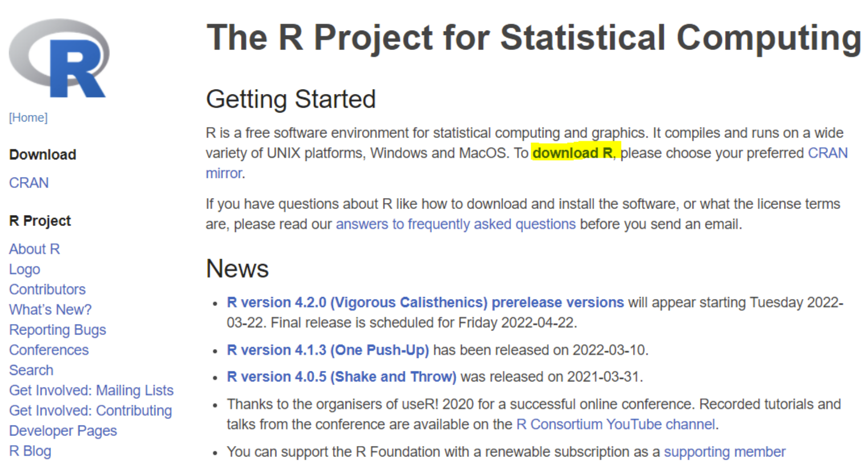
Setup

This chapter will walk you through downloading the R programming language as well as R Studio, which is a popular tool for interacting with the R ecosystem. Additionally, there are alternatives to R Studio listed at the end of the chapter. However, R Studio is the recommended environment for completing this book.

3.1 Install R

Before you do anything, you'll need to download R. This download will allow your computer to interpret the R code you write later on.

1. Download R From R: The R Project for Statistical Computing
2. Select "download R"

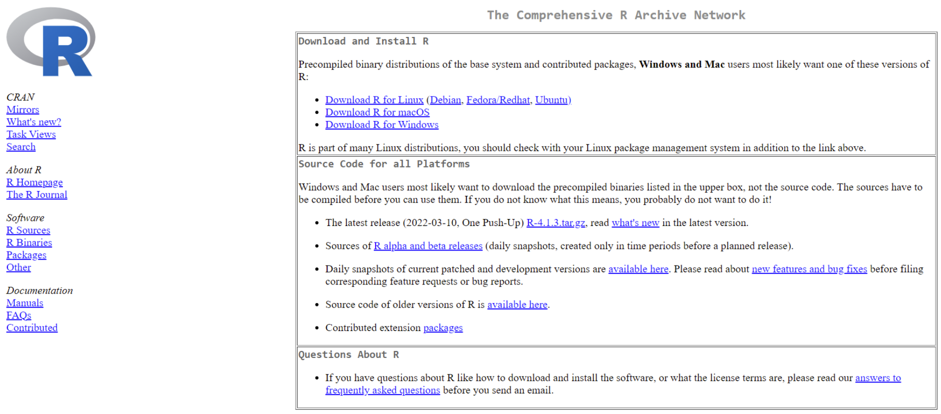


3. Choose any link but preferably the one closest to your physical location

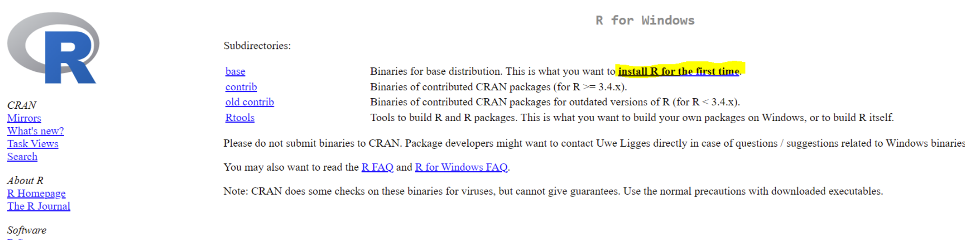
USA

https://mirror.las.iastate.edu/CRAN/	Iowa State University, Ames, IA
http://ftp.ussg.iu.edu/CRAN/	Indiana University
https://rweb.crmda.ku.edu/cran/	University of Kansas, Lawrence, KS
https://repo.miserver.it.umich.edu/cran/	MBNI, University of Michigan, Ann Arbor, MI
http://cran.wustl.edu/	Washington University, St. Louis, MO
https://archive.linux.duke.edu/cran/	Duke University, Durham, NC
https://cran.case.edu/	Case Western Reserve University, Cleveland, OH
https://ftp.osuosl.org/pub/cran/	Oregon State University
http://lib.stat.cmu.edu/R/CRAN/	Statlib, Carnegie Mellon University, Pittsburgh, PA
https://cran.mirrors.hoobly.com/	Hoobly Classifieds, Pittsburgh, PA
https://mirrors.nics.utk.edu/cran/	National Institute for Computational Sciences, Oak Ridge, TN
https://cran.microsoft.com/	Revolution Analytics, Dallas, TX

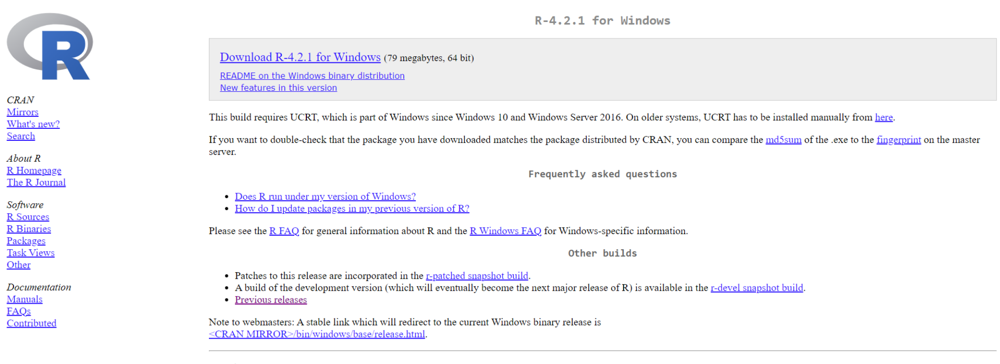
4. Choose your operating system



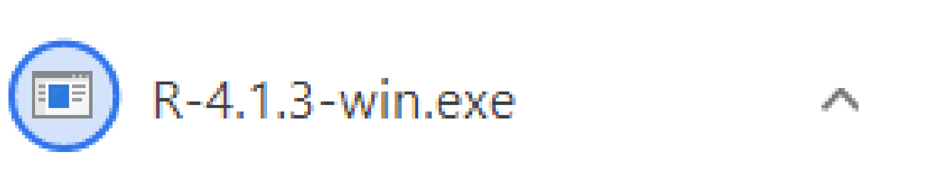
5. Press "Install R for the first time"



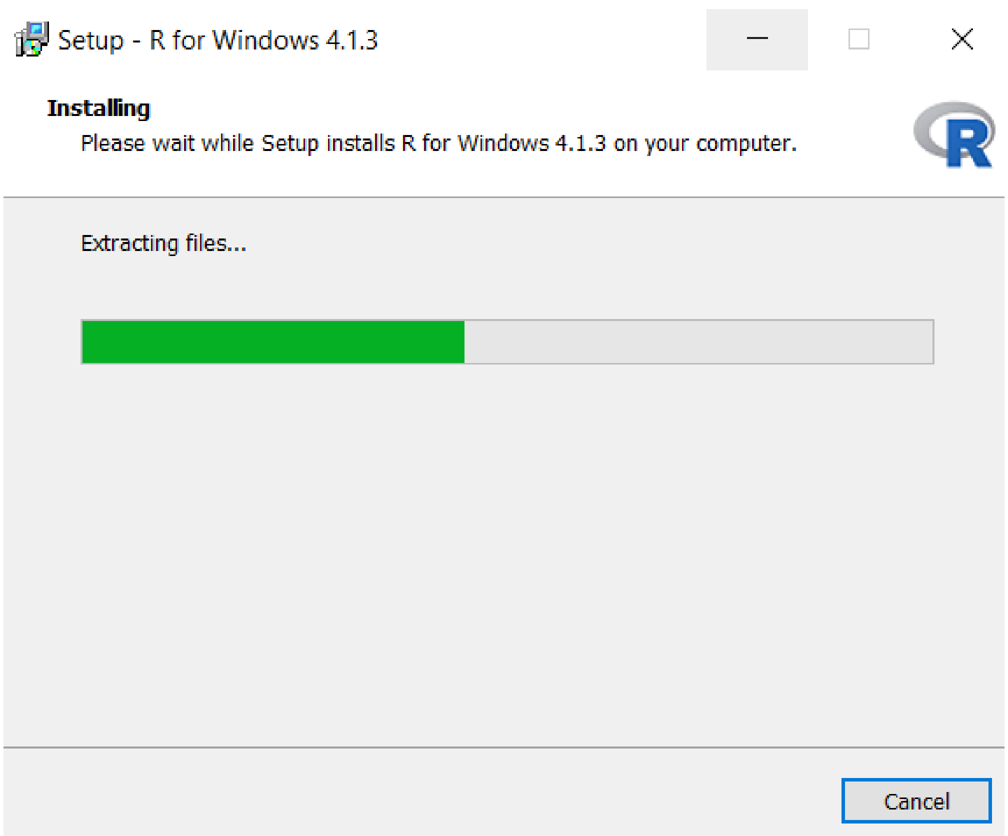
6. Press "download"



7. Open installer



8. Follow the prompts and leave all options set as their default values

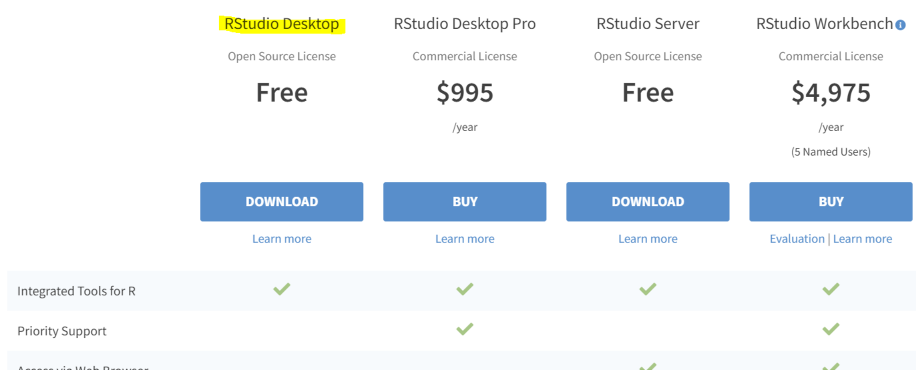


3.2 Install R Studio

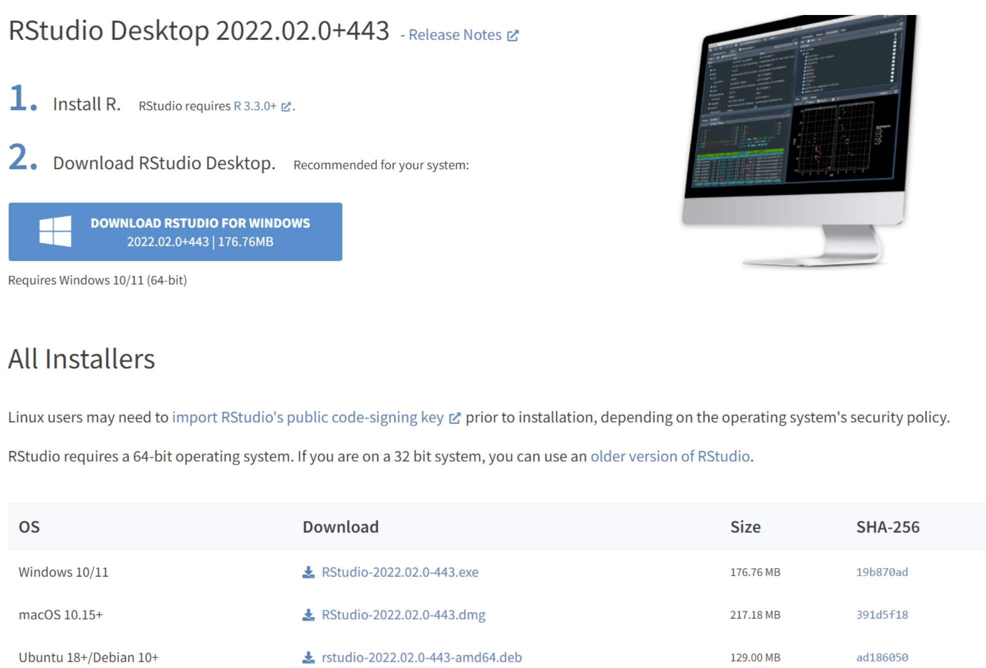
After you install R, you'll need an environment to write and run your code in. Most people use a program called "RStudio" for this. To download RStudio follow the steps listed below:

1. Navigate to the R Studio download site: Download the RStudio IDE

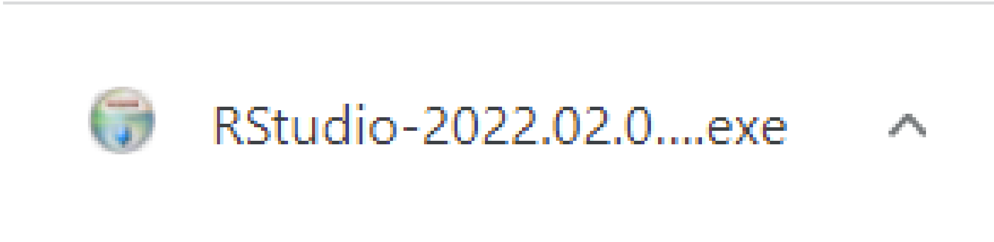
2. Press the "download" button under RStudio Desktop



3. Choose the download option for your operating system



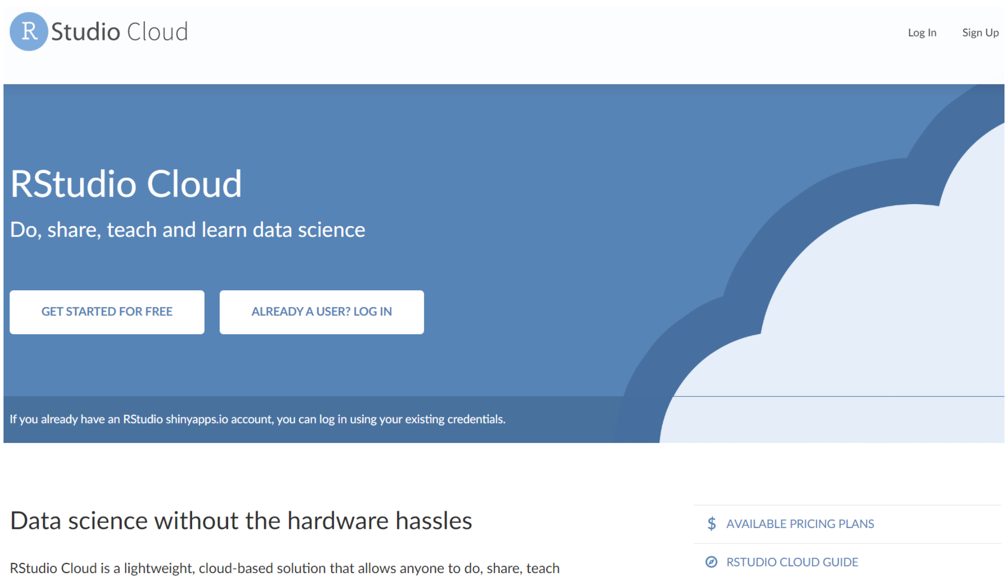
4. Open the installer and accept all defaults



3.3 Alternatives

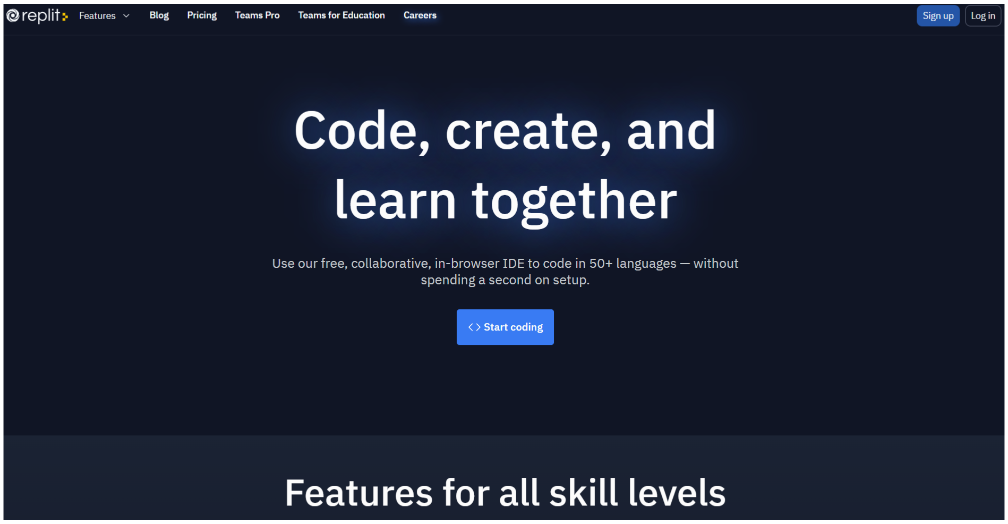
3.3.1 Posit Cloud

Posit Cloud offers users a way to replicate the full RStudio experience without having to download or set anything up on your personal computer. You can sign up for a free account here:



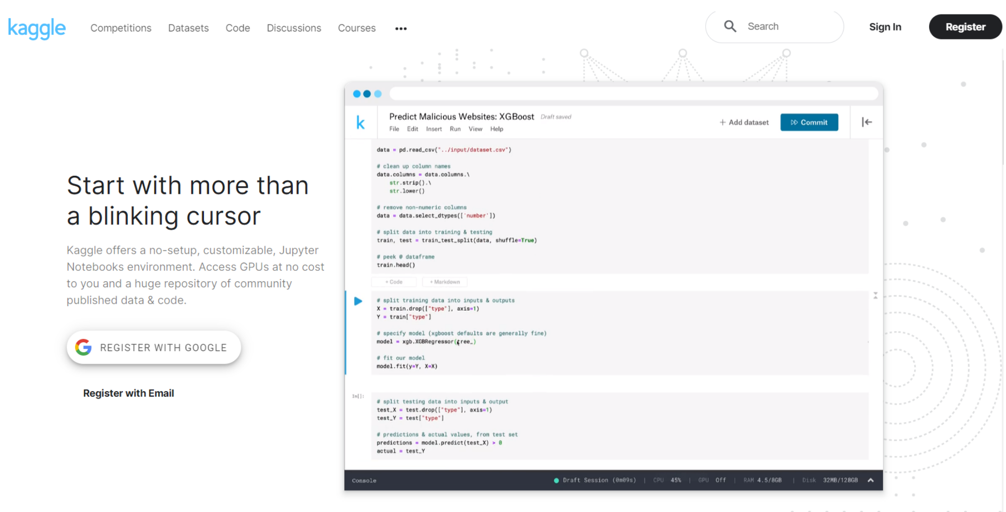
3.3.2 Replit

Replit allows users to code in 50+ languages in the browser. While you won't be able to follow along with the RStudio specific examples, you will be able to run R code. You can sign up for a free account here:



3.3.3 Kaggle

Kaggle is one of the most popular sites for data analysts to compete in data competitions, find data, and discuss data topics. They also have a feature that allows you to write and run R (and Python) code. You can sign up for a free account here:



3.4 Resources

- "R Installation and Administration" by the R Core Team: https://cran.r-project.org/doc/manuals/r-release/R-admin.html

Part II

Part I: Fundamentals

This section will introduce you to the basics of programming in the context of R. There are four chapters in this book. Each chapter has a brief description listed below. After you have finished reading through each of them, you will have the opportunity to attempt practical exercises to reinforce your newly-gained knowledge.

> **Note**
>
> Users with a moderate amount of experience in R or another programming language should feel free to either skip, skim, or leverage this chapter as a reference guide.

- **Getting Familiar with RStudio-** There are four sections in RStudio. These sections are often referred to as "panes". This chapter will introduce you to the "source", "console", "environment", and "files" panes. Additionally, you will learn about the different ways you can customize your version of RStudio such as changing the color scheme.
- **Programming Basics-** While the R language certainly has its unique advantages, it still leverages principles found in many other programming languages such as functions, comments, and loops. Learn how to apply these and other principles in R.
- **Data Types-** Data is stored differently depending on what it represents when programming. For example, a number and a letter are stored as different data types. Learn about the five basic data types in R and how to use them.
- **Data Structures-** In computer science, a data structure refers to the method which one uses to organize their data. Six basic data structures are commonly used in R. Learn about each of them in this chapter.

Chapter 4

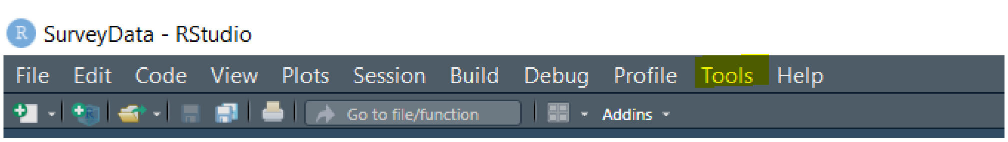
Getting Familiar with RStudio

To begin, we are going to walk through customizing your version of RStudio to make it the most comfortable environment for you personally. Following this, we are going to walk through the four panes of RStudio. At a glance, RStudio may seem overwhelming; however, by the end of this chapter you will have learned the essentials needed to embark on your data analysis journey.

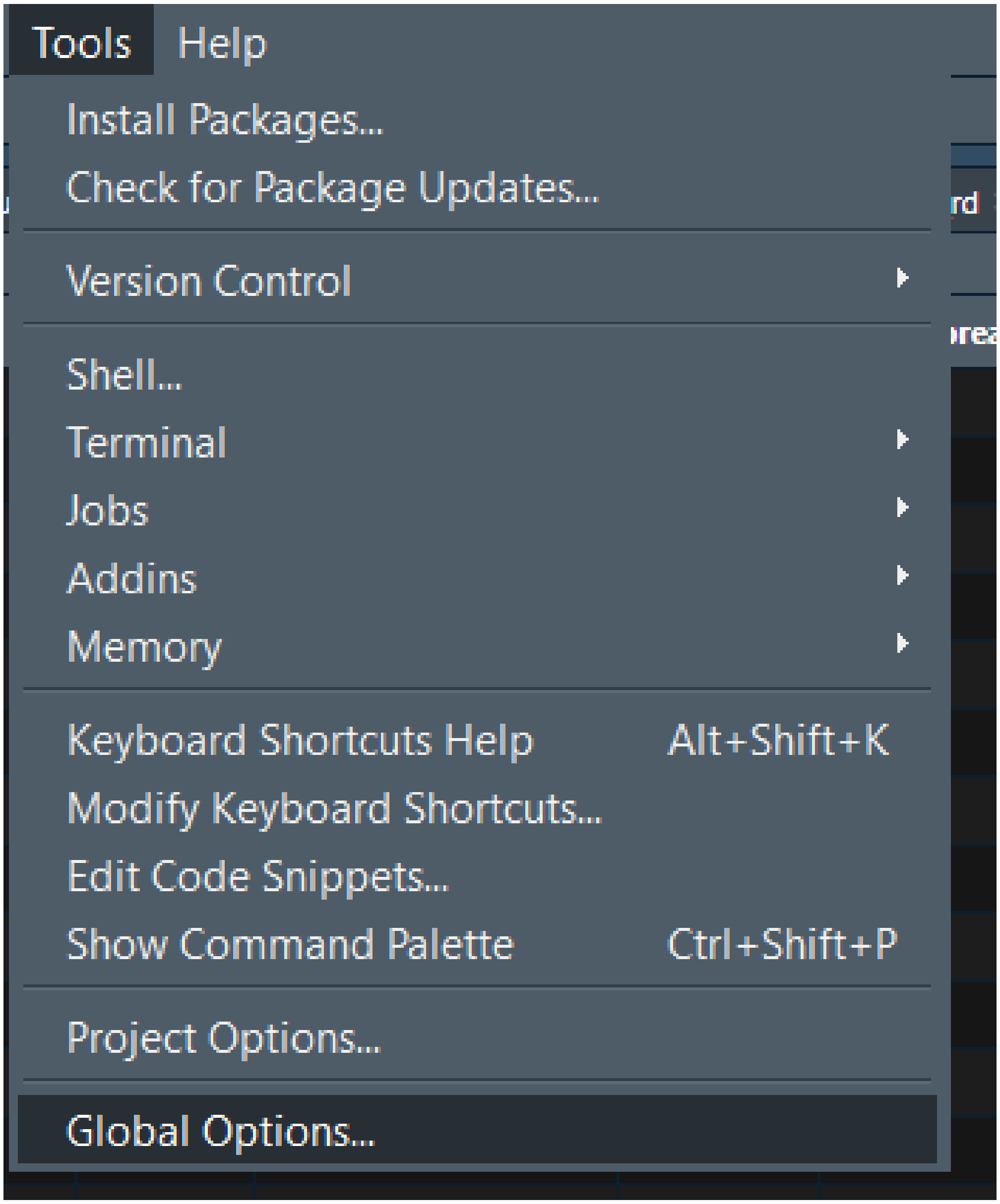
4.1 Customization

You are able to customize how your version of RStudio looks by following these steps:

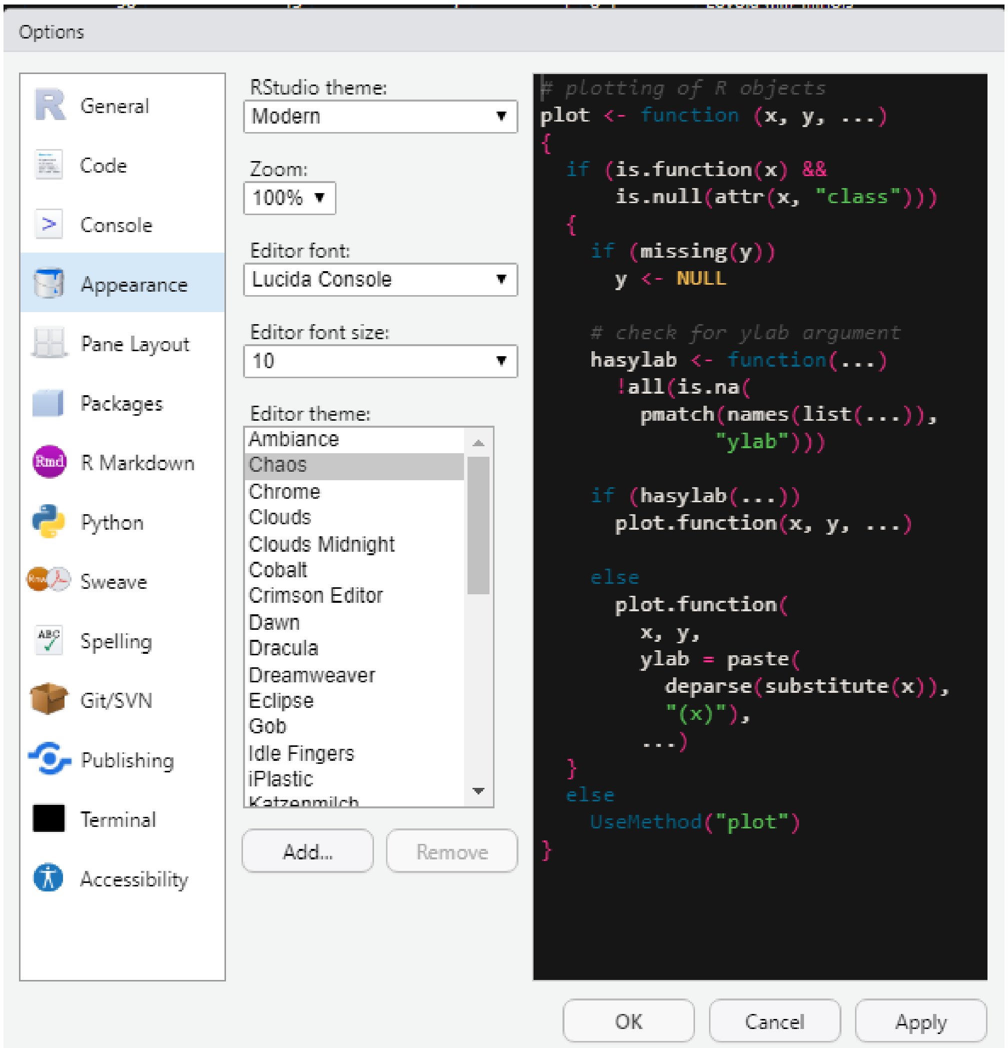
1. Open RStudio and choose ‘tools’ from the toolbar



2. Choose ‘Global Options’



3. Choose 'Appearance' and select your favorite theme from the 'Editor Theme' section

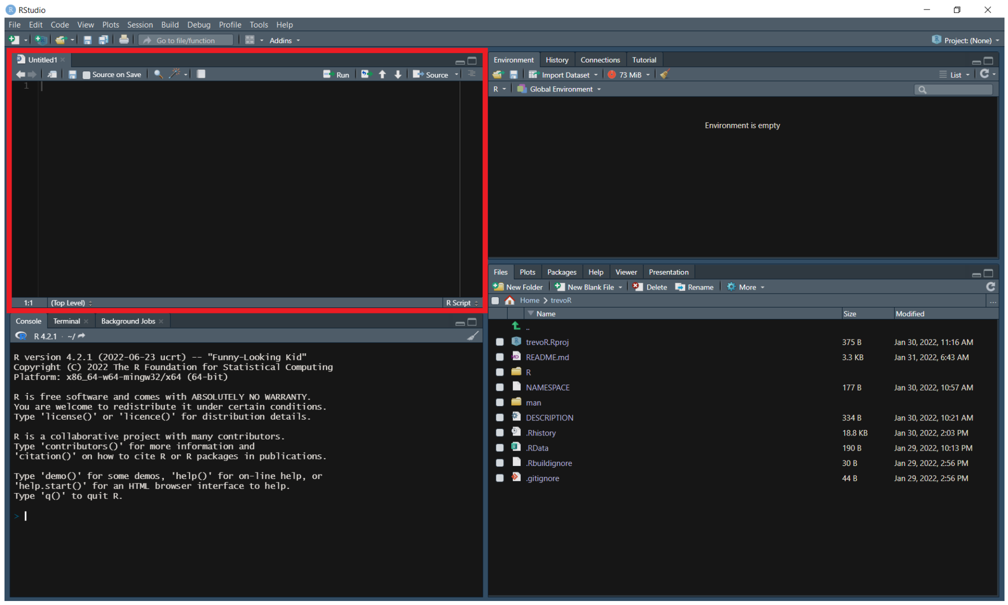


4. Press 'Apply'

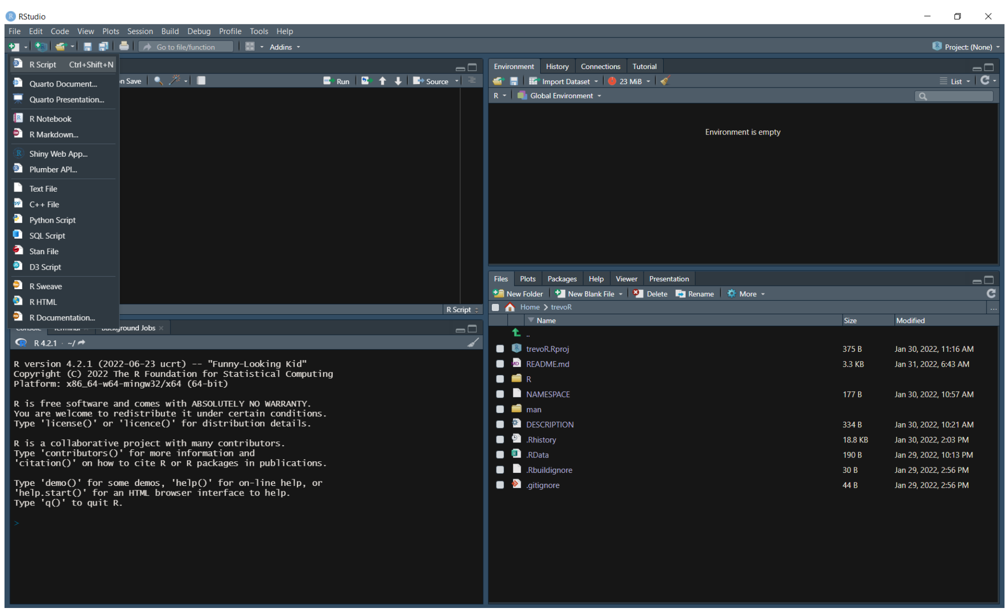
There are other customization options avaialable as well. Feel free to explore the "Global Options" section to make your version of RStudio your own.

4.2 Source Pane

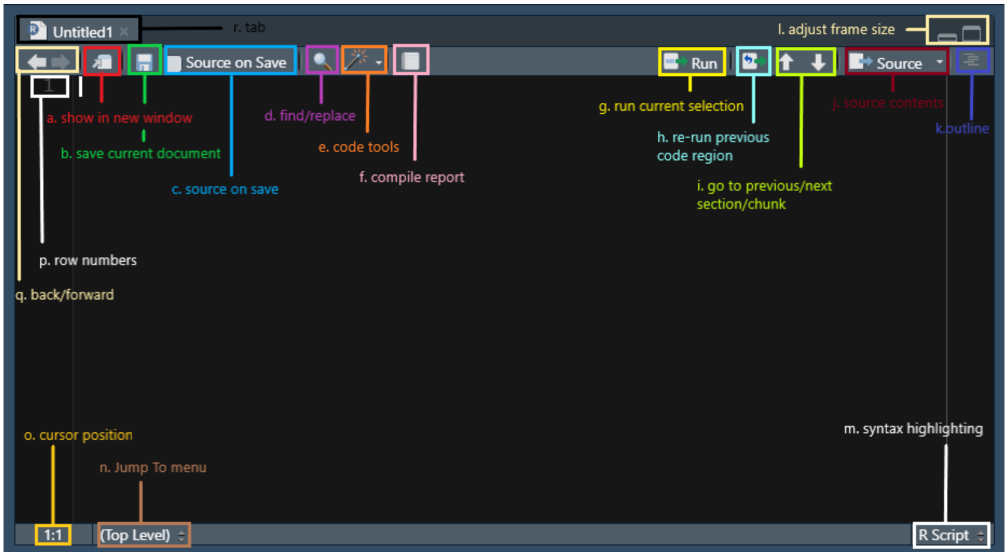
The source pane is the top left pane in RStudio. This is where you will write and edit your code.



If you don't see the source pane, you may need to create a new R script by pressing "Ctrl + Shift + N" ("Cmd + Shift + N" on Mac) or by selecting "R Script" from the "New File" dropdown in the top left corner.



Each element of the source pane is outlined below.



a. **Show in New Window**- This allows you to pop the source pane into a new window by itself.
b. **Save Current Document**- This saves the file contained in the tab you currently have active.
c. **Source on Save**- Automatically sources your file every time you hit save. "Sourcing" is similar to "Running" in the sense that both will execute your code; however, sourcing will execute your saved file rather than copying lines of code into the console.
d. **Find/Replace**- this feature allows you to find and replace specified text, similar to find and replace features in other tools such as Excel.
e. **Code Tools**- This brings up a menu of options which help you to code more efficiently. Some of these tools include formatting your code and help with function definitions.
f. **Compile Report**- This allows you to compile a report directly from an R script without needing to use additional frameworks such as R Markdown.
g. **Run Current Selection**- This allows you to highlight a portion of your code and run only that portion.
h. **Re-run Previous Code Region**- This option will execute the last section of code that you ran.
i. **Go to Previous/Next Section/Chunk**- These up and down arrows allow you to navigate through sections of your code without needing to scroll.
j. **Source Contents**- This option will save your active document if it isn't already saved and then source the file.
k. **Outline**- Pressing this option will pop open an outline of your current file.
l. **Adjust Frame Size**- These two options will adjust the size of the source pane inside of R Studio.

m. **Syntax Highlighting-** This allows you to adjust the syntax highlighting of your active document to match the highlighting of other file types.
n. **"Jump To" Menu-** This menu allows you to quickly jump to different sections of your code.
o. **Cursor Position-** This displays your current cursor position by row and column.
p. **Row Numbers-** Display the row number for each line of your code on the left side of the document.
q. **Back/Forward-** These arrows are navigation tools that will allow you to redo/undo the following actions: opening a document (or switching tabs), going to a function definition, jumping to a line, and jumping to a function using the function menu (Paulson 2022).
r. **Tab-** This is a tab in the traditional sense, meaning you are able to have a collection of documents open displayed as tabs. These tabs will have the title of your document and often an icon of some sort to demonstrate the file type.

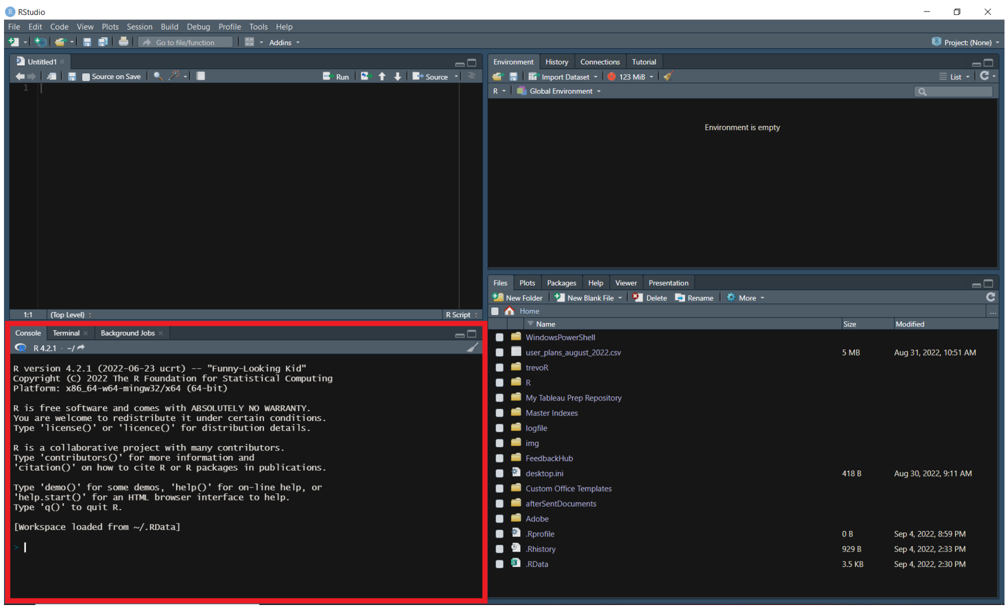
4.3 Console

The console pane is the bottom left pane in RStudio. This pane has three tabs: "Console", "Terminal", and "Background Jobs".

- The "Console" tab is where you will be able to run R code directly without writing a script (this will be covered in the next chapter).

- The "Terminal" tab is the same terminal you have on your computer. This can be adjusted in the global options.

- The "Background Jobs" tab is where you can start and manage processes that need to run behind the scenes.



4.4 Environment

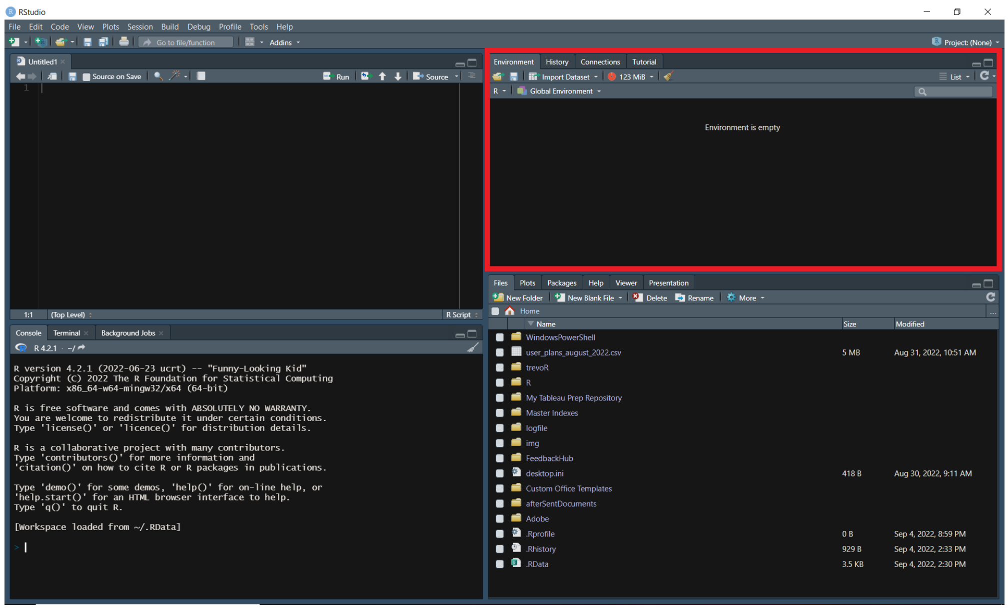
The environment pane is the top right pane in RStudio. This is where you will manage all things related to your development environment. This pane has four tabs: “Environment”, “History”, “Connections”, and “Tutorial”.

- The “Environment” tab will display all information relevant to your current environment. This includes data, variables, and functions. This is also the place where you can view and manage your memory usage as well as your workspace.

- The “History” tab allows you to view the history of your executed code. You can search through these commands and even select and re-execute them.

- The “Connections” tab is where you can create and manage connections to databases.

- The “Tutorial” tab delivers tutorials powered by the “learnr” package.



4.5 Files

The files pane is the bottom right pane in RStudio. This pane has six tabs: “Files”, “Plots”, “Packages”, “Help”, “Viewer”, and “Presentation”.

- The “Files” tab is a file explorer of sorts. You can view the contents of a directory, navigate to new directories, and manage files here.

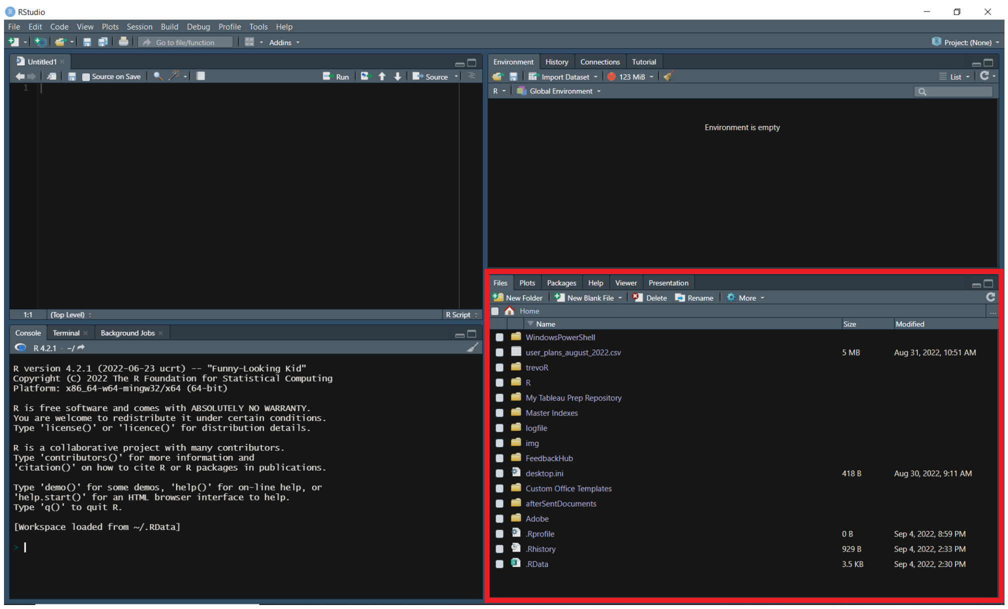
- The “Plots” tab is where the output of your generated plots will show up. You can also export your plots from this tab.

- The “Packages” tab allows you to view all available packages within your environment. From this tab, you can read more about each package as well as update and access packages.

- The “Help” tab allows you to search for information about functions to include examples, descriptions, and available parameters.

- The “Viewer” tab is where certain types of content such as quarto documents will be displayed when rendered.

- The “Presentation” tab is similar to the “Viewer” tab except the content type will be presentations.



4.6 Resources

- "Editing and Executing Code in the RStudio IDE" from the R Studio Support team: https://support.rstudio.com/hc/en-us/articles/200484448-Editing-and-Executing-Code
- "Code Folding and Sections in the RStudio IDE" from the R Studio Support team: https://support.rstudio.com/hc/en-us/articles/200484568-Code-Folding-and-Sections-in-the-RStudio-IDE
- "Keyboard Shortcuts in the RStudio IDE" from the R Studio Support team: https://support.rstudio.com/hc/en-us/articles/200711853-Keyboard-Shortcuts-in-the-RStudio-IDE
- "Navigating Code in the RStudio IDE" from the R Studio Support team: https://support.rstudio.com/hc/en-us/articles/200710523-Navigating-Code-in-the-RStudio-IDE

Chapter 5

Programming Basics

This chapter will walk you through executing code and writing scripts in R. You will then build upon that knowledge by learning about comments, variables, operators, functions, loops, conditionals, and libraries. While this chapter is titled “Programming Basics”, the knowledge you will have learned by the end of this chapter is enough for you to accomplish a huge variety of tasks.

5.1 Executing Code

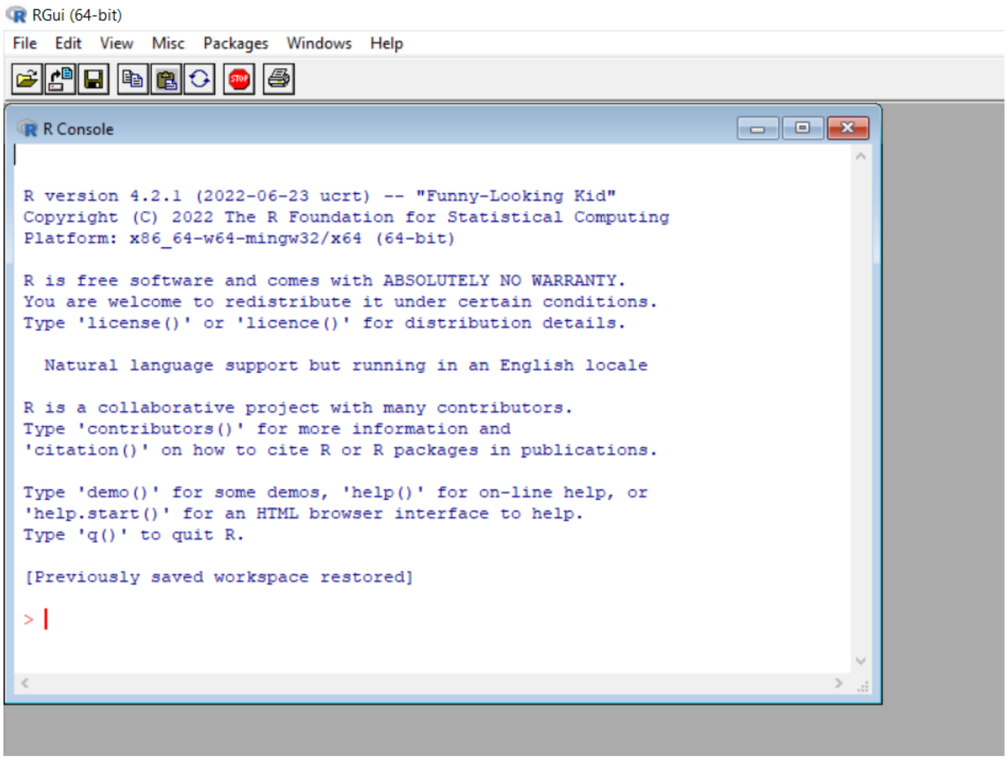
When working in most programming languages, you will generally have the option to execute code one of two ways:

- in the console
- in a script

5.1.1 Console

The first way to run code is directly in the console. If you’re working in RStudio, you will access the console through the “console” pane.

Alternatively, if you downloaded R to your personal computer, you will likely be able to search your machine for an app named “RGui” and access the console this way as well.



In the following example, the text "print(3+2)"" is typed into the console. The user then presses enter and sees the result: "[1] 5".

```
print(3+2)
```

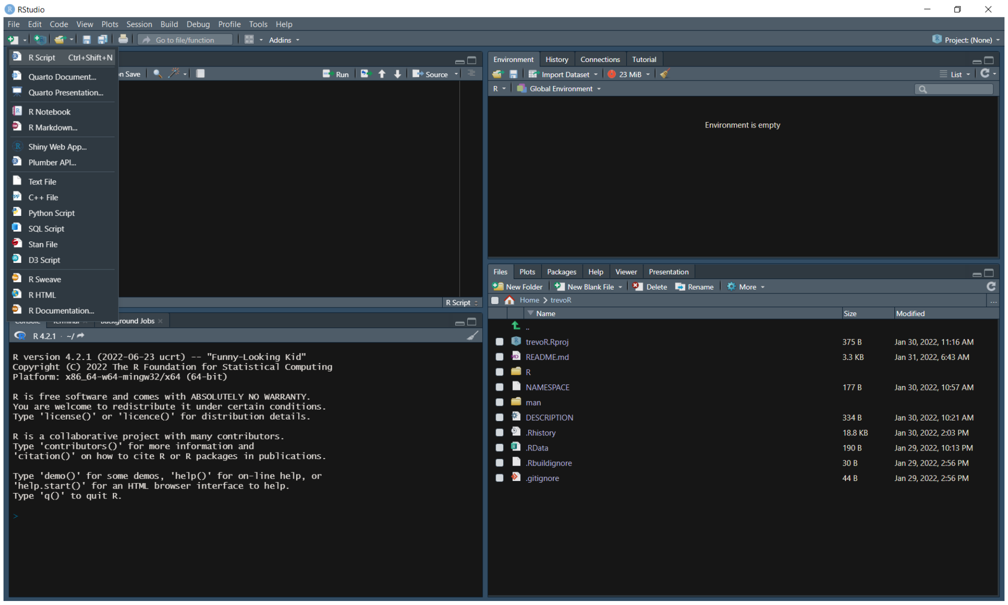
```
[1] 5
```

You may be wondering what "[1]" represents. This is simply a line number in the console and can be ignored for most practical purposes. Additionally, most of the examples in this book will be structured in this way: formatted code immediately followed by the code output.

5.1.2 Script

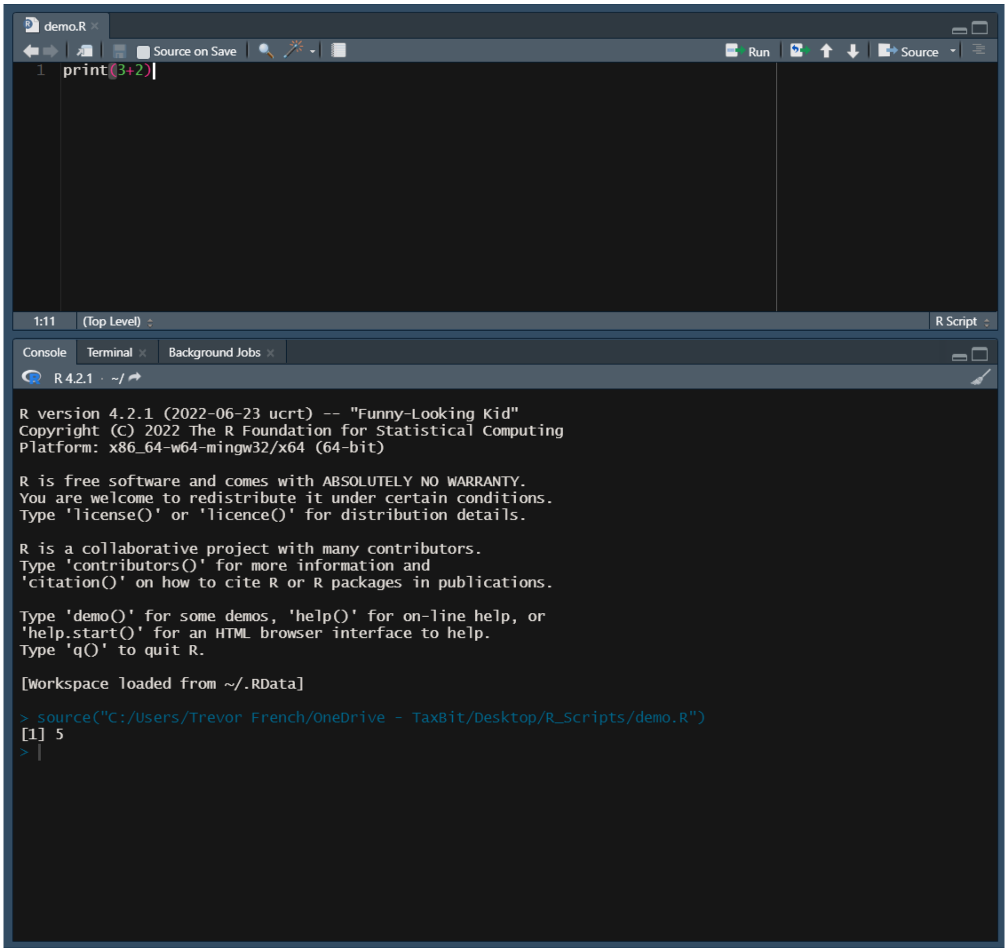
You likely will be using scripts most of the time when working in R. A script is just a file that allows you to type out longer sequences of code and execute them all at once.

For those of you following along in RStudio, you can create a script by pressing "Ctrl + Shift + N" on Windows or by selecting "R Script" from the "New File" dropdown in the top left corner.



From here you can type the same command from before into the source pane. Next, you'll want to save your file by pressing "Ctrl + S" on Windows or by selecting "Save" from the "File" dropdown in the top left corner. Now just give your file a name and your file will automatically be saved as a ".R" file.

Finally, run your newly created R script by pressing the "source" button.



5.2 Comments

Comments are present in most (if not all) programming languages. They allow the user to write text in their code that isn't executed or read by computers. Comments can serve many purposes such as notes, instructions, or formatting.

Comments are created in R by using the "#" symbol. Here's an example:

```r
# This is a comment
print(3+2)
```

```
[1] 5
```

Some programming languages allow you a "bulk-comment" feature which allows you to quickly comment out multiple consecutive lines of text. However, in R, there is no such option. Each line must begin with a "#" symbol, as such:

```
# This is the first line of a comment
# This is the second line of a comment
print(3+2)
```

```
[1] 5
```

Comments don't have to start at the beginning of a line. You are able to start comments anywhere on a line like in this example:

```
print(3+2) # This comment starts mid-line
```

```
[1] 5
```

5.3 Variables

Variables are used in programming to give values to a symbol. In the following example we have a variable named "rate" which is equal to 15, a variable named "hours" which is equal to 4, and a variable named "total_cost" which is equal to rate * hours.

```
rate <- 15
hours <- 4
total_cost <- rate * hours
print(total_cost)
```

```
[1] 60
```

5.4 Operators

An operator is a symbol that allows you to perform an action or define some sort of logic. The following image demonstrates the operators that are available to you in R.

Operators

Arithmetic	
+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Exponent
%%	Modulus
%/%	Integer Division

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

Misc	
:	Creates a series of numbers in a sequence
%in%	Checks if element exists in vector
%*%	Matrix multiplication

Logical	
&	Vectorized AND operator
&&	AND
|	Vectorized OR operator
||	OR
!	NOT

Assignment	
<-, ->	local
<<-, ->>	global

5.4.1 Arithmetic Operators

Arithmetic operators allow users to perform basic mathematical operations. The examples below demonstrate how these operators might be used. For those not familiar, the modulus operator will return the remainder of a division operation while integer (or Euclidean) division returns the result of a division operation without the fractional component.

```
3 + 3
```

```
[1] 6
```

```
3 - 3
```

```
[1] 0
```

```
3 * 3
```

```
[1] 9
```

```
3 ^ 3
```

```
[1] 27
```

```
10 / 7
```

```
[1] 1.428571
```

```
10 %% 7
```

```
[1] 3
```

```
10 %/% 7
```

```
[1] 1
```

5.4.2 Comparison Operators

Comparison operators allow users to compare values. The examples below demonstrate how these operators might be used.

```
3 == 3
```

```
[1] TRUE
```

```
3 != 3
```

```
[1] FALSE
```

```
3 > 3
```

```
[1] FALSE
```

```
3 < 3
```

```
[1] FALSE
```

```
3 >= 3
```

```
[1] TRUE
```

```
3 <= 3
```

```
[1] TRUE
```

5.4.3 Logical Operators

Logical operators allow users to express "AND", "OR", and "NOT". The following examples demonstrate how these operators might be used in conjunction with comparison operators as well as the difference between standard logical operators and "vectorized" logical operators.

In this example, we will evaluate two vectors of the same length from left to right. Each vector has seven observations (-3, -2, -1, 0, 1, 2, 3). Rather than simply returning a single "TRUE" or "FALSE", this will return seven "TRUE" or "FALSE" values. In this case, the first element of each vector ("-3" and "-3") will be evaluated against their respective conditions and return "TRUE" only if both conditions are met. This will then be repeated for each of the remaining elements.

```
# Vectorized "AND" operator
((-3:3) >= 0) & ((-3:3) <= 0)
```

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

This example will return a single "TRUE" only if both conditions are met, otherwise "FALSE" will be returned.

```
# Standard "AND" operator
(3 >= 0) && (-3 <= 0)
```

```
[1] TRUE
```

This example is the same as the previous one with the exception that we have negated the second condition with a “NOT” operator.

```
# Standard "AND" operator with "NOT" operator
(3 >= 0) && !(-3 <= 0)
```

```
[1] FALSE
```

The following two examples are essentially the same as the first two except that we are using “OR” operators rather than “AND” operators

```
# Vectorized "OR" operator
((-3:3) >= 0) | ((-3:3) <= 0)
```

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

```
# Standard "OR" operator
(3 >= 0) || (-3 <= 0)
```

```
[1] TRUE
```

5.4.4 Assignment Operators

Assignment operators allow users to assign values to something. For most users, only “<-” or “->” will ever be used. These are called local assignment operators. However, there is another type of operator called a global assignment operator which is denoted by “«-” or “-»”.

Understanding the difference between local and global assignment operators in R can be tricky to get your head around. Here’s an example which should clear things up.

First, let’s create two variables named “global_var” and “local_var” and give them the values “global” and “local”, respectively. Notice we are using the standard assignment operator “<-” for both variables.

```
global_var <- 'global'
local_var <- 'local'

global_var
```

```
[1] "global"
```

```
local_var
```

```
[1] "local"
```

Next, let's create a function to test out the global assignment operator ("«-"). Inside this function, we will assign a new value to both of the variables we just created; however, we will use the "<-" operator for the local_var and the "«-" operator for the global_var so that we can observe the difference in behavior.

> **Note**
>
> Functions are covered directly after this section. If the concept of functions is unfamiliar to you, feel free to jump ahead and come back later.

```
my_function <- function() {
   global_var <<- 'na'
   local_var <- 'na'
   print(global_var)
   print(local_var)
}

my_function()
```

```
[1] "na"
[1] "na"
```

This function performs how you would expect it to intuitively, right? The interesting part comes next when we print out the values of these variables again.

```
global_var
```

```
[1] "na"
```

```
local_var
```

```
[1] "local"
```

From this result, we can see the difference in behavior caused by the differing assignment operators. When using the "<-" operator inside the function, it's scope is limited to just the function that it lives in. On the other hand, the "«-" operator has the ability to edit the value of the variable outside of the function as well.

You may now be wondering why both the local and the global assignment operators have two separate denotations. The following example demonstrates the difference between the two.

```
x <- 3
3 -> y

x
```

```
[1] 3
```

```
y
```

```
[1] 3
```

There is also a third assignment operator that can be used: "=". You will generally use the local assignment operator; however, you may notice that the "=" operator is used within certain functions as you progress. You can find more information about these three operators in the resources section.

5.4.5 Miscellaneous Operators

The ":" operator allows users to create a series of numbers in a sequence. This was demonstrated in the logical operator section. The %in% operator checks if an element exists in a vector. Both of these operators are demonstrated in the following example.

```
3 %in% 1:3
```

```
[1] TRUE
```

Finally, the "%*%" operator allows users to perform matrix multiplication as is demonstrated below. First, let's create a 2x2 matrix and then let's multiply it by itself.

```
x <- matrix(
  c(1,3,3,7)
  , nrow = 2
  , ncol = 2
  , byrow = TRUE)

x %*% x
```

```
     [,1] [,2]
[1,]   10   24
[2,]   24   58
```

5.5 Functions

Functions allow you to bundle a predefined set of operations into one command. The syntax of functions in R is as follows.

```
# Create a function called function_name
function_name <- function() {
  print("Hello World!")
}

# Call your newly created function
function_name()
```

```
[1] "Hello World!"
```

To go one step further, you can also add "arguments" to a function. Arguments allow you to pass information into the function when it is called. Here's an example:

```
# Create a function called add_numbers which will add
# two specified numbers together and print the result
add_numbers <- function(x, y) {
    print(x + y)
}

# Call your newly created function twice with different inputs
add_numbers(2, 3)
```

```
[1] 5
```

```
add_numbers(50, 50)
```

```
[1] 100
```

Finally, you can return a value from a function as such:

```
# Create a function called calculate_raise which multiplies
# base_salary and annual_adjustment and returns the result
calculate_raise <- function(base_salary, annual_adjustment) {
    raise <- base_salary * annual_adjustment
    return(raise)
}

# Calculate John's raise
johns_raise <- calculate_raise(90000, .05)
```

```
#Calculate Jane's raise
janes_raise <- calculate_raise(100000, .045)

print("John's Raise:")
```

```
[1] "John's Raise:"
```

```
print(johns_raise)
```

```
[1] 4500
```

```
print("Jane's Raise:")
```

```
[1] "Jane's Raise:"
```

```
print(janes_raise)
```

```
[1] 4500
```

5.6 Loops

There are two types of loops in R: while loops and for loops.

5.6.1 While Loops

While loops are executed as follows:

```
# Set i equal to 1
i <- 1

# While i is less than or equal to three, print i
# The loop will increment the value of i after each print
while (i <= 3) {
  print(i)
  i <- i + 1
}
```

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

Additionally, you can add ‘break’ statements to while loops to stop the loop early.

```r
i <- 1

while (i <= 10) {
    print(i)
    if (i == 5) {
        print("Stopping halfway")
        break
    }
    i <- i + 1
}
```

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] "Stopping halfway"
```

5.6.2 For Loops

For loops are executed as follows:

```r
employees <- list("jane", "john")

for (employee in employees) {
  print(employee)
}
```

```
[1] "jane"
[1] "john"
```

5.7 Conditionals

You are also able to execute a command if a condition is met by using "if" statements.

```r
if (2 > 0) {
    print("true")
}
```

```
[1] "true"
```

You can add more conditions by adding "else if" statements.

```r
if (2 > 3) {
    print("two is greater than three")
} else if (2 < 3) {
    print("two is not greater than three")
}
```

```
[1] "two is not greater than three"
```

Finally, you can catch anything that doesn't meet any of your conditions by adding an "else" statement at the end.

```r
x <- 20
if (x < 20) {
    print("x is less than 20")
} else if (x > 20) {
    print("x is greater than 20")
} else {
    print("x is equal to 20")
}
```

```
[1] "x is equal to 20"
```

5.8 R packages

Packages allow you to access functions other people have created and shared in a standard format, e.g. via the Comprehensive R Archive Network (CRAN), Bioconductor, the r-universe or e.g. as github repositories.

To access a package's functionality, you first have to add it to your system's library. Afterward, you can check it out for use in your current session with the `library()` command.

In this example, we will be installing and loading a common package named "dplyr".

You first retrieve it from CRAN with the following command.

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

Next, you make it available in your R session with the `library()` command. (Alternatively, you can also use the `require()` command.)

```r
library(dplyr)
```

You are now able to access all of the functions available in the dplyr library!

Sometimes users in the R community create their own packages that aren't distributed through the CRAN network. You can still use these packages, but you'll just have to perform an extra step or two. One of the most common places to host packages is Github. The following example will demonstrate how to load a package that I created from Github.

First you'll need to install the "remotes" package. As the name might suggest, this package allows you to access other packages from *remote* locations.

```
install.packages("remotes")
```

Next you'll need to install the remote package of your choosing. In our case, we'll execute the following code.

```
remotes::install_github("TrevorFrench/trevoR")
```

In the previous example, we used the "install_github" function from the "remotes" package and then specified the Github path of the remote repository by typing "TrevorFrench/trevoR". This code is functionally the same as the "install.packages" function. You may have noticed a new piece of syntax though. The "::" in between "remotes" and "install_github" tells R to use the "install_github" function from the "remotes" library without the need to require the library via the "library" function. This syntax can be used with any other function from any other library.

Now that the remote package is installed, we can require it in the same way we would any other package.

```
library(trevoR)
```

5.9 Resources

- W3 Schools R Tutorial: https://www.w3schools.com/r/
- Assignment Operators: https://stat.ethz.ch/R-manual/R-devel/library/base/html/assignOps.html

Chapter 6

Data Types

Data is stored differently depending on what it represents when programming. For example, a number is going to be stored as a different data type than a letter is.

There are five basic data types in R that each store a single value:

- **Numeric** - This is the default type for numbers, e.g. integers and doubles.
  - *Double* - A double allows you to store numbers as decimals. This is the default treatment for numbers.
  - *Integer* - An integer is a subset of the numeric data type. This type will only allow whole numbers and is denoted by the letter "L".
- **Complex** - This type is created by using the imaginary variable "i".
- **Character** - This type is used for storing non-numeric text data.
- **Logical** - Sometimes referred to as "boolean", this data type will store either "TRUE" or "FALSE".
- **Raw** - Used less often, this data type will store data as raw bytes.

In addition, each missing values can be specified with the special `NA` type, which can represent each of the data types listed above.

6.1 Numeric

6.1.1 Double

Let's explore the "double" data type by assigning a number to a variable and then check its type by using the "typeof" function. Alternatively, we can use the "is.double" function to check whether or not the variable is a double.

```
x <- 6.2
typeof(x)
```

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

```r
is.double(x)
```

```
[1] TRUE
```

Next, let's check whether or not the variable is numeric by using the "is.numeric" function.

```r
is.numeric(x)
```

```
[1] TRUE
```

This function should return "TRUE" as well, which demonstrates the fact that a double is a subset of the numeric data type.

6.1.2 Integer

Let's explore the "integer" data type by assigning a whole number followed by the capital letter "L" to a variable and then check its type by using the "typeof" function. Alternatively, we can use the "is.integer" function to check whether or not the variable is an integer.

```r
x <- 6L
# By using the "typeof" function, we can check the data type of x
typeof(x)
```

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

```r
is.integer(x)
```

```
[1] TRUE
```

Next, let's check whether or not the variable is numeric by using the "is.numeric" function.

```r
is.numeric(x)
```

```
[1] TRUE
```

This function should return "TRUE" as well, demonstrating that an integer is also a subset of the numeric data type.

6.2 Complex

Complex data types make use of the mathematical concept of an imaginary number through the use of the lowercase letter "i". The following example sets "x" equal to six times i and then displays the type of x.

```
x <- 6i
typeof(x)
```

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

6.3 Character

Character data types store text data. When creating characters, make sure you wrap your text in quotation marks.

```
x <- "Hello!"
typeof(x)
```

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

6.4 Logical

Logical data types store either "TRUE" or "FALSE". Unlike characters, these data should not be wrapped in quotation marks.

```
x <- TRUE
typeof(x)
```

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

6.5 Raw

Used less often, the raw data type will store data as raw bytes. You can convert character data types to raw data types by using the "charToRaw" function. Similarly, you can convert integer data types to raw data types through the use of the "intToBits" function.

```
x <- charToRaw("Hello!")
print(x)
```

```
[1] 48 65 6c 6c 6f 21
```

```
typeof(x)
```

```
[1] "raw"
```

```r
x <- intToBits(6L)
print(x)
```

```
 [1] 00 01 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[26] 00 00 00 00 00 00 00
```

```r
typeof(x)
```

```
[1] "raw"
```

6.6 Resources

- W3 Schools: https://www.w3schools.com/r/r_data_types.asp
- "Advanced R" by Hadley Wickham: https://adv-r.hadley.nz/vectors-chap.html#atomic-vectors
- "Bits and Bytes" from Stanford CS 101: https://web.stanford.edu/class/cs101/bits-bytes.html

Chapter 7

Data Structure

In computer science, a data structure refers to the method which one uses to organize their data. Six basic data structures are commonly used in R:

- **Vectors** - Vectors contain ordered data of a single type.
- **Lists** - Lists are a collection of objects.
- **Matrices** - A matrix is a two-dimensional array where the data is all of the same type.
- **Factors** - Factors are used to designate levels within categorical data.
- **Data Frames** - A data frame contains two-dimensional data where the data can have different types.
- **Arrays** - Arrays are objects which have more than two dimensions (n-dimensional).

7.1 Vectors

We can create a vector by using the "c" function to combine multiple values into a single vector. In the following example, we will combine four separate numbers into a single vector and the output the resulting vector to see what it looks like.

```
x <- c(1, 3, 3, 7)

print(x)
```

```
[1] 1 3 3 7
```

7.2 Lists

Lists are a collection of objects. This means that each element can be a different data type (unlike vectors). In the following example we'll create a list containing two character objects and one vector with the "list" function.

```
first_name <- "John"
last_name <- "Smith"
favorite_numbers <- c(1, 3, 3, 7)

person <- list(first_name, last_name, favorite_numbers)

print(person)
```

```
[[1]]
[1] "John"

[[2]]
[1] "Smith"

[[3]]
[1] 1 3 3 7
```

7.3 Matrices

A matrix is a two-dimensional array where the data is all of the same type. In the following example, we'll create a matrix with three rows and four columns.

```
x <- matrix(
        c(1,3,3,7,1,3,3,7,1,3,3,7)
        , nrow = 3
        , ncol = 4
        , byrow = TRUE)

print(x)
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    3    3    7
[2,]    1    3    3    7
[3,]    1    3    3    7
```

7.4 Factors

Factors are used to designate levels within categorical data. In the following example, we'll use the "factor" function on a vector of assorted color names to receive the "levels" which it contains.

```
x <- c("Red", "Blue", "Red", "Yellow", "Yellow")

colors <- factor(x)

print(colors)
```

```
[1] Red    Blue   Red    Yellow Yellow
Levels: Blue Red Yellow
```

7.5 Data Frames

A data frame contains two-dimensional data. Unlike the matrix data structure, each column of a data frame can contain data of a differing type (but within a column the data must be of the same type). The following example will create a data frame with two rows and two columns.

```
people <- c("John", "Jane")
id <- c(1, 2)
df <- data.frame(id = id, person = people)

print(df)
```

```
  id person
1  1   John
2  2   Jane
```

7.6 Arrays

Arrays are objects that can have more than two dimensions. This is sometimes referred to as being "n-dimensional". The dimensions of the following example are 1 x 4 x 3. You'll see that the data consist of one row and four columns spread out over a third dimension.

```
x <- array(
        c(1,3,3,7,1,3,3,7,1,3,3,7)
        , dim = c(1,4,3))
```

```r
print(x)
```

```
, , 1

     [,1] [,2] [,3] [,4]
[1,]    1    3    3    7

, , 2

     [,1] [,2] [,3] [,4]
[1,]    1    3    3    7

, , 3

     [,1] [,2] [,3] [,4]
[1,]    1    3    3    7
```

7.7 Resources

- W3 Schools: https://www.w3schools.com/r/r_vectors.asp

Exercises

Questions

Exercise: 5-A

Write a function called "multiply" that accepts two numbers as arguments and outputs the product of those two numbers when called as is demonstrated below.

```
multiply(3, 3)
# [1] 9
```

Exercise: 5-B

Write an equation that returns the remainder of 12 divided by 8.

Exercise: 5-C

Write an equation that returns the remainder of 36 divided by 10.

Exercise: 5-D

Write a "while" loop that prints all even numbers from 0 to 10.
It's possible for this task to be accomplished in several ways; however, the output of your program should always look like this:

```
# [1] 0
# [1] 2
# [1] 4
# [1] 6
# [1] 8
# [1] 10
```

Exercise: 5-E

You are given a vector that looks like this:

```
numbers <- c(0:12)
```

Write a for loop that loops through your vector and prints any element greater than or equal to 3.
It's possible for this task to be accomplished in several ways; however, the output of your program should always look like this:

```
# [1] 3
# [1] 4
# [1] 5
# [1] 6
# [1] 7
# [1] 8
# [1] 9
# [1] 10
# [1] 11
# [1] 12
```

Exercise: 6-A

Convert the following character variable to a variable with the data type "raw":

```
x <- "Trevor rocks"
```

You should store your raw data in a variable named "raw_data", print the data to the console, and check the data type with the "typeof" function. Your output should look like the following:

```
print(raw_data)
# [1] 54 72 65 76 6f 72 20 72 6f 63 6b 73
typeof(raw_data)
# [1] "raw"
```

Exercise: 6-B

Create a variable named "spending" and give it a value of 120. Then create a variable named "budget" and give it a value of 100. Next, check whether spending is greater than budget and store the resulting logical data in a variable named "over_budget". Finally, print the value of "over_budget" variable and check it's data type with the "typeof" function.
Your final output should look like this:

```
print(over_budget)
# [1] TRUE
typeof(over_budget)
# [1] "logical"
```

Exercise: 7-A

Create a vector named "animal" and give it the following three values: "cow", "cat", "pig". Create a second vector named "sound" and give it the following three values: "moo", "meow", "oink". Finally, create a data frame named "animal_sounds" and assign each of these vectors to be a column.
After printing the resulting data frame to the console, you should get the following output:

```
#   animal sound
# 1    cow   moo
# 2    cat  meow
# 3    pig  oink
```

Answers

Answer: 5-A

One way you could accomplish this task is demonstrated in the following solution.

```
multiply <- function(x, y) {
  return (x * y)
}

multiply(3, 3)
```

```
[1] 9
```

Answer: 5-B

A remainder is referred to as "modulus" in programming. We can use the "%%" operator to accomplish this. For this example, the output of your equation should be 4.

```
12 %% 8
```

```
[1] 4
```

Answer: 5-C

A remainder is referred to as "modulus" in programming. We can use the "%%" operator to accomplish this. For this example, the output of your equation should be 6.

```
36 %% 10
```

```
[1] 6
```

Answer: 5-D

Here's one way you could write your `while` loop to achieve this output:

```
i <- 0

while (i <= 10) {
  print(i)
  i <- i + 2
}
```

```
[1] 0
[1] 2
[1] 4
[1] 6
[1] 8
[1] 10
```

Answer: 5-E

Here's one way you could write your `for` loop to achieve this output:

```
numbers <- c(0:12)

for (number in numbers) {
  if (number >= 3) {
    print(number)
  }
}
```

```
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
[1] 11
[1] 12
```

Answer: 6-A

You can accomplish this task with the "charToRaw" function.

```
x <- "Trevor rocks"
raw_data <- charToRaw(x)
print(raw_data)
```

```
 [1] 54 72 65 76 6f 72 20 72 6f 63 6b 73
```

```
typeof(raw_data)
```

```
[1] "raw"
```

Answer: 6-B

The following example demonstrates how you can accomplish this task.

```
spending <- 120
budget <- 100
over_budget <- spending > budget
print(over_budget)
```

```
[1] TRUE
```

```
typeof(over_budget)
```

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

Answer: 7-A

The following example demonstrates how you can accomplish this task.

```
animal <- c("cow", "cat", "pig")
sound <- c("moo", "meow", "oink")
animal_sounds <- data.frame(animal = animal, sound = sound)
print(animal_sounds)
```

```
  animal sound
1    cow   moo
2    cat  meow
3    pig  oink
```

Part III

Part II: Data Acquisition

Before conducting an analysis you must first acquire your data, e.g. via manual creation, importing pre-constructed data, or leveraging APIs.

- **Included Datasets-** R comes with a variety of built-in datasets. This chapter will teach you how to view the catalog of included datasets, preview individual datasets, and begin working with the data.
- **Import from Spreadsheets-** Most R users will have to work with spreadsheets at some point in their careers. This chapter will teach you how to import data from spreadsheets, e.g. from a .csv or .xlsx file, and get the imported data into a format that's easy to work with.
- **Working with APIs-** API stands for Application Programming Interface. These sorts of tools are commonly used to programmatically pull data from a third party resource. This chapter demonstrates how you can begin to leverage these tools in your own workflows.

Chapter 8

Included Datasets

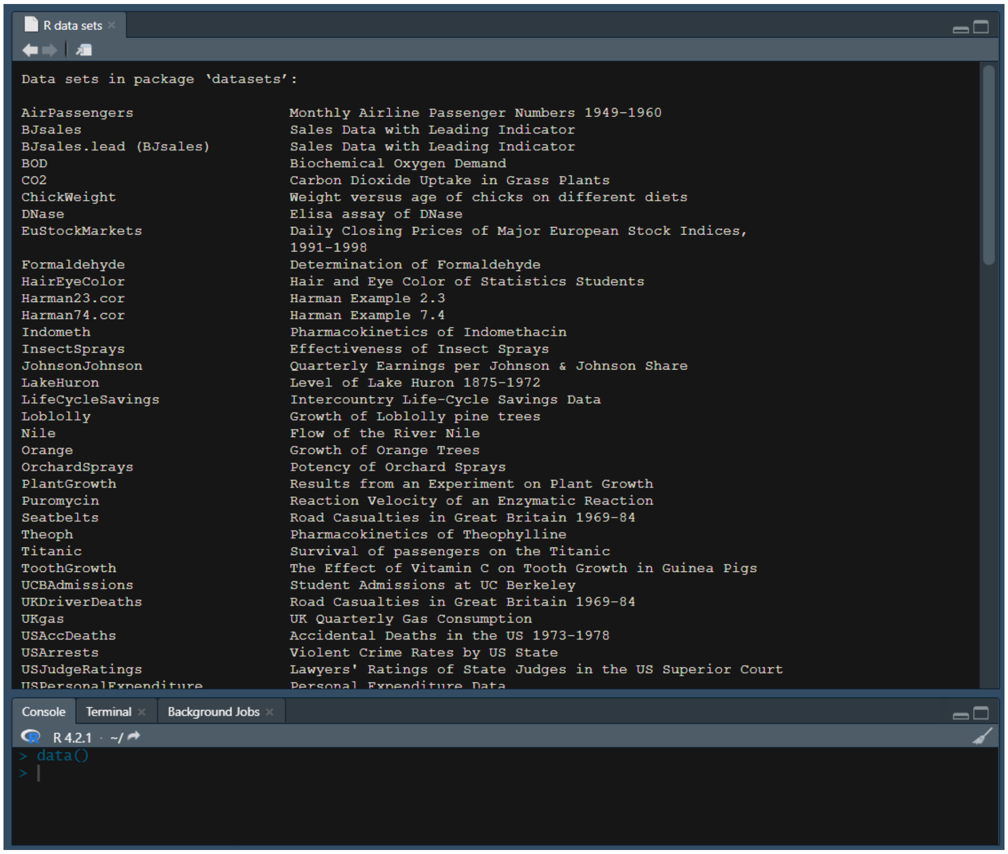
R comes with a variety of datasets already built in. This chapter will teach you how to view the catalog of included datasets, preview individual datasets, and begin working with the data.

8.1 View Catalog

You can view the complete list of datasets available along with a brief description for each one by typing "data()" into your console.

```
data()
```

This will open a new tab in your RStudio instance that looks similar to the following image:



8.2 Working with Included Data

The first step to begin working with your chosen dataset is to load it into your environment by using the "data" function with the quoted name of your dataset inside the parentheses. In the following example, we'll attach the "iris" dataset to our environment.

> **Note**
>
> It may not be necessary for you to load your dataset via the "data" function prior to using it. Additionally, some datasets may require you to add them to your search path by using using the "attach" function (conversely, you can remove datasets from your search path by using the "detach" function).

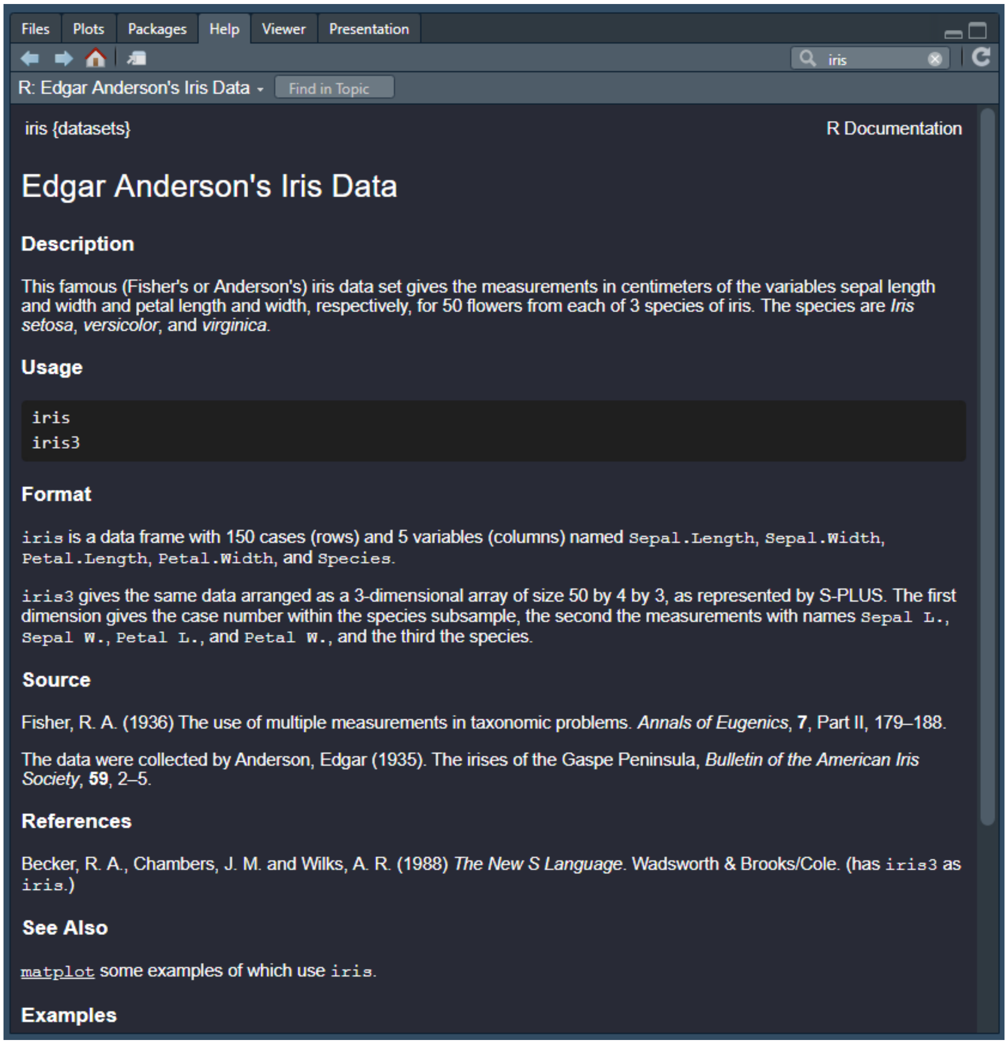
```
data("iris")
```

This command will add a new object "iris" to our R session. Let's preview the "iris" dataset by using the "head" function.

```
head(iris)
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa

Finally, you can view more information about any given dataset by typing its name into the "Help" tab in the "Files" pane.



iris {datasets} R Documentation

Edgar Anderson's Iris Data

Description

This famous (Fisher's or Anderson's) iris data set gives the measurements in centimeters of the variables sepal length and width and petal length and width, respectively, for 50 flowers from each of 3 species of iris. The species are *Iris setosa*, *versicolor*, and *virginica*.

Usage

```
iris
iris3
```

Format

`iris` is a data frame with 150 cases (rows) and 5 variables (columns) named `Sepal.Length`, `Sepal.Width`, `Petal.Length`, `Petal.Width`, and `Species`.

`iris3` gives the same data arranged as a 3-dimensional array of size 50 by 4 by 3, as represented by S-PLUS. The first dimension gives the case number within the species subsample, the second the measurements with names `Sepal L.`, `Sepal W.`, `Petal L.`, and `Petal W.`, and the third the species.

Source

Fisher, R. A. (1936) The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, **7**, Part II, 179–188.

The data were collected by Anderson, Edgar (1935). The irises of the Gaspe Peninsula, *Bulletin of the American Iris Society*, **59**, 2–5.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (has `iris3` as `iris`.)

See Also

`matplot` some examples of which use `iris`.

Examples

8.3 Common Datasets

Here are a few other datasets commonly used in the R community to practice and to teach.

8.3.1 mtcars

```
head(mtcars)
```

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

8.3.2 faithful

```
head(faithful)
```

eruptions	waiting
3.600	79
1.800	54
3.333	74
2.283	62
4.533	85
2.883	55

8.3.3 ChickWeight

```
head(ChickWeight)
```

weight	Time	Chick	Diet
42	0	1	1
51	2	1	1
59	4	1	1
64	6	1	1
76	8	1	1
93	10	1	1

8.3.4 Titanic

```r
head(Titanic)
```

```
, , Age = Child, Survived = No

      Sex
Class  Male Female
  1st     0      0
  2nd     0      0
  3rd    35     17
  Crew    0      0

, , Age = Adult, Survived = No

      Sex
Class  Male Female
  1st   118      4
  2nd   154     13
  3rd   387     89
  Crew  670      3

, , Age = Child, Survived = Yes

      Sex
Class  Male Female
  1st     5      1
  2nd    11     13
  3rd    13     14
  Crew    0      0

, , Age = Adult, Survived = Yes

      Sex
Class  Male Female
  1st    57    140
  2nd    14     80
  3rd    75     76
  Crew  192     20
```

8.4 Resources

- List of datasets available in Base R: https://www.rdocumentation.org/packages/datasets/versions/3.6.2

Chapter 9

Import from Spreadsheets

Most R users will have to work with spreadsheets at some point in their careers. This chapter will teach you how to import data from a .csv or .xlsx file, and how to get the imported data into a format that's easy to work with. Additionally, this chapter will demonstrate how to import multiple files at once and combine them all into a single dataframe.

9.1 Import from .csv Files

R has a function called "read.csv" which allows you to read a csv file directly into a dataframe. The following code snippet is a simple example of how to execute this function.

> **Note**
>
> It's worth noting that it isn't necessary to store the file path as a variable before calling the function; however, this habit may save you time down the road.

```r
input <- "C:/File Location/example.csv"
df <- read.csv(input)
```

Alternatively, if you have multiple files from the same directory that need to be imported, you could do something more like the following code snippet.

```r
directory <- "C:/File Location/"
first_file <- paste(directory, "first_file.csv", sep="")
second_file <- paste(directory, "second_file.csv", sep="")
```

```
first_df <- read.csv(first_file)
second_df <- read.csv(second_file)
```

9.2 Import from .xlsx Files

Excel files are handled very similarly to CSV files with the exception being that you will need to use the "read_excel" function from the "readxl" library. The following code snippet demonstrates how to import an Excel file into R.

```
library(readxl)
input <- "C:/File Location/example.xlsx"
df <- read_excel(input)
```

9.3 Import and Combine Multiple Files

You may come across a situation where you have multiple CSV files in a folder that need to be combined into a single data frame. The `read_csv()` function from the `readr` package accepts the paths to multiple files and will automatically concatenate them along their rows (if the columns match).

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

You can list the paths to all `.csv` files in a directory with the `dir()` command:

```
wd <- "C:/YOURWORKINGDIRECTORY"
dir(wd, full.names = TRUE, pattern = ".csv")
```

And read them into a single data.frame with a single command:

```
df <- read_csv(dir(wd, full.names = TRUE, pattern = ".csv"))
```

> **Note**
>
> All of the headers must match in your CSV files must match exactly for this function to work as expected.

9.4 Resources

- trevoR package documentation: https://github.com/TrevorFrench/trevoR

Chapter 10

Working with APIs

API stands for Application Programming Interface. These sorts of tools are commonly used to programmatically pull data from a third party resource. This chapter demonstrates how one can begin to leverage these tools in their own workflows.

The following example uses the Helium API to return data about its blockchain network.

10.1 Install Packages

```
install.packages(c('httr', 'jsonlite'))
```

10.2 Load packages from the library

```
library('httr')
library('jsonlite')
```

10.3 Make Request

Pass a URL into the 'GET' function and store the response in a variable called 'res'.

```
res = GET("https://api.helium.io/v1/stats")
print(res)
```

```
Response [https://api.helium.io/v1/stats]
  Date: 2022-08-04 01:25
  Status: 200
  Content-Type: application/json; charset=utf-8
  Size: 922 B
```

10.4 Parse & Explore Data

Use the 'fromJSON' function from the 'jsonlite' package to parse the response data and then print out the names in the resulting data set.

```
data = fromJSON(rawToChar(res$content))

names(data)
```

```
[1] "data"
```

Go one level deeper into the data set and print out the names again.

```
data = data$data

names(data)
```

```
[1] "token_supply"     "election_times"   "counts"           "challenge_counts" "blo
```

Alternatively, you can loop through the names as follows.

```
for (name in names(data)){print(name)}
```

```
[1] "token_supply"
[1] "election_times"
[1] "counts"
[1] "challenge_counts"
[1] "block_times"
```

Get the 'token_supply' field from the data.

```
token_supply = data$token_supply

print(token_supply)
```

```
[1] 124675821
```

10.5 Adding Parameters to Requests

Add 'min_time' and 'max_time' as parameters on a different endpoint and print the resulting 'fee' data.

```
res = GET("https://api.helium.io/v1/dc_burns/sum",
    query = list(min_time = "2020-07-27T00:00:00Z"
                 , max_time = "2021-07-27T00:00:00Z"))

data = fromJSON(rawToChar(res$content))
fee = data$data$fee
print(fee)
```

```
[1] 10112755000
```

10.6 Adding Headers to Requests

Execute the same query as above except this time specify headers. This will likely be necessary when working with an API that requires an API Key.

```
res = GET("https://api.helium.io/v1/dc_burns/sum",
    query = list(min_time = "2020-07-27T00:00:00Z"
                 , max_time = "2021-07-27T00:00:00Z"),
    add_headers(`Accept`='application/json', `Connection`='keep-live'))

data = fromJSON(rawToChar(res$content))
fee = data$data$fee
print(fee)
```

```
[1] 10112755000
```

10.7 Resources

- Blog post by Trevor French: https://medium.com/trevor-french/api-calls-in-r-136290ead81d

10.7.1 Helpful APIs

- Meta Graph API: https://developers.facebook.com/docs/graph-api/

- Twitter API: https://developer.twitter.com/en/docs/twitter-api
- NASA APIs: https://api.nasa.gov/
- Etherscan API: https://etherscan.io/apis
- Covalent API: https://www.covalenthq.com/docs/api/#/0/0/USD/1
- EDGAR APIs from the SEC: https://www.sec.gov/edgar/sec-api-documentation
- Weather API: https://openweathermap.org/api
- Helium API: https://docs.helium.com/api/

Exercises

Questions

Exercise: 8-A

Create a data frame called "cars" that contains the first five rows of the mtcars dataset by using the "head" function. After printing to the console, you should get the following result:

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

Exercise: 9-A

Write a function named "read_file" which will accept a file name as a parameter named "file_name". The function should then read in a csv with the specified name, store it as a data frame named "df", and return "df" as the final output.

Exercise: 9-B

In exercise 9-A you created a function that will allow you to read a csv file. Extend this function by adding a second parameter named "csv" which will accept either "TRUE" or "FALSE". The functionality shouldn't change if the parameter is equal to "TRUE"; however, if the function is equal to "FALSE", the function should allow the user to read in an xlsx file instead. For example, if a user wanted to read in a csv file they would use the function in this way:

```
read_file("iris.csv", TRUE)
```

If the user wanted to read in an xlsx file they would use the function in this way:

```
read_file("iris.xlsx", FALSE)
```

Answers

Answer: 8-A

This task can be accomplished with the following code:

```
cars <- head(mtcars, 5)
print(cars)
```

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

Answer: 9-A

This task can be accomplished with the following code:

```
read_file <- function(file_name) {
    df <- read.csv(file_name)
    return(df)
}
```

Answer: 9-B

Here's one way you could write your function to accomplish this task:

```r
library(readxl)

read_file <- function(file_name, csv) {
    if (csv == TRUE) {
        df <- read.csv(file_name)
        return(df)
    }

    if (csv == FALSE) {
        df <- read_excel(file_name)
        return(df)
    }
}
```

Part IV

Part III: Data Preparation

Most data will not be received in the precise format you need to begin your analysis. The process of data preparation is where you will structure and add features to your data.

- **Data Cleaning-** This chapter will cover the basics of cleaning your data, including renaming variables, splitting text, replacing values, dropping columns, and dropping rows. These basic actions will be essential to preparing your data prior to developing insights.
- **Handling Missing Data-** You may encounter situations where some of your data are missing. This chapter will cover best practices on dealing with missing data and introduce the tools to do so.
- **Outliers-** Outliers are observations that fall outside the expected scope of the dataset. It's important to identify outliers and either choose analyses strategies that are robust to their presence or deal with them appropriately before moving into the next analysis phase.
- **Organizing Data-** This chapter will focus on sorting, filtering, and grouping your datasets.

Chapter 11

Data Cleaning

This chapter will cover the basics of cleaning your data including renaming variables, splitting text, replacing values, dropping columns, and dropping rows. These basic actions will be essential to preparing your data prior to developing insights.

11.1 Renaming Variables

Let's begin by creating a dataset we can use to work through some examples. In our case, we'll take the first few rows from the "iris" dataset and create a new dataframe called "df".

```r
df <- head(iris)
print(df)
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa

Now, let's change our column names (which contain different properties of iris species) into "snake case", e.g. all words are lowercase and separated by underscores. We'll do this through the use of the "colnames" function. In the following example, we are renaming each column individually by specifying what number column to adjust.

```
colnames(df)[1] <- "sepal_length"
colnames(df)[2] <- "sepal_width"
colnames(df)[3] <- "petal_length"
colnames(df)[4] <- "petal_width"
colnames(df)[5] <- "species"
```

sepal_length	sepal_width	petal_length	petal_width	species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa

Let's change the column names again, but use "camel case" this time, e.g. the first word will be lowercase, and all subsequent words will have the first letter capitalized. Instead of using the column number though, this time we'll use the actual name of the column we want to adjust.

```
colnames(df)[colnames(df) == "sepal_length"] <- "sepalLength"
colnames(df)[colnames(df) == "sepal_width"] <- "sepalWidth"
colnames(df)[colnames(df) == "petal_length"] <- "petalLength"
colnames(df)[colnames(df) == "petal_width"] <- "petalWidth"
```

sepalLength	sepalWidth	petalLength	petalWidth	species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa

Alternatively, you can use the "rename" function from the "dplyr" package.

```
library(dplyr)
df <- rename(df, "plantSpecies" = "species")
```

sepalLength	sepalWidth	petalLength	petalWidth	plantSpecies
5.1	3.5	1.4	0.2	setosa

sepalLength	sepalWidth	petalLength	petalWidth	plantSpecies
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa

11.2 Splitting Text

If you've worked in a spreadsheet application before, you're likely familiar with the "text-to-columns" tool. This tool allows you to split one column of data into multiple columns based on a delimiter. This same functionality is also achievable in R through functions such as the "separate" function from the "tidyr" library.

To test this function out, let's first attach the "tidyr" package and then create a test data frame for us to use.

```
library(tidyr)
df <- data.frame(person = c("John_Doe", "Jane_Doe"))
```

person
John_Doe
Jane_Doe

We now have a data frame with one column that contains a first name and a last name combined by an underscore. Let's now split the two names into their own separate columns.

```
df <- df %>% separate(person, c("first_name", "last_name"), "_")
```

first_name	last_name
John	Doe
Jane	Doe

Let's break down what just happened. We first declared that "df" was going to be equal to the output of the function that followed by typing "df <-". Next we told the separate function that it would be altering the existing dataframe called "df" by typing "df %>%".

We then gave the separate function three arguments. The first argument was the column we were going to be editing, "person". The second argument was the names of our two new columns, "first_name" and "last_name". Finally, the third argument was our desired delimiter, "_".

11.3 Replace Values

We'll next go over how you can replace specific values in a dataset. Let's begin by creating a dataset to work with. The following example will create a dataframe which contains student names and their respective grades on a test.

```
students <- c("John", "Jane", "Joe", "Janet")
grades <- c(83, 97, 74, 27)
df <- data.frame(student = students, grade = grades)
```

student	grade
John	83
Jane	97
Joe	74
Janet	27

Now that our dataset is assembled, let's decide that we're going to institute a minimum grade of 60. To do this we're going to need to replace any grade lower than 60 with 60. The following example demonstrates one way you could accomplish that.

```
df[which(df$"grade" < 60), "grade"] <- 60
```

student	grade
John	83
Jane	97
Joe	74
Janet	60

11.4 Drop Columns

Let's use the "mtcars" dataset to demonstrate how to drop columns

```
df <- head(mtcars)
print(df)
```

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

Next, we can either drop columns by specifying the columns we want to keep or by specifying the ones we want to drop. The following example will get rid of the “carb” column by specifying that we want to keep every other column.

```
df <- subset(df, select = c(mpg, cyl, disp, hp, drat, wt, qsec, vs, am, gear))
```

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

Alternatively, let’s try getting rid of the “gear” column directly. We can do this by putting a “-” in front of the “c” function.

```
df <- subset(df, select = -c(gear))
```

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

One other way you could drop columns if you wanted to use index numbers rather than column names is demonstrated below.

```
df <- df[,-c(1,3:7)]
```

	cyl	vs	am
Mazda RX4	6	0	1
Mazda RX4 Wag	6	0	1
Datsun 710	4	1	1
Hornet 4 Drive	6	1	0
Hornet Sportabout	8	0	0
Valiant	6	1	0

As you can see, we used the square brackets to select a subset of our dataframe and then pasted our values after the comma to declare that we were choosing columns rather than rows. After that we used the "-" symbol to say that we were choosing columns to drop rather than columns to keep. Finally, we chose to drop columns 1 as well as columns 3 through 7.

11.5 Drop Rows

We are also able to drop rows with the same method we just used to drop columns with the difference being that we would place our values in front of the comma rather than after the comma. For example, if we wanted to drop the first two rows (otherwise known as observations) from our previous dataframe, we could do the following.

```
df <- df[-c(1:2),]
```

	cyl	vs	am
Datsun 710	4	1	1
Hornet 4 Drive	6	1	0
Hornet Sportabout	8	0	0
Valiant	6	1	0

11.6 Resources

- "Separate" function documentation: https://tidyr.tidyverse.org/reference/separate.html

Chapter 12

Handling Missing Data

You may encounter situations while analysing data that some of your data are missing. This chapter will cover best practices in regards to handling these situations as well as the technical details on how to remedy the data.

Missing data will often be represented by either "NA" or " " in R. Sometimes you will be able to manage by just ignoring this data; however, other times you will need to "impute" the missing data. This just means you end up coming up with a value that makes sense to use in place of the missing data. The three imputation methods we are going to cover in this chapter are constant value imputation, central tendency imputation, and multiple imputation.

12.1 Handling NA/Blank Values

This section will cover common methods and formulas for identifying and isolating missing data. Let's start by creating a a vector with one " " value and a vector with one "NA" value.

```
blanks <- c("John", "Jane", "")
nas <- c(NA, "Jane", "Joe")
```

```
print(blanks)
```

```
[1] "John" "Jane" ""
```

```
print(nas)
```

```
[1] NA     "Jane" "Joe"
```

We can use the "is.na" function to identify data with "NA" values. The following example demonstrates how the function works. The output ends up being a "TRUE" or "FALSE" to designate whether each observation is an "NA" value.

```
is.na(nas)
```

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

We can then take this one step further and use the function to filter for "NA" values.

```
only_nas <- nas[is.na(nas)]
print(only_nas)
```

```
[1] NA
```

This works great; however, it's more likely that you would want to see the values which aren't equal to "NA". This can be accomplished by using the "NOT" operator "!".

```
no_nas <- nas[!is.na(nas)]
print(no_nas)
```

```
[1] "Jane" "Joe"
```

If your missing data is just an empty string ("") rather than an"NA" value, you can use simple comparison operators to accomplish the same thing.

```
blanks == ""
```

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

```
only_blanks <- blanks[blanks == ""]
print(only_blanks)
```

```
[1] ""
```

```
no_blanks <- blanks[blanks != ""]
print(no_blanks)
```

```
[1] "John" "Jane"
```

When working with dataframes rather than just vectors, you can also use the "na.omit" function to remove complete rows with "NA" values.

```r
students <- c("John", "Jane", "Joe")
scores <- c(100, 80, NA)
df <- data.frame(student = students, score = scores)
print(df)
```

```
  student score
1    John   100
2    Jane    80
3     Joe    NA
```

```r
df <- na.omit(df)
print(df)
```

```
  student score
1    John   100
2    Jane    80
```

12.2 Constant Value Imputation

Many datasets you encounter will likely be missing data. The temptation may be to immediately disregard these observations; however, it's important to consider what missing data represents in the context of your dataset as well as the context of what your analysis is hoping to achieve. For example, say you are a teacher and you are trying to determine the average test scores of your students. You have a dataset which lists your students names along with their respective test scores. However, you find that one of your students has an "NA" value in place of a test score.

```r
students <- c("John", "Jane", "Joe")
scores <- c(100, 80, NA)
df <- data.frame(student = students, score = scores)

print(df)
```

```
  student score
1    John   100
2    Jane    80
3     Joe    NA
```

Depending on the context, it may make sense for you to ignore this observation prior to calculating the average score. It could also make sense for you to assign a value of "0" to this student's test score.

Let's demonstrate how you would replace "NA" values with a constant value of "0".

```
df[is.na(df)] <- 0
print(df)
```

```
  student score
1    John   100
2    Jane    80
3     Joe     0
```

12.3 Central Tendency Imputation

Two of the most common measures of central tendency are “mean” and “median”. Suppose you have a dataset that tracks the time employees spend performing a certain task. After review, you realize that several employees have not historically tracked their time. Instead of just ignoring these entries, you decide to try imputing these values.

```
employees <- c("John", "Jane", "Joe", "Janet")
hours_spent <- c(12, 14, NA, 9)
df <- data.frame(employee = employees, hours_spent = hours_spent)

print(df)
```

```
  employee hours_spent
1     John          12
2     Jane          14
3      Joe          NA
4    Janet           9
```

The following example demonstrates how you can replace missing values with an average of the rest of the employees’ time spent.

```
mean_value <- mean(df$hours_spent[!is.na(df$hours_spent)])
print(mean_value)
```

```
[1] 11.66667
```

```
df$hours_spent[is.na(df$hours_spent)] <- mean_value
print(df)
```

```
  employee hours_spent
1     John    12.00000
2     Jane    14.00000
3      Joe    11.66667
4    Janet     9.00000
```

Alternatively, we can reset our dataframe and replace "NA" values with the median value by doing the following.

```
# RESET DATAFRAME
df$hours_spent <- hours_spent

# SET MISSING VALUES TO MEDIAN
median_value <- median(df$hours_spent[!is.na(df$hours_spent)])
print(median_value)
```

```
[1] 12
```

```
df$hours_spent[is.na(df$hours_spent)] <- median_value
print(df)
```

```
  employee hours_spent
1     John          12
2     Jane          14
3      Joe          12
4    Janet           9
```

12.4 Multiple Imputation

The two previous examples are types of "single value imputation" as both examples took one value and applied it to every missing value in the dataset. At a very basic level, multiple imputation requires users to come up with some sort of model to fill in missing values. In the following example we are going to demonstrate how you might use a simple linear regression model to perform multiple imputation.

> **Note**
>
> Linear regression is covered more in-depth later in this book. Don't worry if this example feels completely unfamiliar at this point.

We'll begin by creating a dataframe with both an "x" and a "y" variable.

```
y <- c(10, 8, NA, 9, 4, NA)
x <- c(8, 6, 9, 7, 2, 12)
df <- data.frame(y = y, x = x)

print(df)
```

```
   y  x
```

```
1 10  8
2  8  6
3 NA  9
4  9  7
5  4  2
6 NA 12
```

Next, let's use the "lm" function to create a linear model and then print out a summary of that model.

```r
model <- lm(y ~ x)
summary(model)
```

```
Warning in summary.lm(model): essentially perfect fit: summary may be unreliable

Call:
lm(formula = y ~ x)

Residuals:
1 2 4 5
0 0 0 0

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)        2          0     Inf   <2e-16 ***
x                  1          0     Inf   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0 on 2 degrees of freedom
  (2 observations deleted due to missingness)
Multiple R-squared:      1, Adjusted R-squared:      1
F-statistic:   Inf on 1 and 2 DF,  p-value: < 2.2e-16
```

From the model summary, we can see that we have a model with a high level of statistical significance. Let's now use the model coefficients to impute our missing values.

```r
imputed <- predict(model, newdata = list(x = df$x[is.na(df$y)]))
df$y[is.na(df$y)] <- imputed
print(df)
```

```
   y  x
1 10  8
2  8  6
3 11  9
4  9  7
```

```
5  4  2
6 14 12
```

12.5 Resources

- “Missing-data Imputation” from Columbia: http://www.stat.columbia.edu/~gelman/arm/missing.pdf

Chapter 13

Outliers

Outliers are observations that fall outside the expected scope of the dataset. It's important to identify outliers in your data and determine the necessary treatment for them before moving into the next analysis phase.

For example, it might be necessary to impute values, remove a row, perform sensitivity analysis, or choose analysis methods that are robust in the presence of outliers.

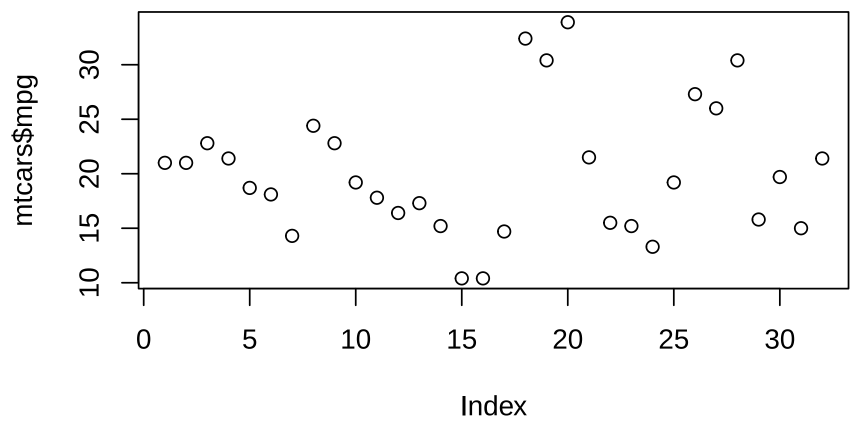
13.1 Finding Outliers Visually

One common first step many people employ when looking for outliers is visualizing their datasets so that extreme values can quickly be spotted This section will briefly cover several common visualizations used to identify outliers; however, each of these plots will be explored more in-depth later in the book.

13.1.1 Scatter Plot

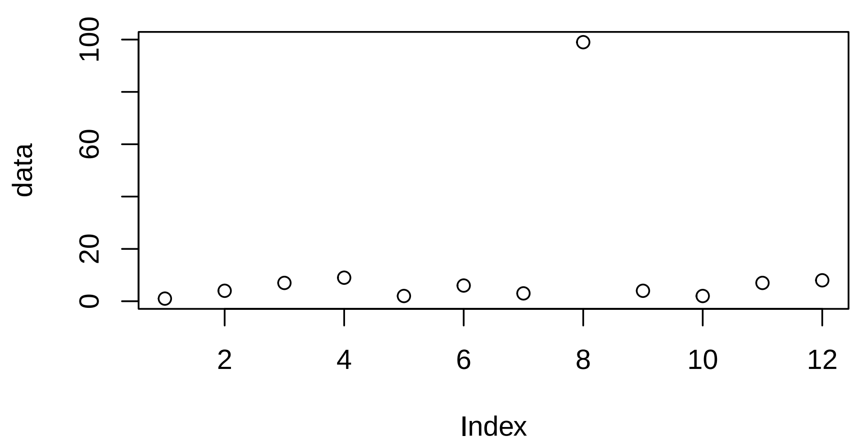
This is probably the first plot you'll reach for when trying to visualize outliers in your data. The scatter plot is a great tool to quickly visualize your data at a high level and see if anything major stands out.

```
plot(mtcars$mpg)
```



Here's how a scatter plot with an extreme outlier might look.

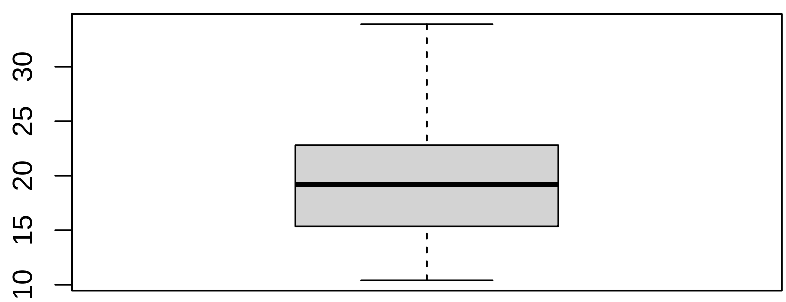
```
data <- c(1,4,7,9,2,6,3,99,4,2,7,8)
plot(data)
```



13.1.2 Box Plot

Another way to quickly visualize outliers is to use the “boxplot” function. This plot will allow you to evaluate outliers in a more systematic way.

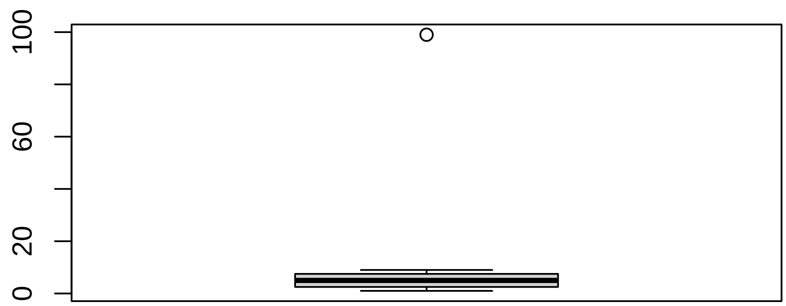
```
boxplot(mtcars$mpg)
```



The solid black line represents the median value of your dataset. The top and bottom “whiskers” represent your extreme values (minimum and maximum). The top and bottom of the “box” represent the first and third quartile.

Here’s an example of a box plot with an extreme outlier.

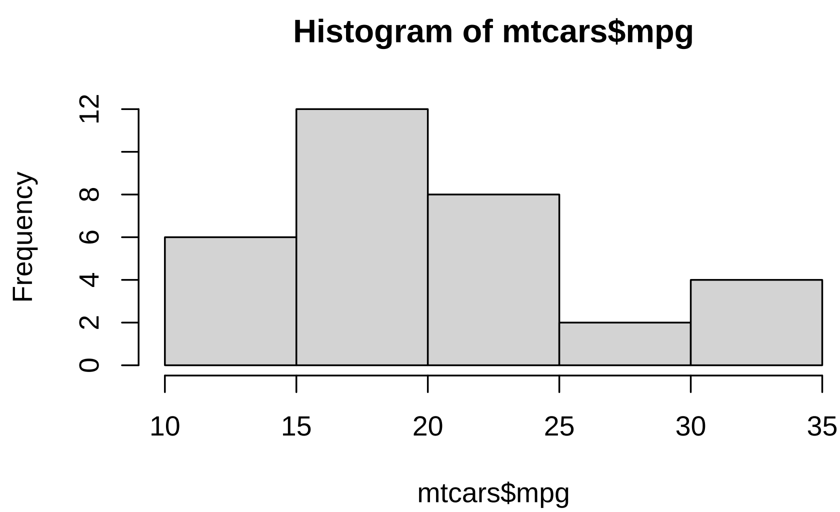
```
boxplot(data)
```



13.1.3 Histogram

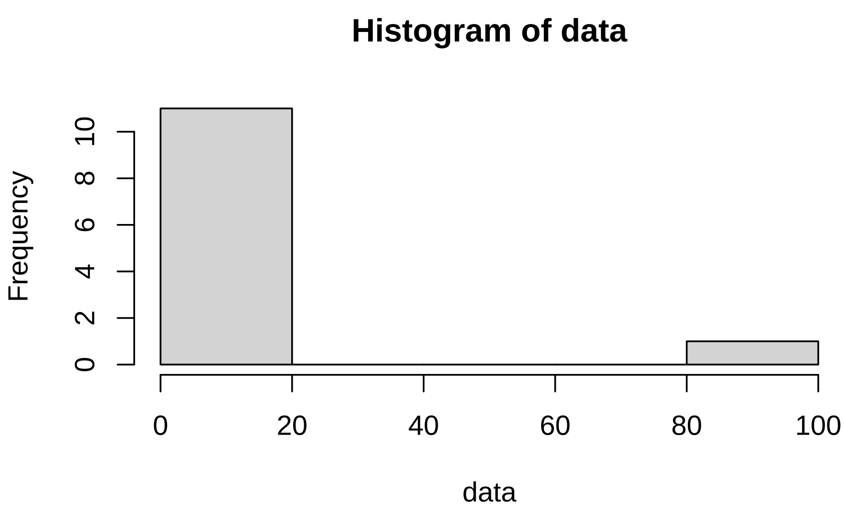
Histograms will allow you to see how often values occur within certain buckets.

```
hist(mtcars$mpg)
```



Here's a histogram with data that contains an outlier.

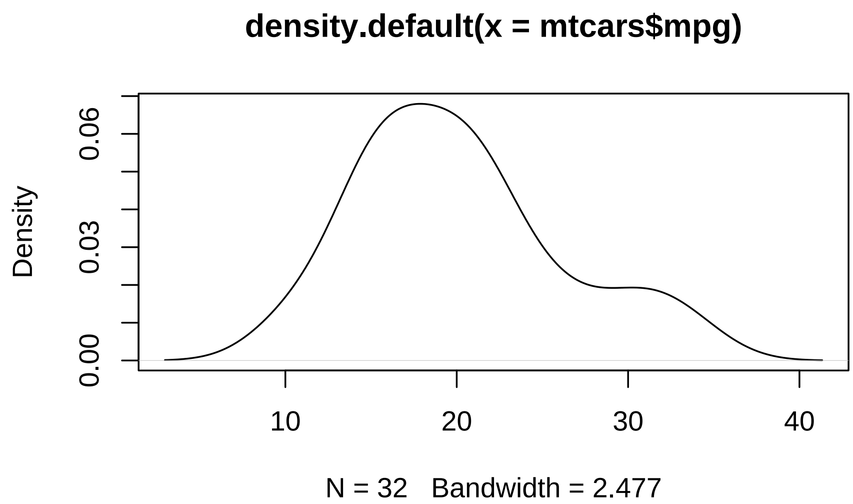
```r
hist(data)
```



13.1.4 Density Plot

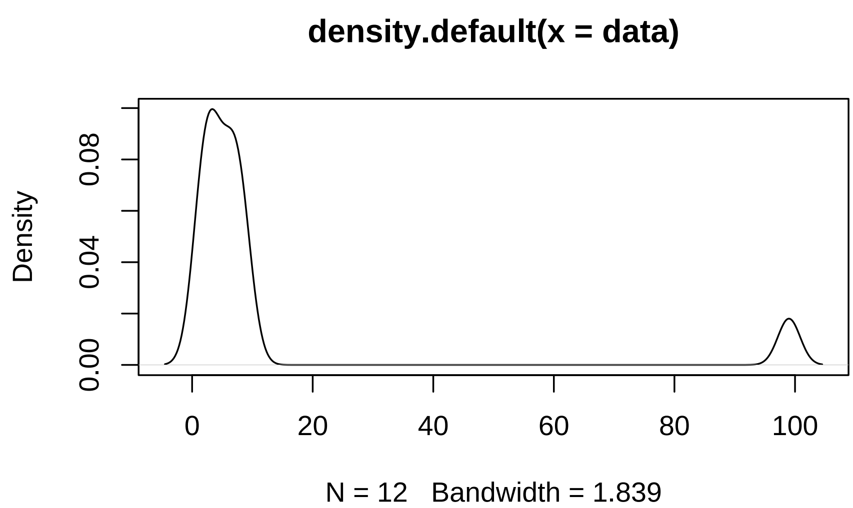
Density plots can be thought of as a smoothed version of a histogram. (You can tune the degree of smoothing, e.g. via the `adjust` argument to the `density()` function.)

```r
plot(density(mtcars$mpg))
```



Here's an example of a density plot with data that contains an outlier.

```
plot(density(data))
```



13.2 Finding Outliers Statistically

While examining your data visually may be a convenient and sufficient way to detect outliers in your data, sometimes you may require a more rigorous approach to outlier detection.

13.2.1 Standard Deviation

One simple way to check the extremity of your observation is to calculate how many standard deviations it falls from the mean.

Let's start by calculating the standard deviation of our dataset by using the "sd" function.

```
sd <- sd(data)
print(sd)
```

```
[1] 27.31078
```

Next, let's calculate the mean of our dataset.

```
mean <- mean(data)
print(mean)
```

```
[1] 12.66667
```

Finally, for each record in our vector, let's calculate how many standard deviations it falls from the mean.

```
extremity <- abs(data - mean) / sd
print(extremity)
```

```
 [1] 0.4271817 0.3173350 0.2074883 0.1342571 0.3905661 0.2441038 0.3539506
 [8] 3.1611447 0.3173350 0.3905661 0.2074883 0.1708727
```

13.3 Removing Outliers

After identifying your outliers you have several options to remove them.

Your first option would be to manually remove a specific outlier.

```
manually_cleaned <- data[data != 99]
print(manually_cleaned)
```

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

A more robust option would be to rely on your previously performed calculations to remove any observations which are located too far away from the mean.

```
statistically_cleaned <- data[extremity < 3]
print(statistically_cleaned)
```

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

13.4 Resources

"Statistics - Standard Deviation" by W3 Schools: https://www.w3schools.com/statistics/statistics_standard_deviation.php "Identifying outliers with the 1.5xIQR rule": https://www.khanacademy.org/math/statistics-probability/summarizing-quantitative-data/box-whisker-plots/a/identifying-outliers-iqr-rule

Chapter 14

Organizing Data

This chapter will focus on sorting, filtering, and grouping your datasets.

14.1 Sort, Order, and Rank

Three functions you may use to organize your data are “sort”, “order”, and “rank”. The following examples will go through each one and show you how to use them.

Let’s start by creating a vector to work with.

```
completed_tasks <- c(5, 9, 3, 2, 7)
print(completed_tasks)
```

```
[1] 5 9 3 2 7
```

Next we’ll sort our data by using the “sort” function. This function will return your original data but sorted in ascending order.

```
sort(completed_tasks)
```

```
[1] 2 3 5 7 9
```

Alternatively, you can set the “decreasing” parameter to “TRUE” to sort your data in descending order.

```
sort(completed_tasks, decreasing = TRUE)
```

```
[1] 9 7 5 3 2
```

The "order" function will return the index of each item in your vector in sorted order. This function also has a "decreasing" parameter which can be set to "TRUE".

```
order(completed_tasks)
```

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

Finally, the "rank" function will return the rank of each item in your vector in ascending order.

```
rank(completed_tasks)
```

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

14.2 Filtering

You may have noticed in previous chapters that we've used comparison operators to filter our data. Let's review by filtering out completed tasks greater than or equal to 7.

```
completed_tasks[completed_tasks < 7]
```

```
[1] 5 3 2
```

Alternatively, you can use the "filter" function from the "dplyr" library. Let's use this function with the "iris" dataset to filter out any species other than virginica.

```
head(iris)
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa

```
library(dplyr)
virginica <- filter(iris, Species == "virginica")
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
6.3	3.3	6.0	2.5	virginica
5.8	2.7	5.1	1.9	virginica
7.1	3.0	5.9	2.1	virginica
6.3	2.9	5.6	1.8	virginica
6.5	3.0	5.8	2.2	virginica
7.6	3.0	6.6	2.1	virginica

14.3 Grouping

One final resource for you to leverage as you organize your data is the "group_by" function from the "dplyr" library.

If we wanted to group the iris dataset by species we might do something similar to the following example.

```
library(dplyr)
grouped_species <- iris %>% group_by(Species)
```

Now if we print out our resulting dataset you'll notice that the "group_by" operation we just performed doesn't change how the data looks by itself.

```
head(grouped_species)
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa

In order to change the structure of our dataset we'll need to specify how our groups should be treated by combining the "group_by" function with another dplyr "verb" such as "summarise".

```
grouped_species <- grouped_species %>% summarise(
    sepal_length = mean(Sepal.Length),
    sepal_width = mean(Sepal.Width),
    petal_length = mean(Petal.Length),
    petal_width = mean(Petal.Width)
)
```

```
head(grouped_species)
```

Species	sepal_length	sepal_width	petal_length	petal_width
setosa	5.006	3.428	1.462	0.246
versicolor	5.936	2.770	4.260	1.326
virginica	6.588	2.974	5.552	2.026

Now each of the three species in the iris dataset have their average sepal length, sepal width, petal length, and petal width displayed.

You can find more information about the "group_by" function and other dplyr "verbs" in the resources section below.

14.4 Resources

- dplyr "filter" function documentation: https://dplyr.tidyverse.org/reference/filter.html
- dplyr "group_by" function documentation: https://dplyr.tidyverse.org/reference/group_by.html

Exercises

Questions

Exercise: 11-A

Create a dataframe named "df" which is equal to the first three columns and the first five rows of the "mtcars" dataset. Next, rename the "mpg" column to "miles_per_gallon".
After printing the resulting dataframe to the console you should have the following results:

```
#                   miles_per_gallon cyl disp
# Mazda RX4                     21.0   6  160
# Mazda RX4 Wag                 21.0   6  160
# Datsun 710                    22.8   4  108
# Hornet 4 Drive                21.4   6  258
# Hornet Sportabout             18.7   8  360
```

Exercise: 12-A

You are given the following dataframe:

```
var_1 <- c(3, 4, 2, 9, NA, 2, 7)
var_2 <- c(8, NA, 6, 4, 8, 5, 5)
df <- data.frame(var_1 = var_1, var_2 = var_2)
print(df)
#   var_1 var_2
# 1     3     8
# 2     4    NA
# 3     2     6
# 4     9     4
# 5    NA     8
# 6     2     5
# 7     7     5
```

Create a new dataframe called "cleaned_df" which is equal to "df" except with both rows which contain "NA" values removed.
The final output of "cleaned_df" should look like this:

```
#   var_1 var_2
# 1     3     8
# 3     2     6
# 4     9     4
# 6     2     5
# 7     7     5
```

Exercise: 12-B

Take the original "df" dataframe from exercise 12-A and apply a constant value of "5" to each "NA" value. Store this new dataframe in a variable named "constant_value".
Your final output after printing "constant_value" to the console should look like this:

```
print(constant_value)
#   var_1 var_2
# 1     3     8
# 2     4     5
# 3     2     6
# 4     9     4
# 5     5     8
# 6     2     5
# 7     7     5
```

Exercise: 12-C

Take the same "df" dataframe from exercises 12-A and 12-B and apply an average value of each column to "NA" values in each respective column. Store this new dataframe in a variable named "mean_value".
Your final output after printing "mean_value" to the console should look like this:

```
print(mean_value)
#   var_1 var_2
# 1   3.0     8
# 2   4.0     6
# 3   2.0     6
# 4   9.0     4
# 5   4.5     8
# 6   2.0     5
# 7   7.0     5
```

Exercise: 13-A

Use the "Nile" dataset to create a histogram to view the distribution of it's data.

Exercise: 14-A

Take the dataframe created in exercise 11-A and drop any row where the "disp" column is equal to "160".
You should receive the following results when you print the resulting dataframe to the console.

```
#                   miles_per_gallon cyl disp
# Datsun 710                    22.8   4  108
# Hornet 4 Drive                21.4   6  258
# Hornet Sportabout             18.7   8  360
```

Answers

Answer: 11-A

This task could be accomplished in the following way:

```r
library(dplyr)
```

```
Attaching package: 'dplyr'
The following objects are masked from 'package:stats':

    filter, lag
The following objects are masked from 'package:base':

    intersect, setdiff, setequal, union
```

```r
df <- mtcars[1:5, 1:3]
df <- rename(df, "miles_per_gallon" = "mpg")
print(df)
```

```
                  miles_per_gallon cyl disp
Mazda RX4                     21.0   6  160
Mazda RX4 Wag                 21.0   6  160
Datsun 710                    22.8   4  108
Hornet 4 Drive                21.4   6  258
Hornet Sportabout             18.7   8  360
```

Answer: 12-A

This task could be accomplished in the following way:

```r
var_1 <- c(3, 4, 2, 9, NA, 2, 7)
var_2 <- c(8, NA, 6, 4, 8, 5, 5)
df <- data.frame(var_1 = var_1, var_2 = var_2)
cleaned_df <- na.omit(df)
print(cleaned_df)
```

```
  var_1 var_2
1     3     8
3     2     6
4     9     4
6     2     5
7     7     5
```

Answer: 12-B

There are several ways this task could be accomplished; however, the following example demonstrates one way to do it.

```r
var_1 <- c(3, 4, 2, 9, NA, 2, 7)
var_2 <- c(8, NA, 6, 4, 8, 5, 5)
df <- data.frame(var_1 = var_1, var_2 = var_2)

constant_value <- df
constant_value[is.na(constant_value)] <- 5
print(constant_value)
```

```
  var_1 var_2
1     3     8
2     4     5
3     2     6
4     9     4
5     5     8
6     2     5
7     7     5
```

Answer: 12-C

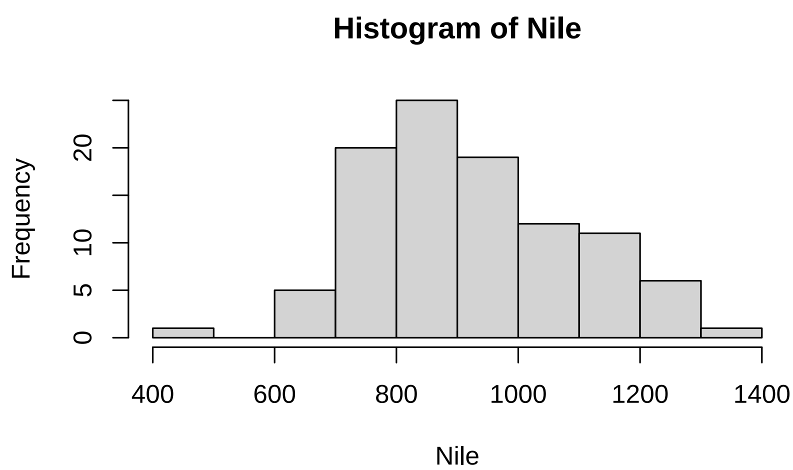
There are several ways this task could be accomplished; however, the following example demonstrates one way to do it.

```r
var_1 <- c(3, 4, 2, 9, NA, 2, 7)
var_2 <- c(8, NA, 6, 4, 8, 5, 5)
df <- data.frame(var_1 = var_1, var_2 = var_2)

mean_1 <- mean(df$var_1[!is.na(df$var_1)])
mean_2 <- mean(df$var_2[!is.na(df$var_2)])

mean_value <- df
mean_value$var_1[is.na(mean_value$var_1)] <- mean_1
mean_value$var_2[is.na(mean_value$var_2)] <- mean_2
print(mean_value)
```

```
  var_1 var_2
1   3.0     8
2   4.0     6
3   2.0     6
4   9.0     4
5   4.5     8
6   2.0     5
7   7.0     5
```

Answer: 13-A

```
hist(Nile)
```



Answer: 14-A

This task could be accomplished in the following way:

```
library(dplyr)
df <- mtcars[1:5, 1:3]
df <- rename(df, "miles_per_gallon" = "mpg")

df <- filter(df, disp != 160)
print(df)
```

```
                  miles_per_gallon cyl disp
Datsun 710                    22.8   4  108
Hornet 4 Drive                21.4   6  258
Hornet Sportabout             18.7   8  360
```

Part V

Part IV: Developing Insights

> "A learning organization is an organization skilled at creating, acquiring, and transferring knowledge, and at modifying its behavior to reflect new knowledge and insights." -David A. Garvin (Garvin 1993)

Once your data is prepared, you can begin to make sense of it and develop insights about its meaning. For many, this is where the data analysis process becomes the most fulfilling. This is the point where you get to reap what you've sown in the previous phases of the data analysis lifecycle.

- **Summary Statistics-** Summary statistics are usually where one starts when beginning to develop insights. You may hear the phrase "Exploratory Data Analysis" (sometimes abbreviated "EDA") throughout your career. This is the point where you try to get a high-level understanding of your data through methods such as summary statistics.
- **Regression-** Regression is a common statistical technique employed by many to make generalizations as well as predictions about data.
- **Plotting-** This chapter will cover the basics of creating plots in R. It will begin by demonstrating the plotting capabilities available in R "out of the box". You will also be given resources to learn more about "ggplot2" which is one of the most common plotting libraries in R.

Chapter 15

Summary Statistics

Summary statistics (otherwise known as descriptive statistics) are usually where one starts when beginning to develop insights. You may hear the phrase "Exploratory Data Analysis" (sometimes abbreviated "EDA") throughout your career. This is the point where you try to get a high-level understanding of the distributions and relationships within your dataset.

15.1 Quantitative Data

When dealing with continuous data, one of the quickest ways to get a high level view of your data is by using the "summary" function. This function will return your extreme (minimum and maximum) values, your median, mean, 1st quantile, and 3rd quantile.

```
summary(mtcars$mpg)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
  10.40   15.43   19.20   20.09   22.80   33.90 
```

Alternatively, you can use the following eight functions to retrieve specific information about your data.

```
# Returns the average
mean(mtcars$mpg)
```

```
[1] 20.09062
```

```
# Returns the median
median(mtcars$mpg)
```

```
[1] 19.2
```

```
# Returns the standard deviation
sd(mtcars$mpg)
```

```
[1] 6.026948
```

```
# Returns the sample variance
var(mtcars$mpg)
```

```
[1] 36.3241
```

```
# Returns the minimum value
min(mtcars$mpg)
```

```
[1] 10.4
```

```
# Returns the maximum value
max(mtcars$mpg)
```

```
[1] 33.9
```

```
# Returns the minimum and maximum value
range(mtcars$mpg)
```

```
[1] 10.4 33.9
```

```
# Returns quantile data
quantile(mtcars$mpg)
```

```
    0%    25%    50%    75%   100% 
10.400 15.425 19.200 22.800 33.900 
```

15.2 Qualitative Data

If you're working with data that is categorical and encoded as a factor, you can view all categories by using the "levels" function.

```
levels(iris$Species)
```

```
[1] "setosa"     "versicolor" "virginica" 
```

However, if you want to count the number of occurrences for each level, you can use the "table" function.

```
table(iris$Species)
```

```
    setosa versicolor  virginica 
        50         50         50 
```

If you need to keep digging for insights, you can represent your categories however you'd like to using the "group_by" function covered in the last chapter.

15.3 Resources

- "Exploring Data and Descriptive Statistics (using R)" from princeton: https://www.princeton.edu/~otorres/sessions/s2r.pdf

Chapter 16

Regression

Regression is a common statistical technique employed by many to make generalizations as well as predictions about data.

> **Note**
>
> The purpose of this chapter is to give readers a high-level overview of how to apply regression techniques in R rather than to give a full introduction to regression itself. However, there are multiple comprehensive resources in the resources section for interested readers.

16.1 Linear Regression

The first kind of regression we'll cover is linear regression. Linear regression will use your data to come up with a linear model that describes the general trend of your data. Generally speaking, a linear model will consist of a dependent variable (y), at least one independent variable (x), coefficients to go along with each independent variable, and an intercept. Here's one common linear model you may remember:

$$y = mx + b$$

This is a simple linear model many people begin with where x and y are the independent and dependent variables, respectively, m is the slope (or coefficient of x), and b is the intercept.

To perform linear regression in R, you'll use the "lm" function. Let's try it out on the "faithful" dataset.

```r
head(faithful)
```

eruptions	waiting
3.600	79
1.800	54
3.333	74
2.283	62
4.533	85
2.883	55

The “lm” function will accept at least two parameters which represent “y” and “x” in this format:

```r
lm(y ~ x)
```

Let’s try this out by setting the y variable to eruptions and the x variable to waiting. We can then view the output of our linear model by using the “summary” function.

```r
lm <- lm(faithful$eruptions ~ faithful$waiting)
summary(lm)
```

```
Call:
lm(formula = faithful$eruptions ~ faithful$waiting)

Residuals:
     Min       1Q   Median       3Q      Max 
-1.29917 -0.37689  0.03508  0.34909  1.19329 

Coefficients:
                  Estimate Std. Error t value Pr(>|t|)    
(Intercept)      -1.874016   0.160143  -11.70   <2e-16 ***
faithful$waiting  0.075628   0.002219   34.09   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.4965 on 270 degrees of freedom
Multiple R-squared:  0.8115,    Adjusted R-squared:  0.8108 
F-statistic:  1162 on 1 and 270 DF,  p-value: < 2.2e-16
```

This summary will show us the statistical significance of our model along with all relevant statistics to correctly interpret the significance. Additionally, we now have our model coefficients. From this summary we can assume that our model looks something like this:

$$eruptions = waiting * 0.075628 - 1.874016$$

Let's break down everything that the model summary returns.

- The "Call" section calls the model that you created
- The "Residuals" section gives you a summary of all of your model residuals. Simply put, a residual denotes how far away any given point falls from the predicted value.
- The "Coefficients" section gives us our model coefficients, our intercept, and statistical values to determine their significance.
  - For each coefficient, we are given the respective standard error. The standard error is used to measure the precision of coefficient's estimate.
  - Next, we have a t value for each coefficient. The t value is calculated by dividing the coefficient by the standard error.
  - Finally, you have the p value accompanied by symbols to denote the corresponding significance level.
- The residual standard error gives you a way to measure the standard deviation of the residuals and is calculated by dividing residual sum of squares by the residual degrees of freedom and taking the square root of that where the residual degrees of freedom is equal to total observations - total model parameters - 1.
- R-squared gives you the proportion of variance that can be explained by your model. Your adjusted R-squared statistic will tell you the same thing but will adjust for the number of variables you've included in your model.
- Your F-statistic will help you to understand the probability that all of your model parameters are actually equal to zero.

16.2 Multiple Regression

If you had more x variables you wanted to add to your linear model, you could add them just like you would in any other math equation. Here's an example:

```
lm(data$y ~ data$x1 + data$x2 + data$x3 + data$x4)
```

Additionally, you can use the "data" parameter rather than putting the name of the dataset before every variable.

```
lm(y ~ x1 + x2 + x3 + x4, data = data)
```

Let's try a real example with the mtcars dataset.

```
head(mtcars)
```

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

Now, let's try to predict mpg and use every other column as a variable then see what the results look like.

```r
lm <- lm(mpg ~ cyl + disp + hp + drat + wt + qsec + vs + am + gear + carb
            , data = mtcars)
summary(lm)
```

```
Call:
lm(formula = mpg ~ cyl + disp + hp + drat + wt + qsec + vs +
    am + gear + carb, data = mtcars)

Residuals:
    Min      1Q  Median      3Q     Max
-3.4506 -1.6044 -0.1196  1.2193  4.6271

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 12.30337   18.71788   0.657   0.5181
cyl         -0.11144    1.04502  -0.107   0.9161
disp         0.01334    0.01786   0.747   0.4635
hp          -0.02148    0.02177  -0.987   0.3350
drat         0.78711    1.63537   0.481   0.6353
wt          -3.71530    1.89441  -1.961   0.0633 .
qsec         0.82104    0.73084   1.123   0.2739
vs           0.31776    2.10451   0.151   0.8814
am           2.52023    2.05665   1.225   0.2340
gear         0.65541    1.49326   0.439   0.6652
carb        -0.19942    0.82875  -0.241   0.8122
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.65 on 21 degrees of freedom
Multiple R-squared:  0.869, Adjusted R-squared:  0.8066
```

```
F-statistic: 13.93 on 10 and 21 DF,  p-value: 3.793e-07
```

From here, you would likely tweak your model further based on the significance statistics we see here; however, that's outside the scope of what we're doing in this book. Take a look in the resources section at the end of this chapter to dive deeper into developing regression models.

16.3 Logistic Regression

Logistic regression is commonly used when your dependent variable (y) binomial (0 or 1). Instead of using the "lm" function though, you will use the "glm" function. Let's try this out on the mtcars dataset again but this time with "am" as the dependent variable.

```
glm <- glm(am ~ cyl + hp + wt, family = binomial, data = mtcars)
summary(glm)
```

```
Call:
glm(formula = am ~ cyl + hp + wt, family = binomial, data = mtcars)

Deviance Residuals:
     Min        1Q    Median        3Q       Max
-2.17272  -0.14907  -0.01464   0.14116   1.27641

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept) 19.70288    8.11637   2.428   0.0152 *
cyl          0.48760    1.07162   0.455   0.6491
hp           0.03259    0.01886   1.728   0.0840 .
wt          -9.14947    4.15332  -2.203   0.0276 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 43.2297  on 31  degrees of freedom
Residual deviance:  9.8415  on 28  degrees of freedom
AIC: 17.841

Number of Fisher Scoring iterations: 8
```

16.4 Resources

- "Lecture 9 - Linear regression in R" by Professor Alexandra Chouldechova at Carnegie Mellon University: https://www.andrew.cmu.edu/user/

achoulde/94842/lectures/lecture09/lecture09-94842.html

- “Logistic Regression” by Erin Bugbee and Jared Wilber: https://mlu-explain.github.io/logistic-regression/
- “Visualizing OLS Linear Regression Assumptions in R” by Trevor French https://medium.com/trevor-french/visualizing-ols-linear-regression-assumptions-in-r-e762ba7afaff

Chapter 17

Plotting

This chapter will cover the basics of creating plots in R. It will begin by demonstrating the plotting capabilities available in R out of the box. These capabilities are often referred to as “Base R”. In the resources section, you can also find resources to learn more about “ggplot2” which is one of the most common plotting libraries in R.

17.1 Plotting your Regression Model

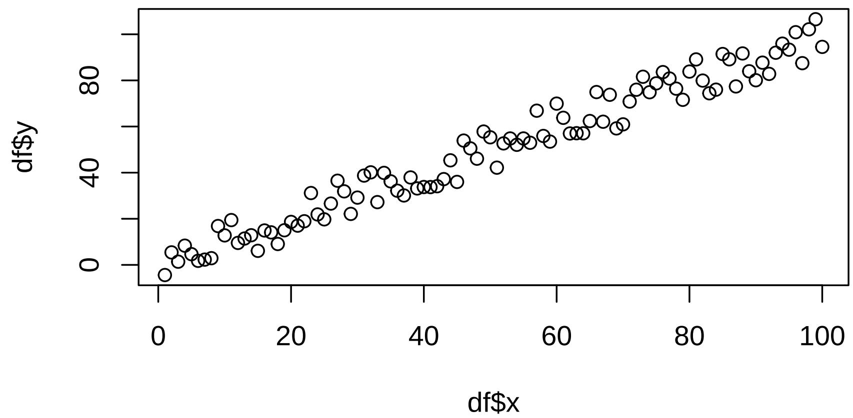
Now that you’ve learned how create a linear regression model, let’s look at how you might go about representing it visually.

Here’s a preview of the dataset we’ll be using:

y	x
-4.400327	1
5.428396	2
1.401835	3
8.347445	4
4.653595	5
1.768966	6

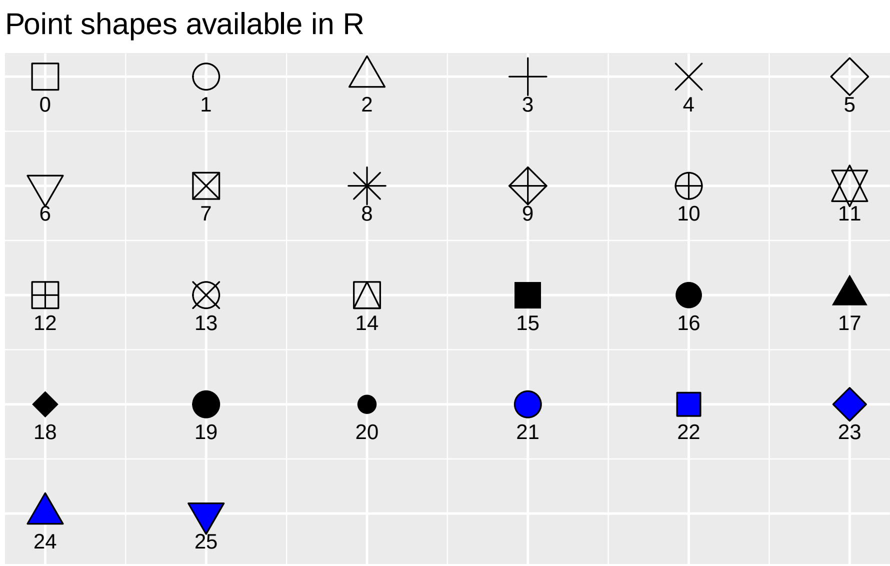
We’ll begin by just creating a scatter plot of the raw data.

```
plot(df$x, df$y)
```



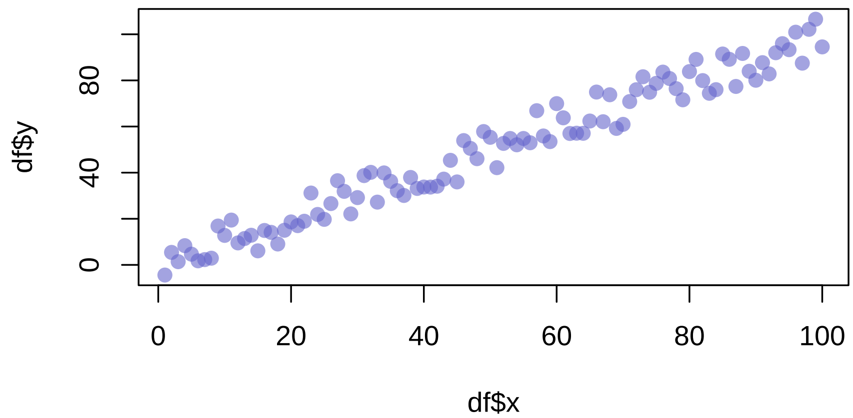
Additionally, you can alter the appearance of your points by using the "pch", "cex", and "col" options. PCH stands for Plot Character and will adjust the symbol used for your points. The available point shapes are listed in the image below.

```
ggpubr::show_point_shapes()
```



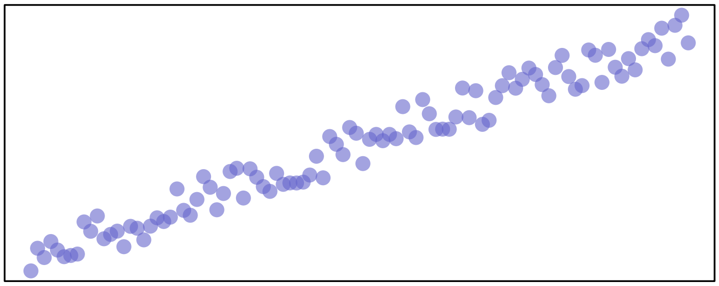
The "cex" option allows you to adjust the symbol size. The default value is 1. If you were to change the value to .75, for example, the plot symbol would be scaled down the 3/4 of the default size. The "col" option allows you to adjust the color of your plot symbols.

```
plot(df$x
        , df$y
        , col=rgb(0.4,0.4,0.8,0.6)
        , pch=16
        , cex=1.2)
```



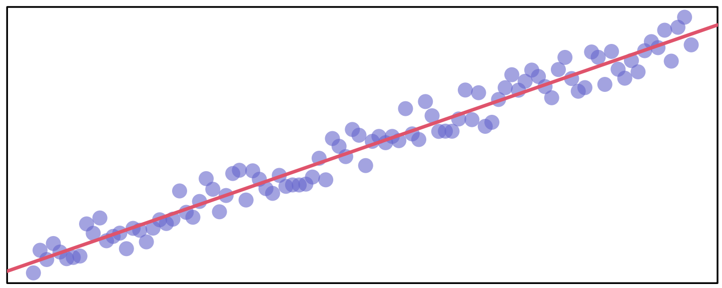
You can adjust the axes with the “xlab”, “ylab”, “xaxt”, and “yaxt” options (amongst other available options). In the following example we will remove the axes altogether.

```
plot(df$x
        , df$y
        , col=rgb(0.4,0.4,0.8,0.6)
        , pch=16
        , cex=1.2
        , xlab=""
        , ylab=""
        , xaxt="n"
        , yaxt="n")
```



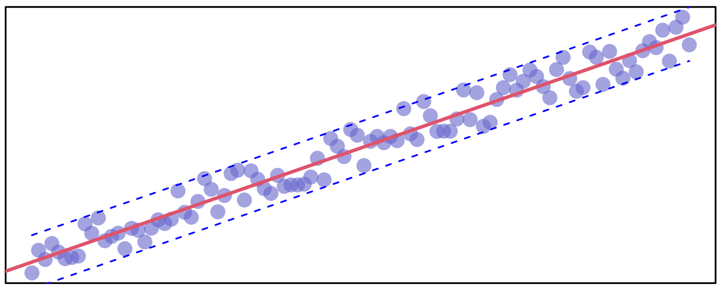
Finally, you can add a trend line by creating a model and adding the fitted values to the graph. We'll also adjust the line width and color with the "lwd" and "col" parameters, respectively.

```
plot(df$x
        , df$y
        , col=rgb(0.4,0.4,0.8,0.6)
        , pch=16
        , cex=1.2
        , xlab=""
        , ylab=""
        , xaxt="n"
        , yaxt="n")

model <- lm(df$y ~ df$x)
abline(model, col=2, lwd=2)
```



The model also returns confidence intervals for the predictions, which can be added

```
# Extract the upper and lower 95% confidence intervals of the predictions
conf_interval <- predict(
  model,
  newdata=data.frame(x=df$x),
  interval = "prediction",
  level = 0.95)

plot(df$x
        , df$y
        , col=rgb(0.4,0.4,0.8,0.6)
        , pch=16
        , cex=1.2
        , xlab=""
        , ylab=""
        , xaxt="n"
        , yaxt="n")
abline(model, col=2, lwd=2)
lines(df$x, conf_interval[,2], col="blue", lty=2)
lines(df$x, conf_interval[,3], col="blue", lty=2)
```



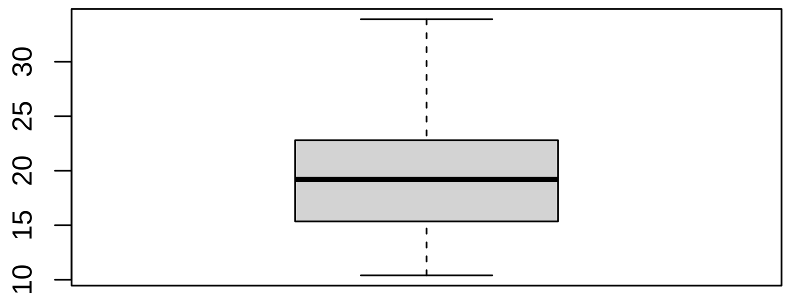
17.2 Plots Available in Base R

Now that you've seen how to build a scatterplot in R, let's take a look at other plots available in Base R.

17.2.1 Box Plot

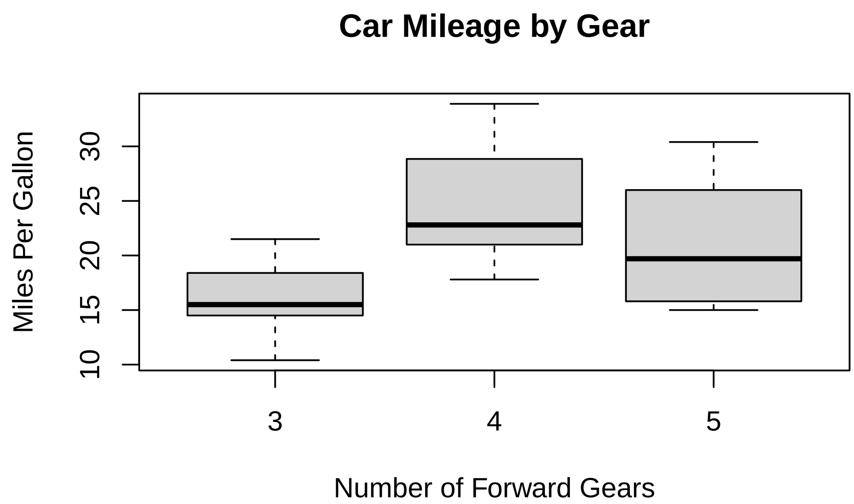
One plot you've already seen in the outliers chapter is the box plot. These plots can be created via the "boxplot" function.

```
boxplot(mtcars$mpg)
```



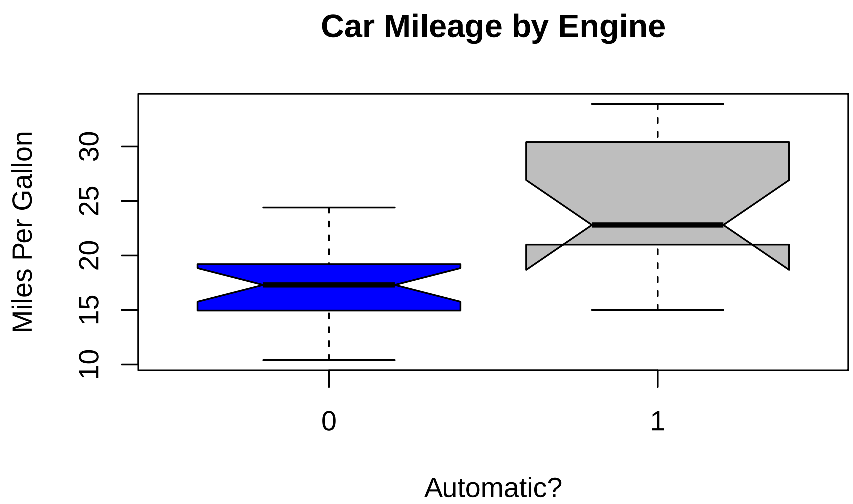
We can build on this plot by specifying the dataset with the "data" parameter, removing the "mtcars$" prefix from our variable, adding a plot title with the "main" parameter, and adding axis labels with the "xlab" and "ylab" parameters. Additionally, we are going to add an additional variable for our data to be categorized by.

```
boxplot(mpg ~ gear
            , data = mtcars
            , main = "Car Mileage by Gear"
            , xlab = "Number of Forward Gears"
            , ylab = "Miles Per Gallon")
```



Finally, we can set the box colors with the "col" parameter and set "notch" equal to "TRUE" to give our boxes notches. If the notches of two plots do not overlap this is 'strong evidence' that the two medians differ Chambers and Tukey (1983).

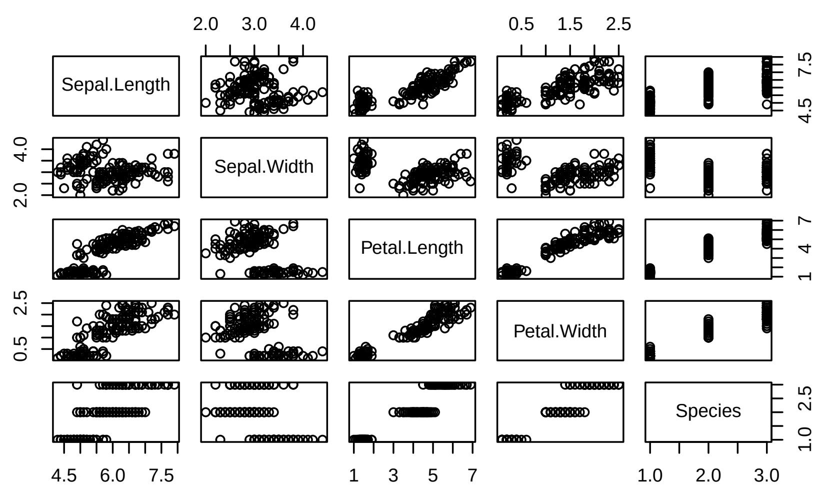
```
boxplot(mpg ~ am
            , data = mtcars
            , notch = TRUE
            , col = (c("blue", "grey"))
            , main = "Car Mileage by Engine"
            , xlab = "Automatic?"
            , ylab = "Miles Per Gallon")
```



17.2.2 Plot Matrix

You can use the “pairs” function to create a plot matrix. Let’s use the iris dataset to demonstrate this.

```
pairs(iris)
```

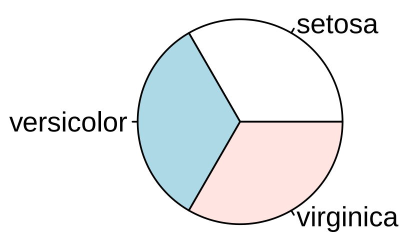


This plot gives us the ability to see how each variable interacts with one another.

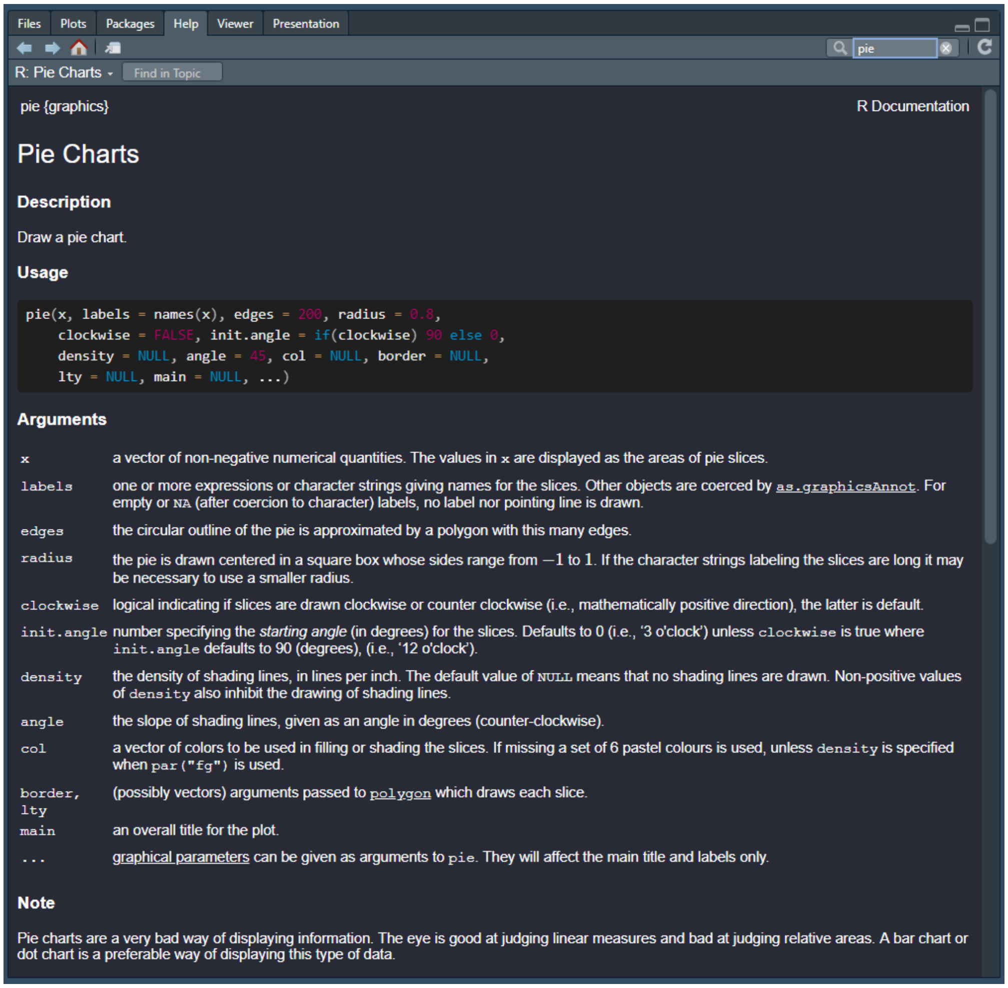
17.2.3 Pie Chart

Let's try plotting a pie chart of species in the iris dataset via the "pie" function. This function accepts numerical values so we'll need to use the "table" function on our column as well.

```
pie(table(iris$Species))
```



You can view the full list of available parameters for this and other functions through the help tab in the files pane in R Studio.



Files Plots Packages Help Viewer Presentation

R: Pie Charts Find in Topic

pie {graphics} R Documentation

Pie Charts

Description

Draw a pie chart.

Usage

```
pie(x, labels = names(x), edges = 200, radius = 0.8,
    clockwise = FALSE, init.angle = if(clockwise) 90 else 0,
    density = NULL, angle = 45, col = NULL, border = NULL,
    lty = NULL, main = NULL, ...)
```

Arguments

`x`	a vector of non-negative numerical quantities. The values in `x` are displayed as the areas of pie slices.
`labels`	one or more expressions or character strings giving names for the slices. Other objects are coerced by `as.graphicsAnnot`. For empty or `NA` (after coercion to character) labels, no label nor pointing line is drawn.
`edges`	the circular outline of the pie is approximated by a polygon with this many edges.
`radius`	the pie is drawn centered in a square box whose sides range from -1 to 1. If the character strings labeling the slices are long it may be necessary to use a smaller radius.
`clockwise`	logical indicating if slices are drawn clockwise or counter clockwise (i.e., mathematically positive direction), the latter is default.
`init.angle`	number specifying the *starting angle* (in degrees) for the slices. Defaults to 0 (i.e., '3 o'clock') unless `clockwise` is true where `init.angle` defaults to 90 (degrees), (i.e., '12 o'clock').
`density`	the density of shading lines, in lines per inch. The default value of `NULL` means that no shading lines are drawn. Non-positive values of `density` also inhibit the drawing of shading lines.
`angle`	the slope of shading lines, given as an angle in degrees (counter-clockwise).
`col`	a vector of colors to be used in filling or shading the slices. If missing a set of 6 pastel colours is used, unless `density` is specified when `par("fg")` is used.
`border, lty`	(possibly vectors) arguments passed to `polygon` which draws each slice.
`main`	an overall title for the plot.
`...`	graphical parameters can be given as arguments to `pie`. They will affect the main title and labels only.

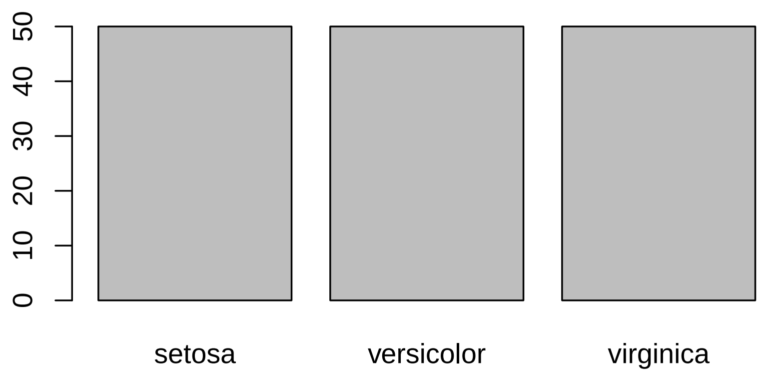
Note

Pie charts are a very bad way of displaying information. The eye is good at judging linear measures and bad at judging relative areas. A bar chart or dot chart is a preferable way of displaying this type of data.

17.2.4 Bar Plot

Let's try a bar plot on the same dataset with the "barplot" function.

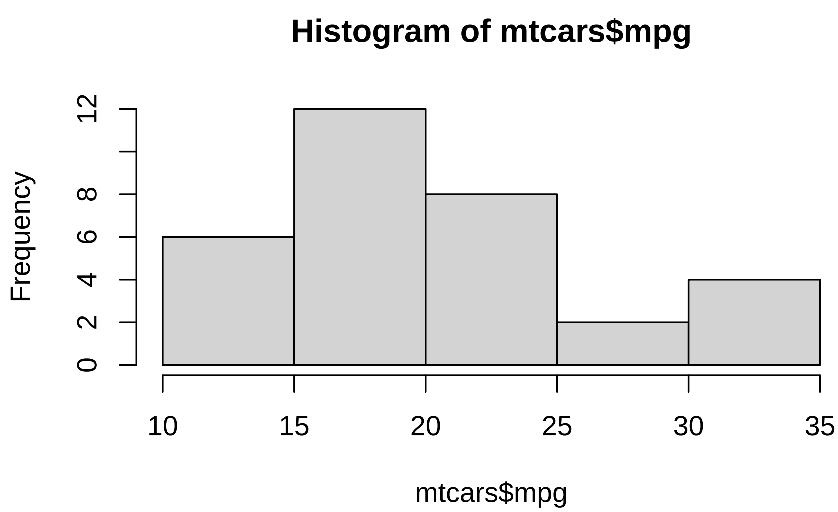
```
barplot(table(iris$Species))
```



17.2.5 Histogram

You may recall that we also used histigrams in the outliers chapter to try to visually identify extreme values. Here's a quick recap:

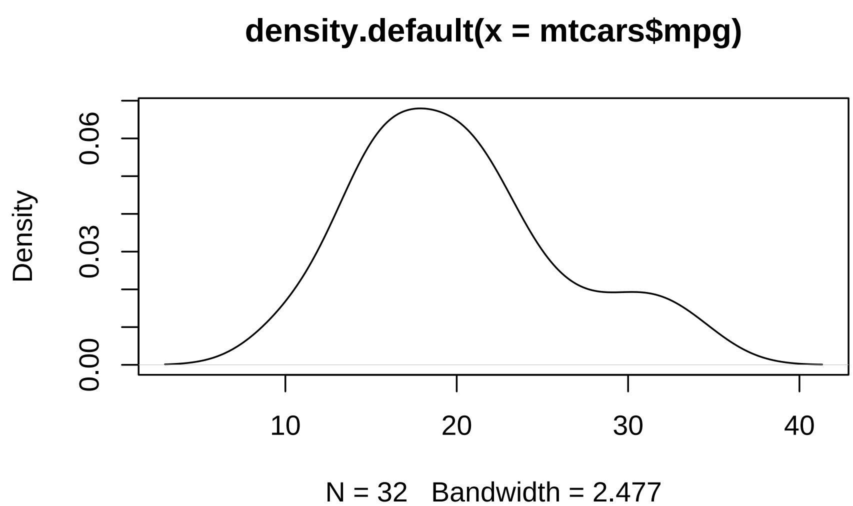
```
hist(mtcars$mpg)
```



17.2.6 Density Plot

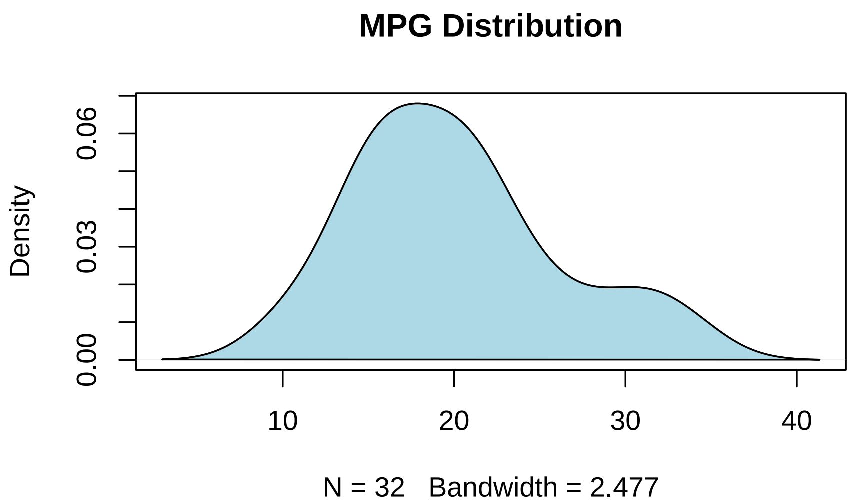
We also used the following example in the outliers chapter to create a density plot:

```
plot(density(mtcars$mpg))
```



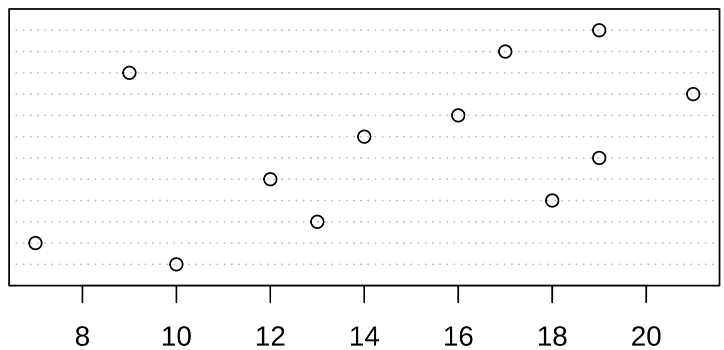
We can take this one step further by adding a title and a shape to the plot.

```
mpg <- density(mtcars$mpg)
plot(mpg, main="MPG Distribution")
polygon(mpg, col="lightblue", border="black")
```

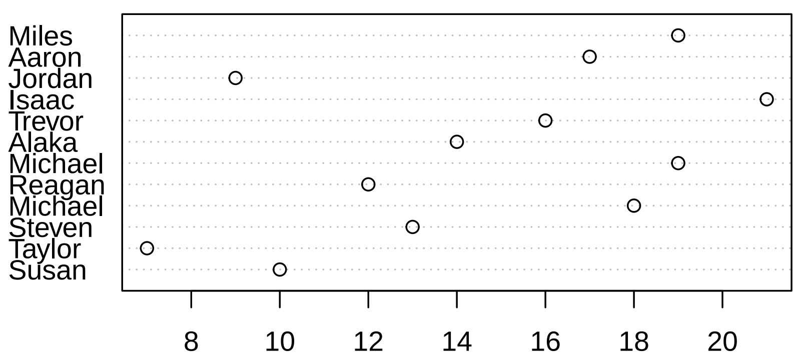


17.2.7 Dot Chart

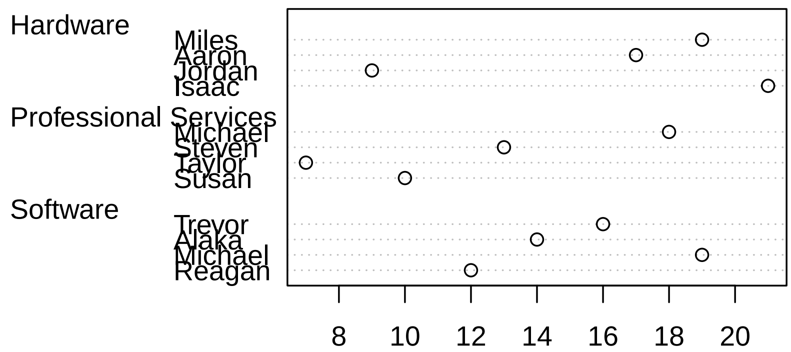
```
salesperson <- c("Susan", "Taylor", "Steven"
                    , "Michael", "Reagan", "Michael"
                    , "Alaka", "Trevor", "Isaac"
                    , "Jordan", "Aaron", "Miles")
product <- c("Professional Services", "Professional Services"
                , "Professional Services", "Professional Services"
                , "Software",  "Software",  "Software",  "Software"
                , "Hardware", "Hardware", "Hardware", "Hardware")
sales <- c(10, 7, 13, 18, 12, 19, 14, 16, 21, 9, 17, 19)
df <- data.frame(salesperson = salesperson, product = product, sales = sales)

dotchart(df$sales)
```



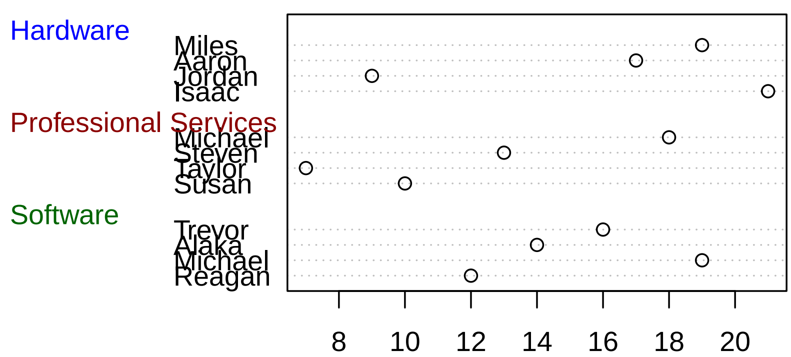

```
dotchart(df$sales, labels = df$salesperson)
```



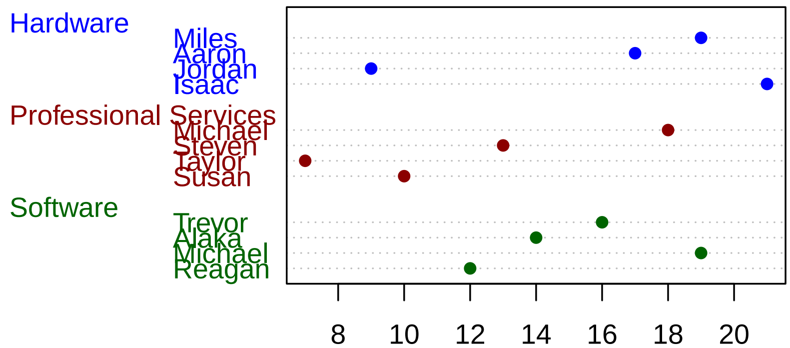

```
groups <- as.factor(df$product)
dotchart(df$sales, labels = df$salesperson, groups = groups)
```




```
group_colors <-  c("blue", "darkred", "darkgreen")
dotchart(df$sales
            , labels = df$salesperson
            , groups = groups
            , gcolor = group_colors)
```



```
dotchart(df$sales
             , labels = df$salesperson
             , groups = groups
             , gcolor = group_colors
             , color = group_colors[groups]
             , pch = 16)
```



17.3 Resources

- ggplot2 documentation: https://ggplot2.tidyverse.org/
- ggplot2 cheat sheet: https://github.com/rstudio/cheatsheets/blob/main/data-visualization.pdf
- ggplot2 extension gallery: https://exts.ggplot2.tidyverse.org/gallery/
- R colors: http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf

Exercises

Questions

Exercise: 15-A

Use the "summary" function to get summary statistics for all columns in the "mtcars" dataset.
Your final output should resemble the following:

```
#       mpg             cyl             disp             hp
#  Min.   :10.40   Min.   :4.000   Min.   : 71.1   Min.   : 52.0
#  1st Qu.:15.43   1st Qu.:4.000   1st Qu.:120.8   1st Qu.: 96.5
#  Median :19.20   Median :6.000   Median :196.3   Median :123.0
#  Mean   :20.09   Mean   :6.188   Mean   :230.7   Mean   :146.7
#  3rd Qu.:22.80   3rd Qu.:8.000   3rd Qu.:326.0   3rd Qu.:180.0
#  Max.   :33.90   Max.   :8.000   Max.   :472.0   Max.   :335.0
#       drat             wt             qsec             vs
#  Min.   :2.760   Min.   :1.513   Min.   :14.50   Min.   :0.0000
#  1st Qu.:3.080   1st Qu.:2.581   1st Qu.:16.89   1st Qu.:0.0000
#  Median :3.695   Median :3.325   Median :17.71   Median :0.0000
#  Mean   :3.597   Mean   :3.217   Mean   :17.85   Mean   :0.4375
#  3rd Qu.:3.920   3rd Qu.:3.610   3rd Qu.:18.90   3rd Qu.:1.0000
#  Max.   :4.930   Max.   :5.424   Max.   :22.90   Max.   :1.0000
#        am              gear             carb
#  Min.   :0.0000   Min.   :3.000   Min.   :1.000
#  1st Qu.:0.0000   1st Qu.:3.000   1st Qu.:2.000
#  Median :0.0000   Median :4.000   Median :2.000
#  Mean   :0.4062   Mean   :3.688   Mean   :2.812
#  3rd Qu.:1.0000   3rd Qu.:4.000   3rd Qu.:4.000
#  Max.   :1.0000   Max.   :5.000   Max.   :8.000
```

Exercise: 16-A

Use the "lm" function to create a linear model using the "ChickWeight" dataset. Your model should predict the "weight" variable using the "Diet" and "Time" variables.
Name your linear model "lm" and then view a summary of your model using the "summary" function. The output of your summary should look like this:

```
# Call:
# lm(formula = weight ~ Diet + Time, data = ChickWeight)

# Residuals:
#      Min       1Q   Median       3Q      Max
# -136.851  -17.151   -2.595   15.033  141.816

# Coefficients:
#             Estimate Std. Error t value Pr(>|t|)
# (Intercept)  10.9244     3.3607   3.251  0.00122 **
# Diet2        16.1661     4.0858   3.957 8.56e-05 ***
# Diet3        36.4994     4.0858   8.933  < 2e-16 ***
# Diet4        30.2335     4.1075   7.361 6.39e-13 ***
# Time          8.7505     0.2218  39.451  < 2e-16 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

# Residual standard error: 35.99 on 573 degrees of freedom
# Multiple R-squared:  0.7453,  Adjusted R-squared:  0.7435
# F-statistic: 419.2 on 4 and 573 DF,  p-value: < 2.2e-16
```

Exercise: 17-A

Create a density plot using the "Nile" dataset.

Answers

Answer: 15-A

Here's how you can accomplish this task:

```
summary(mtcars)
```

```
      mpg             cyl             disp             hp
 Min.   :10.40   Min.   :4.000   Min.   : 71.1   Min.   : 52.0
 1st Qu.:15.43   1st Qu.:4.000   1st Qu.:120.8   1st Qu.: 96.5
 Median :19.20   Median :6.000   Median :196.3   Median :123.0
 Mean   :20.09   Mean   :6.188   Mean   :230.7   Mean   :146.7
 3rd Qu.:22.80   3rd Qu.:8.000   3rd Qu.:326.0   3rd Qu.:180.0
 Max.   :33.90   Max.   :8.000   Max.   :472.0   Max.   :335.0
      drat             wt             qsec             vs
 Min.   :2.760   Min.   :1.513   Min.   :14.50   Min.   :0.0000
 1st Qu.:3.080   1st Qu.:2.581   1st Qu.:16.89   1st Qu.:0.0000
 Median :3.695   Median :3.325   Median :17.71   Median :0.0000
 Mean   :3.597   Mean   :3.217   Mean   :17.85   Mean   :0.4375
 3rd Qu.:3.920   3rd Qu.:3.610   3rd Qu.:18.90   3rd Qu.:1.0000
 Max.   :4.930   Max.   :5.424   Max.   :22.90   Max.   :1.0000
       am              gear            carb
 Min.   :0.0000   Min.   :3.000   Min.   :1.000
 1st Qu.:0.0000   1st Qu.:3.000   1st Qu.:2.000
 Median :0.0000   Median :4.000   Median :2.000
 Mean   :0.4062   Mean   :3.688   Mean   :2.812
 3rd Qu.:1.0000   3rd Qu.:4.000   3rd Qu.:4.000
 Max.   :1.0000   Max.   :5.000   Max.   :8.000
```

Answer: 16-A

You can create your model with the following code:

```r
lm <- lm(weight ~ Diet + Time, data = ChickWeight)
summary(lm)
```

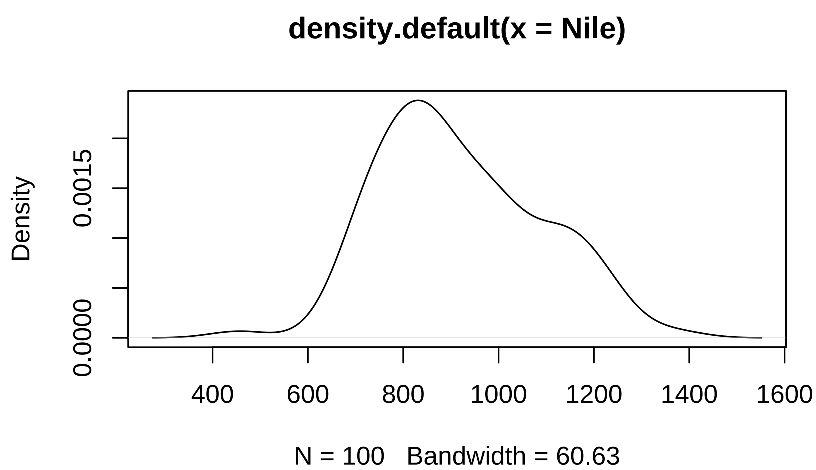
```
Call:
lm(formula = weight ~ Diet + Time, data = ChickWeight)

Residuals:
     Min       1Q   Median       3Q      Max
-136.851  -17.151   -2.595   15.033  141.816

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  10.9244     3.3607   3.251  0.00122 **
Diet2        16.1661     4.0858   3.957 8.56e-05 ***
Diet3        36.4994     4.0858   8.933  < 2e-16 ***
Diet4        30.2335     4.1075   7.361 6.39e-13 ***
Time          8.7505     0.2218  39.451  < 2e-16 ***
---
```

```
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 35.99 on 573 degrees of freedom
Multiple R-squared:  0.7453,    Adjusted R-squared:  0.7435
F-statistic: 419.2 on 4 and 573 DF,  p-value: < 2.2e-16
```

Answer: 17-A

You can create your density plot with the following code:

```
plot(density(Nile))
```



Part VI

Part V: Reporting

> "It feels like we're all suffering from information overload or data glut. And the good news is there might be an easy solution to that, and that's using our eyes more. So, visualizing information, so that we can see the patterns and connections that matter and then designing that information so it makes more sense, or it tells a story, or allows us to focus only on the information that's important. Failing that, visualized information can just look really cool." -David McCandless (McCandless 2010)

Finally, it's important to report on your data to make it easy for others to extract and understand the information that is most relevant.

- **Spreadsheets-** Spreadsheets are a common way to communicate information to stakeholders. This chapter will go over how to export .xlsx and .csv files from R, how to format those spreadsheets, and how to add formulas to them.
- **R Markdown-** R Markdown allows you to create documents in a programmatic fashion that improves reproducibility. This chapter will cover some of the different formats that are available in R as well as how to create them.
- **R Shiny-** R Shiny is a tool used to develop web applications and is commonly deployed in the use of creating dashboards, hosting static reports, and custom tooling.

Chapter 18

Spreadsheets

Spreadsheets are a common way to communicate information to stakeholders. This chapter will go over how to export .xlsx and .csv files from R, how to format those spreadsheets, and how to add formulas to them.

18.1 Export

18.1.1 Export .csv Files

In order to export a dataframe to a CSV file, you can use the "write.csv" function. This function will accept a dataframe followed by the desired output location of your file. Let's start by creating a sample dataframe to work with.

```
people <- c("John", "Jane", NA)
id <- c(12, 27, 23)
df <- data.frame(id = id, person = people)
```

id	person
12	John
27	Jane
23	NA

Now, let's specify the location we want to store the CSV file at and execute the "write.csv" function. (We use the `file.path()` to specify a path to the `example.csv` file in a temporary directory that will automatically be erased when your R session ends.)

```
output <- file.path(tempdir(), "example.csv")
write.csv(df, output)
```

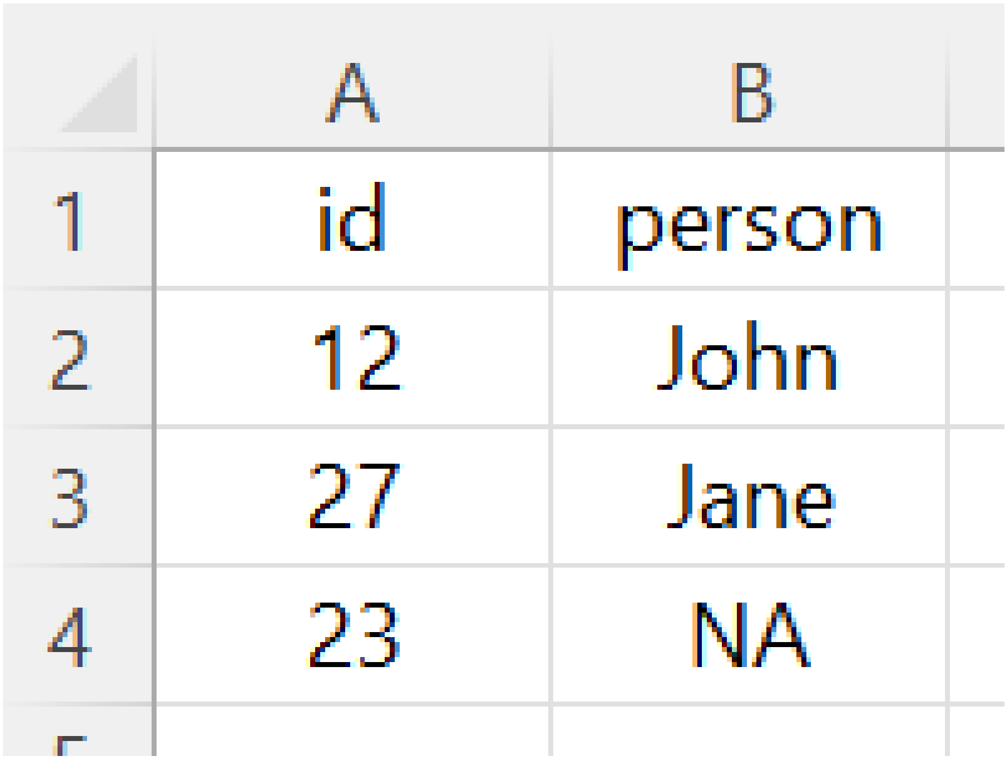
This will give you a file that looks like the following image.

	A	B	C
1		id	person
2	1	12	John
3	2	27	Jane
4	3	23	NA

You'll notice that the first column contains the row numbers of the dataframe. This can be remedied by setting "row.names" to "FALSE" as follows.

```
write.csv(df, output, row.names = FALSE)
```

This will yield the following result.

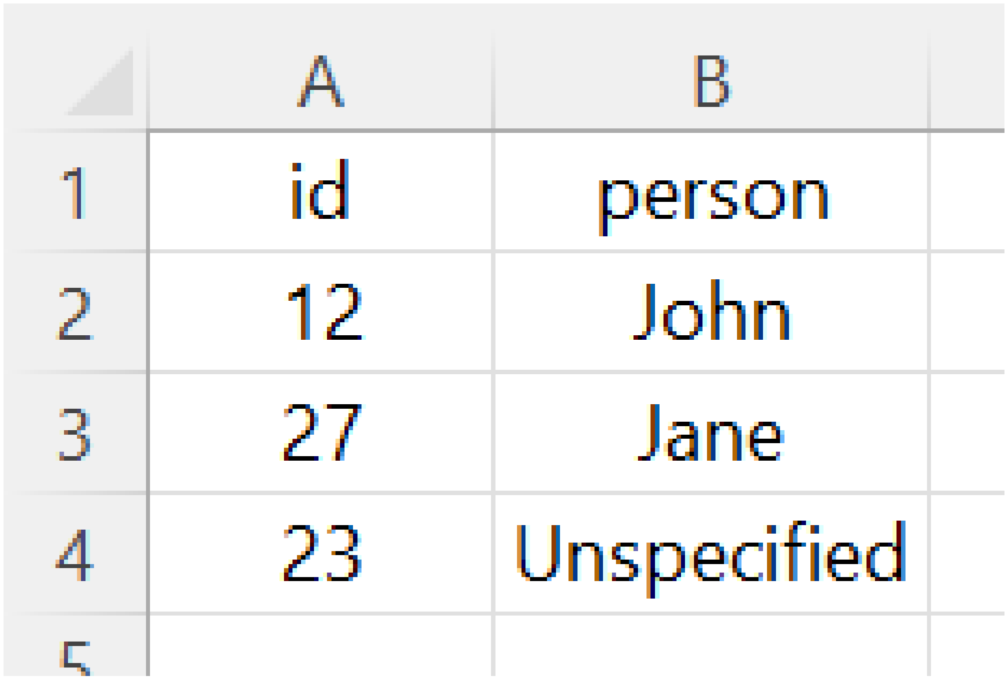


	A	B
1	id	person
2	12	John
3	27	Jane
4	23	NA

Finally, you'll notice that one of the names is an "NA" value. You can tell R how to handle these values at the time of exporting your file with the "na" argument. This argument will replace any "NA" values with the value of your choice. Let's try replacing the "NA" value with "Unspecified".

```
write.csv(df, output, row.names = FALSE, na = "Unspecified")
```

This results in the following output:



	A	B
1	id	person
2	12	John
3	27	Jane
4	23	Unspecified
5		

18.1.2 Export .xlsx Files

Excel files are handled very similarly to CSV files except that you will need to use the "write_excel" function from the "writexl" package. The following code snippet demonstrates how to export your data to an Excel file.

```
library(writexl)
output <- "C:/File Location/example.xlsx"
write_xlsx(df, output)
```

18.2 Formatting

When saving Excel workbooks, you can also leverage the "openxlsx" library to format and add formulas to your workbook. Let's use the iris dataset to demonstrate these capabilities.

```
library(openxlsx)
```

Next, let's break down the iris dataset into three separate datasets based on species.

```
setosa <- iris[which(iris$"Species" == "setosa"),]
versicolor <- iris[which(iris$"Species" == "versicolor"),]
virginica <- iris[which(iris$"Species" == "virginica"),]
```

Now, we'll use the "createWorkbook" function from the "openxlsx" library to create a blank workbook object.

```
wb <- createWorkbook()
```

We'll now add three worksheets to our workbook. These worksheets will ultimately be tabs in our Excel workbook.

```
addWorksheet(wb, "Setosa")
addWorksheet(wb, "Versicolor")
addWorksheet(wb, "Virginica")
```

We can also create styles to apply to our workbook. Let's create a style for our headers as well as a style for the body of our data.

```
heading <- createStyle(fontName = "Segoe UI"
                        , fontSize = 12
                        , fontColour = "#FFFFFF"
                        , bgFill = "#244062"
                        , textDecoration = "bold")

body <- createStyle(fontName = "Segoe UI", fontSize = 12)
```

Let's now apply our three datasets to the workbook object we previously created.

```
# Write the setosa dataset to the "Setosa" worksheet
writeData(wb
            , "Setosa"
            , setosa
            , startCol = 1
            , startRow = 1
            , rowNames = FALSE)

# Write the versicolor dataset to the "Versicolor" worksheet
writeData(wb
            , "Versicolor"
            , versicolor
            , startCol = 1
            , startRow = 1
            , rowNames = FALSE)
```

```r
# Write the virginica dataset to the "Virginica" worksheet
writeData(wb
            , "Virginica"
            , virginica
            , startCol = 1
            , startRow = 1
            , rowNames = FALSE)
```

Now let's apply our styles to each worksheet.

```r
# Apply styles to "Setosa" worksheet
addStyle(wb
          , "Setosa"
          , cols = 1:length(setosa)
          , rows = 1
          , style = heading
          , gridExpand = TRUE)

addStyle(wb
          , "Setosa"
          , cols = 1:length(setosa)
          , rows = 2:nrow(setosa)
          , style = body
          , gridExpand = TRUE)

# Apply styles to "Versicolor" worksheet
addStyle(wb
          , "Versicolor"
          , cols = 1:length(versicolor)
          , rows = 1
          , style = heading
          , gridExpand = TRUE)

addStyle(wb
          , "Versicolor"
          , cols = 1:length(versicolor)
          , rows = 2:nrow(versicolor)
          , style = body
          , gridExpand = TRUE)

# Apply styles to "Virginica" worksheet
addStyle(wb
          , "Virginica"
          , cols = 1:length(virginica)
```

```
          , rows = 1
          , style = heading
          , gridExpand = TRUE)

addStyle(wb
          , "Virginica"
          , cols = 1:length(virginica)
          , rows = 2:nrow(virginica)
          , style = body
          , gridExpand = TRUE)
```

Finally, we will save our workbook to a file named “iris.xlsx”.

```
saveWorkbook(wb, "iris.xlsx", overwrite = TRUE)
```

This will result in a workbook that looks like the following image.

	A	B	C	D	E
1	**Sepal.Length**	**Sepal.Width**	**Petal.Length**	**Petal.Width**	**Species**
2	5.1	3.5	1.4	0.2	setosa
3	4.9	3	1.4	0.2	setosa
4	4.7	3.2	1.3	0.2	setosa
5	4.6	3.1	1.5	0.2	setosa
6	5	3.6	1.4	0.2	setosa
7	5.4	3.9	1.7	0.4	setosa
8	4.6	3.4	1.4	0.3	setosa
9	5	3.4	1.5	0.2	setosa
10	4.4	2.9	1.4	0.2	setosa
11	4.9	3.1	1.5	0.1	setosa
12	5.4	3.7	1.5	0.2	setosa
13	4.8	3.4	1.6	0.2	setosa
14	4.8	3	1.4	0.1	setosa
15	4.3	3	1.1	0.1	setosa
16	5.8	4	1.2	0.2	setosa
17	5.7	4.4	1.5	0.4	setosa
18	5.4	3.9	1.3	0.4	setosa
19	5.1	3.5	1.4	0.3	setosa
20	5.7	3.8	1.7	0.3	setosa
21	5.1	3.8	1.5	0.3	setosa
22	5.4	3.4	1.7	0.2	setosa
23	5.1	3.7	1.5	0.4	setosa
24	4.6	3.6	1	0.2	setosa
25	5.1	3.3	1.7	0.5	setosa

Setosa | Versicolor | Virginica

The full script is shown below.

```
library(openxlsx)

# Create datasets
setosa <- iris[which(iris$"Species" == "setosa"),]
versicolor <- iris[which(iris$"Species" == "versicolor"),]
virginica <- iris[which(iris$"Species" == "virginica"),]

# Create workbook object
wb <- createWorkbook()

#Add worksheets
addWorksheet(wb, "Setosa")
addWorksheet(wb, "Versicolor")
addWorksheet(wb, "Virginica")

# Create Styles
heading <- createStyle(fontName = "Segoe UI"
                       , fontSize = 12
                       , fontColour = "#FFFFFF"
                       , bgFill = "#244062"
                       , textDecoration = "bold")

body <- createStyle(fontName = "Segoe UI", fontSize = 12)

# Write the setosa dataset to the "Setosa" worksheet
writeData(wb
            , "Setosa"
            , setosa
            , startCol = 1
            , startRow = 1
            , rowNames = FALSE)

# Write the versicolor dataset to the "Versicolor" worksheet
writeData(wb
            , "Versicolor"
            , versicolor
            , startCol = 1
            , startRow = 1
            , rowNames = FALSE)

# Write the virginica dataset to the "Virginica" worksheet
writeData(wb
            , "Virginica"
            , virginica
```

```
             , startCol = 1
             , startRow = 1
             , rowNames = FALSE)

# Apply styles to "Setosa" worksheet
addStyle(wb
           , "Setosa"
           , cols = 1:length(setosa)
           , rows = 1
           , style = heading
           , gridExpand = TRUE)

addStyle(wb
           , "Setosa"
           , cols = 1:length(setosa)
           , rows = 2:nrow(setosa)
           , style = body
           , gridExpand = TRUE)

# Apply styles to "Versicolor" worksheet
addStyle(wb
           , "Versicolor"
           , cols = 1:length(versicolor)
           , rows = 1
           , style = heading
           , gridExpand = TRUE)

addStyle(wb
           , "Versicolor"
           , cols = 1:length(versicolor)
           , rows = 2:nrow(versicolor)
           , style = body
           , gridExpand = TRUE)

# Apply styles to "Virginica" worksheet
addStyle(wb
           , "Virginica"
           , cols = 1:length(virginica)
           , rows = 1
           , style = heading
           , gridExpand = TRUE)

addStyle(wb
           , "Virginica"
```

```
          , cols = 1:length(virginica)
          , rows = 2:nrow(virginica)
          , style = body
          , gridExpand = TRUE)

saveWorkbook(wb, "iris.xlsx", overwrite = TRUE)
```

You may notice that this script is a little longer than it needs to be. Let's try to simplify it with a loop.

The following script will accomplish the exact same thing as the first script.

```
library(openxlsx)

setosa <- iris[which(iris$"Species" == "setosa"),]
versicolor <- iris[which(iris$"Species" == "versicolor"),]
virginica <- iris[which(iris$"Species" == "virginica"),]

wb <- createWorkbook()

heading <- createStyle(fontName = "Segoe UI"
                       , fontSize = 12
                       , fontColour = "#FFFFFF"
                       , bgFill = "#244062"
                       , textDecoration = "bold")

body <- createStyle(fontName = "Segoe UI", fontSize = 12)

datasets <- list(setosa, virginica, versicolor)
worksheets <- c("Setosa", "Virginica", "Versicolor")

for (i in 1:3) {
  df <- as.data.frame(datasets[i])
  addWorksheet(wb, worksheets[i])
  writeData(wb
            , worksheets[i]
            , df
            , startCol = 1
            , startRow = 1
            , rowNames = FALSE)
  addStyle(wb
           , worksheets[i]
           , cols = 1:length(df)
           , rows = 1
           , style = heading
```

```
             , gridExpand = TRUE)
  addStyle(wb
             , worksheets[i]
             , cols = 1:length(df)
             , rows = 2:nrow(df)
             , style = body
             , gridExpand = TRUE)
}

saveWorkbook(wb, "iris.xlsx", overwrite = TRUE)
```

18.3 Formulas

If we wanted to add another column to each of our worksheets that used an Excel formula to determine the ratio between the sepal length and the sepal width, we could use the "writeFormula" function to accomplish that.

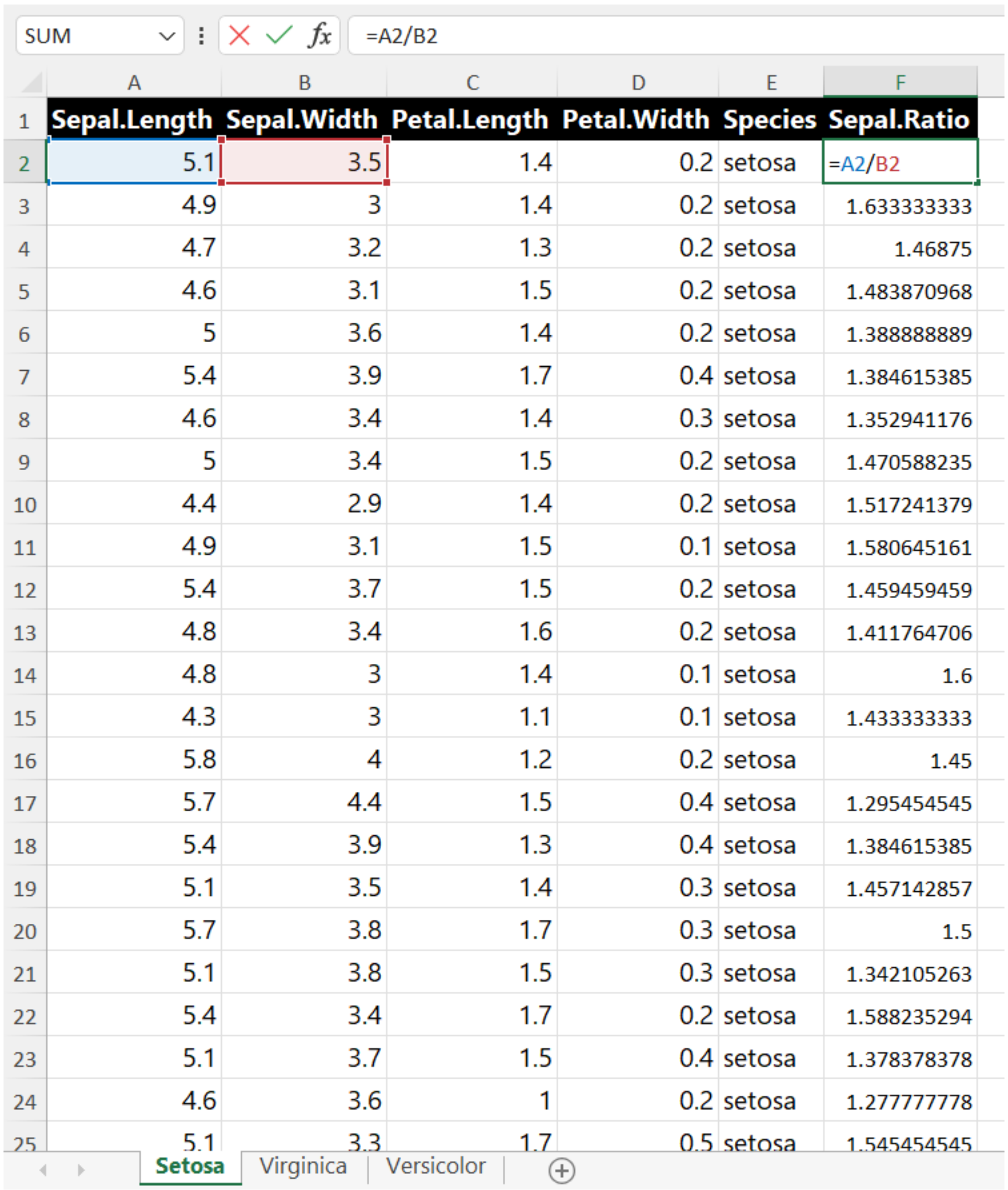
The following example uses a loop that creates a formula for each row which divides the respective value in column A by the the respective value in column B. Next we add the heading style to the first row in column six and add a header named "Sepal.Ratio". Finally, we write the formula vector to column six beginning on row 2.

```
library(openxlsx)

setosa <- iris[which(iris$"Species" == "setosa"),]
versicolor <- iris[which(iris$"Species" == "versicolor"),]
virginica <- iris[which(iris$"Species" == "virginica"),]

wb <- createWorkbook()

heading <- createStyle(fontName = "Segoe UI"
                        , fontSize = 12
                        , fontColour = "#FFFFFF"
                        , bgFill = "#244062"
                        , textDecoration = "bold")

body <- createStyle(fontName = "Segoe UI", fontSize = 12)

datasets <- list(setosa, virginica, versicolor)
worksheets <- c("Setosa", "Virginica", "Versicolor")

for (i in 1:3) {
  df <- as.data.frame(datasets[i])
```

```
addWorksheet(wb, worksheets[i])
writeData(wb
          , worksheets[i]
          , df
          , startCol = 1
          , startRow = 1
          , rowNames = FALSE)
addStyle(wb
          , worksheets[i]
          , cols = 1:length(df)
          , rows = 1
          , style = heading
          , gridExpand = TRUE)
addStyle(wb
          , worksheets[i]
          , cols = 1:length(df)
          , rows = 2:nrow(df)
          , style = body
          , gridExpand = TRUE)

formula <- c()

for (x in 2:(nrow(df) + 1)) {
  formula <- append(formula, paste("A", x, "/B", x, sep = ''))
}

addStyle(wb
          , worksheets[i]
          , cols = 6
          , rows = 1
          , style = heading
          , gridExpand = TRUE)
writeData(wb
            , worksheets[i]
            , "Sepal.Ratio"
            , startCol = 6
            , startRow = 1
            , rowNames = FALSE)
writeFormula(wb
              , worksheets[i]
              , formula
              , startCol = 6
              , startRow = 2)
```

```
}

saveWorkbook(wb, "iris.xlsx", overwrite = TRUE)
```

This gives us an Excel workbook that looks like the following image.



	A	B	C	D	E	F
1	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species	Sepal.Ratio
2	5.1	3.5	1.4	0.2	setosa	=A2/B2
3	4.9	3	1.4	0.2	setosa	1.633333333
4	4.7	3.2	1.3	0.2	setosa	1.46875
5	4.6	3.1	1.5	0.2	setosa	1.483870968
6	5	3.6	1.4	0.2	setosa	1.388888889
7	5.4	3.9	1.7	0.4	setosa	1.384615385
8	4.6	3.4	1.4	0.3	setosa	1.352941176
9	5	3.4	1.5	0.2	setosa	1.470588235
10	4.4	2.9	1.4	0.2	setosa	1.517241379
11	4.9	3.1	1.5	0.1	setosa	1.580645161
12	5.4	3.7	1.5	0.2	setosa	1.459459459
13	4.8	3.4	1.6	0.2	setosa	1.411764706
14	4.8	3	1.4	0.1	setosa	1.6
15	4.3	3	1.1	0.1	setosa	1.433333333
16	5.8	4	1.2	0.2	setosa	1.45
17	5.7	4.4	1.5	0.4	setosa	1.295454545
18	5.4	3.9	1.3	0.4	setosa	1.384615385
19	5.1	3.5	1.4	0.3	setosa	1.457142857
20	5.7	3.8	1.7	0.3	setosa	1.5
21	5.1	3.8	1.5	0.3	setosa	1.342105263
22	5.4	3.4	1.7	0.2	setosa	1.588235294
23	5.1	3.7	1.5	0.4	setosa	1.378378378
24	4.6	3.6	1	0.2	setosa	1.277777778
25	5.1	3.3	1.7	0.5	setosa	1.545454545

18.4 Resources

- openxlsx documentation: https://cran.r-project.org/web/packages/openxlsx/openxlsx.pdf

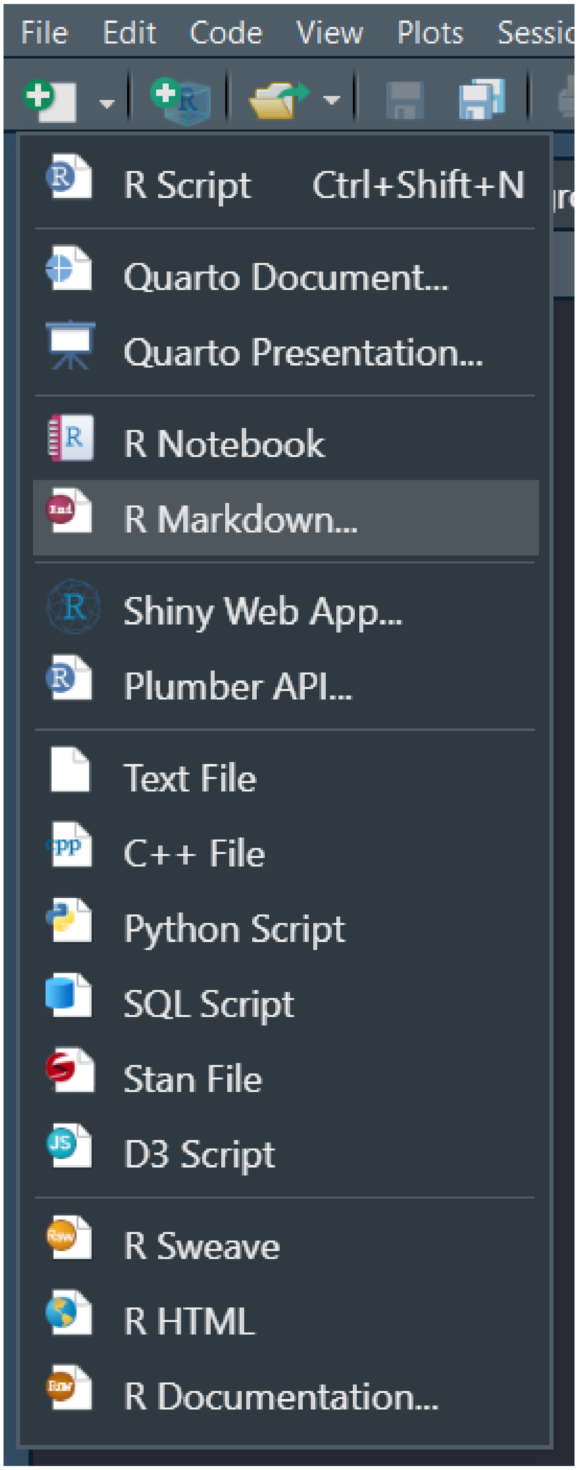
Chapter 19

R Markdown

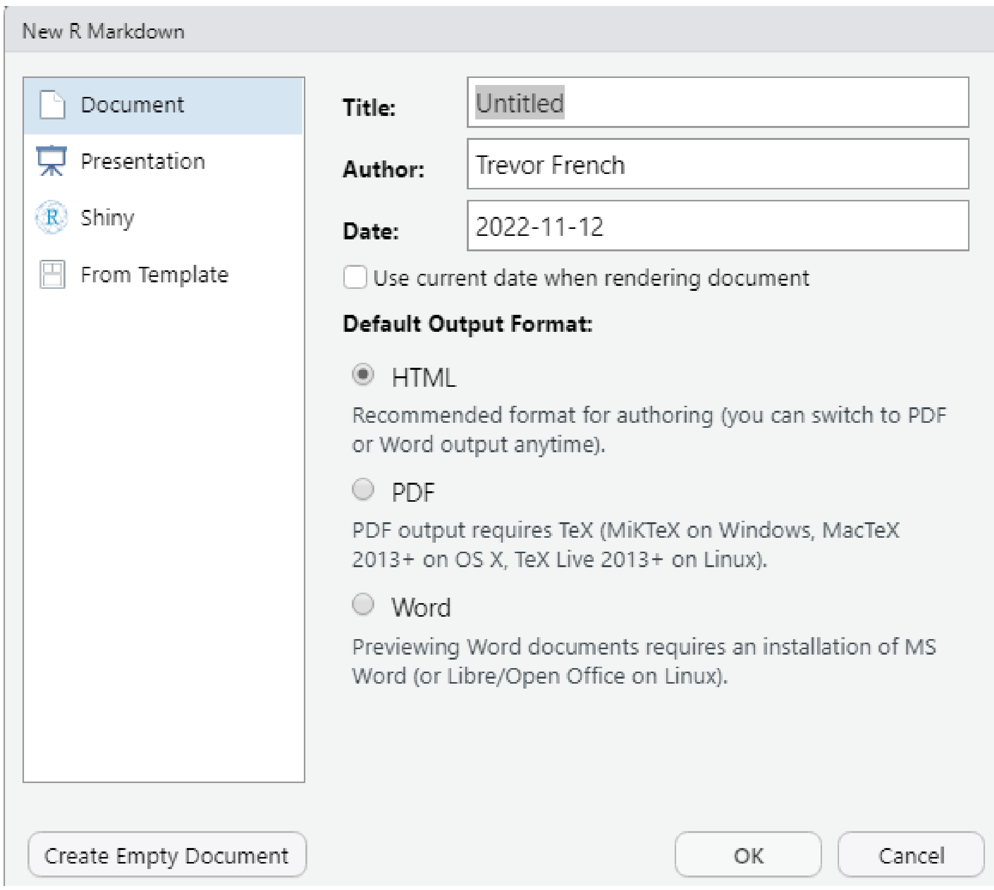
R Markdown allows you to create documents in a programmatic fashion that lends itself towards reproducibility. This chapter will cover the different formats that are available in R as well as how to create them.

19.1 Format Options

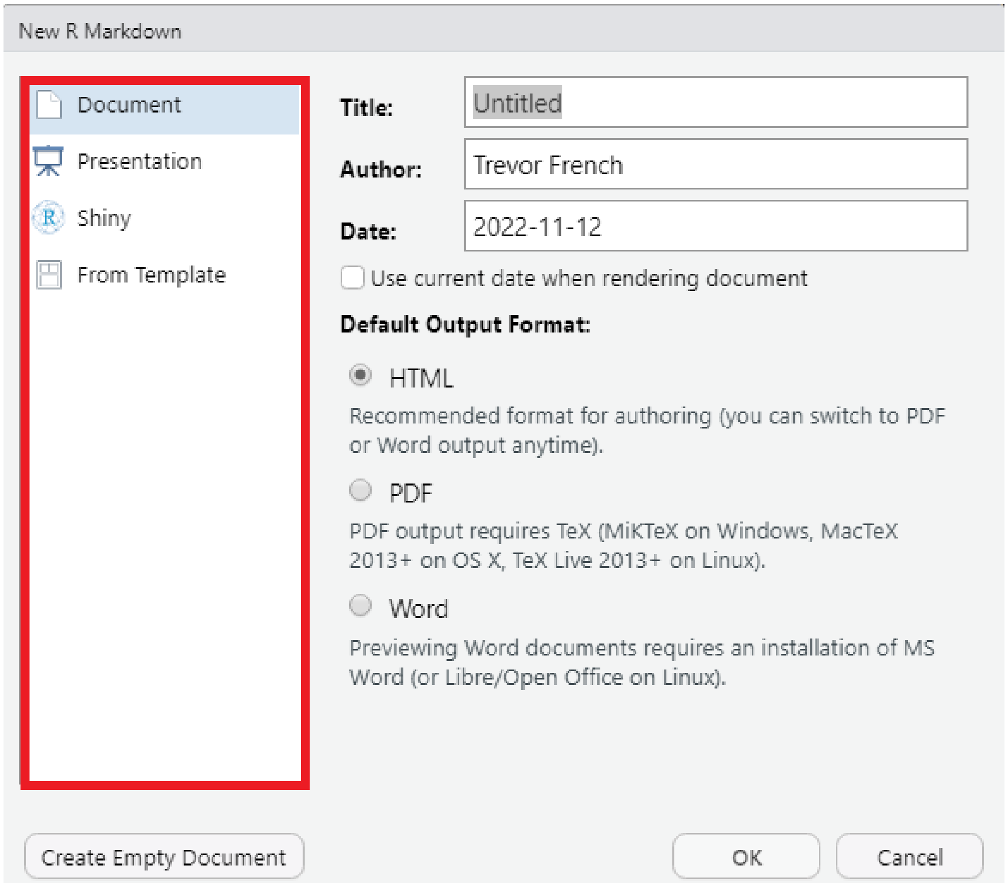
We'll begin by creating a new document by selecting the "New File" button towards the top left corner of R Studio and choosing "R Markdown" from the dropdown menu.


File
Edit
Code
View
Plots
R Script
Ctrl+Shift+N
Quarto Document...
Quarto Presentation...
R Notebook
R Markdown...
Shiny Web App...
Plumber API...
Text File
C++ File
Python Script
SQL Script
Stan File
D3 Script
R Sweave
R HTML
R Documentation...

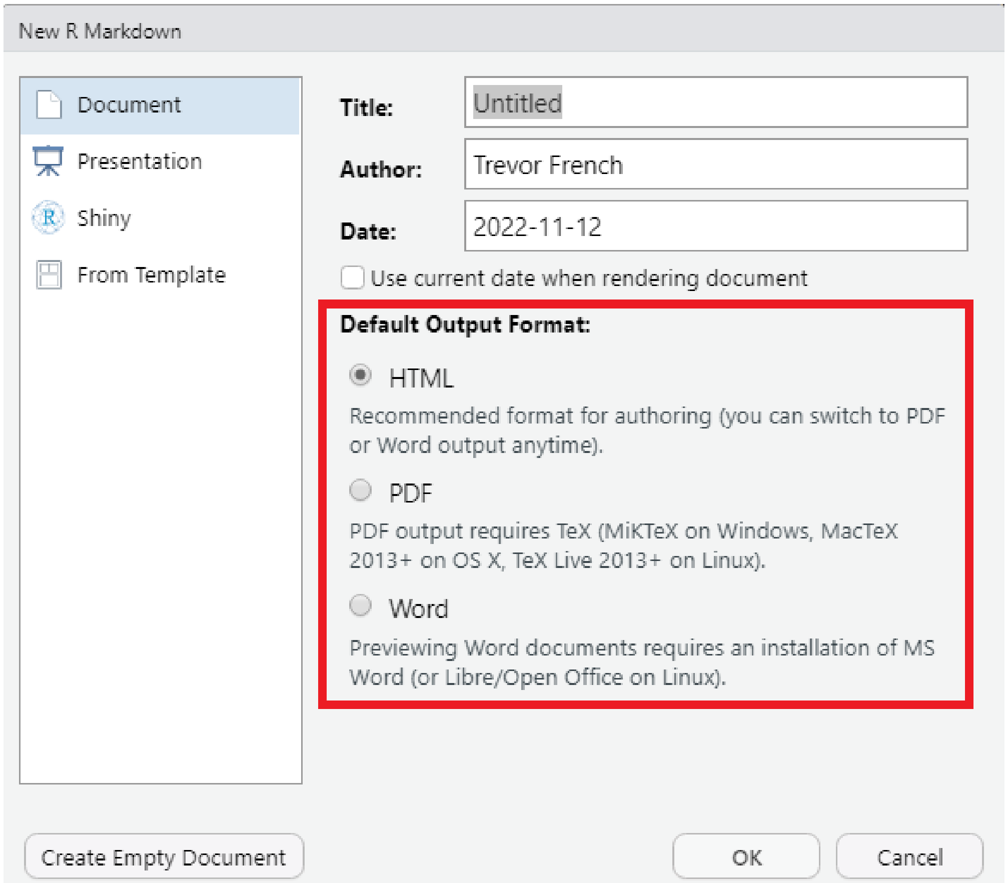
This will display a menu that looks like the following image.



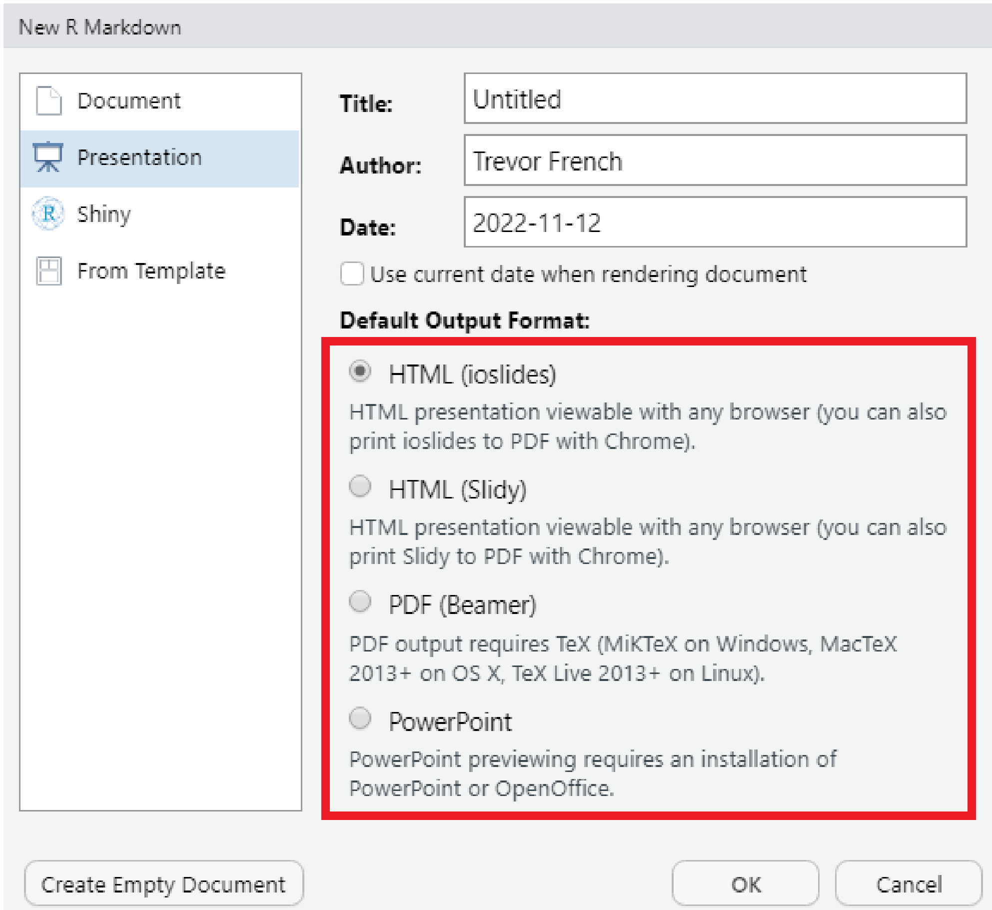
You'll notice that you have four main options on the left-hand side: "Document", "Presentation", "Shiny", and "From Template".



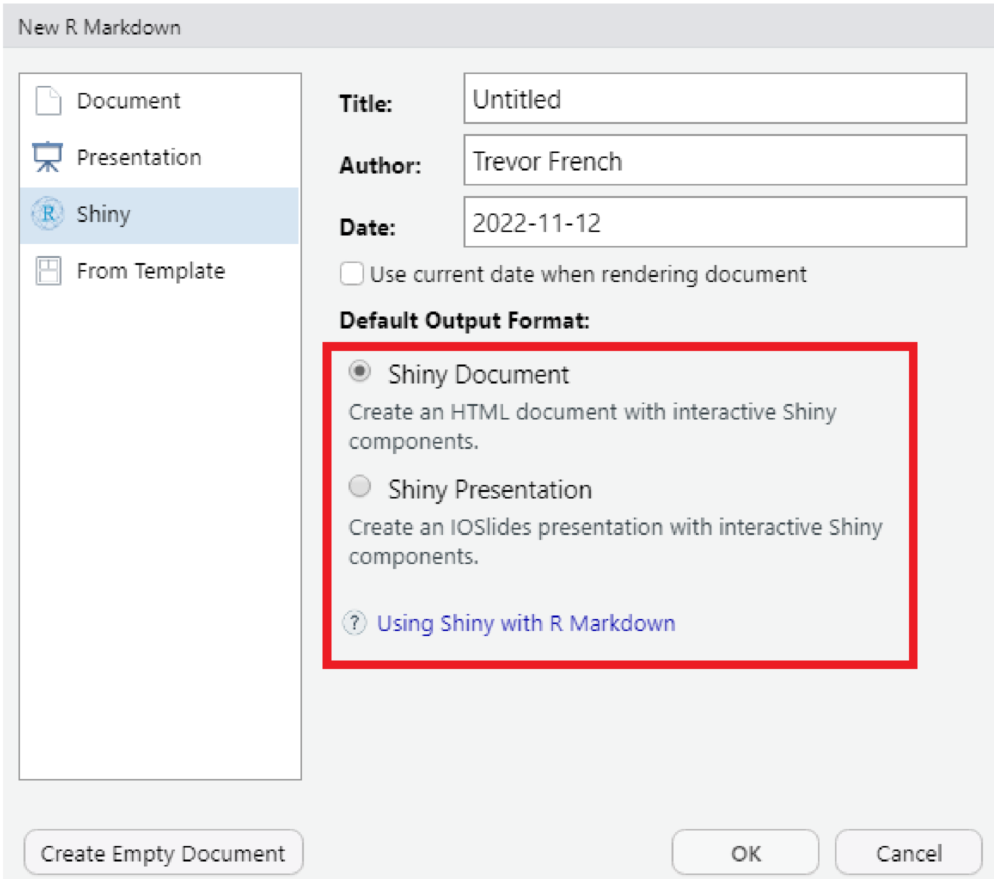
Each of these options will have several sub-options. The "Document" option, for example is selected by default and you can see there are three sub-options on the right-hand side: "HTML", "PDF", and "Word".



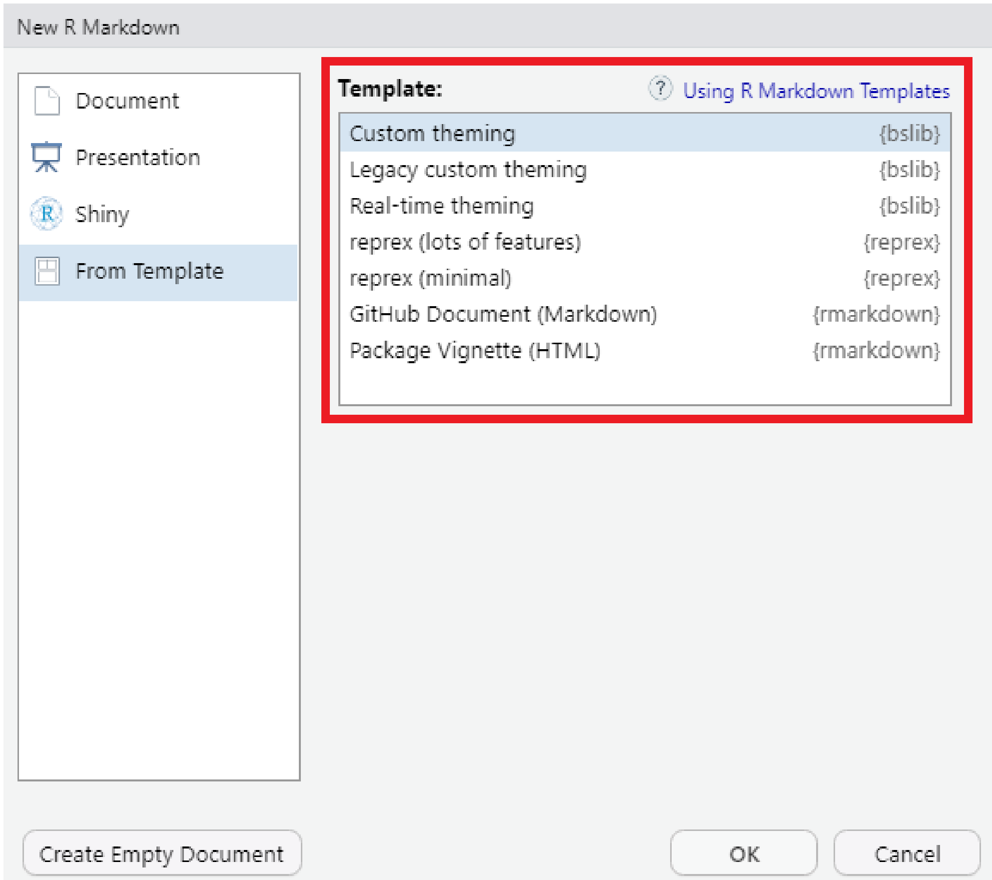
The "Presentation" option allows you to create slide-based presentations in either HTML, PDF, or PowerPoint format.



The "Shiny" option allows you to create either presentations or documents which include interactive Shiny components.

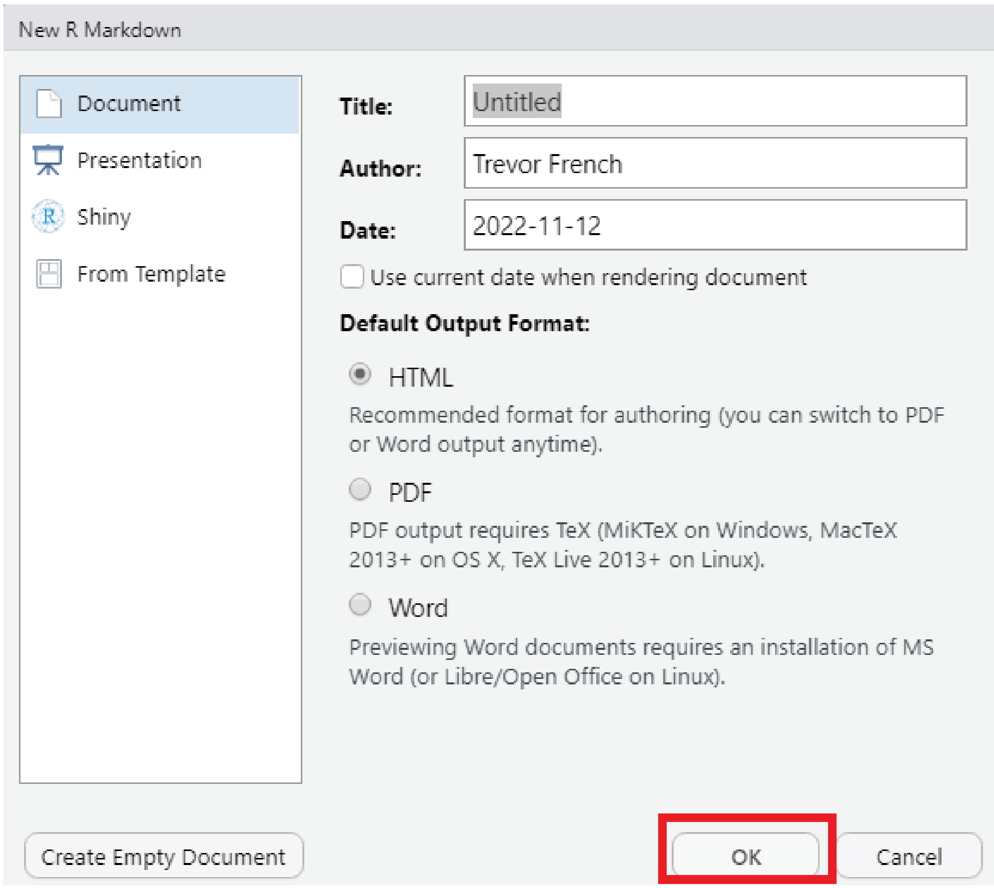


Finally, the "From Template" option will display several options for you to leverage pre-made templates.

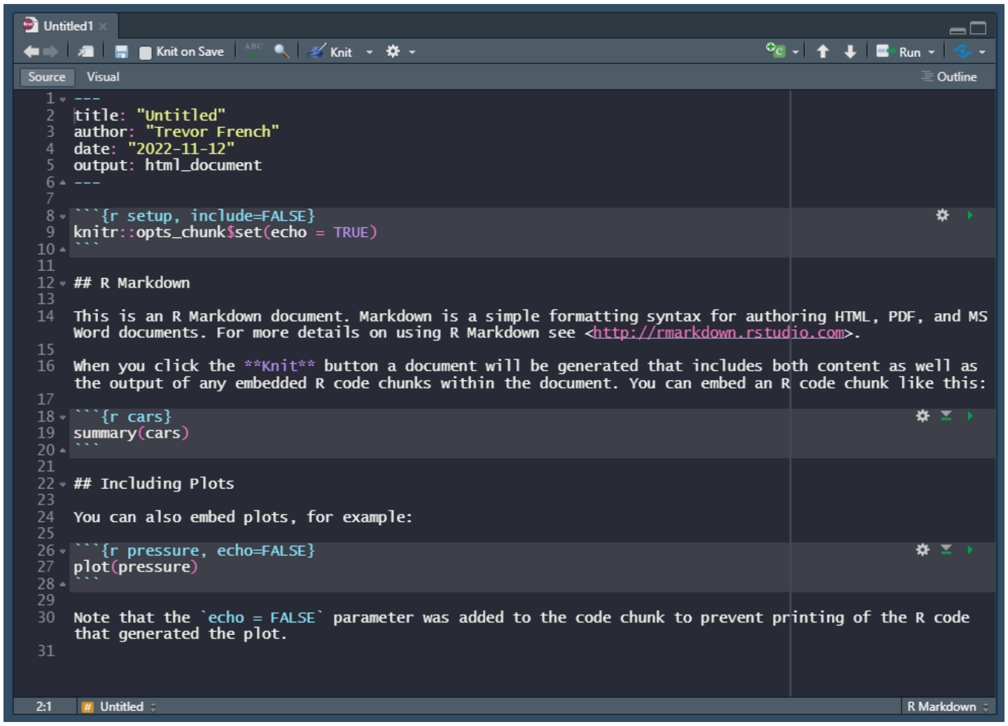


19.2 HTML Document Example

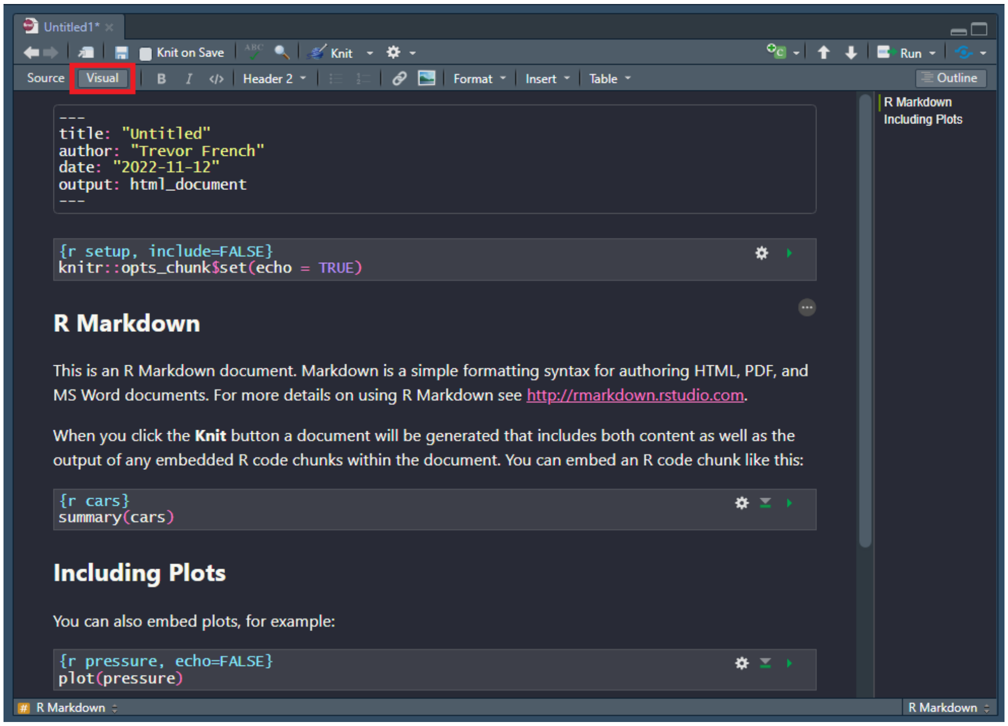
Let's choose the HTML sub-option from the Document option and select "OK".



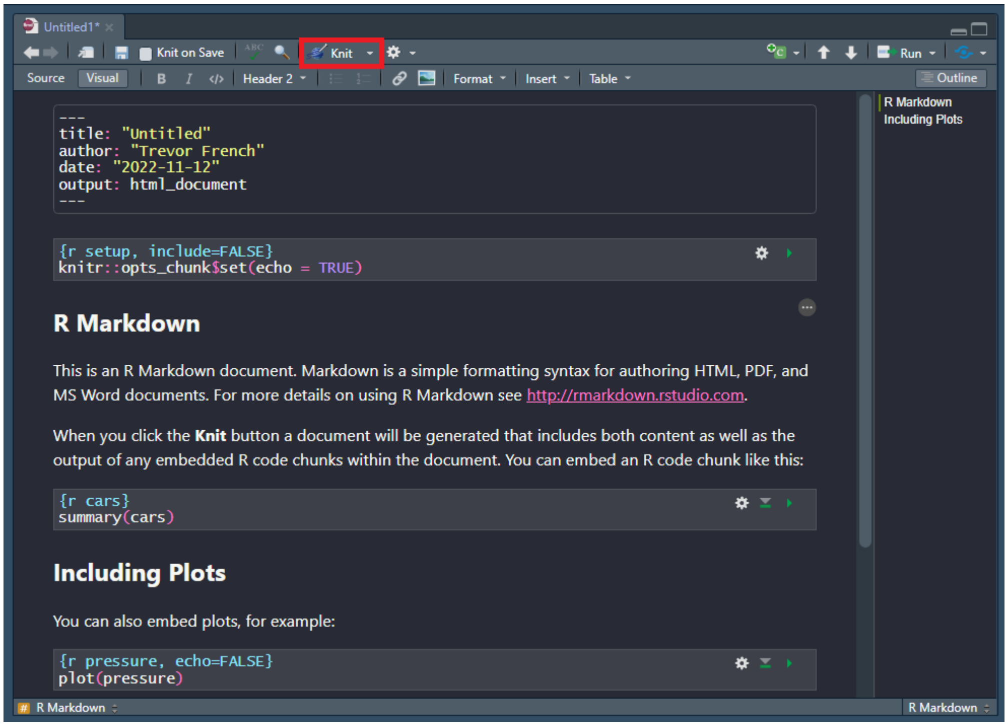
This will result in a new file in your source pane that looks similar to the following image.



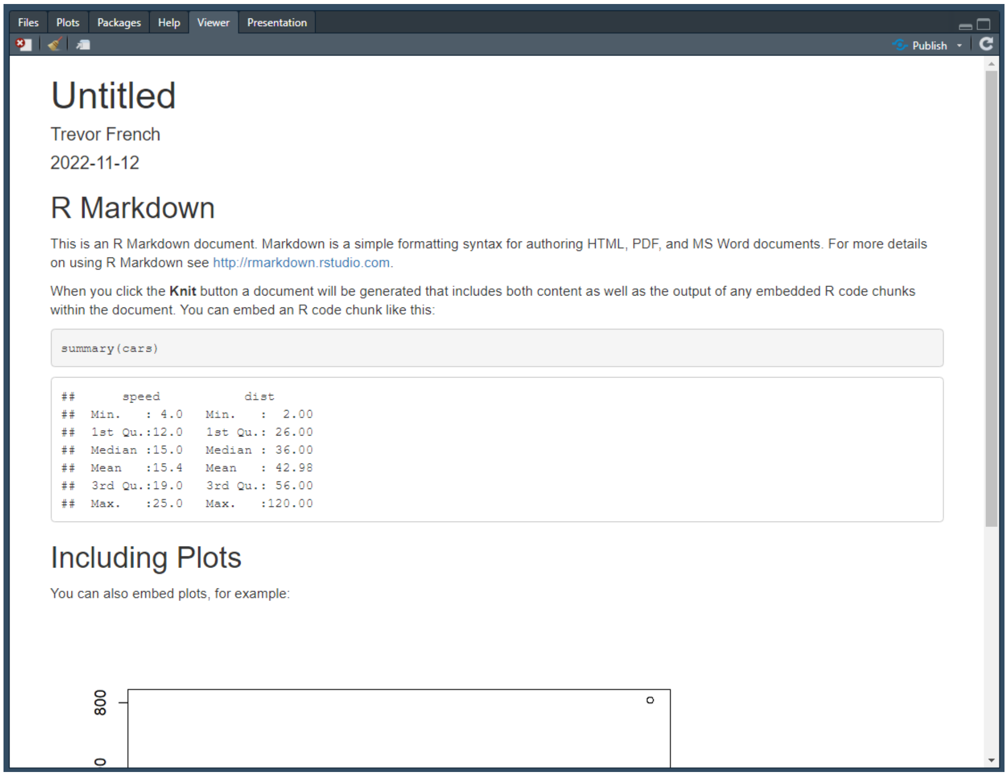
You can either continue to edit your document with markdown code or you can select the "visual" option towards the top-left corner of the source pane to have more of a traditional text editor experience.



Finally, we can render our document by selecting the "knit" button.



Selecting this will prompt you to save your file. After you do so, your rendered document will appear in your viewer tab.

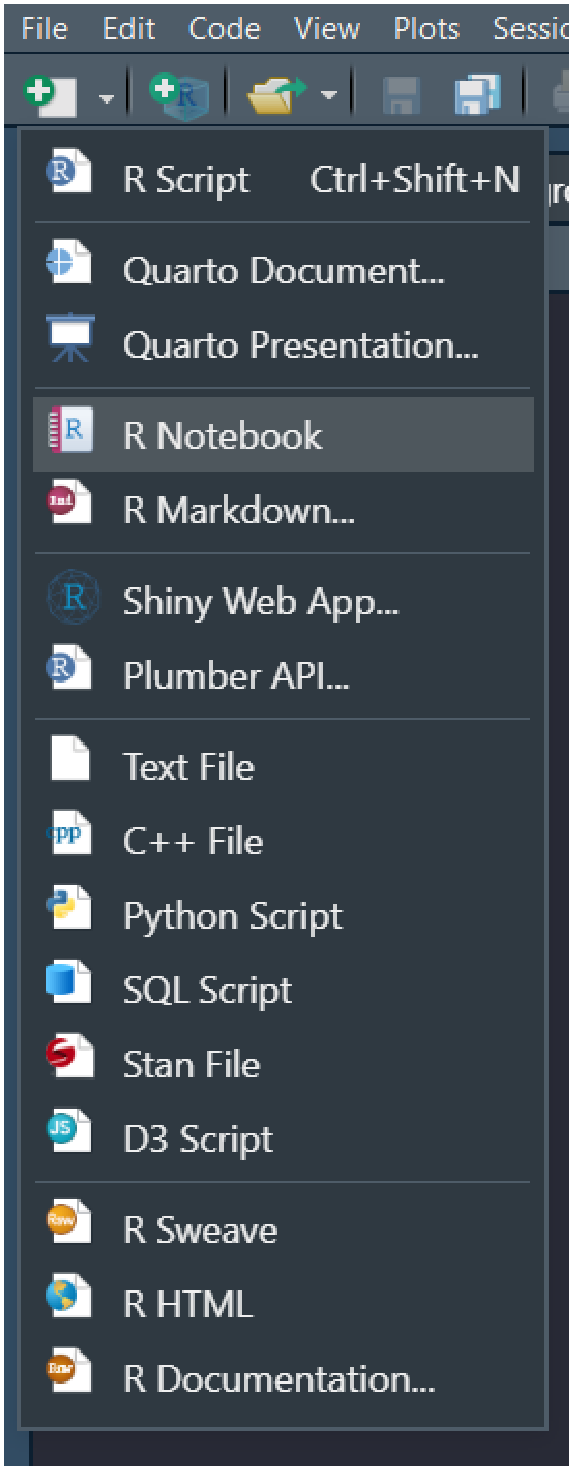


In addition to the preview being displayed in your viewer tab, you should now also have an HTML file located in the same place as you saved your R Markdown file. You can select this file to preview it in your browser as well as send it to others for them to preview.

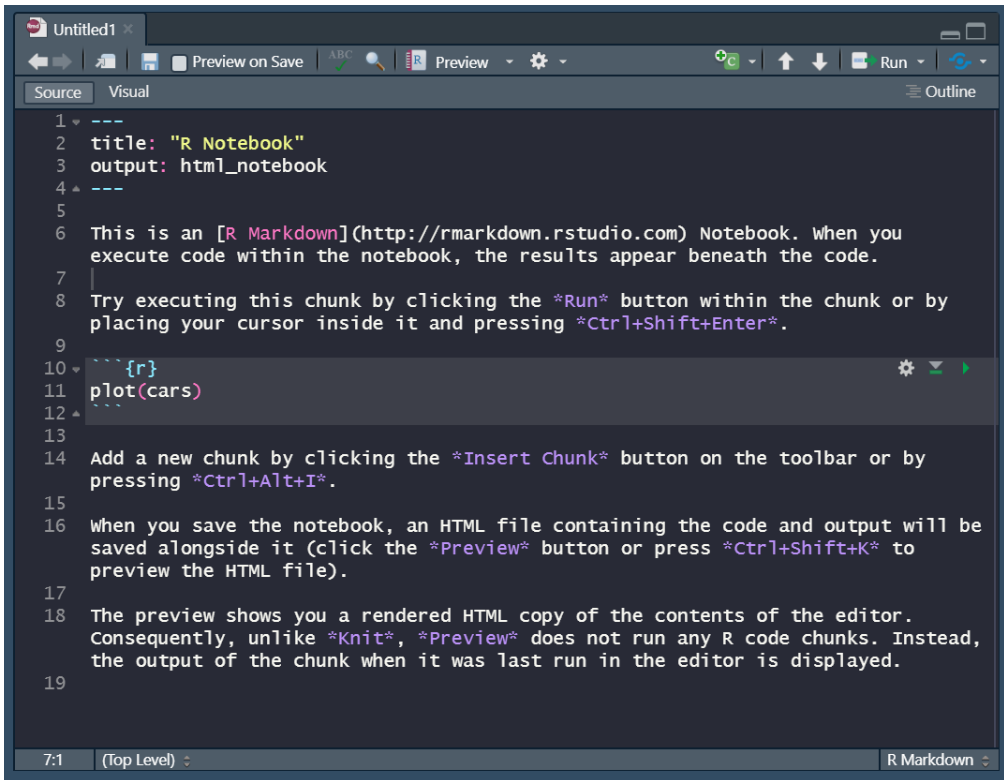
19.3 R Notebook

Another subset of R Markdown is R Notebooks. There is a lot of crossover between regular R Markdown documents and R Notebooks; however, R notebooks will generally be used for more technical audiences such as other R users or even just to organize your own thought processes while coding.

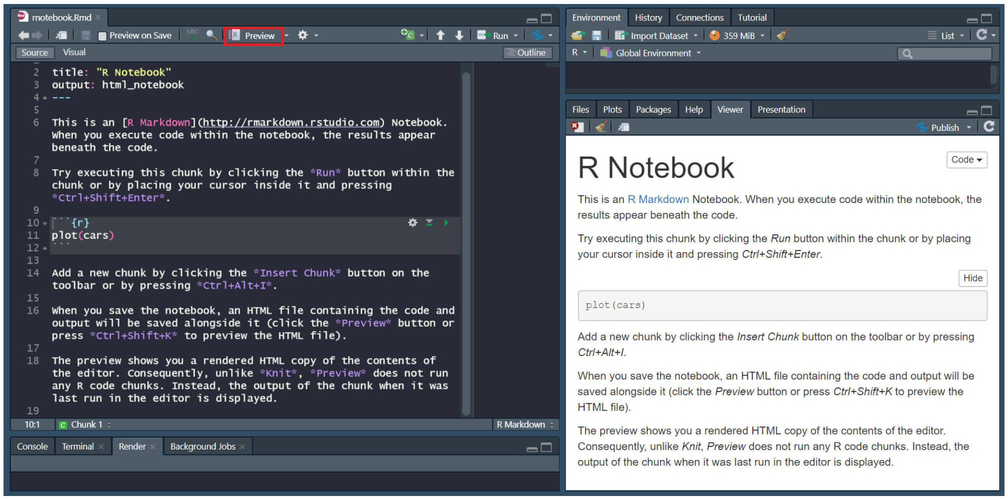
Let's try creating a notebook by selecting the "New File" button towards the top left corner of R Studio and choosing "R Notebook" from the dropdown menu.


File Edit Code View Plots Sessi
R Script Ctrl+Shift+N
Quarto Document...
Quarto Presentation...
R Notebook
R Markdown...
Shiny Web App...
Plumber API...
Text File
C++ File
Python Script
SQL Script
Stan File
D3 Script
R Sweave
R HTML
R Documentation...

This will generate a new file in your source pane that looks like the following image.

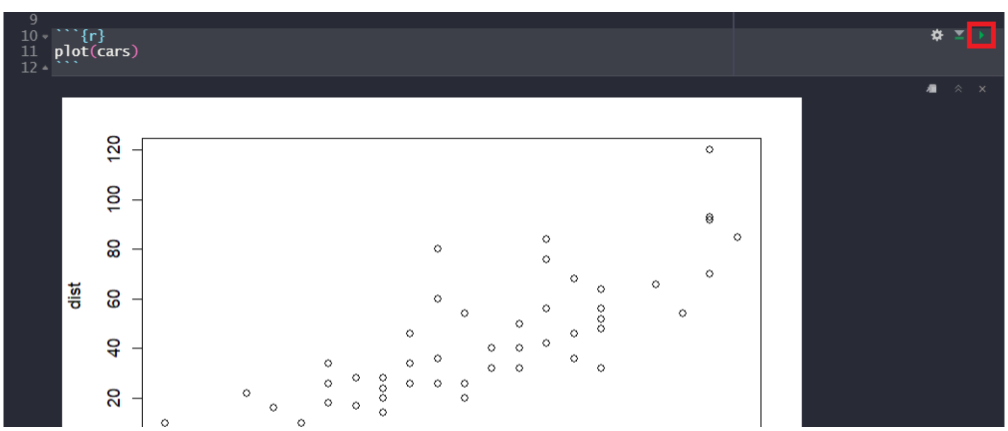


You'll notice that there is no "knit" option like there is in an ordinary R Markdown file. This is because this file is meant to be shared in its current format rather than as a rendered document. The "knit" option is replaced by a "preview" option. Selecting this option will result in the following output.

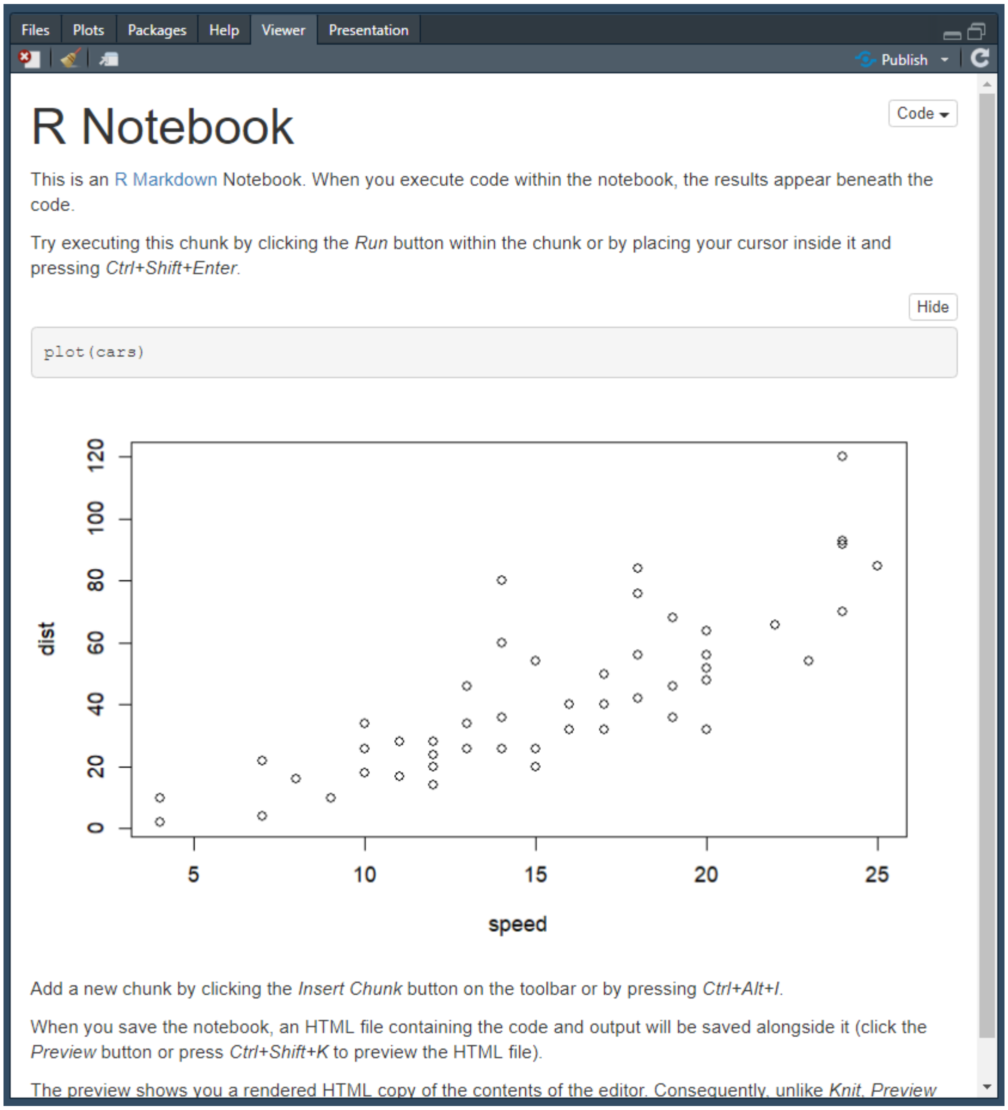


This generates a preview of your file in the viewer tab. You may also notice that the output of the `plot(cars)` code has not been rendered in the preview. This is because code has to be explicitly run in R Notebooks in order for it to be displayed in the rendered preview.

Let's run the code by pressing the green play button inside the code chunk.



Now if you preview the notebook again you'll see the plot output included.



19.4 Resources

- "Document Templates" from "R Markdown: The Definitive Guide": https://bookdown.org/yihui/rmarkdown/document-templates.html?version=2022.07.2%2B576&mode=desktop
- R Markdown Formats: https://rmarkdown.rstudio.com/formats.html
- R Markdown Home Page: https://rmarkdown.rstudio.com/
- R Markdown Notebooks: https://rmarkdown.rstudio.com/lesson-10.html

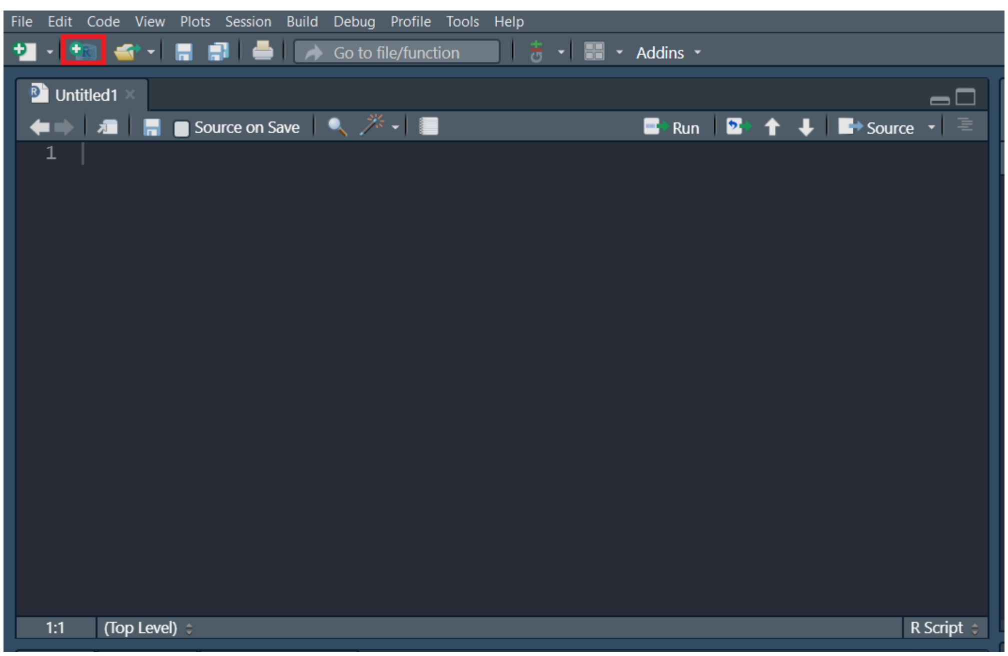
Chapter 20

R Shiny

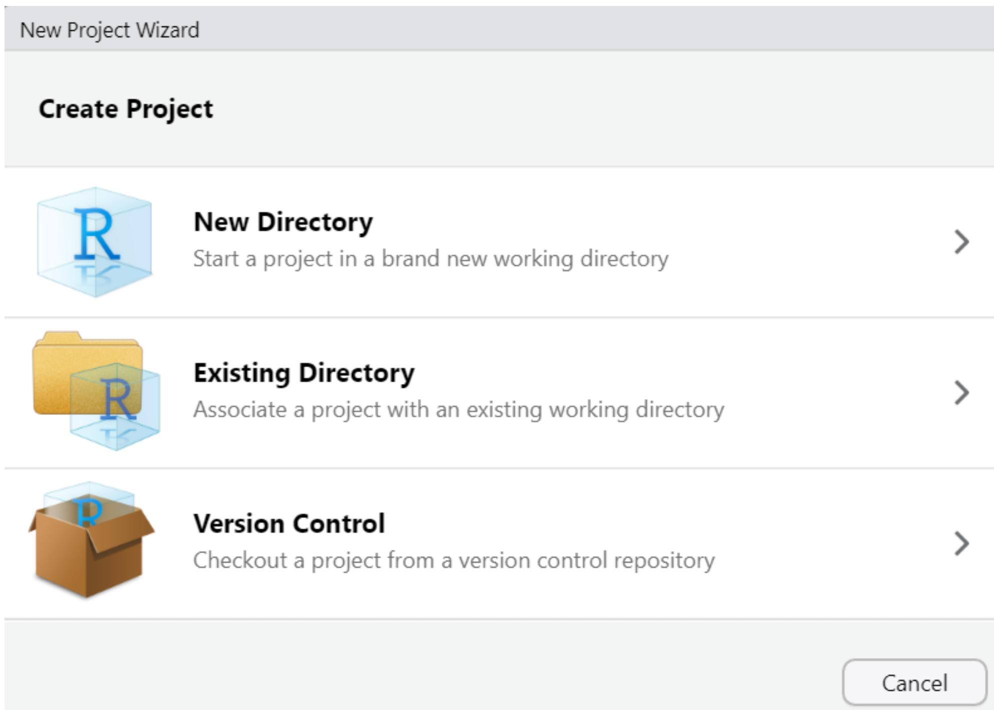
R Shiny is a tool used to develop web applications and is commonly deployed in the use of creating dashboards, hosting static reports, and custom tooling.

20.1 Quickstart

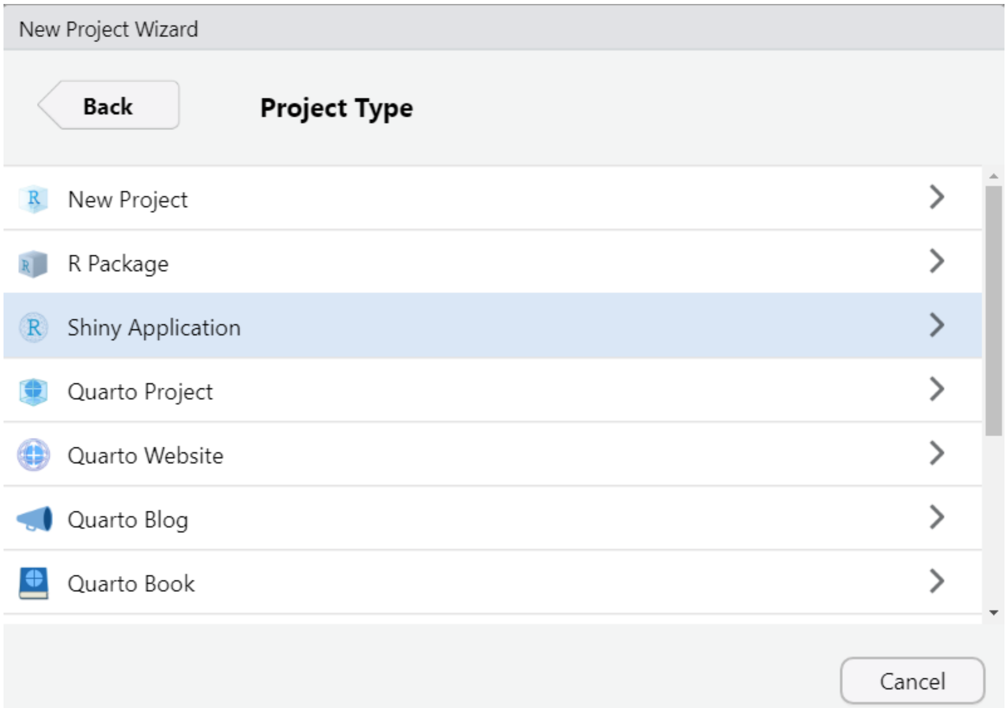
Let's create a new project containing a shiny application. Projects allow you to bundle multiple files into a a single workspace. You can create a new project via the "Create a new project" button towards the top left corner in RStudio.



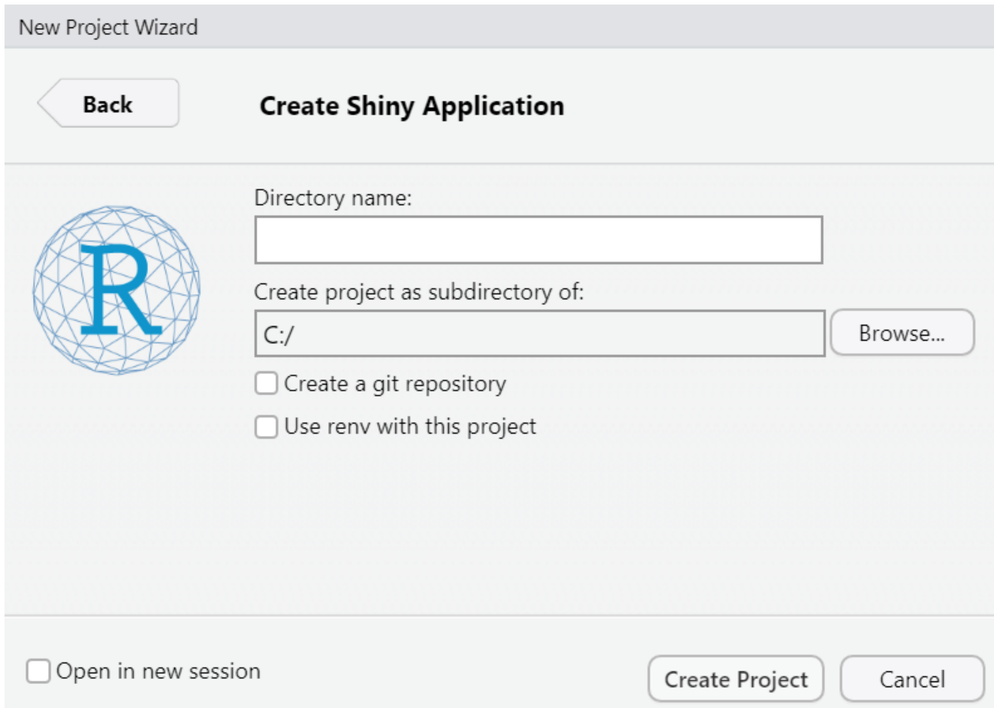
Since we are starting this project from scratch, let's choose the "New Directory" option.



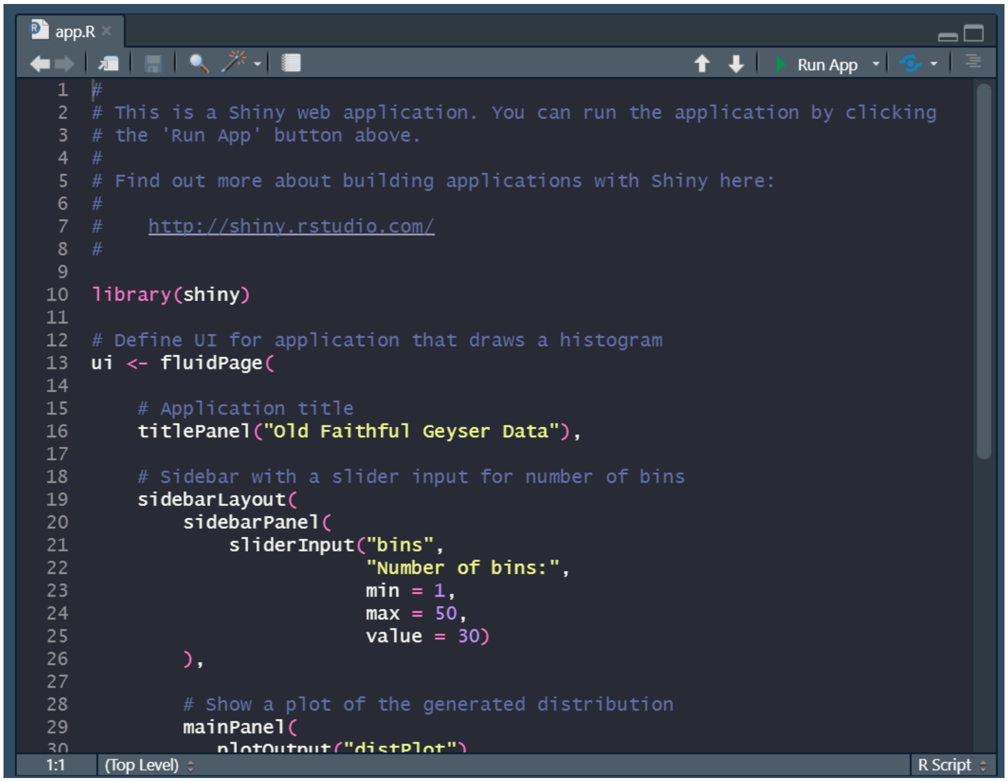
Now you can see there are many types of projects that you can create (not just Shiny Applications). However, we are going to choose "Shiny Application" for this example.



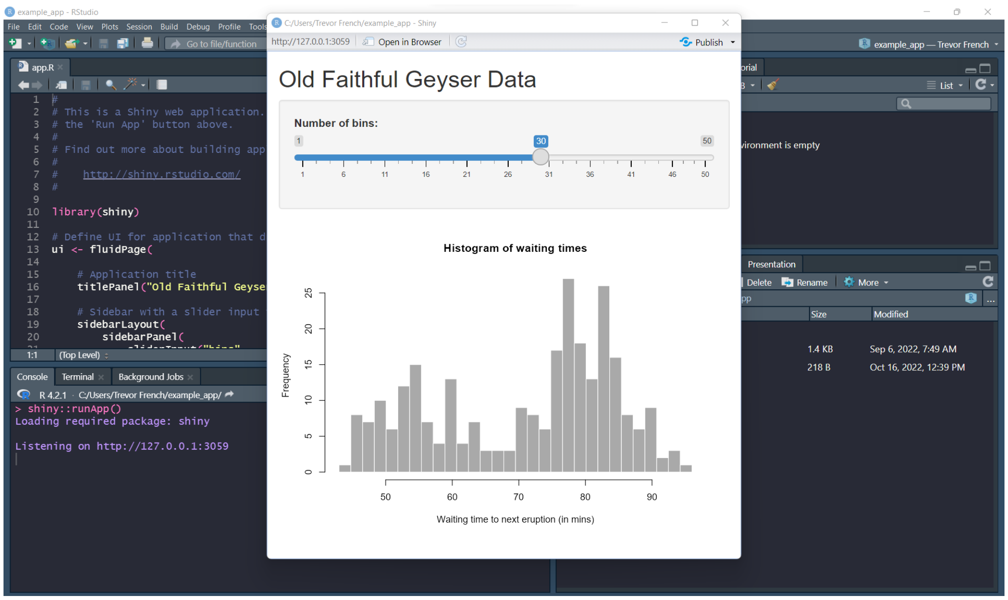
This is going to create a new folder containing your project files. Choose what you would like to name that folder and where you would like for it to be saved.



If you're working in RStudio, you should now have a sample application in your source pane. We'll go more in depth into what all of this means later.



For now, let's demo what this app looks like by pressing the "Run App" button towards the top right corner of your source pane. You should see a screen pop up that looks like this.



We can see that the application is using the faithful dataset to create a histogram which accepts user input to dynamically adjust the number of bins presented in the histogram.

20.2 Basic Components of a Shiny Application

Shiny applications consist of two main components: a server function and a UI object. The server function will handle any logic you need to put into your application while the UI object will create a user interface. Additionally, you will need to include the "shiny" library and any other libraries that you use in your code. Let's break down everything that is happening in this sample application

20.2.1 Libraries

One library you will always need to include in your shiny applications is the "shiny" library. Make sure you include any other libraries you plan on using in your code.

```
library(shiny)
```

20.2.2 UI

The next thing we see in our code is the creation of our UI object. This is where the application layout is created. The first function is the "fluidPage" function.

This is probably one of the most common ways to create user interfaces in shiny applications. Layouts created with the fluid page methodology are organized into rows and columns and scale to fit varying browser sizes.

The "titlePanel" function creates a panel with your title inside of it. In our case, this function is responsible for "Old Faithful Geyser Data" being displayed at the top of the page.

Next, we see the "sidebarLayout" function. This is essentially a pre-constructed layout which consists of a "sidebar" panel and a "main" panel which are created using the "sidebarPanel" and "mainPanel" functions, respectively. You'll notice that our sidebar is actually located above our main panel rather than to the side. This is just because the size of our browser was small enough that they collapsed to be stacked on top of each other. If you increase the size of your browser, you will see the sidebar return to it's original location.

Inside of the "sidebarPanel" function, we have a function called "sliderInput". The "sliderInput" function creates the component which allows the user to select the number of bins in our app. We can see this function gives the component the name "bins", the title "Number of Bins", a minimum input of "1", a maximum value of "50", and a default value of "30".

The last component of our UI object is the "mainPanel" function. This main panel designates the section where our output plot will ultimately go as can be observed by the "plotOutput" function nested inside of it. This "plotOutput" function is given the name "distPlot". This is done so that it can be referenced later in our server function.

```
ui <- fluidPage(

    # Application title
    titlePanel("Old Faithful Geyser Data"),

    # Sidebar with a slider input for number of bins
    sidebarLayout(
        sidebarPanel(
            sliderInput("bins",
                        "Number of bins:",
                        min = 1,
                        max = 50,
                        value = 30)
        ),

        # Show a plot of the generated distribution
        mainPanel(
           plotOutput("distPlot")
        )
```

```
    )
)
```

20.2.3 Server

After we create the UI object, we'll need to create our server function. We'll pass two arguments into the function: "input" and "output". The input argument allows us to access data from the user interface while the output argument allows us to pass data back to the user interface.

Inside the function, we reference the "distPlot" component of the UI by typing "output$distPlot". After this, we pass a plot to the UI with the "renderPlot" function.

> **Note**
>
> The UI can only accept the plot we are going to send it because it is using the "plotOutput" function. If you were going to send a different form of data, the UI would need to have the corresponding function in order to accept it.
> For example, if your server was going to send a table to the UI your server would need to use the "renderTable" function and your UI would need to use the "tableOutput" function.

```
server <- function(input, output) {

    output$distPlot <- renderPlot({
        # generate bins based on input$bins from ui.R
        x    <- faithful[, 2]
        bins <- seq(min(x), max(x), length.out = input$bins + 1)

        # draw the histogram with the specified number of bins
        hist(x, breaks = bins, col = 'darkgray', border = 'white',
             xlab = 'Waiting time to next eruption (in mins)',
             main = 'Histogram of waiting times')
    })
}
```

20.2.4 Putting it Together

Finally, you will combine your server and your UI and actually run your app with the "shinyApp" function.

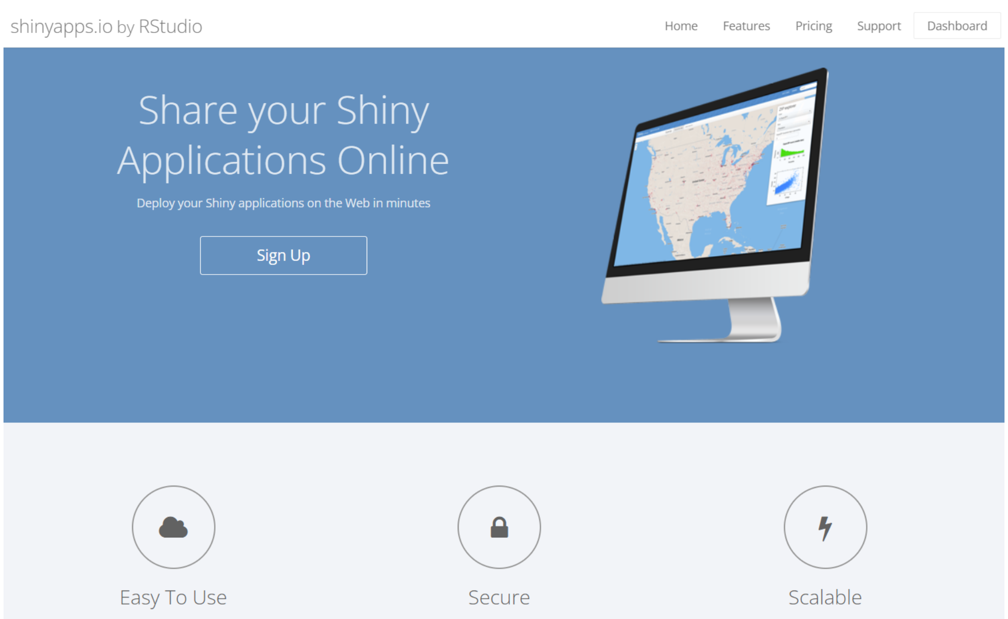
```
shinyApp(ui = ui, server = server)
```

20.3 Deploying Application

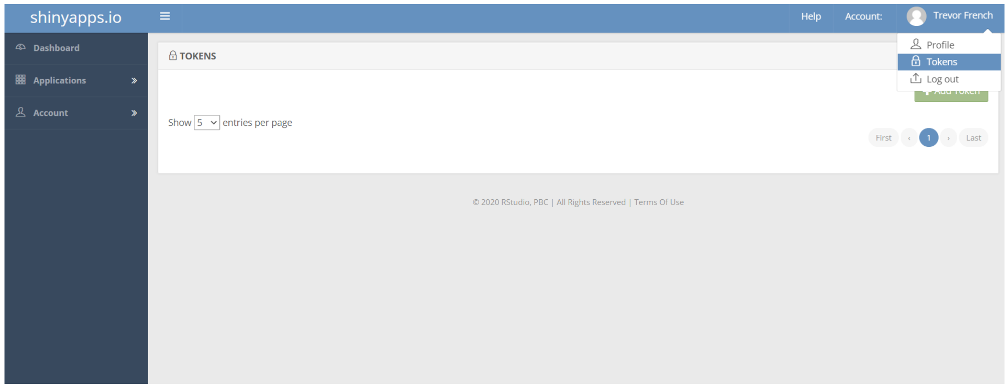
Now that you've built an application, you can actually deploy it for the whole world to see. There are many ways to do this; however, probably the easiest way to get started is to create a free account with ShinyApps.io.

20.3.1 ShinyApps.io

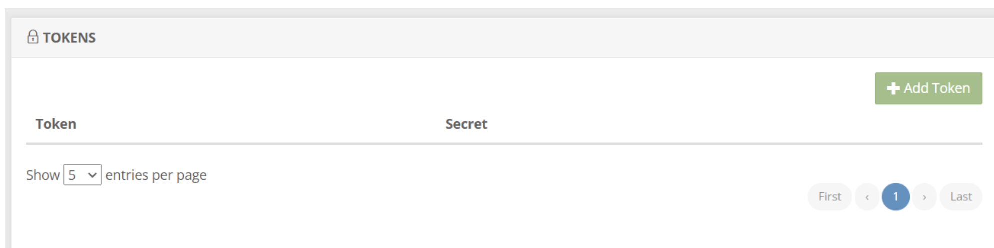
Navigate to https://www.shinyapps.io/, select the "Sign Up" button and follow the steps to create a free account.



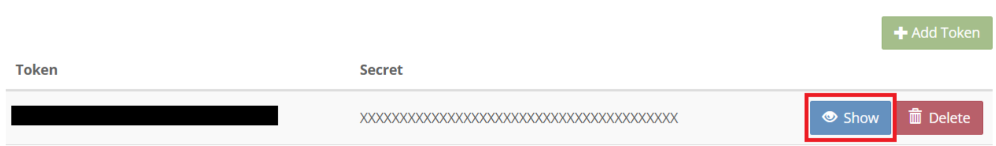
Once you create your account and see your dashboard, you can navigate to your "tokens" by selecting your name in the top right corner and choosing "tokens" from the dropdown menu.



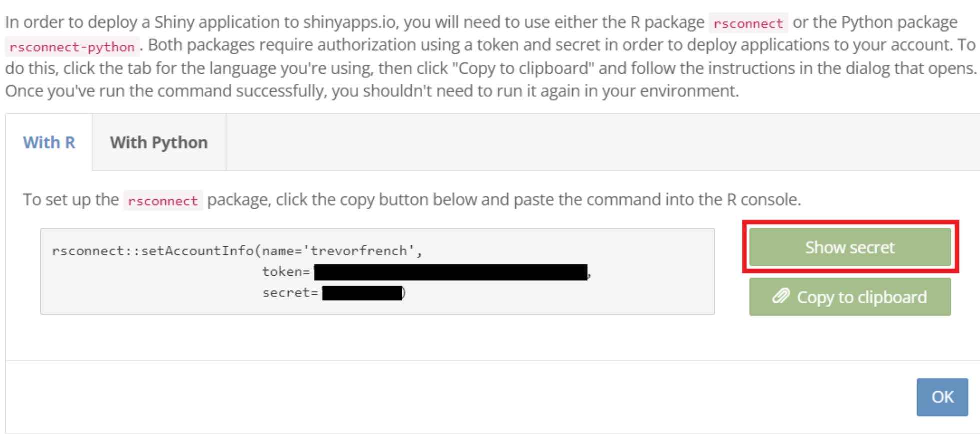
Choose the green "Add Token" button to create a new token.



Now that your token has been generated, select the blue "Show" button to view it.

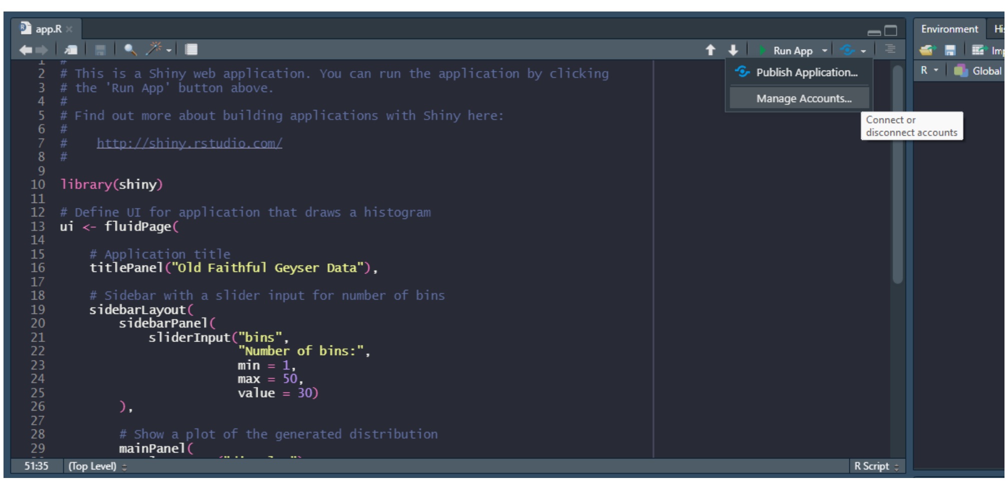


You should now have a window that resembles the following image. Select the "Show secret" button and then copy the code to your clipboard for use later.

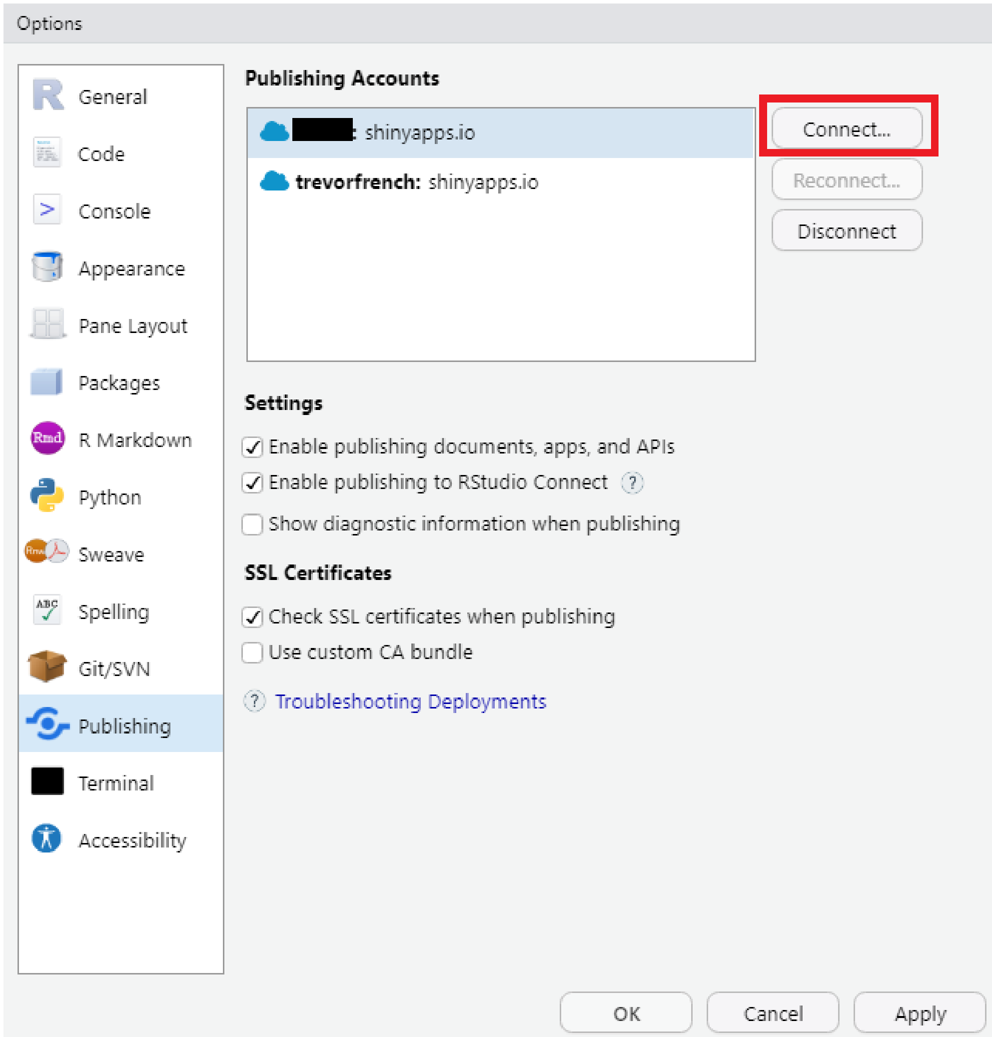


20.3.2 Configuring Account

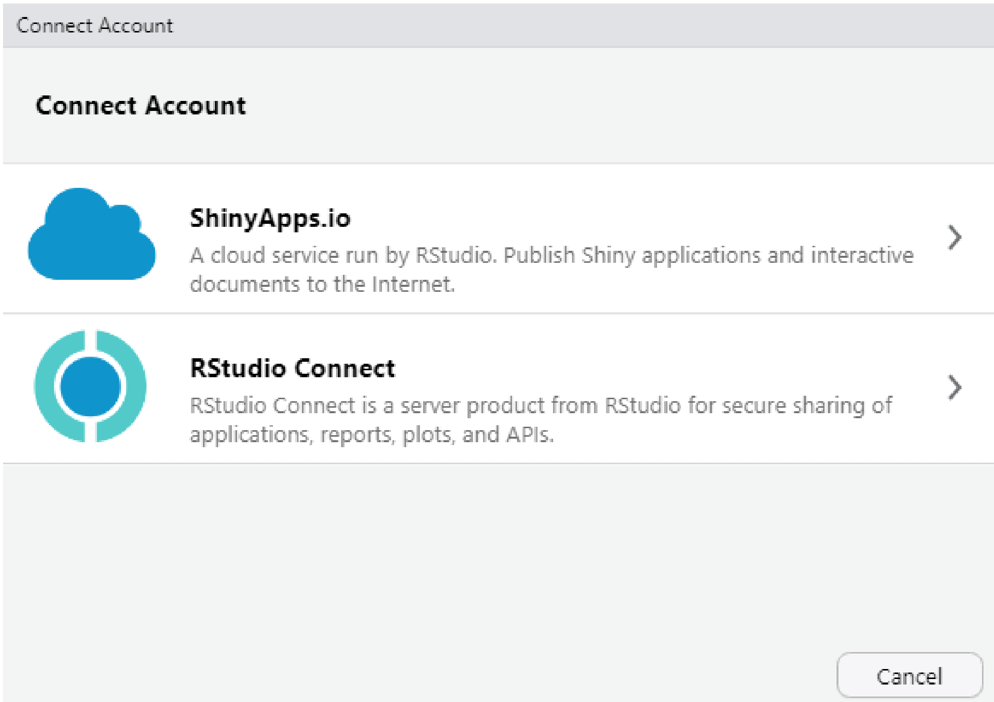
The next thing we'll need to do is to link RStudio to your ShinyApps.io account. You can do this by navigating back to RStudio and choosing the dropdown menu next to the publish button. From here, select the "Manage Accounts" option.



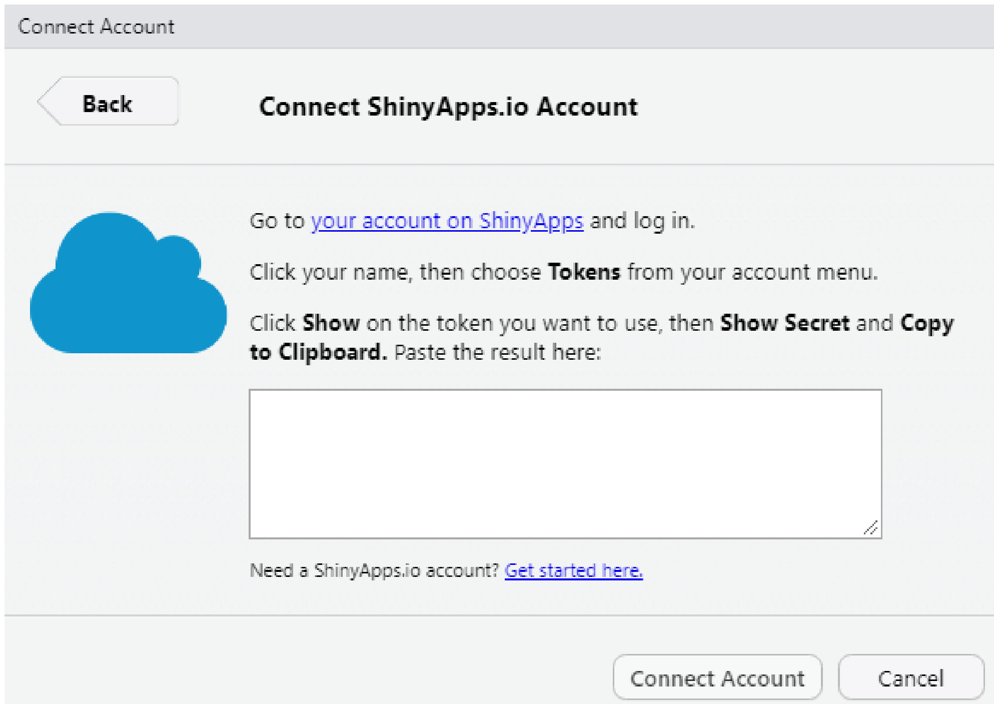
You'll then get a window the resembles the following image. Choose the "Connect" button to continue.



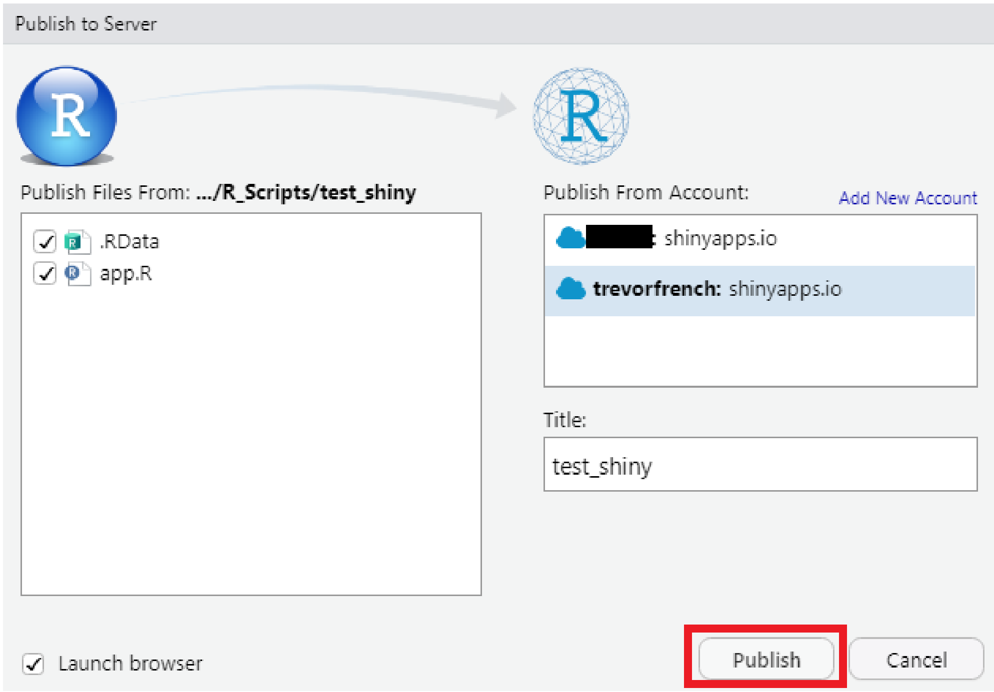
Next, you'll see the following options. Choose "ShinyApps.io" to continue.



Now you'll have the oppportunity to paste your token from you ShinyApps.io account. After you do so, press the "Connect Account" button.



Now that RStudio is linked to your ShinyApps.io account, you can press the publish button. You'll then get a window which allows you to name your app before publishing. Once you are satisfied with the name you choose, select "Publish".



After a few moments, your browser should launch displaying your newly created Shiny App!

20.4 Resources

- Shiny Home Page: https://shiny.rstudio.com/
- Shiny UI Editor: https://rstudio.github.io/shinyuieditor/

Exercises

Questions

Exercise: 18-A

Write the first seven rows of the "faithful" dataset to a csv file named "faithful.csv". Make sure you do not include any row names in your output file.

Exercise: 18-B

Write the entire "faithful" dataset to an xlsx file using the "saveWorkbook" function. Name the tab (worksheet) that the data is on "data" and make the text in the header row bold.

Answers

Answer: 18-A

You can accomplish this through the use of the "write.csv" function.

```
write.csv(head(faithful, 7), "faithful.csv", row.names = FALSE)
```

Answer: 18-B

The following code will allow you to accomplish this task.

```
library(openxlsx)
wb <- createWorkbook()
heading <- createStyle(textDecoration = "bold")
addWorksheet(wb, "data")
writeData(wb
            , "data"
            , faithful
            , startCol = 1
            , startRow = 1
            , rowNames = FALSE)
addStyle(wb
            , "data"
            , cols = 1:length(faithful)
            , rows = 1
            , style = heading
            , gridExpand = TRUE)
saveWorkbook(wb, "faithful.xlsx", overwrite = TRUE)
```

References

Chambers, Cleveland, J. M., and P. A. Tukey. 1983. *Graphical Methods for Data Analysis*. Wadsworth & Brooks/Cole.

Eremenko, Kirill. 2020. "Hadley Wickham Talks Integration and Future of r and Python [Audio Podcast]." SuperDataScience. https://www.superdatascience.com/podcast/hadley-wickham-talks-integration-and-future-of-python-and-r.

Garvin, David A. 1993. "Building a Learning Organization." *Harvard Business Review* July-August 1993.

Hermans, Felienne. 2021. "Hadley Wickham on r and Tidyverse [Audio Podcast]." Software Engineering Radio. https://www.se-radio.net/2021/03/episode-450-hadley-wickham-on-r-and-tidyverse/.

Hofmann, J. R. 1996. *Enlightenment and Electrodynamics*. Cambridge University Press.

Ihaka, Ross. 1998. "R : Past and Future History." https://www.stat.auckland.ac.nz/~ihaka/downloads/Interface98.pdf.

McCandless, David. 2010. "The Beauty of Data Visualization." https://www.ted.com/talks/david_mccandless_the_beauty_of_data_visualization/transcript?language=en.

Paulson, Josh. 2022. *Navigating Code in the RStudio IDE*. https://support.rstudio.com/hc/en-us/articles/200710523-Navigating-Code-in-the-RStudio-IDE.