Intro to R Workshop

Dr. Na Li, Mark Ly

2023-04-24

Contents

1 About the workshop **5**

1.1 Audience 5

1.2 Learning objectives 5

2 R and R studio **7**

2.1 Getting Started (Desktop) 7

2.2 Getting Started (Cloud) 10

3 Basics **21**

3.1 Basic Operations 21

3.2 Code Chunk 22

3.3 Storing Variables 23

3.4 Using Variables 24

3.5 Logical operators 27

3.6 Data types 27

4 Data Structures **31**

4.1 Vectors 31

4.2 Lists 32

4.3 Matrices 36

4.4 Dataframes 41

4.5 Factors 44

5 Data Wrangling **45**
5.1 R Packages 45
5.2 dplyr 46
5.3 Loading datasets 48
5.4 Understanding your dataset 49
5.5 Combining your dataset 50
5.6 Missing values 55

6 Data Exploration **59**
6.1 Summary statistics 59
6.2 Standard deviation and variance 61

7 Data visualization **63**
7.1 Plots 63
7.2 ggplot2 68
7.3 gtsummary 75

8 Regression **79**
8.1 Linear regression 79

9 Other capabilties **85**
9.1 Machine learning 85
9.2 R-bookdown 87
9.3 R-Shiny 87

Chapter 1

About the workshop

Introduction to R is a 6-hour workshop, split into two 3-hour in-person sessions, introducing basic programming concepts in R and learning to execute data manipulations, calculations, basic statistical analyses, and produce useful figures and tables. Participants will also learn to write simple functions that can be used to automate analyses, practical statistical computing, and general programming concepts.

1.1 Audience

This workshop is targeted towards researchers who are interested in learning R programming for data analysis within the University of Calgary and AHS.

No prior programming knowledge is required.

Having previous research experience or working in a research setting is preferred.

1.2 Learning objectives

By the end of the workshop, participants will be able to:

1. Install and configure R and R studio
2. Be familiar with the R studio IDE
3. Clean and prepare a dataset for analysis with common packages and functions
4. Manipulate a data set to create meaningful tables and figures

5. Perform some common statistical analysis
6. Learn about some other advanced capabilities that R has to offer.

Chapter 2

R and R studio

R is programming language like Javascript, Python, Java, C and C++, that is mostly used for statistical computing and visualizations.

RStudio is a integrated development environment (*IDE*) created to help organize and streamline your programming with R.

There is a desktop version of RStudio, where you can download and work on a local environment or if you prefer, there is a cloud version where you can do cloud computing instead.

Both the desktop and cloud version will be able to produce the same results and it really depends on your workstation capabilities, what types of scripts you are planning to run and, how often you are planning to use RStudio.

2.1 Getting Started (Desktop)

2.1.1 Installation

To get started we want to download both **R**, the programming language, and **R studio** the IDE.

You can get both from a quick Google search or from the website. https://posit.co/download/rstudio-desktop/

We want to install **R** before we install **RStudio**

2.1.2 Creating .RMD file

Once you have downloaded both **R** and **RStudio** you can load up the **RStudio IDE** and it will come up with something like this.

Figure 2.1: Download screen for R and RStudio

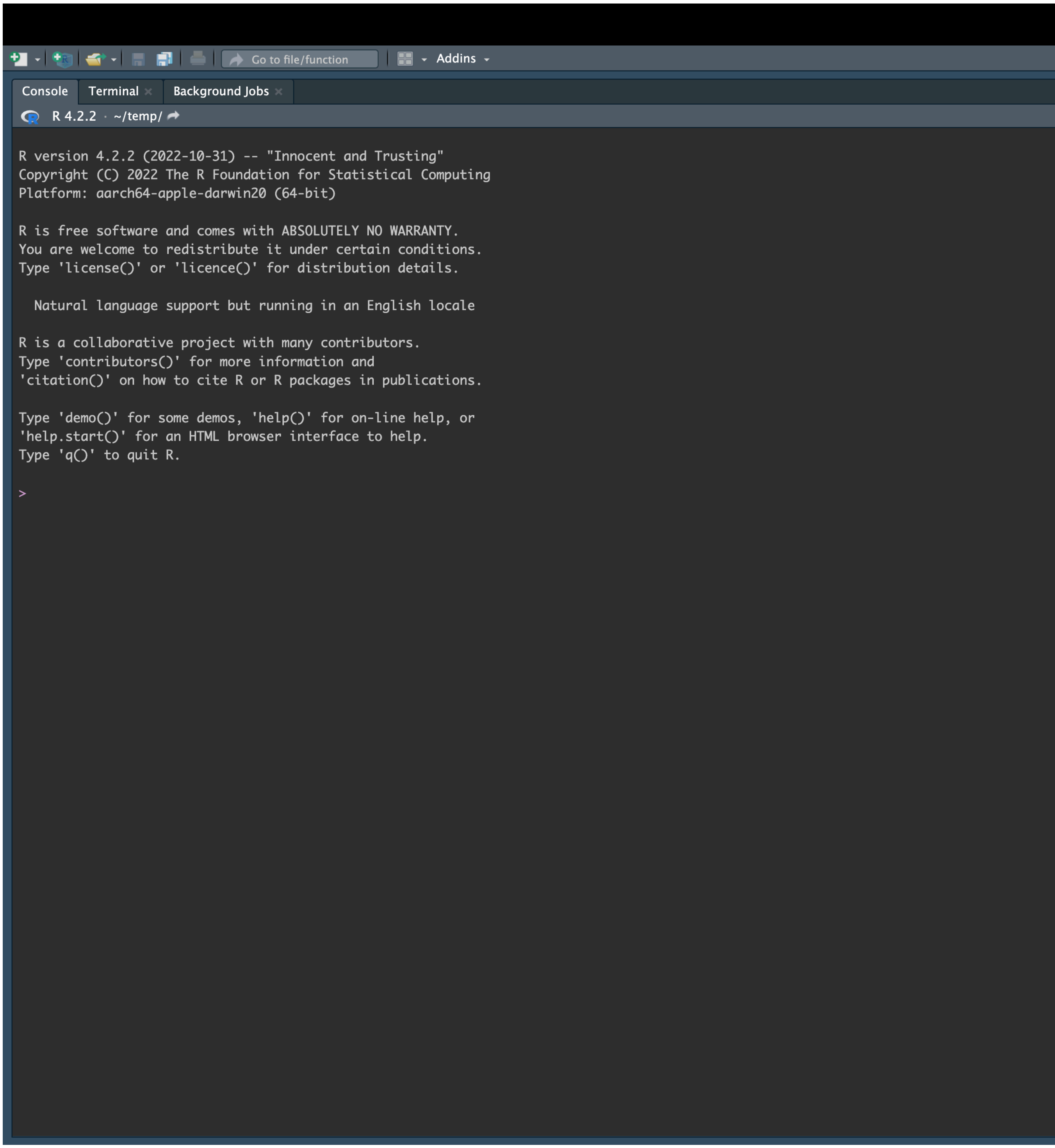


Figure 2.2: RStudio IDE

From here we want create a **R Markdown** file by going to `File -> New File -> R Markdown`

R Markdown files are unique to R which is document that combines both text and code and allows you to format your document for HTML, PDF, MS Word. You can tell it is a **R Markdown** file when it has the extension ***.rmd***

A popup window should come up and we need to title our **R Markdown** file.

You can type in ***2023 Rworkshop*** for the title and then click on ok to create the ***.rmd*** file.

Once you hit `OK`, you should see a tab at the top that says `Untitled1` and your **RStudio IDE** should have 4 distinct panels.

1. Source - Places where most of the coding happens. The source can look different depending the type of file you are working with (.rmd, .R, .MD). Any dataset you want to view will also show up in this window.

2. Environment/History - This is were you can find any stored variables (objects), imported scripts, loaded databases that are defined in memory. The history tab will contain a history of all the R commands that you have executed in this session

3. Console/Terminal - This is were the commands that are written in the source window are actually executed and started to run. This would be the same if you were to use R using a command line instead of an IDE. You can run and enter in commands and scripts in this window, but they will be executed as soon as you hit `ENTER/RETURN`. Can be used to quickly check a snip of code, do some basic calculations or install some packages. Runtime errors will also show up in this window which can be useful when you are debugging.

4. Files/Plots/Pkgs/Help/Viewer - This is more a directory window where you can cycle between files, plots, packages, help, and Viewer.

2.2 Getting Started (Cloud)

If you would like to use the cloud version of **RStudio** you can sign up for the free version here:

https://posit.cloud/plans/free

If you are just planning to use R occasionally and don't need heavy computing, then the free version of RStudio cloud will work just fine.

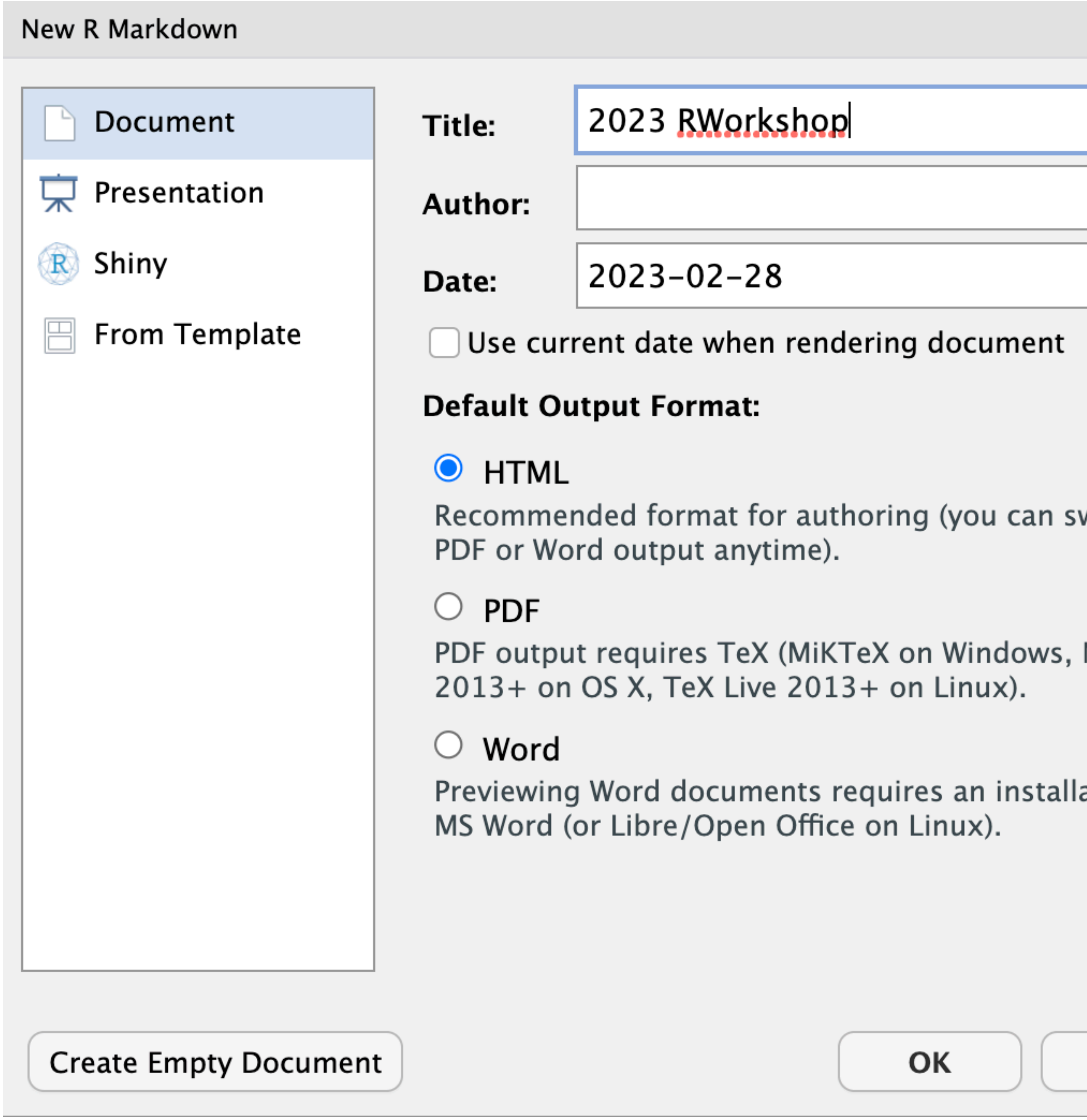


Figure 2.3: RStudio markdown creation

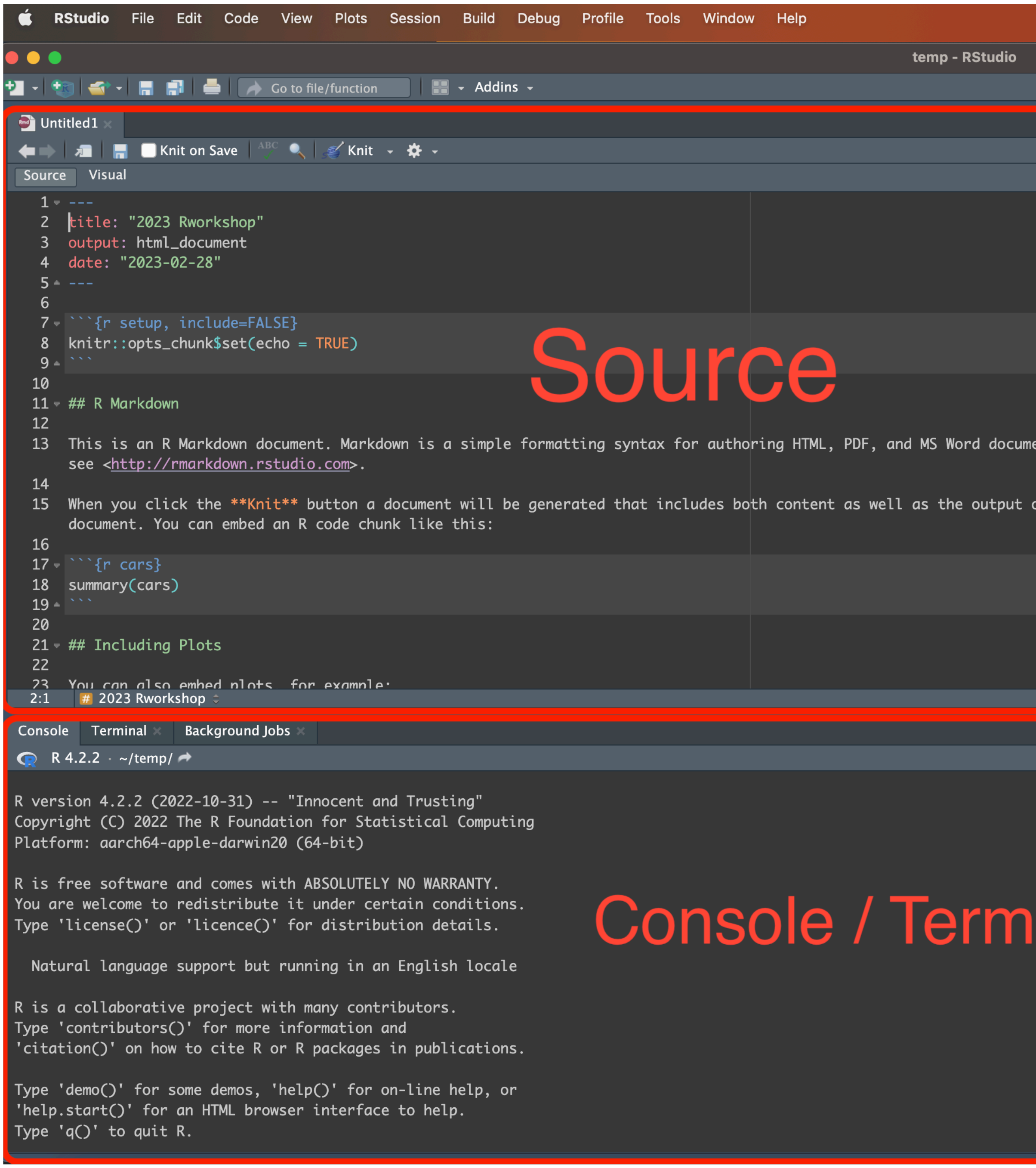


Figure 2.4: New .rmd file

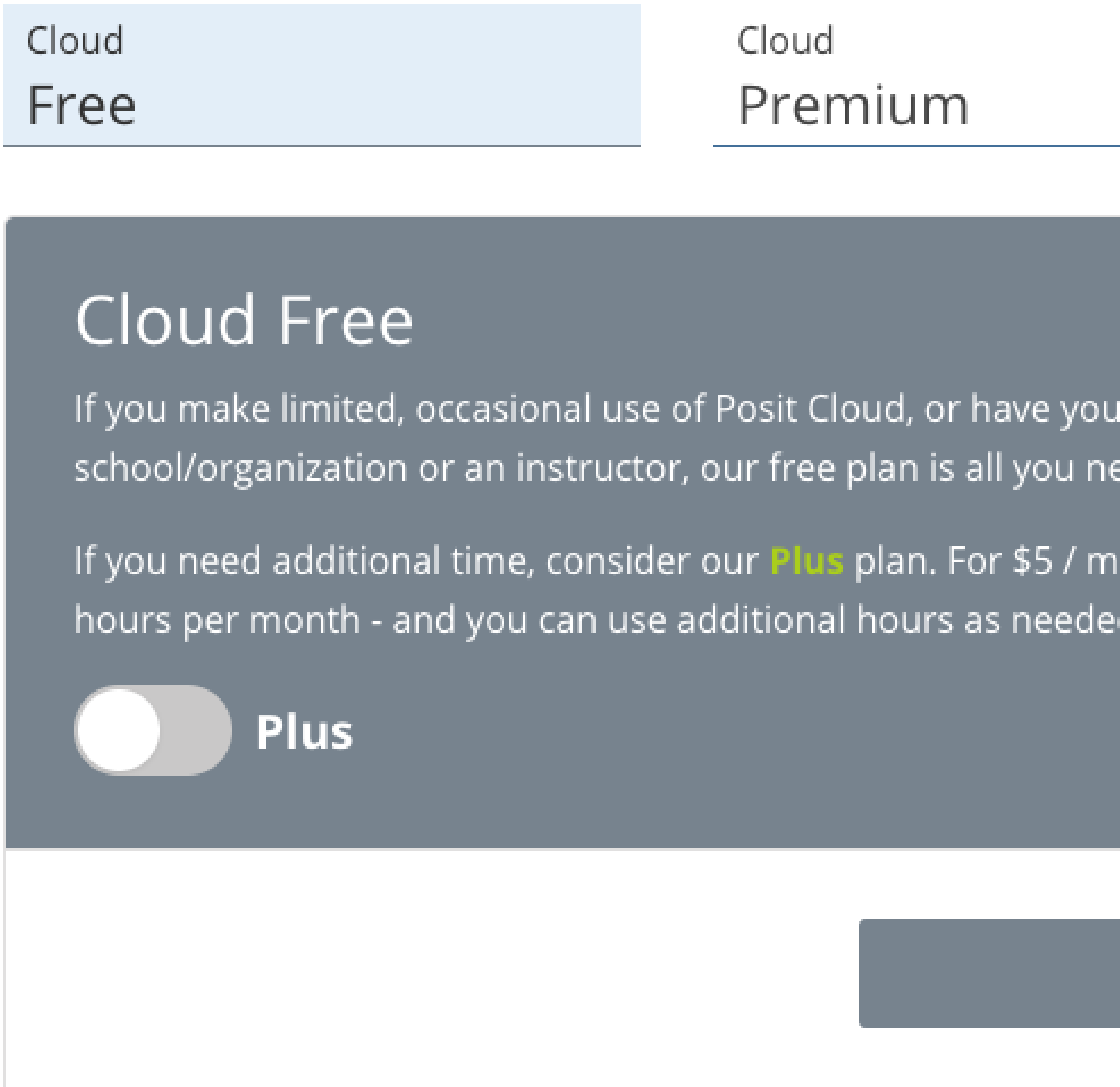


Figure 2.5: Sign up for RStudio cloud

2.2.1 Creating .RMD File

After logging in to the free account, you can click on `New Project` on the right hand side and select `New RStudio Project`

Once you open the new project, you will get a screen similar to this.

From here we want create a **R Markdown** file by going to `File -> New File -> R Markdown`

A pop-up will appear saying it will need to install some packages to create a **R Markdown** file. You can install these by selecting `Yes`

Another popup window should come up and we need to title our **R Markdown** file.

You can type in ***2023 Rworkshop*** for the title and then click on ok to create the ***.rmd*** file.

Once you hit `OK`, you should see a tab at the top that says `Untitled1` and your **RStudio IDE** should have 4 distinct panels.

The panels are the same as the ones described in Getting Started (Desktop)

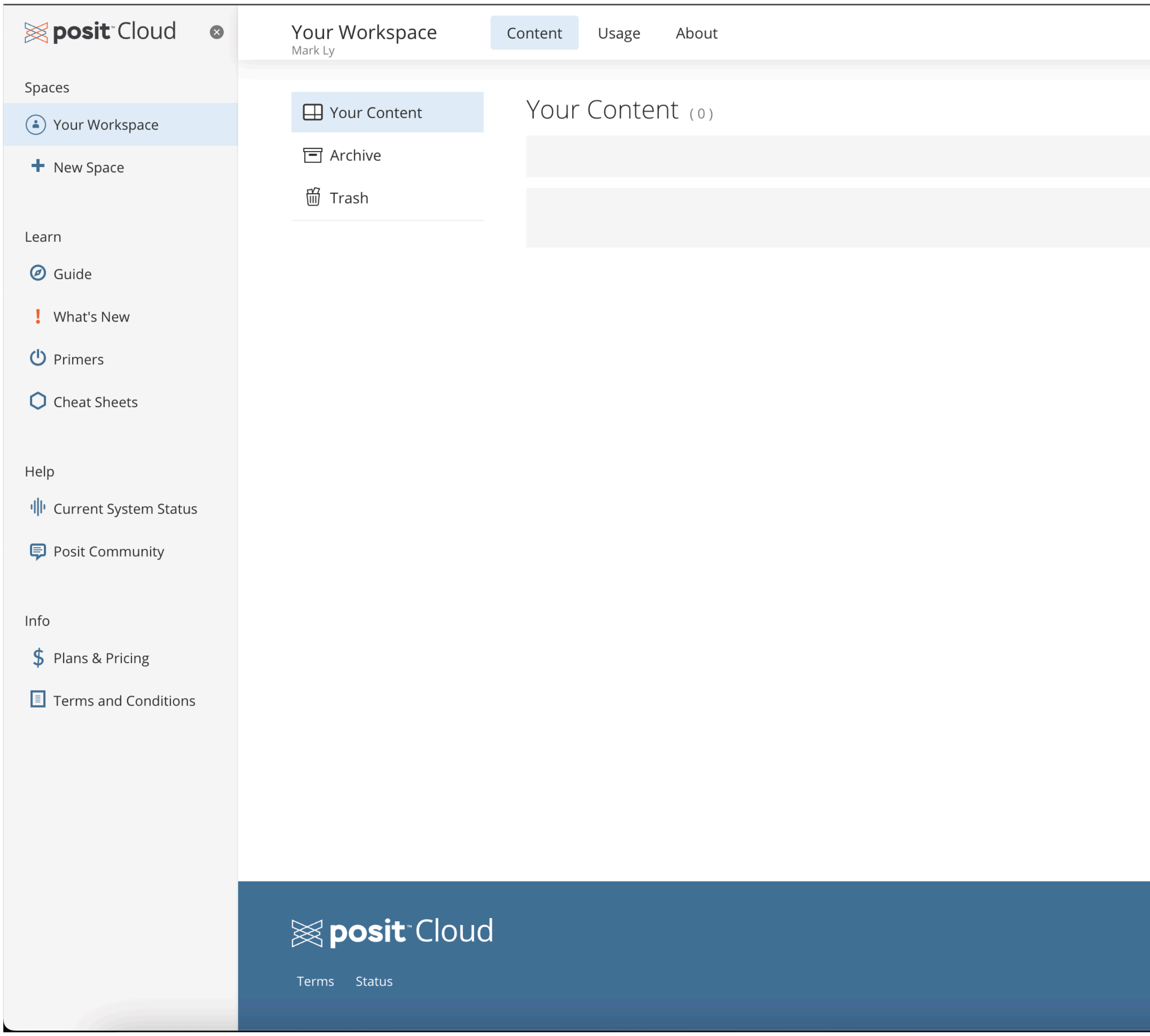


Figure 2.6: RStudio Cloud Creating a new project

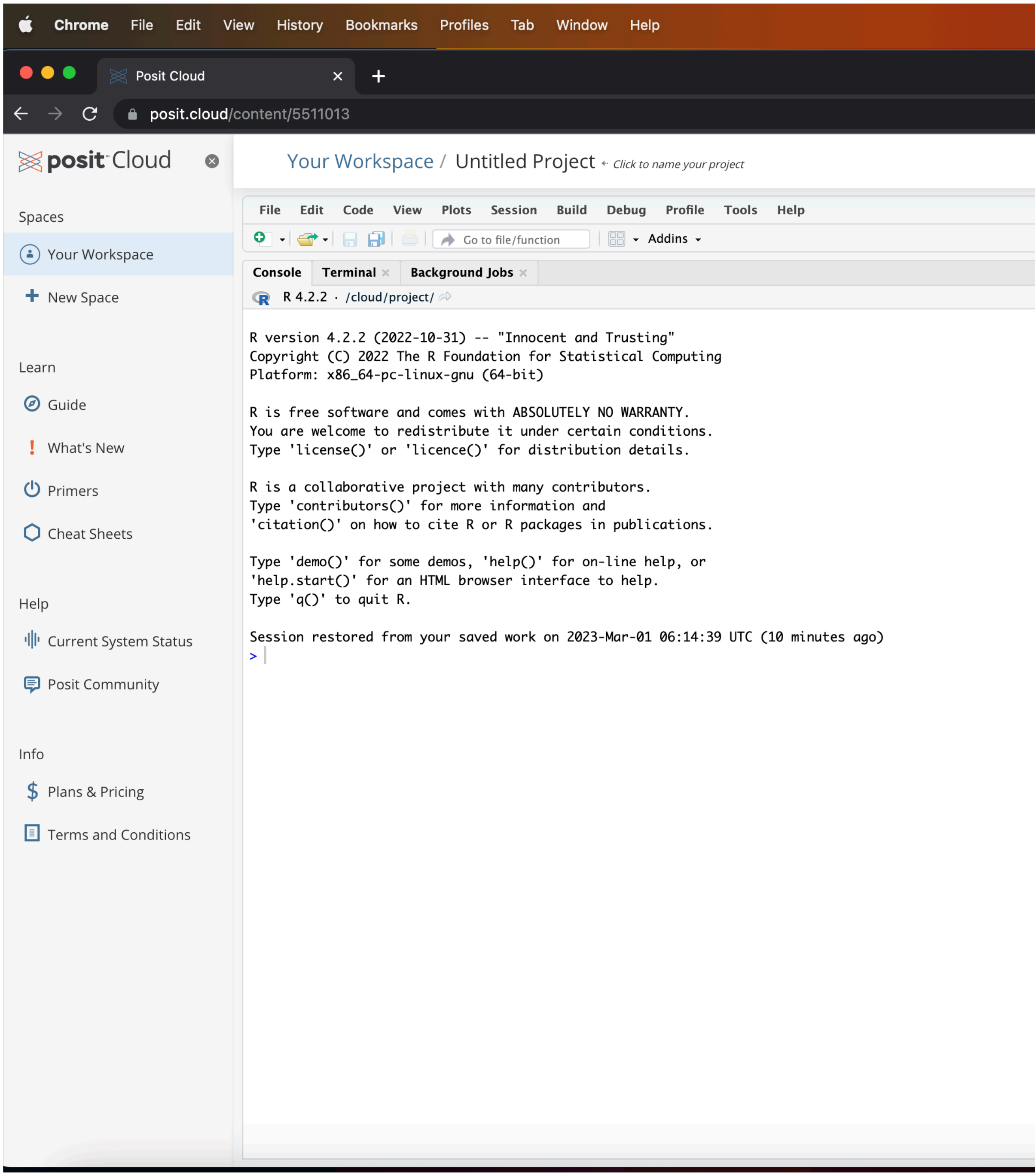


Figure 2.7: RStudio Cloud Creating a new project

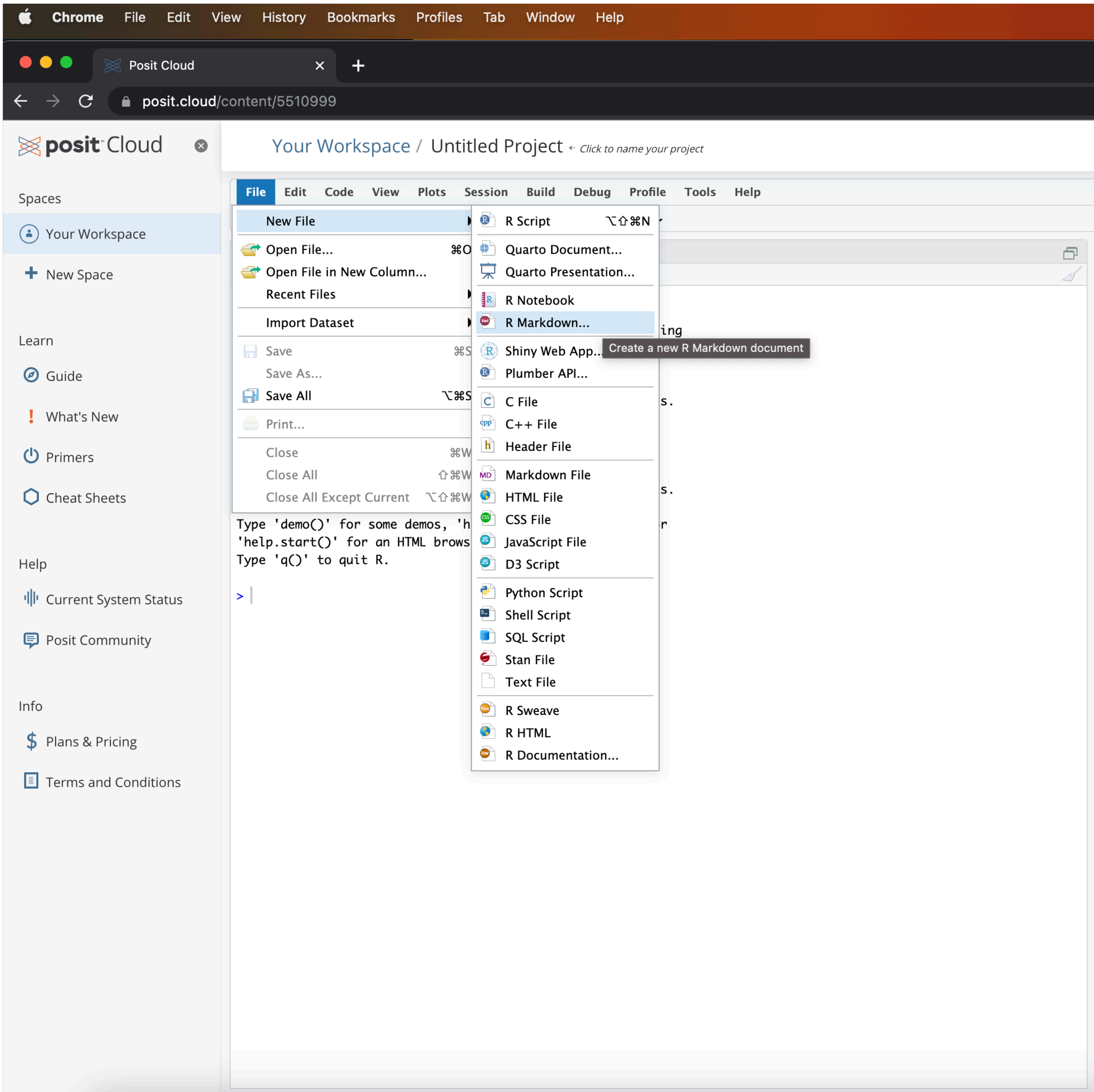


Figure 2.8: RStudio Cloud new markdown file

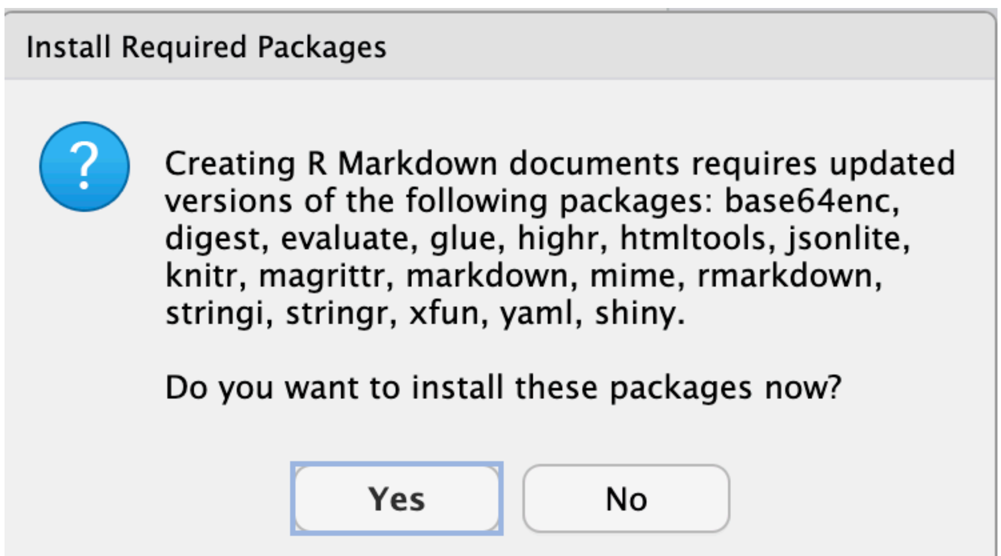


Figure 2.9: RStudio cloud markdown packages

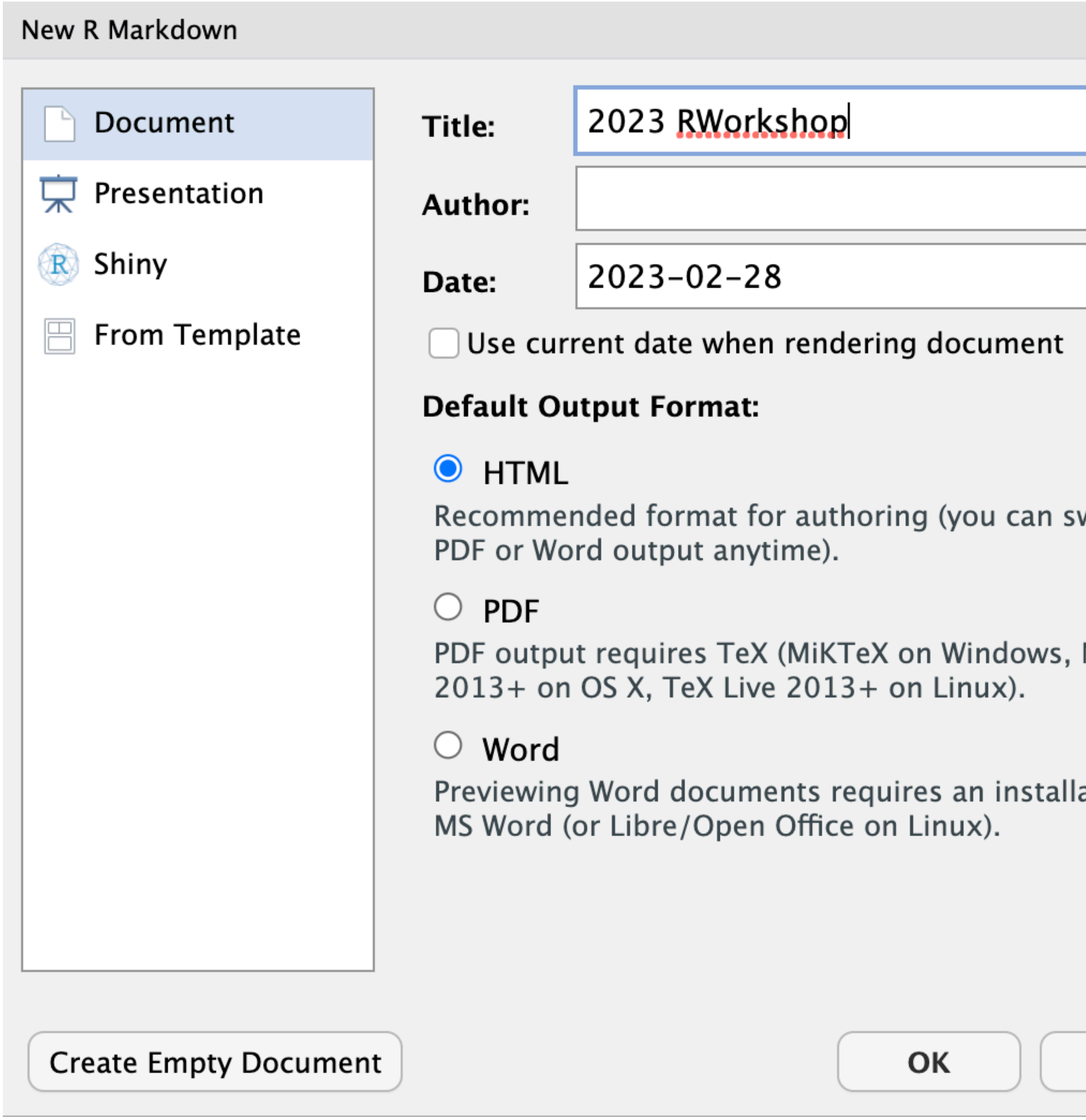


Figure 2.10: RStudio cloud markdown name

Chapter 3

Basics

Now that we set-up our **R Markdown** file we can start exploring what we can do with **R**

The examples will be shown on **R Desktop** but will work the same if you are using **R Cloud**. Differences will be highlighted if they occur later on in the workshop.

In this section, we will learn about some simple coding operations you can perform with **R**, learn about different data types and, how to create and manipulate variables.

3.1 Basic Operations

All the basic arithmetic operators can be done in using **R** which includes

- Addition: `+`
- Subtraction: `-`
- Multiplication: `*`
- Division: `/`
- Exponentiation: `^`
- Modulo: `%%`

Modulo is an operation that will return the remainder of the division. For example;

$$11 \bmod 4 = 3$$

This is because 11 divides by 4 (twice) and you are left with 3 remaining.

$$25 \bmod 5 = 0$$

Alternatively, 25 divides into 5 evenly into 5 so you are left with no remainder.

Try

> You can try out the following operations in the **Console** window in R studio.

```
4 + 5
24 - 8
4 * 4
11 / 3
11 %% 3
```

Alternatively, we can use our **R Markdown** file we created to do these operations as well.

3.2 Code Chunk

To use the **R Markdown** file we will need to create a ***Code Chunk***. For **R Markdown** files, each line outside of a code chunk will be text. To execute and run your code, it will need to be inside a code chunk.

To create a code chunk you can go to the top and find `Code` and then click on `Insert Chunk`.

NOTE

For Mac users the shortcut for inserting a code chunk is:

```
Command + Option + i
```

For Windows users the shortcut for inserting a code chunk is:

```
Ctrl + Alt + i
```

Try

> Try to create a code chunk using either the menu at the top to insert or keyboard shortcuts in the source window where your *R Markdown* file is. You should see a new section appear like the image below

You we can re-run the same operations from before in this code chunk instead of running it in the console. When you are ready, you can click on the green arrow at the top right of the code chunk to execute the entire chunk. The answers will be evaluated in order right beneath the code chunk.

Try

~~~~
```{r}

```
~~~~

Figure 3.1: RStudio code chunk

Try running the same equations as before but this time in the code chunk. Use the green arrow on the top right of code chunk to evaluate all the equations in the chunk.

~~~~
```{r}
4 + 5
24 - 8
4 * 4
11 / 3
11 %% 3
```

[1] 9
[1] 16
[1] 16
[1] 3.666667
[1] 2
~~~~

Figure 3.2: RStudio code chunk evaluated

3.3 Storing Variables

We can use the code chunk to help us store variables we might want to reuse later on instead of having to type it out each time.

To assign a value of 8 to the variable `var1`, you can use the following commands

> var1 <- 4

or

> var1 = 4

Try

> Create a new code chunk and try storing our previous results to the variables a-e

```
a <- 4 + 5
b <- 24 - 8
c <- 4 * 4
d <- 11 / 3
e <- 11 %% 3
```

> Note: When you assign a variable, RStudio will store it in the environment panel.

Once your variable is assigned you can recall the result by calling the variable. We can use the use these variables directly in the console or together in the *R Markdown* file.

Try

> Try recalling the new variables in both the **console** and in a **code chunk**. Create another new code chunk and just type in the variable name. Click on the green arrow when you are ready to evaluate

3.4 Using Variables

We can also perform the same mathematical operations using stored variables instead of needing to write out.

Try

> Using a **code chunk** or just in the **console**, perform the following operations.

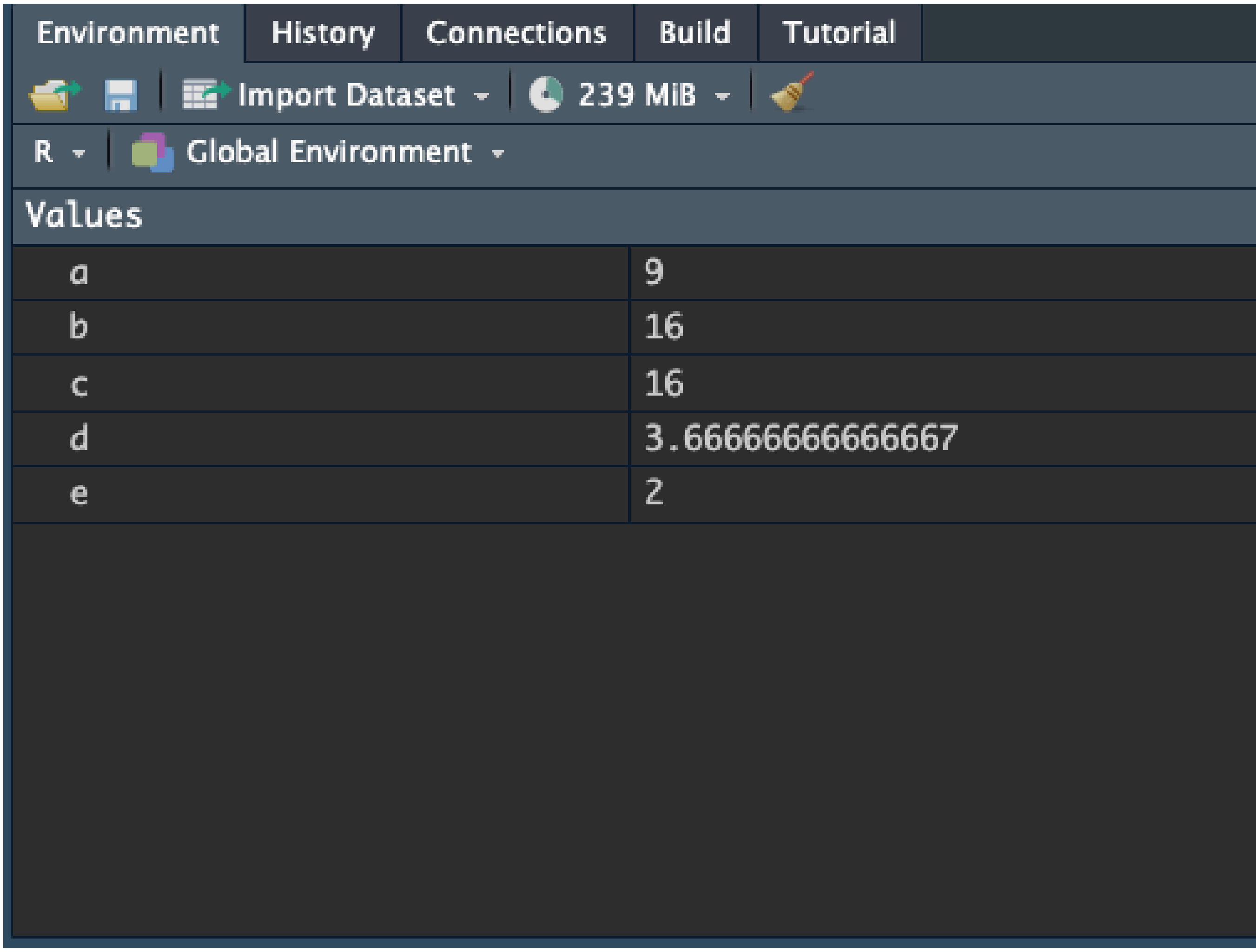


Figure 3.3: Updated environemnt

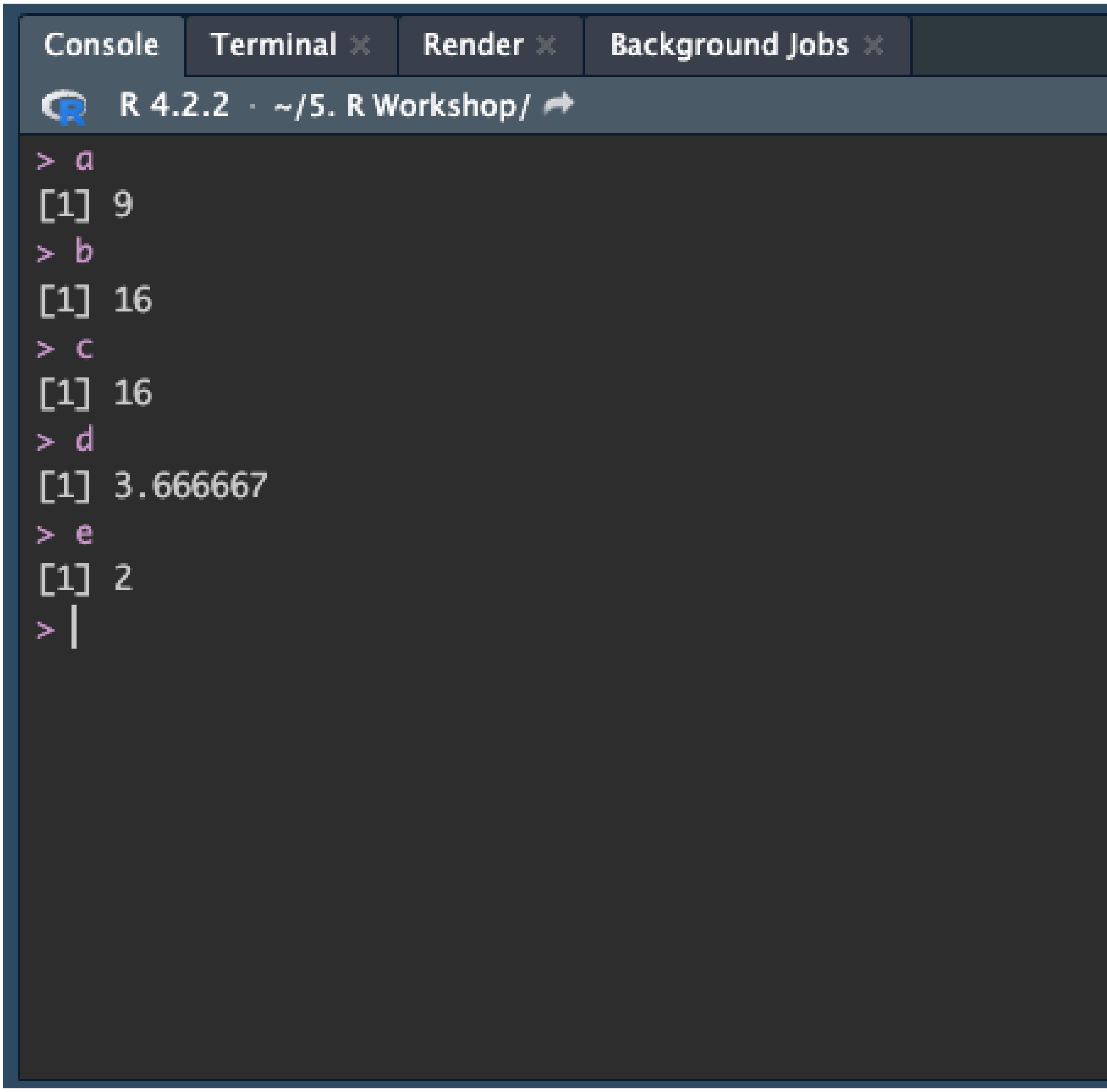


Figure 3.4: Using variables console

```
a + a
b - c
a * d
b %% c
```

3.5 Logical operators

R can also perform logical operations as well that include

- Less than: `<`
- Less than or equal to: `<=`
- Greater than: `>`
- Greater than or equal to: `>=`
- Exactly Equal to: `==`
- Not equal to: `!=`
- OR: `|`
- AND: `&`

In any type of programming you do, you will likely run into these logical operations. You will commonly see these types of operators when we are cleaning and preparing data sets for analysis. For health data, we can use logical operators to help us determine disease status or help us separate age groups.

Try

> Try using the following logical operators on the variables that we created in the *console* or in a *code chunk*

```
a > b
b == c
e <= b
```

3.6 Data types

There are different data types you will run into while you are working on a data set including

- **Numeric**: All real numbers with or without decimals `8.4`
- **Integers**: Whole numbers `29`
- **Logical**: Boolean values `TRUE or FALSE`
- **Characters**: Characters or String values. A single letter is a character `A`. A word or a sentence would be a string `Orange`

It is important to know the data type you are working with since some of the common mistakes in cleaning and working with a data set is trying to combine data types that are not compatible.

Try

> Let's create a new code chunk and create one of each variable time and try to perform some arithmetic operations on them to see what happens.

Note

> Logical variables: The proper syntax is all caps TRUE or FALSE

> Characters: For strings and characters, you need to surround the word or the letter with single or double quotations. "Apple" "Orange" 'Cat'

```
new_num <- 8.4

new_int <- 29

new_logi <- TRUE

new_stringA <- "R Workshop"

new_stringB <- "2023"
```

Note

> A quick way to check what type of variable you are dealing with is to use the class function. i.e., class(new_num)

```
new_num + new_int

new_stringA == new_stringB
```

To combine strings together we want to use the `paste()` function

```
paste(new_stringA, new_stringB)
```

Chapter 4

Data Structures

So far we've learned some basics of what you can do in *R* and *R Studio* including the creation and storage of variables. When processing data sets, we need to use data structures for processing, retrieving and storing data. These data structures are

- **Vectors**: Elements of the same type
- **Lists**: Contains elements of different types. Can contain store numerical values, strings and characters all together.
- **Matrices**: arranged in a 2d layout with rows and columns. s
- **Data frames**: a 2d table-like structure where each column can have a different data type.
- **Factors**: Used to categorize the data and store it in levels.

4.1 Vectors

This is a one dimensional data structure where all the elements in the vector are the same. Similar to vectors that are found in mathematics.

Imagine you are. Imagine that you're going on a vacation and you need to pack all the essentials in your luggage. To make most space you want to use packing cubes and pack all your similar items together. So all your shirts go in one cube, all your pant in another and, all your electric devices in the third one. You can think of your packing cube as a vector in R, and the items you're packing as the elements of the vector.

4.1.1 Creating simple vectors

Try

> Try creating a vector using the **c( )** function.

```
num_vector <- c(2,0,2,3)

shirt_vector <- c("Grey shirt","White Shirt","Black Shirt")

pant_vector <- c("Blue jeans", "Beige chinos", "Black shorts")

electronic_vector <-c("Phone charger", "Phone cable", "Laptop")
```

4.1.2 Simple vector operations

One thing that's unique about vectors in R is that you can perform operations on them all at once. Because of your organized packing you are able to double the number of shirts you can pack. If **num_vector** represents the number of each shirt, you can simply multiply this by 2 and you can increase the number of shirts you are bringing with you.

Try

> Try doubling the **num_vector** variable you created. Also, try to see what happens when you try to double the **shirt_vector**.

4.2 Lists

A list in R is like a container that can hold any type of data. So in this case, your luggage will be a list because it used to hold all the the different items together (shirts, pants, electronics). We will have a luggage list which contains a shirt vector, a pant vector and electronic vector.

4.2.1 Creating a list of vectors

Try

> Try to make the luggage list with the our previous vectors by using the **list( )** function

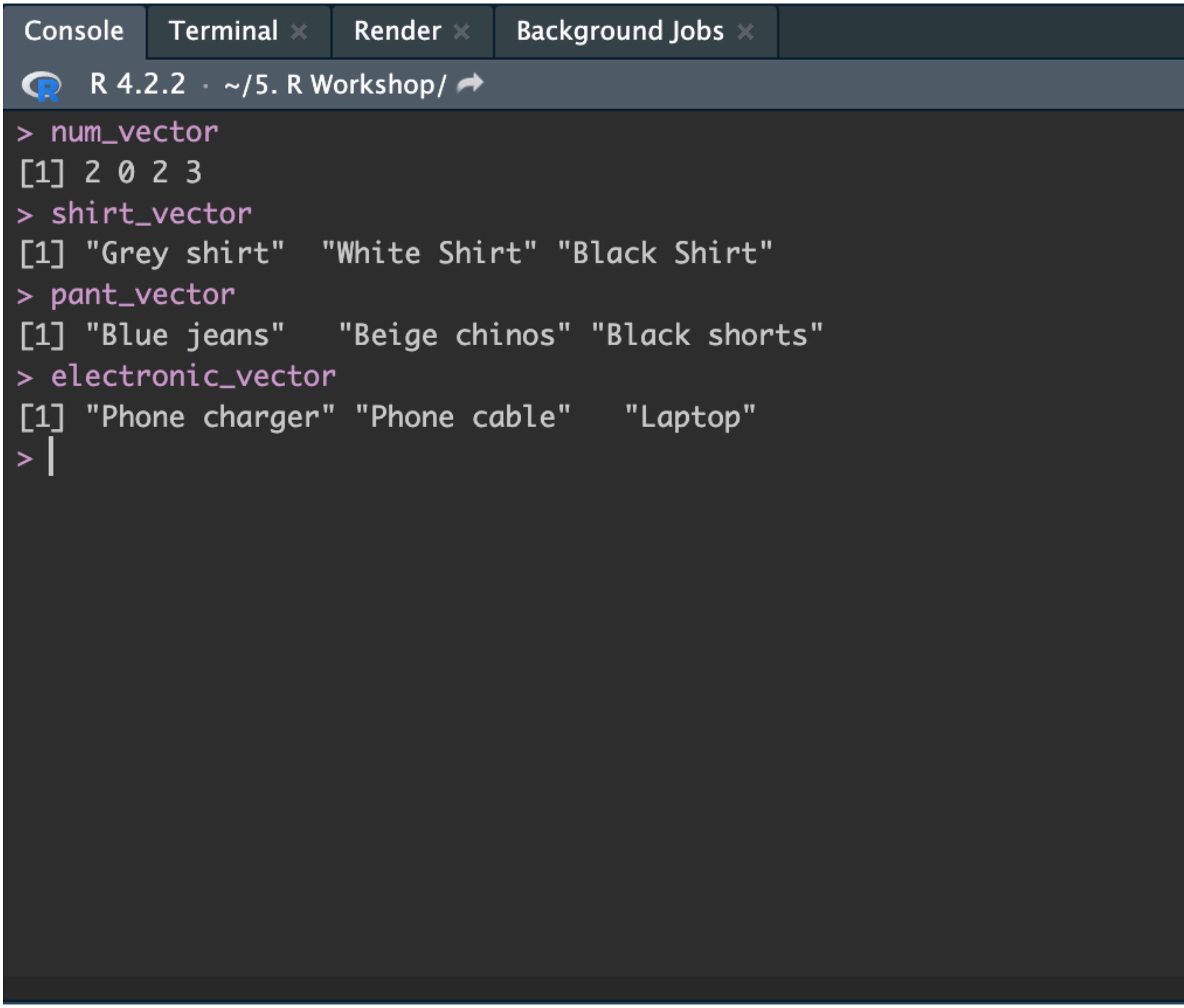


Figure 4.1: Basic Vectors

```
luggage_list <- list(shirt_vector, pant_vector,electronic_vector)
```

4.2.2 Accessing elements in the list

Notice the output we get when we call our list. Our shirts are the first item in our list, pants the second and electronics the third. If you want to check what **jeans** you packed in your luggage list, you can use **luggage_list**[[**2**]]

Try

Try accessing the pants section in our luggage list.

4.2.3 Adding items to lists

4.2.3.1 Add to the end

Let's say you need to add another packing cube but this time it has all your toiletries. To do this you need try the following:

```
luggage_list[[length(luggage_list)+1]] <- c("Toothbrush", "Toothpaste","Floss")
```

If we were to check our list again, we can see that our new toiletry vector has been added to the end of the list.

4.2.3.2 Add to the front

We still have a little bit of space in our luggage and decide to pack some shoes. If we want to add shoes to the front our list try the following

```
shoes_vector <- c("Running shoes","Sandles")

luggage_list <- c(list(shoes_vector), luggage_list)
```

Try

> Try adding toiletries vector to the end of the list and the shoes vector to the front of our luggage list using the code provided above.

Your luggage list should now have shoes, shirts, jeans, electronics and toiletries.

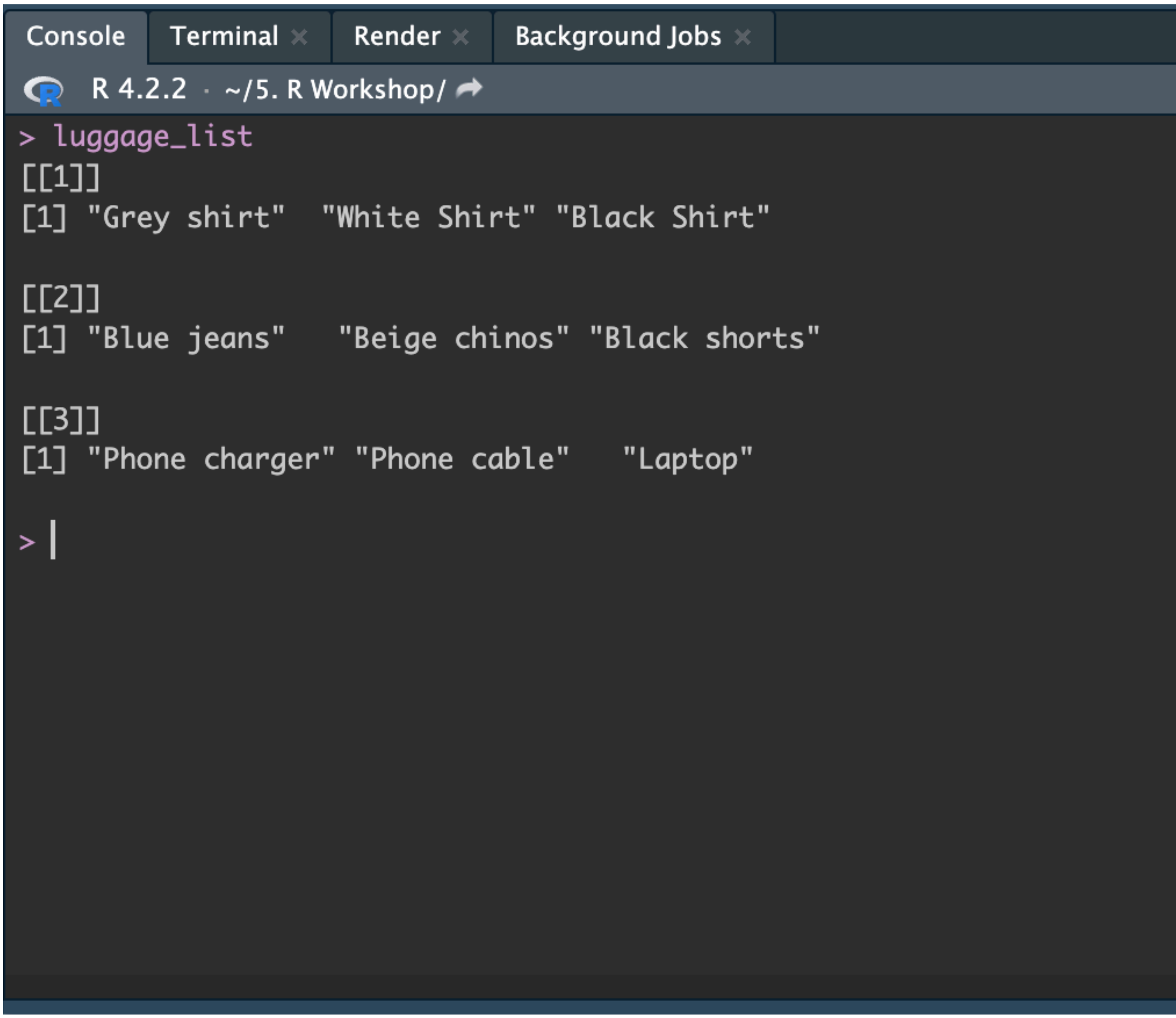


Figure 4.2: Basic Lists

4.2.4 Labeling items within lists

Now that our list has grown, we should label each item incase we forget the of our items. To rename the items in the list you can use the **names( )** function.

```
names(luggage_list) <- c("Shoes","Shirts","Pants","Electronics","Tolietries")
```

Now we can we can access our items by using the **$** symbol which make it easier to check what lists we have.

Try

> Try accessing the **Electronics** vector in our luggage_list using the **$** Symbol.

We have created a **luggage_list** that has 5 vectors that are labeled shoes, shirts, pants, electronics and tolietries.

4.3 Matrices

A matrix 2d data structure that contains rows and columns. Matrices can only contain elements of the same data type, so all the elements in the matrix must be either numeric, character, or logical. Matrices are useful for organizing and manipulating data in a structure and efficient manner since we are able to perform mathematic operations on them, like linear algebra.

In our current example, we can consider a matrix as a packing checklist where each row represents a particular item to pack (such as shirts, pants, or shoes) and each column represents a each of our travel partners luggage. The elements of the matrix could then represent the quantity of each item to pack in each suitcase

4.3.1 Creating a matrix

We can generate a matrix using the **matrix( )** function

```
packing_matrix <- matrix(0,nrow = 5, ncol=3)

rownames(packing_matrix) <- c("Shoes", "Shirts", "Pants", "Electronics", "Toiletries")

colnames(packing_matrix) <- c("My_Luggage", "Traveler_2", "Travler_3")

print(packing_matrix)
```

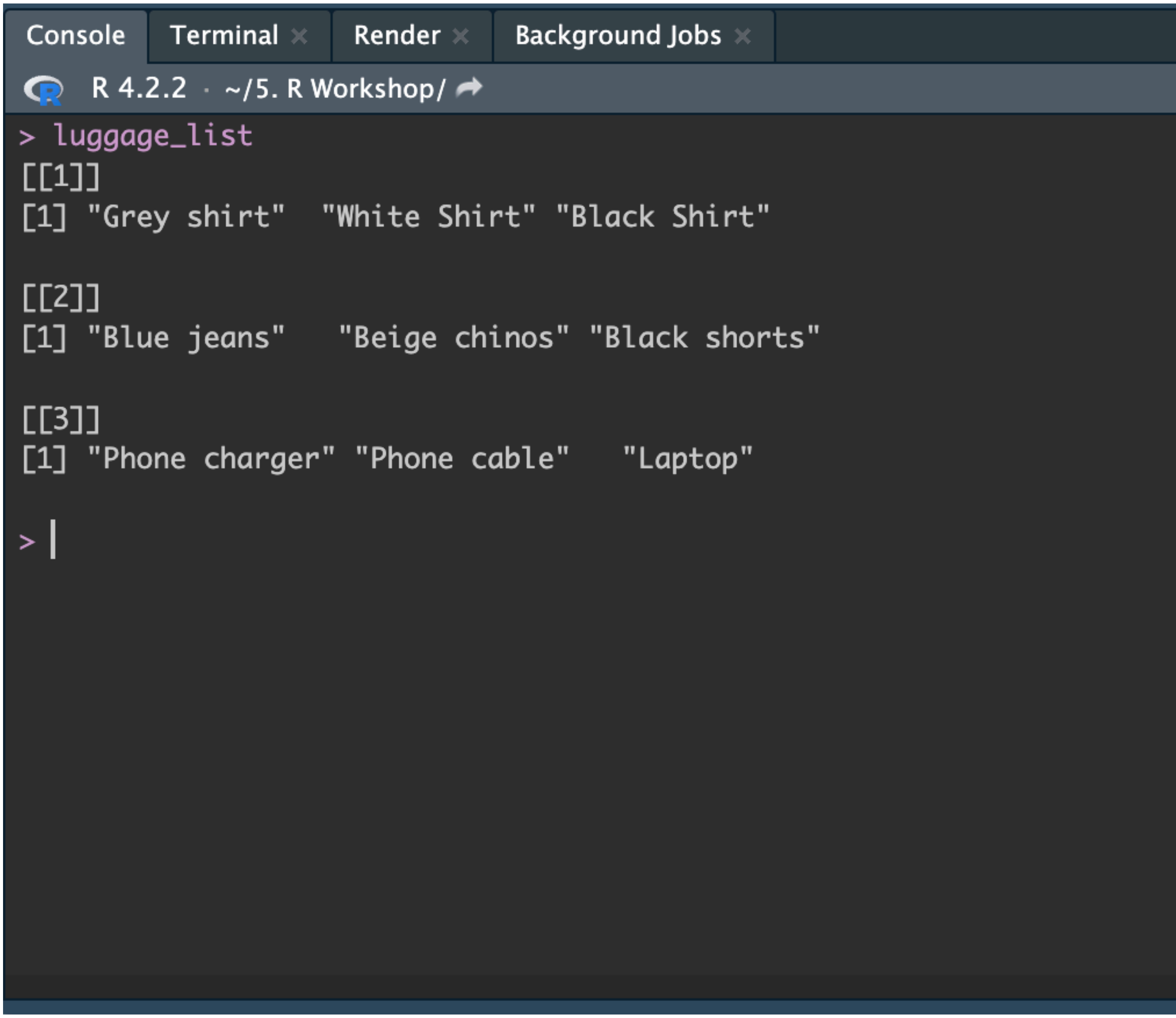


Figure 4.3: Basic Lists

```
##             My_Luggage Traveler_2 Travler_3
## Shoes                0          0         0
## Shirts               0          0         0
## Pants                0          0         0
## Electronics          0          0         0
## Toiletries           0          0         0
```

Try

> Try creating a packing matrix using the code provided above

4.3.2 Navigating the matrix

4.3.2.1 Filling in values

Now that we have created our matrix, we can access certain columns and rows by indexing which is done using square brackets *[ ]*. The synatax for using square brackets would be **matrix[row,column]**. Currently, we have no values in our matrix but we can fill them using indexing.

Let's say you ended up packing 2 shoes, 6 shirts, 3 pants, 4 electronics, and 2 toiletries. To add this to your matrix, you would create a vector and then pass that vector into the first column using the indexing syntax

```
packing_matrix[, 1] <- c(2, 6, 3, 4, 2)
```

Now when check our matrix we should have the first column filled out with the number of items that we packed.

Try

> Try filling putting values for Traveler_2 and Traveler_3. You can select them yourself or just generate them randomly.

Note

> To randomly generate some numbers we can use the sample( ) function.
>
> i.e., sample(1:8, 5, replace = TRUE)

```
packing_matrix[,2] <- sample(1:8, 5, replace = TRUE)
packing_matrix[,3] <- sample(1:8, 5, replace = TRUE)
```

Now that our have filled our packing matrix, we can easily showcase how to access columns and rows in a matrix.

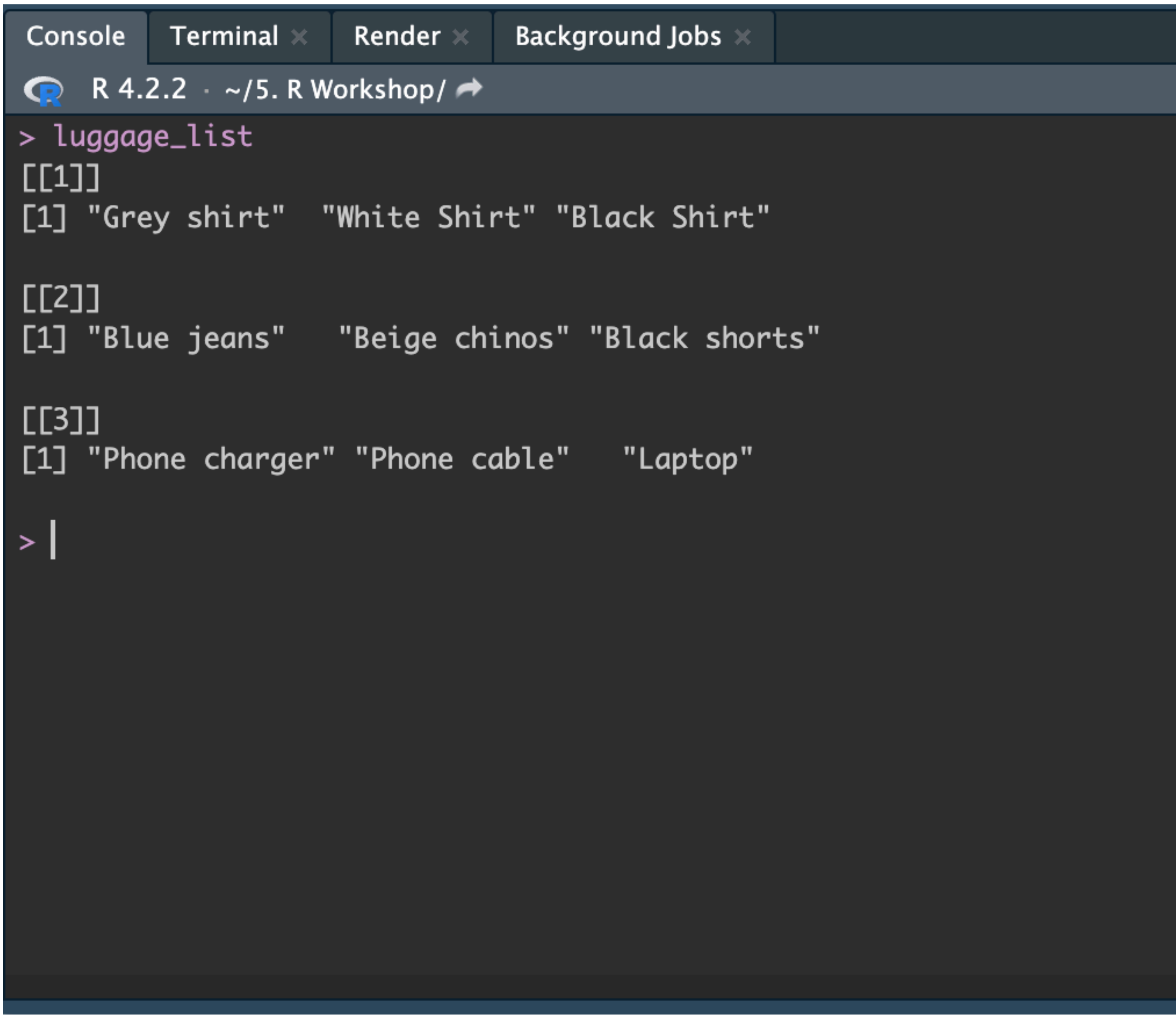


Figure 4.4: Matrix filled

4.3.2.2 Selecting columns

Let's say we want to double check all the things that were packed for myself. To index a column of a matrix you simply have to use the square brackets. Since we know we are the first column in the matrix we can use **packing_matrix[,1]** to find out what we packed.

```
packing_matrix[,1]
```

```
##       Shoes      Shirts       Pants Electronics  Toiletries
##           2           6           3           4           2
```

Try

> Try see what Traveler_2 and Traveler_3 packed

4.3.2.3 Selecting rows

Let's say we are curious on how many shirts each person going on this trip packed. We will still use square brackets, but this time we will be indexing the row instead of the column. **packing_matrix[1,]**

```
packing_matrix[1,]
```

```
## My_Luggage Traveler_2  Travler_3
##          2          3          2
```

4.3.2.4 Multiple selections

If we want to see how our packing compares to our travelling partners packing we can use a vector to index 2 columns at the same time.

```
packing_matrix[,c(1,3)]
```

```
##             My_Luggage Travler_3
## Shoes                2         2
## Shirts               6         1
## Pants                3         4
## Electronics          4         1
## Toiletries           2         7
```

We can also index a certain range instead of selecting specific rows or columns. Let's say we want to check the what each person packed for shoes, shirts and pants. We could index using a vector by putting all 3 numbers or we can use a colon *:* to check the range.

```
packing_matrix[1:3,]
```

```
##        My_Luggage Traveler_2 Travler_3
## Shoes           2          3         2
## Shirts          6          4         1
## Pants           3          1         4
```

4.4 Dataframes

Dataframes is a very popular data structure in R since they are easy to work with and allows you do organize and work with data very effciently. A dataframe is another tabular object like the matrix but the difference between the two is that you can store different types of data in a dataframe. Think of it simliar to an excel spreadsheet where you can different types of data for each column (age, gender, income, etc.).

4.4.1 Creating a dataframe

So with our vacation example, we can use a dataframe to keep track of the preferences of each traveler with the following variables.

- Age (numerical)
- Gender (factor)
- Budget (numerical)
- Number of luggages (numerical)
- Weight of luggages (numerical)
- Food allergies (string)
- Activities (string)
- Must see places (string)

Note

> To create a dataframe you can use the function **data.frame( )**

```
travelers <- data.frame(
  Age = c(25, 30, 35),
  Gender = factor(c("Female", "Male", "Non-binary"), levels = c("Male", "Female", "Non-
  Budget = c(1500, 2500, 2000),
  Num_luggages = c(2, 3, 1),
  Weight_luggages = c(20, 15, 25),
  Food_allergies = c("Peanuts, shellfish", "Gluten, dairy", "None"),
  Activities = c("Hiking, sightseeing", "Museums, beach", "Shopping, nightlife"),
  Must_see_places = c("Eiffel Tower, Colosseum", "Statue of Liberty, Grand Canyon", "Go
)

print(travelers)
```

```
##   Age     Gender Budget Num_luggages Weight_luggages     Food_allergies
## 1  25     Female   1500            2              20 Peanuts, shellfish
## 2  30       Male   2500            3              15      Gluten, dairy
## 3  35 Non-binary   2000            1              25               None
##            Activities                  Must_see_places
## 1 Hiking, sightseeing          Eiffel Tower, Colosseum
## 2      Museums, beach  Statue of Liberty, Grand Canyon
## 3 Shopping, nightlife Golden Gate Bridge, Machu Picchu
```

4.4.2 Using the a dataframe

4.4.2.1 Manipulating data

Like spreedsheets, we can manipulate the dataframe to create new variables. If we wanted to find out the average weight of the luggages we can use build in mean function

```
mean(travelers$Weight_luggages)
```

```
## [1] 20
```

We can also find the median as well using the median function

```
median(travelers$Weight_luggages)
```

```
## [1] 20
```

4.4.2.2 Subsetting data

If we don't want all the columns, we can subset what we need into a new dataframe using square brackets **df[row,col]**. If we wanted to to look **Age**, **Gender** and, **Budget** we can use the following code.

Note:

> This is using base R, we will be using a package later on called 'dplyr' to also subset the data

```
travelers[,1:3]
```

```
##   Age     Gender Budget
## 1  25     Female   1500
## 2  30       Male   2500
## 3  35 Non-binary   2000
```

Note:

> If we didn't know the names the columns we can use the **names( )** function to find out the column names.
>
> If we knew the names, we can also subset using a vector and the names of the columns we want to subset **travelers[,c("Age","Gender","Budget")]**

4.4.2.3 Filtering data

Let's say that we are only interested in those who have a budget that is $<$ **2500**. We can use the logical statements that were introduced in Chapter 3 to do this.

```
travelers[travelers$Budget <2500, ]
```

```
##   Age     Gender Budget Num_luggages Weight_luggages     Food_allergies
## 1  25     Female   1500            2              20 Peanuts, shellfish
## 3  35 Non-binary   2000            1              25               None
##             Activities                  Must_see_places
## 1 Hiking, sightseeing          Eiffel Tower, Colosseum
## 3 Shopping, nightlife Golden Gate Bridge, Machu Picchu
```

Try

> Try to subset the dataframe for **Age** $>$ **25**

```
travelers[travelers$Age >25, ]
```

```
##   Age     Gender Budget Num_luggages Weight_luggages Food_allergies
## 2  30       Male   2500            3              15  Gluten, dairy
## 3  35 Non-binary   2000            1              25           None
##            Activities                  Must_see_places
## 2      Museums, beach  Statue of Liberty, Grand Canyon
## 3 Shopping, nightlife Golden Gate Bridge, Machu Picchu
```

4.5 Factors

Factors are used to represent categorical variables such as **Gender** or **Income levels**. Using the *factor( )* function, we can change text data types to factor data types and use built in-functions to work with categorical data.

You may have noticed before when we created our travel dataframe that gender was coded using the *factor( )*

```
factor(c("Female", "Male", "Non-binary"), levels = c("Male", "Female", "Non-binary"))
```

```
## [1] Female     Male       Non-binary
## Levels: Male Female Non-binary
```

we can use the table function to show us the number observations in each category.

```
table(travelers$Gender)
```

```
##
##       Male     Female Non-binary
##          1          1          1
```

the *levels( )* function will show the order of our categorical variable. In our **travelers** dataframe, we set the levels as **Male, Female, Non-binary**. If want to know the integer representation we can use the **as.numeric( )** function to show us the order. In our example, the first traveler is **"Female"**, second is **"Male"** and, third is **"Non-binary"**.

```
as.numeric(travelers$Gender)
```

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

Chapter 5

Data Wrangling

5.1 R Packages

Packages are a collection of functions that extend the functionality of R. They are tools that help with data analysis, modelling, data visualization. Some of the most common packages in R is **ggplot2** for data visualization, **dplyr** for data wrangling/manipulation and *caret* for machine learning.

5.1.1 Installing Packages

To use packages we will have the the **install.packages( )** function and put in the package. We can either does this as a *chunk* in a R-markdown file or we can type it directly into the console.

Try

> Try installing **dplyr** and **ggplot2** packages

Note

> You can install multiple packages are the same time if you put then in a vector and before using the **install.packages( )** function.
>
> i.e., install.packages(c('dplyr','ggplot2'))

5.1.2 Using Packages

5.1.2.1 Loading Packages

After installing our packages, we have to load them using the **library( )** function or the **require( )** function. This is similar to opening up a new app or program that you just installed.

```
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
```

5.2 dplyr

The **dplyr** is used for data manipulation and can allow us to work with data more efficiently than with base R. We can do the same operations in our previous *travelers* dataframe using **dplyr**.

5.2.1 Pipe function (%>%)

This is one of the most powerful functions in the **dplyr** package. What the pipe function does is takes the output of one function and then passes it along to the next one. So instead of saving results to multiple variables, we can perform a sequence of commands in as one command. Using our previous **travelers** dataframe, we were able to calculate the *mean luggage weight* for all travelers but what if we are interested in the total weight of the luggage each traveler is bringing? We can do this in **dplyr** using the pipe function (%>%) and **mutate( )** function, which creates a new column. We can also subset the data as in the same function as well. Let's say we are only interested in all the numerical values, we can use the **select** function to ask it to select all the columns that hold numerical data.

```
travelers %>%
  mutate(mean_weight = Weight_luggages/Num_luggages) %>%
  select(c(where(is.numeric)))
```

```
##   Age Budget Num_luggages Weight_luggages mean_weight
## 1  25   1500            2              20          10
## 2  30   2500            3              15           5
## 3  35   2000            1              25          25
```

You can see our new column **mean_weight** added to the end and that the dataframe only contains numerical values. If we were to do this without the pipe function, we would first need to get add the **mean_weight** column, save it, then subset our data for all numeric values including our new column.

Lets say we want to filter our data finding travelers with a **age >25** or if they have a **budget >2500**. We can do this all together in one command using dplyr using the **filter( )** command.

```
travelers %>%
  filter(Age > 25 | Budget > 2500)
```

```
##   Age     Gender Budget Num_luggages Weight_luggages Food_allergies
## 1  30       Male   2500            3              15  Gluten, dairy
## 2  35 Non-binary   2000            1              25           None
##            Activities                    Must_see_places
## 1      Museums, beach  Statue of Liberty, Grand Canyon
## 2 Shopping, nightlife Golden Gate Bridge, Machu Picchu
```

Note

> The / is the logical statment **OR** in R. *&* is the logical statment for **AND**

We can see that we have 2 entries that are either **Age >25** or have a **Budget >2500**

There are a lot more functions that are part of dplyr package. The most common ones for data manipulation are

- mutate() - Adds a new variable to your dataframe based on existing variables
- select() - Picks variables from a dataframe based on their names
- filter() - Picks out cases based on criteria

- summarise() - Reduces down values down to a single summary
- arrange() - Changes the order of the rows based

For more information or more practice you can go to the dplyr website which also has a R-bookdown on data transformation.

https://dplyr.tidyverse.org/

5.3 Loading datasets

More often than not, we will be working with a datasets instead of creating our own. In **R** we can load the different types of files, the most common being a comma separated file (**CSV**). Now that we have our travelers preferences and budget we need to find a destination that would be suitable for each of our travelers. We will be using a few datasets from **Numbeo.com** and **The World bank**.

We want to save these as variables so we can access them later.

To load a dataset, we will use the **read.csv( )** function and save the dataset as a variable in our environment. The first one we want to load is from **numbeo.com** which shows us the cost of a inexpensive meal at a restaurant.

Note

> The data from **numbeo.com** is aggregated by user submissions and calculates the average costs of certain products

```
meals <- read.csv("data/numbeo_meal.csv")
```

Another way to load your data is using the import function in R studio.

When you select the file you want to import a window will appear and you can do some quick modification to our data if needed.

Note

> One of the most common issues when using this feature is the setting the headers. There is a radial button for the header which you can toggle on and off.

Try

> We will be making use of the ticket, arrivals and, country codes dataset. You can try loading these files with the interface or using a code chunk.

```
ticket <- read.csv("data/numbeo_ticket.csv")
arrival <- read.csv("data/world_bank_arrival.csv")
country_codes <- read.csv("data/country_codes_iso.csv")
```

5.4 Understanding your dataset

Understanding your datasets is important when performing data analysis. The more familiar you are with the shape of your dataset, the more insights you are able to pull out from it. This includes knowing the number of rows and columns as well as the types of columns that you are working with.

We can use the **View( )** function in order to open up a new tab to view the dataset like an excel sheet.

Note

> This function is case sensitive and must be called with a uppercase **V**

```
View(meals)
```

To quickly look at the first 6 rows of the data we can use the **head( )** function.

```
head(meals,3)
```

```
##         Country Meal_Inexpensive_Restaurant
## 1   Switzerland                       37.35
## 2       Denmark                       27.58
## 3 United States                       26.72
```

Note

> You can specify the number of rows you want to examine by adding the number to function
>
> i.e., **head(meals,4)**

You can also view the last rows of the column using the **tail( )** function

```
tail(meals,3)
```

```
##       Country Meal_Inexpensive_Restaurant
## 103 Sri Lanka                        2.05
## 104   Nigeria                        2.02
## 105  Pakistan                        1.65
```

We can do some quick summary statistics using the **summary( )** function as well. Depending on the type of data in each column, we can see the length or a number summary including the min, max, mean, median, mode, and the 1st and 3rd quartiles.

```
summary(meals)
```

```
##    Country          Meal_Inexpensive_Restaurant
##  Length:105         Min.   : 1.65
##  Class :character   1st Qu.: 5.87
##  Mode  :character   Median :10.21
##                     Mean   :11.53
##                     3rd Qu.:15.52
##                     Max.   :37.35
```

5.5 Combining your dataset

Currently we have 4 separate datasets that contain all the information we need. Our goal is to combine all the dataset to a singular one which we can start our data analysis. To do this need something that is unique for all the entries. In our case, the ISO country code and the country name are something that is unique for each entry.

We will be using the country_codes dataset as our base.

5.5.1 Binds

There are different types of ways to join your dataset depending on your desired outcome. Sometimes we just need to add columns or rows to the base dataset. For this, we can use either **rbind( )** or **cbind( )** if the conditions are appropriate.

- rbind() - row bind will add rows to the base dataset if they number of rows and the row names are the same.
- cbind() - column bind will add more columns to the base dataset if the they have the same number of rows.

5.5.2 Joins

Most times, we will have an unequal amount of rows or columns and we want to match with something unique to eaach row. For this we can utilize joins. There are different types of joins you can use that all perform their task differently.

- left_join() - Matches all paired variables to the left dataframe
- right_join() - Matches all paired variables to the right dataframe
- inner_join() - Returns a dataframe with only matching variables. If their is no matching variable, it does not get included.
- full_join() - Keeps all variables from both dataframes even if they do not match.

Note

> left_join() is the most common join you will be using during a data analysis

Let's try to combine all 4 datasets into one working dataset. First we want to combine our *meal* and *ticket* dataframes since they have the same number of rows. We will use a *left_join( )* to combine these based on the country name.

Try

> Try to perform a *left_join( )* on the *meal* and *ticket* based on *Country* name.

```
travel_meal_tickets <- left_join(meals,ticket,by = "Country")
head(travel_meal_tickets,3)
```

```
##         Country Meal_Inexpensive_Restaurant one_way_ticket_local
## 1   Switzerland                       37.35                 5.38
## 2       Denmark                       27.58                 4.73
## 3 United States                       26.72                 3.34
```

Using a left join, we now have a single dataframe with 3 columns, Country name, the price for the restaurants and price for a one way ticket.

Next, lets combine this dataframe with the country codes dataframe. Looking at the **country code** dataframe, we have 4 columns and 249 rows. We do need some cleaning on this dataframe before continuing. Let's keep the Country column and just the 3 digit alpha code using **dplyr** and the **select( )** function

```
country_code_subset <- country_codes %>%
  select(c(Country,alpha_3_code)) %>%
  rename(country_code = alpha_3_code)

head(country_code_subset,3)
```

```
##       Country country_code
## 1 Afghanistan          AFG
## 2     Albania          ALB
## 3     Algeria          DZA
```

We want to use our new subset of the country_codes as our base model and join the meal/ticket dataframe to this one. Since the country code has 249 rows and the meal/ticket one has 105 rows. We want to use a full join here because we aren't sure of the country names are spelled the same in each dataframes. Using the **filter( )** function, we can filter out which alpha_3_codes are missing and we notice there is just one which is Kosovo.

```
country_ticket_meal_code <- full_join(country_code_subset,travel_meal_tickets, by = "Co

head(country_ticket_meal_code,3)
```

```
##       Country country_code Meal_Inexpensive_Restaurant one_way_ticket_local
## 1 Afghanistan          AFG                          NA                   NA
## 2     Albania          ALB                        7.79                 0.52
## 3     Algeria          DZA                        2.96                 0.25
```

Using the **filter( )** function, we can filter out which alpha_3_codes are missing and we notice there is just one which is Kosovo.

Note

> Kosovo is not country recognized by ISO 3166 standards.

Finally we want to join our last database which is the from the world bank that contains the number of arrivals to the country from 1960 - 2021 however, we need to perform some data cleaning before we can combine them together. Having a look at our dataframe we see each columns for years has a ***X*** in front of it. We will want to rename that by using the **rename_all( )** which is part of the **dplyr** package.

Note

> The period **(.)** being used here is a special character here that means for this "For this current dataframe"

```
arrival_clean <- arrival %>%
  rename_all(~stringr::str_replace(.,"^X",""))

head(arrival,3)
```

```
##                        Country country_code X1960 X1961 X1962 X1963 X1964 X1965
## 1                        Aruba          ABW    NA    NA    NA    NA    NA    NA
## 2 Africa Eastern and Southern          AFE    NA    NA    NA    NA    NA    NA
## 3                  Afghanistan          AFG    NA    NA    NA    NA    NA    NA
##   X1966 X1967 X1968 X1969 X1970 X1971 X1972 X1973 X1974 X1975 X1976 X1977 X1978
## 1    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
## 2    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
## 3    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
##   X1979 X1980 X1981 X1982 X1983 X1984 X1985 X1986 X1987 X1988 X1989 X1990 X1991
## 1    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
## 2    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
## 3    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
##   X1992 X1993 X1994    X1995    X1996    X1997    X1998    X1999    X2000
## 1    NA    NA    NA   912000   957000   947000   906000   972000  1211000
## 2    NA    NA    NA 11583545 13088654 13456246 14403852 15309378 15353177
## 3    NA    NA    NA       NA       NA       NA       NA       NA       NA
##      X2001    X2002    X2003    X2004    X2005    X2006    X2007    X2008
## 1  1178000  1225000  1184000  1304000  1286000  1285000  1254000  1383000
## 2 15854696 17383375 17844385 18745951 19917566 22650321 25114898 25413098
## 3       NA       NA       NA       NA       NA       NA       NA       NA
##      X2009    X2010    X2011    X2012    X2013    X2014    X2015    X2016
## 1  1420000  1394000  1469000  1481000  1667000  1739000  1832000  1758000
## 2 25964418 29071501 31650244 32748552 34426633 35738392 35318681 37645888
## 3       NA       NA       NA       NA       NA       NA       NA       NA
##      X2017    X2018    X2019 X2020 X2021
## 1  1863000  1897000  1951000    NA    NA
## 2 38258348 41189145 39826701    NA    NA
## 3       NA       NA       NA    NA    NA
```

```
head(arrival_clean,3)
```

```
##                        Country country_code 1960 1961 1962 1963 1964 1965 1966
## 1                        Aruba          ABW   NA   NA   NA   NA   NA   NA   NA
## 2 Africa Eastern and Southern          AFE   NA   NA   NA   NA   NA   NA   NA
## 3                  Afghanistan          AFG   NA   NA   NA   NA   NA   NA   NA
##   1967 1968 1969 1970 1971 1972 1973 1974 1975 1976 1977 1978 1979 1980 1981
## 1   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA
## 2   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA
## 3   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA
```

```
##   1982 1983 1984 1985 1986 1987 1988 1989 1990 1991 1992 1993 1994     1995
## 1   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   912000
## 2   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA 11583545
## 3   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA       NA
##       1996     1997     1998     1999     2000     2001     2002     2003
## 1   957000   947000   906000   972000  1211000  1178000  1225000  1184000
## 2 13088654 13456246 14403852 15309378 15353177 15854696 17383375 17844385
## 3       NA       NA       NA       NA       NA       NA       NA       NA
##       2004     2005     2006     2007     2008     2009     2010     2011
## 1  1304000  1286000  1285000  1254000  1383000  1420000  1394000  1469000
## 2 18745951 19917566 22650321 25114898 25413098 25964418 29071501 31650244
## 3       NA       NA       NA       NA       NA       NA       NA       NA
##       2012     2013     2014     2015     2016     2017     2018     2019 2020
## 1  1481000  1667000  1739000  1832000  1758000  1863000  1897000  1951000   NA
## 2 32748552 34426633 35738392 35318681 37645888 38258348 41189145 39826701   NA
## 3       NA       NA       NA       NA       NA       NA       NA       NA   NA
##   2021
## 1   NA
## 2   NA
## 3   NA
```

Next, we don't actually really don't want to use all the years but just want to have a look at some of the most recent years. In this dataset, we have values from 1995 - 2020. There are a few ways we can filter this data.

- We can filter by the column type by using a combination of the **where( )** function and the **is.numeric( )** function
- We can specify a specific range of dates using the range operator **(:)**

You can try either method. Remember to save your filtered dataframe to a new variable.

```r
arrival_clean_subset <- arrival_clean %>%
  select(c("Country","country_code", where(is.numeric)))

head(arrival_clean_subset,3)
```

```
##                         Country country_code     1995     1996     1997     1998
## 1                         Aruba          ABW   912000   957000   947000   906000
## 2 Africa Eastern and Southern          AFE 11583545 13088654 13456246 14403852
## 3                   Afghanistan          AFG       NA       NA       NA       NA
##       1999     2000     2001     2002     2003     2004     2005     2006
## 1   972000  1211000  1178000  1225000  1184000  1304000  1286000  1285000
## 2 15309378 15353177 15854696 17383375 17844385 18745951 19917566 22650321
```

```
## 3       NA       NA       NA       NA       NA       NA       NA       NA
##       2007     2008     2009     2010     2011     2012     2013     2014
## 1  1254000  1383000  1420000  1394000  1469000  1481000  1667000  1739000
## 2 25114898 25413098 25964418 29071501 31650244 32748552 34426633 35738392
## 3       NA       NA       NA       NA       NA       NA       NA       NA
##       2015     2016     2017     2018     2019 2020
## 1  1832000  1758000  1863000  1897000  1951000   NA
## 2 35318681 37645888 38258348 41189145 39826701   NA
## 3       NA       NA       NA       NA       NA   NA
```

```
arrival_clean %>%
   select(c(Country,country_code,"1995":"2020"))
```

Now that we have all the dataframes ready to be combined we can decided to use the country code, the country name or both. Using the country code it self would be great but to be sure that we maximize the matching, we can use both country name and country code.

```
travel_full_clean <- inner_join(country_ticket_meal_code,arrival_clean_subset, by=c('country_code
head(travel_full_clean,3)
```

```
##       Country country_code Meal_Inexpensive_Restaurant one_way_ticket_local
## 1 Afghanistan          AFG                          NA                   NA
## 2     Albania          ALB                        7.79                 0.52
## 3     Algeria          DZA                        2.96                 0.25
##      1995   1996   1997   1998   1999   2000   2001   2002    2003    2004
## 1      NA     NA     NA     NA     NA     NA     NA     NA      NA      NA
## 2 304000 287000 119000 184000 371000 317000 354000 470000  557000  645000
## 3 520000 605000 635000 678000 749000 866000 901000 988000 1166000 1234000
##       2005    2006    2007    2008    2009    2010    2011    2012    2013
## 1       NA      NA      NA      NA      NA      NA      NA      NA      NA
## 2   748000  937000 1127000 1420000 1856000 2417000 2932000 3514000 3256000
## 3  1443000 1638000 1743000 1772000 1912000 2070000 2395000 2634000 2733000
##       2014    2015    2016    2017    2018    2019    2020
## 1       NA      NA      NA      NA      NA      NA      NA
## 2  3673000 4131000 4736000 5118000 5927000 6406000 2658000
## 3  2301000 1710000 2039000 2451000 2657000 2371000  591000
```

5.6 Missing values

One aspect that is important when working with any dataset is how we deal with missing data. Our goal is to process our dataset so that we have complete rows of data with no missing or duplicate values. Here are some methods we can use when we are trying to deal with missing values.

- Removing the value from the dataset. We have to consider how many values are missing and will removing them impact the analysis we are trying to achieve.
- Imputation. This involves filling in the missing data with a value that makes sense. You can input using the mean, median or regression. Again, you have to determine if imputing your data makes sense with the data you are working with.
- Statistical models. In some cases we can use the existing data to predict the missing data with regression models.
- Find other sources. Sometimes the dataset you get is incomplete. You can also try to find other data sources to fill it in.

In our cases we have 121 countries with missing values and 55 countries for full analysis. We can use the function **complete.cases( )** to check this.

```
sum(complete.cases(travel_full_clean))
```

```
## [1] 55
```

Note

> The exclamation mark **(!)** is a special character that is used when you want to say "is not"

For this workshop we will only worry about the 55 countries that have complete rows of data. We can use the **na.omit( )** function in order to select all the rows with complete data.

```
travel_full_clean_subset <- na.omit(travel_full_clean)
head(travel_full_clean_subset,3)
```

```
##   Country country_code Meal_Inexpensive_Restaurant one_way_ticket_local   1995
## 2 Albania          ALB                        7.79                 0.52 304000
## 3 Algeria          DZA                        2.96                 0.25 520000
## 9 Armenia          ARM                       10.13                 0.34  12000
##     1996   1997   1998   1999   2000   2001   2002    2003    2004    2005
## 2 287000 119000 184000 371000 317000 354000 470000  557000  645000  748000
## 3 605000 635000 678000 749000 866000 901000 988000 1166000 1234000 1443000
## 9  13000  23000  32000  41000  45000 123000 162000  206000  263000  319000
##      2006    2007    2008    2009    2010    2011    2012    2013    2014
## 2  937000 1127000 1420000 1856000 2417000 2932000 3514000 3256000 3673000
```

```
## 3 1638000 1743000 1772000 1912000 2070000 2395000 2634000 2733000 2301000
## 9  382000  511000  558000  575000  684000  758000  963000 1084000 1204000
##      2015    2016    2017    2018    2019    2020
## 2 4131000 4736000 5118000 5927000 6406000 2658000
## 3 1710000 2039000 2451000 2657000 2371000  591000
## 9 1192000 1260000 1495000 1652000 1894000  375000
```

Our dataset is now ready for the exploration an analysis.

Chapter 6

Data Exploration

Exploratory data analysis (EDA) is the first step in the getting insights from a dataset. In this section we will see look at our cleaned dataset and perform some summary statistics and visualizations to see what would be a good destinations for our travellers to head to.

6.1 Summary statistics

We can make use of the **summary( )** function that was introduced in Chapter 4 to determine the mean and mean costs for **meals** and **one way ticket** costs. We can make use of indexing to select just these two columns.

```
summary(travel_full_clean_subset[3:4])
```

```
## Meal_Inexpensive_Restaurant one_way_ticket_local
## Min.   : 2.050              Min.   :0.210
## 1st Qu.: 6.315              1st Qu.:0.495
## Median :10.530              Median :1.200
## Mean   :11.836              Mean   :1.715
## 3rd Qu.:17.490              3rd Qu.:2.200
## Max.   :26.720              Max.   :5.140
```

We can see that the **median** costs for a meal $10.53 CAD, and the **mean** costs is $11.84 CAD. The lowest priced meals are shown as **min** $2.05 CAD, and the most expensive is **max** $26.72 CAD.

We can use this information to filter out countries that we want to visit based on food or ticket costs. Lets consider filtering out food by the interquartile range (**IQR**). Knowing that the 1st quartile is $6.32 CAD and the 3rd quartile is $17.49 we can use the **filter( )** function and the **between( )** function to select only those countries in this range.

```
travel_full_clean_subset %>%
  filter(between(Meal_Inexpensive_Restaurant,6.32,17.49)) %>%
  select(c(Country,Meal_Inexpensive_Restaurant)) %>%
  arrange(Meal_Inexpensive_Restaurant) %>%
  head(5)
```

```
##    Country Meal_Inexpensive_Restaurant
## 1 Ethiopia                        6.68
## 2  Jamaica                        6.96
## 3  Ukraine                        7.18
## 4   Jordan                        7.53
## 5  Albania                        7.79
```

If we want to rank the previous countries based on meals from least expensive to most expensive we can use the **arrange( )** function. If we would like to go from most expensive to least expensive, we can add the **desc( )** function.

Note

> We can also use the **tail( )** function instead of the **head( )** function to get the most expensive.

```
travel_full_clean_subset %>%
  filter(between(Meal_Inexpensive_Restaurant,6.32,17.49)) %>%
  select(c(Country,Meal_Inexpensive_Restaurant)) %>%
  arrange(desc(Meal_Inexpensive_Restaurant)) %>%
  head(5)
```

```
##       Country Meal_Inexpensive_Restaurant
## 1 Puerto Rico                       17.37
## 2      Sweden                       15.52
## 3      Latvia                       14.67
## 4    Slovenia                       14.67
## 5     Georgia                       14.37
```

On the lower meal price end we have

- Ethiopia
- Jamaica

- Ukraine
- Jordon
- Albania

and the higher end we have

- Puerto Rico
- Sweden
- Latvia
- Solenia
- Georgia

We combining what we the data wrangling from the previous section, we can determine what the average number of arrivals for each country is from the lat 5 years using the functions **mutate( )** and **arrange( )**

```
travel_full_clean_subset %>%
  mutate(
    mean5year = rowMeans(select(., "2015":"2020"))) %>%
  select(c(Country,mean5year)) %>%
  arrange(desc(mean5year)) %>%
  head(5)
```

```
##         Country mean5year
## 1 United States 151042948
## 2         China 130066667
## 3         Spain 105691333
## 4        Mexico  87727000
## 5         Italy  80495100
```

6.2 Standard deviation and variance

We can easily calculate the dispersion of the data using standard deviaton and variance that are built in functions with R.

6.2.1 Standard deviation

Standard deviation explains how far away a group of numbers is from the mean. If we wanted to find the standard deviation for our meals, we can just the **sd( )** function.

Note

> The **sd( )** function will generate the sample standard deviation.
>
> To calculate the population standard deivation we would need to multiple the standard deviation by sqrt((n-1)/n) * sd(x)

```
sd(travel_full_clean_subset$Meal_Inexpensive_Restaurant)
```

```
## [1] 7.042914
```

6.2.2 Varience

Variance is how spread out the data is around the mean. To calculate variance the meals variable we can use the **var( )** function.

Note

> The **var( )** function will generate the sample varience. If you ever need to calculate the population varience we would need to multiple the varience by (n-1)/n * var(x)

```
var(travel_full_clean_subset$Meal_Inexpensive_Restaurant)
```

```
## [1] 49.60264
```

Although it is easy to perform these functions, it is important to understand what you are calculating and if it is appropriate to use that calculation.

Chapter 7

Data visualization

Another way we can represent our insights is through data visualization through different graphs. Visualizations can be important tool in exploratory data analysis for identify patterns in our data. Creating meaningful visualization can help communicate your findings and ideas to a wide audience.

7.1 Plots

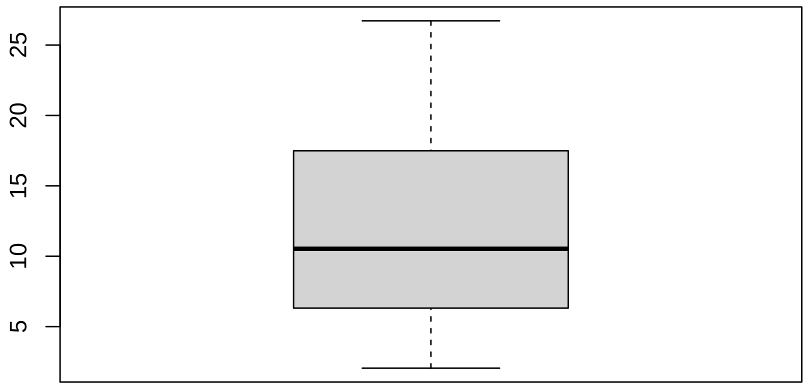
There a few basic plots that can be used for exploratory analysis. In R, we can build most plots using the base R but we will also be exploring a package called **ggplot2**. We will use the display the previous summary statistics using:

- Boxplots
- Histograms
- Scatterplots

7.1.1 Boxplots

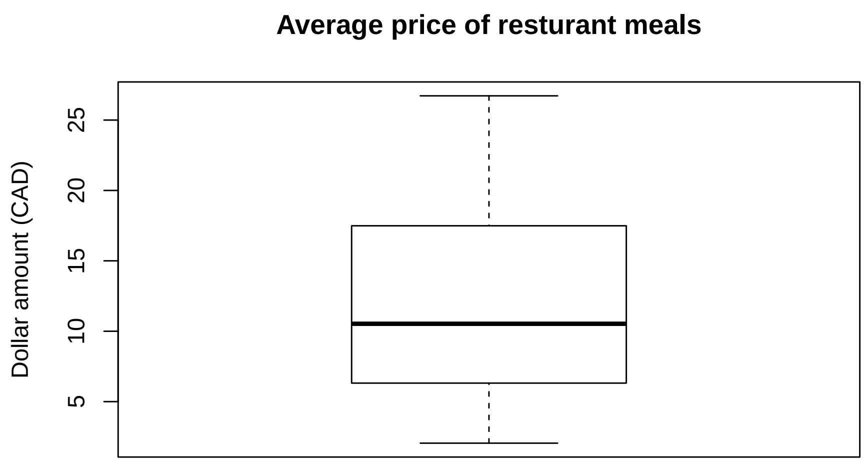
Boxplots are used to summarize the same 5 number summary as the **summary( )** function. They are also a great way to quickly detect outliers in your dataset. To create a basic boxplot with base R, we can use the **boxplot( )** function.

```
boxplot(travel_full_clean_subset$Meal_Inexpensive_Restaurant)
```



With the basic boxplot we can quickly see that no outliers are present in our data and that our data is slight right skewed with the median is below the center of the boxplot. To label our boxplot, we can add the following arguments.

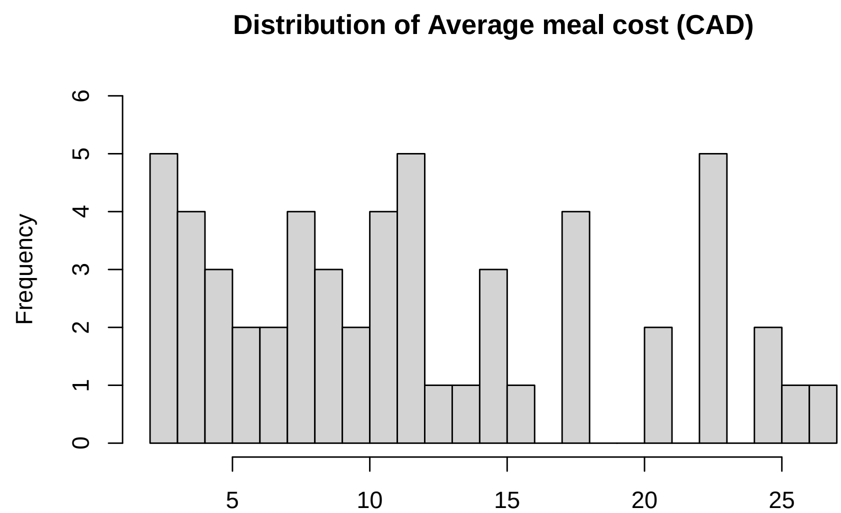
```
boxplot(travel_full_clean_subset$Meal_Inexpensive_Restaurant,
        main = "Average price of resturant meals", # Title of the graph
        xlab = "", # x-axis label
        ylab= "Dollar amount (CAD)", # y-axis label
        col = "white") # color of the boxplot
```



7.1.2 Histogram

Histograms can show the how our data is distributed. We can use the **hist( )** function to do this.

```
hist(travel_full_clean_subset$Meal_Inexpensive_Restaurant,
     breaks= 30, # number of breaks
     ylim = c(0,6),
     main = "Distribution of Average meal cost (CAD)",
     xlab = "") # y-axis limit
```



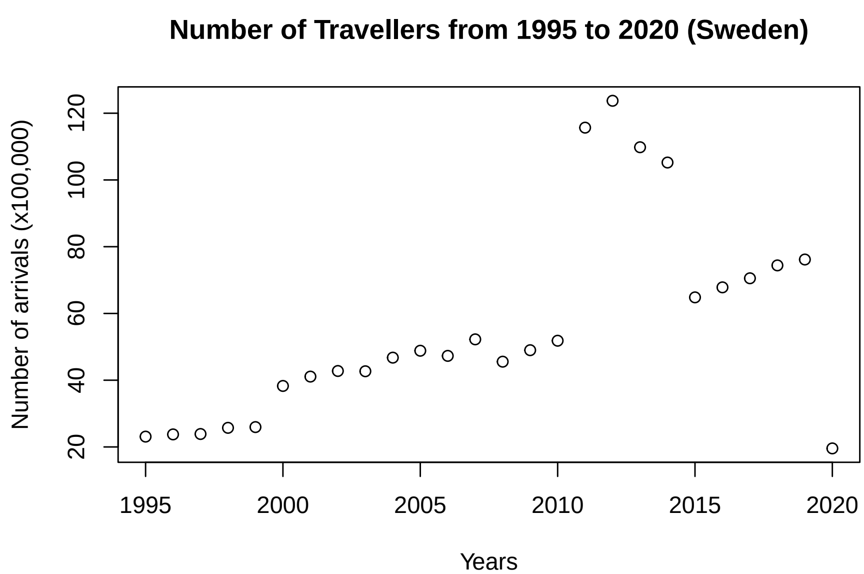
7.1.3 Scatterplots

Scatterplots are useful to show a relationship between two different variables. For example, we can plot the the arrivals over time. Let's examine how Finland has changed over time. There are a few steps we need to take

- Filter our dataset to show data for Finland
- Select the columns related to arrival (1995-2020)
- Pivot our data from wide format to long format
- Rename our columns to meaningful columns
- Draw the scatterplot

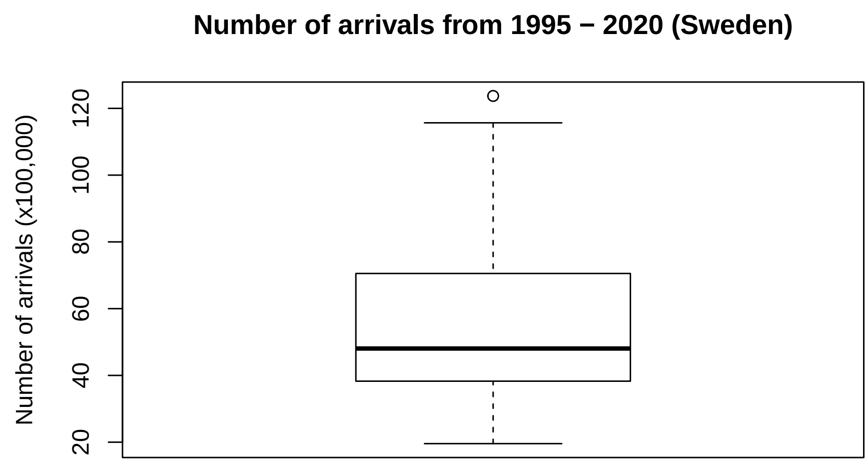
```
library(tidyr)
sweden_travel <- travel_full_clean_subset %>%
  filter(country_code=="SWE" ) %>%
  select(c("1995":"2020")) %>%
  gather() %>%
  rename(years= key,
         arrivals= value)

plot(sweden_travel$years, sweden_travel$arrivals/100000,
```

```
    main="Number of Travellers from 1995 to 2020 (Sweden)",
    xlab="Years",
    ylab="Number of arrivals (x100,000)")
```



From the scatterplot, we see a steady increase in arrivals to sweeden over the years. We do see a few years between 2010, and 2015 with a high spike in arrivals which could indicate a few outliers in our data. To check we can run another boxplot on the arrivals data.

```
boxplot(sweden_travel$arrivals/100000,
        main = "Number of arrivals from 1995 - 2020 (Sweden)", # Title of the graph
        xlab = "", # x-axis label
        ylab= "Number of arrivals (x100,000)", # y-axis label
        col = "white") # color of the boxplot)
```



From the boxplot, we can see that there is at least 1 point that is an outlier in the data indicated by the circle above the maximum value

7.2 ggplot2

You can use the **ggplot2** package to create plots and figures instead of using the base R. It gives you more control over your plots to specify how you want it to look. Let's remake the previous 3 plots using **ggplot2**

Note

> You can refer to the **ggplot2** https://ggplot2.tidyverse.org/reference/index.html to find all the possible plots that you can build with ggplot2

First, we will need to install the package and then load the pack using the **library( )** function.

```
library(ggplot2)
```

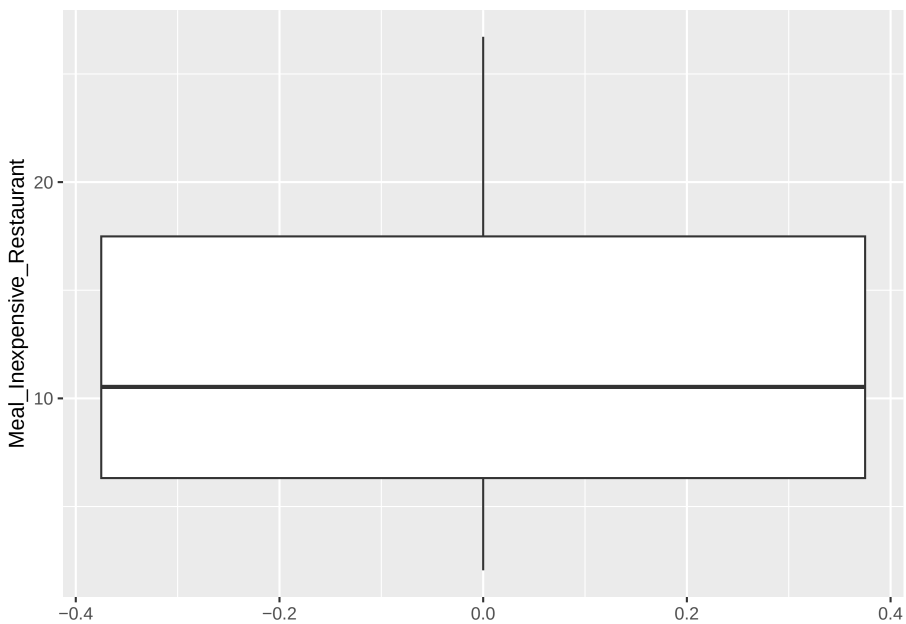
7.2.1 Boxplot

The basic structure for using **ggplot( )** is

ggplot(data = x, aes(x = x-axis, y = y-axis)) + geom_typeOfPlot()

where *aes* stands for aesthetic. This defines what variables you want to plot. For our boxplot, we want look at the meal prices as a whole so we only need to declare the y-variable.

```
ggplot(data = travel_full_clean_subset, aes(y=Meal_Inexpensive_Restaurant)) +
  geom_boxplot()
```



We can also rename our axis in a simliar way to our base R plot by using the **labs( )** argument

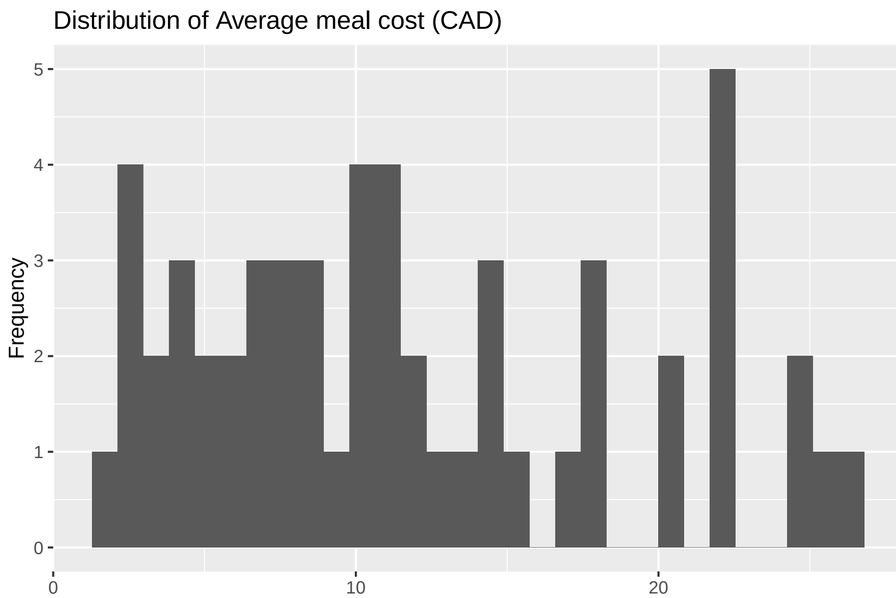
```
ggplot(data = travel_full_clean_subset, aes(y=Meal_Inexpensive_Restaurant)) +
  geom_boxplot() +
  labs(title="Average price of resturant meals",
       x="",
       y = "Dollar amount (CAD)")
```

From here, we can customize the look of our boxplot with other arguments.

7.2.2 Histogram

We can do the same with our histogram. Instead of using **geom_boxplot( )** we will use **geom_histogram( )**

```
ggplot(data = travel_full_clean_subset, aes(x=Meal_Inexpensive_Restaurant)) +
  geom_histogram(bins=30) +
  labs(title="Distribution of Average meal cost (CAD)",
       y= "Frequency",
       x="")
```



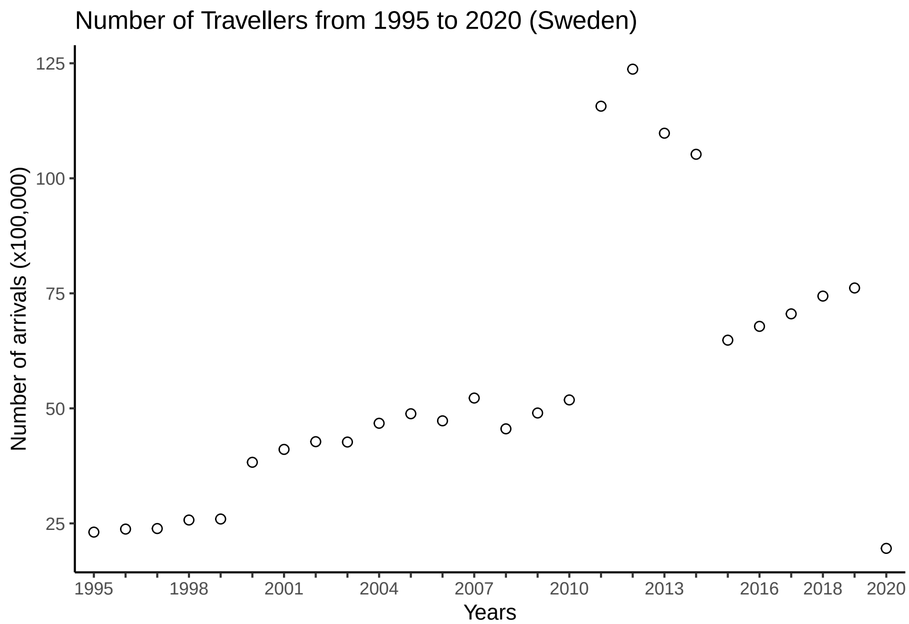
7.2.3 Scatterplot

Similarly with the scatterplot, we can use **geom_point**( )

```
ggplot(data = sweden_travel, aes(x=years, y = arrivals/100000)) +
  geom_point() +
  labs(title="Number of Travellers from 1995 to 2020 (Sweden)",
     x="Years",
     y="Number of arrivals (x100,000)")
```

We can see that the scatterplot has the same trend but looks a little different. We can easily adjust the graph so it matches our previous one.

```
ggplot(data = sweden_travel, aes(x=years, y = arrivals/100000)) +
  geom_point(shape=1,
             size=2) +
  labs(title="Number of Travellers from 1995 to 2020 (Sweden)",
     x="Years",
     y="Number of arrivals (x100,000)")+
  scale_x_discrete(guide = guide_axis(check.overlap = TRUE)) +
  theme_classic()
```



By adding a few arguments we can create clean and meaningful graphs using **ggplot2**

```
ggplot(data = sweden_travel, aes(y=arrivals/100000)) +
  geom_boxplot(outlier.color = "red", outlier.shape = 1, outlier.size = 3) +
  labs(title="Number of Travellers from 1995 to 2020 (Sweden)",
       x="",
       y = "Arrivals")
```

7.2.4 Exporting plots

Once you are finished with creating your desired graph, you can export the file as a pdf or other vector image formats for journals. Let's use our last boxplot on the number of arrivals as a example. First we need to save the boxplot as a variable.

```
boxplot_arrivals_swe <- ggplot(data = sweden_travel, aes(y=arrivals/100000)) +
  geom_boxplot(outlier.color = "red", outlier.shape = 1, outlier.size = 3) +
  labs(title="Number of Travellers from 1995 to 2020 (Sweden)",
       x="",
       y = "Arrivals")
```

Next, we will use the **ggsave( )** function and specify the location, the format, and the height and width we want to save the image as.

```
ggsave("/Users/markly/boxplot_swe_arrival.pdf", plot = boxplot_arrivals_swe, device = "
```

Note:

> We can save images in different formats including

- png, eps, ps, tex, jpeg, tiff, bmp, svg or wmf

7.3 gtsummary

Another powerful package that helps with data visiluzuation is **gtsummary** which can quickly generate summary tables. We will need to install the package and load the package using the **library( )** function.

```
library(gtsummary)
```

Let's say we are interested in summarizing the meal and ticket costs in a table. First, we will need to subset that from our dataset and then pass that into the **tbl_summary()** function.

```
travel_full_clean_subset %>%
  select(c(Meal_Inexpensive_Restaurant,one_way_ticket_local)) %>%
  tbl_summary()
```

```
## Table printed with `knitr::kable()`, not {gt}. Learn why at
## https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
## To suppress this message, include `message = FALSE` in code chunk header.
```

Characteristic	**N = 55**
Meal_Inexpensive_Restaurant	11 (6, 17)
one_way_ticket_local	1.20 (0.50, 2.20)

This produced a summary table of all the meals and ticket information giving us the **median** and **IQR** without any additional steps. We can customize this table a bit further by adding a few arguments.

```
travel_full_clean_subset %>%
  select(c(Meal_Inexpensive_Restaurant,one_way_ticket_local)) %>%
  tbl_summary(label = list("Meal_Inexpensive_Restaurant"~"Meal Price",
                           "one_way_ticket_local"~"One-way Ticket Price")) %>%
  modify_header(label = "**Variable**") %>% # update the column header
  bold_labels()
```

```
## Table printed with `knitr::kable()`, not {gt}. Learn why at
## https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
## To suppress this message, include `message = FALSE` in code chunk header.
```

Variable	**N = 55**
___Meal Price___	11 (6, 17)
___One-way Ticket Price___	1.20 (0.50, 2.20)

gtsummary is also great for separating and summarizing groups in the same table. We can add region information to our data set using a left join. After saving our new dataframe, we can select only the variables needed, region, meal price, and ticket price, and apply the **gtsummary( )** function to output a table based on region.

```
region<- read.csv("data/csv_region.csv")
region_select <-region %>%
  select(c(alpha.3,region)) %>%
  rename(country_code = alpha.3)

travel_region <- travel_full_clean_subset %>%
  select(c(1:4)) %>%
  inner_join(.,region_select,by="country_code")

travel_region %>%
  select(-c(Country,country_code))%>%
  tbl_summary(by=region,
              label = list("Meal_Inexpensive_Restaurant"~"Meal Price",
                           "one_way_ticket_local"~"One-way Ticket Price")) %>%
  modify_header(label = "**Variable**") %>% # update the column header
  bold_labels()
```

```
## Table printed with `knitr::kable()`, not {gt}. Learn why at
## https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
## To suppress this message, include `message = FALSE` in code chunk header.
```

Variable	**Africa**, N = 6	**Americas**, N = 10	**Asia**, N = 14	**E
___Meal Price___	5 (4, 8)	10 (6, 12)	6 (3, 10)	
___One-way Ticket Price___	0.49 (0.33, 1.01)	0.85 (0.76, 1.29)	0.49 (0.35, 0.93)	2.

```
travel_full_clean_subset %>%
  inner_join(.,region_select,by="country_code") %>%
  select(c(Meal_Inexpensive_Restaurant,one_way_ticket_local,"2015":"2020",region)) %>%
  tbl_summary(by=region)
```

```
## Table printed with `knitr::kable()`, not {gt}. Learn why at
## https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
## To suppress this message, include `message = FALSE` in code chunk header.
```

Characteristic	**Africa**, N = 6	**Americas**, N = 10	**A
Meal_Inexpensive_Restaurant	5 (4, 8)	10 (6, 12)	
one_way_ticket_local	0.49 (0.33, 1.01)	0.85 (0.76, 1.29)	0.4
2015	3,534,500 (1,308,000, 9,246,250)	3,531,500 (2,639,500, 5,859,500)	6,151,000 (
2016	3,881,500 (1,490,000, 9,438,750)	3,756,000 (2,585,250, 6,327,500)	6,511,000 (
2017	4,751,500 (1,641,000, 10,418,000)	4,166,000 (2,695,500, 6,703,750)	7,014,000 (
2018	5,478,000 (1,737,500, 11,441,500)	4,289,500 (2,693,500, 6,762,500)	7,855,500 (
2019	5,900,000 (1,656,250, 12,189,000)	4,382,000 (2,761,500, 6,895,250)	8,413,000 (
2020	1,301,500 (536,250, 2,604,500)	1,362,850 (771,875, 3,598,500)	1,837,000

With both **ggplot2** and **gtsummary**, we have lots of tools to use to display and communicate our data.

Chapter 8

Regression

R is a statistical computing language which is capable of performing multiple statistical models including

- lm(y ~ x) linear regression model with one explanatory variable
- lm(y ~ x1 + x2 + x3) multiple regression, a linear model with multiple explanatory variables
- glm(y ~ x, family = poisson) generalized linear model, poisson distribution; see ?family to see those supported, including binomial, gaussian, poisson, etc.
- glm(y ~ x + y, family = binomial) glm for logistic regression
- aov(y ~ x) analysis of variance (same as lm except in the summary)
- gam(y ~ x) generalized additive models
- tree(y ~ x) or rpart(y ~ x) regression/classification trees

8.1 Linear regression

Linear regression is a statistical method used for predictive analysis where we want to show if there is a linear relationship between an **independent predictor** variable and a **dependent output** variable. The goal is to build a mathematical formula that defines y as a function of the x variable. Once, we built a statistically significant model, it's possible to use it for predicting future outcome on the basis of new x values.

From the previous scatterplot, we noticed that there might be a relationship between the years and arrivals for Sweden. We can try to perform a linear regression to determine if that is the case. The hypothesis we want to propose is that the number of arrivals to Sweden increases linearly per year.

The general formula for a linear model is

$$Y_i = \beta_0 + \beta_1 X_1 + \epsilon$$

where

$$\begin{aligned}
Y_i &= \text{Dependent Variable} \\
\beta_0 &= \text{Constant/Intercept} \\
\beta_1 &= \text{Slope/Intercept} \\
X_1 &= \text{Independent Variable} \\
\epsilon &= \text{Error Term}
\end{aligned}$$

For our model, we would write out our linear regression equation as

$$\text{arrivals} = \beta_0 + \beta_1 * \text{Years} + \epsilon$$

Assumptions of Linear Regression

Before we begin with linear regression, there are some assumptions that need to be satisfied. We can use the *LINE* acronym to explain these assumptions.

- **L**inearity of residuals - There needs to be a linear relationship between the dependent and independent variables(s)
- **I**ndependance of residuals - The error terms should not be dependent on one another. For example, in time series, the next value is dependent on the previous one.
- **N**ormal distribution of residuals - The mean residuals should follow a normal distribution with a mean equal to zero or close to zero.
- **E**qual variance of the residuals - The error terms must have constant variance

8.1.1 Residuals

Residuals represent the difference between the predicted value and the observed value. The formula for residuals is.

$$\text{Residual} - \text{actual y value} - \text{predicted y value}$$

Visually, if we were to draw a line

8.1.2 Creating our intial model

To create a linear model, we use the function **lm([target] ~ [predictor / features], data = [data source])** when we have one explanatory variable. In this case our y-variable is the number of **arrivals** and the x-variable is the **years**

Note:

> The years variable in our dataframe is a *character*. We need to convert it to a numeric to perform the linear regression.

```
swe_model <- lm(arrivals~as.numeric(sweden_travel$years),data=sweden_travel)
swe_model
```

```
##
## Call:
## lm(formula = arrivals ~ as.numeric(sweden_travel$years), data = sweden_travel)
##
## Coefficients:
##                     (Intercept)  as.numeric(sweden_travel$years)
##                      -470402333                           237113
```

8.1.3 Interpretation of results

We can print out a summary of our model using the **summary( )** function. This will give us information on the intercept, standard error, p-value, test-statistic, r2 value.

```
summary(swe_model)
```

```
##
## Call:
## lm(formula = arrivals ~ as.numeric(sweden_travel$years), data = sweden_travel)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -6608564  -826391  -507921   -40090  5703338
##
## Coefficients:
##                                  Estimate Std. Error t value Pr(>|t|)
## (Intercept)                    -470402333  126815668  -3.709  0.00109 **
## as.numeric(sweden_travel$years)     237113      63170   3.754  0.00098 ***
## ---
```

```
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2416000 on 24 degrees of freedom
## Multiple R-squared:  0.3699, Adjusted R-squared:  0.3436
## F-statistic: 14.09 on 1 and 24 DF,  p-value: 0.0009798
```

In this output, the estimate column shows the estimates of our beta coefficients β_0 and β_1. The intercept (β_0) is -470,402,333 and the coefficient of years variable is 237,113. We can rewrite our original equation with our new results

$$\text{arrivals} = \beta_0 + \beta_1 * \text{Years} + \epsilon$$

$$\text{arrivals} = -470,402,333 + 237,113 * \text{Years} + \epsilon$$

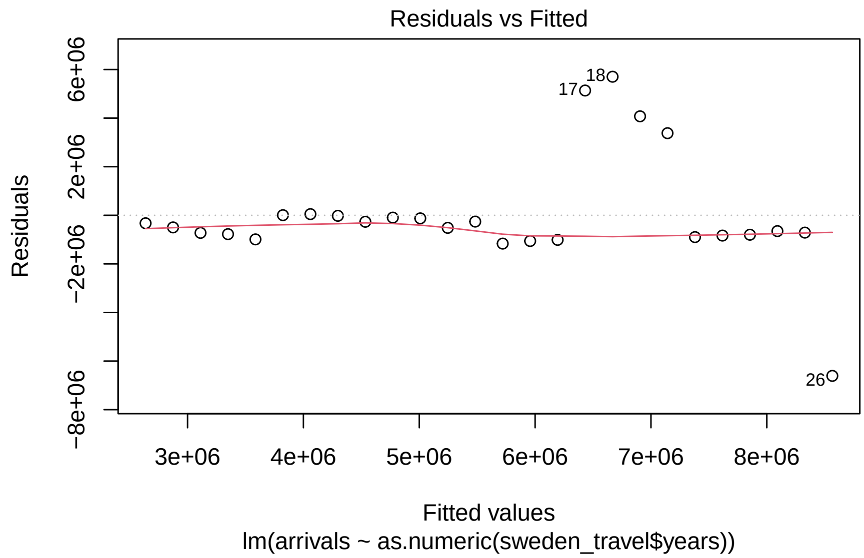
In writen form, we would say

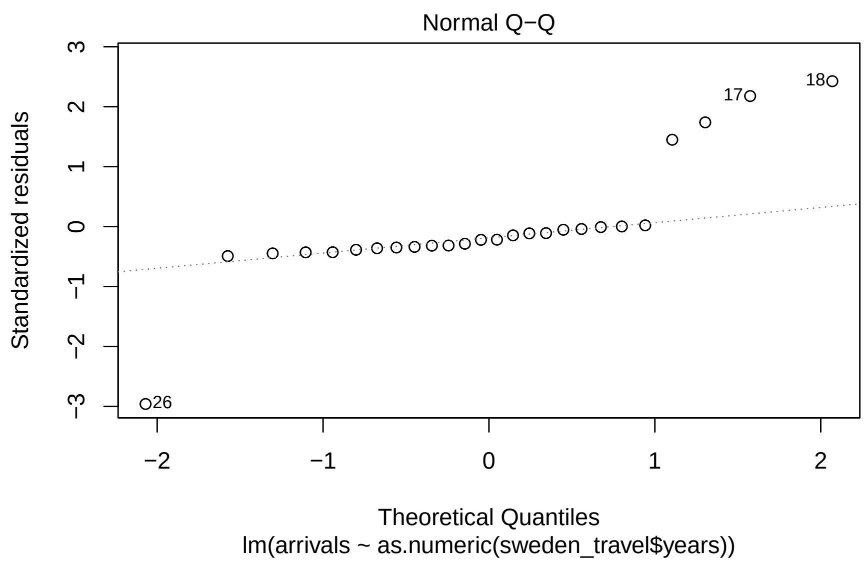
- If the Years is equal to zero, we can expect -470,402,333 arrivals
- For every 1 increase in Years, we can expect 237,113 arrivals

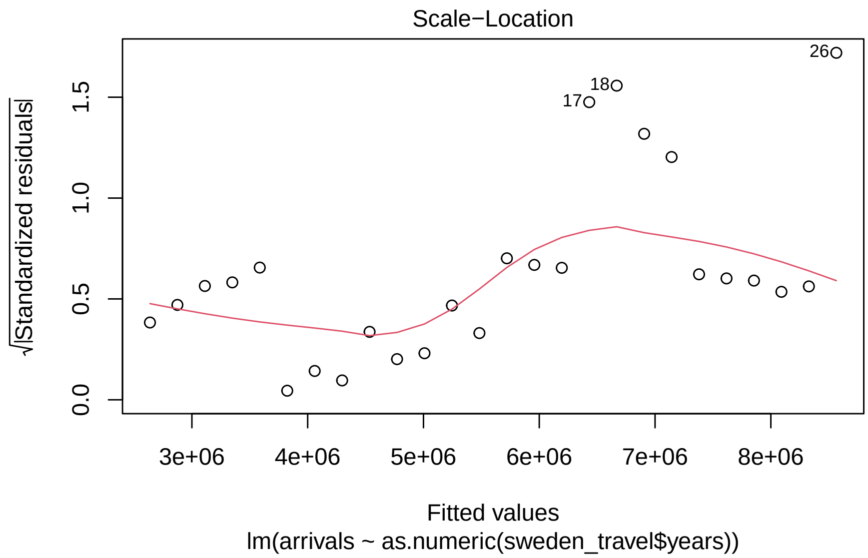
If we plot our linear regression, it will give us diagnostic plots that will help determine if our linear model satisfies certain assumptions

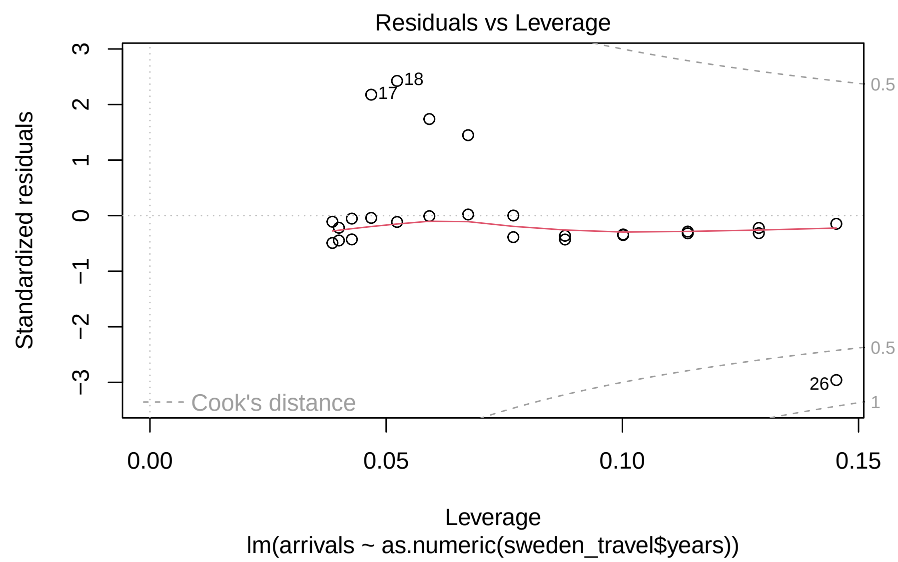
- **Residual vs Fitted** - Used to check linear relationship assumptions.
- **Normal Q-Q-** Used to check if the residuals are normally distributed.
- **Scale-Location** - Used to check homogeneity of variance.
- **Residuals vs Leverage** - Used to check for influential cases or extreme values that might influence the regression results.

```
plot(swe_model)
```


Residuals vs Fitted
Residuals
Fitted values
lm(arrivals ~ as.numeric(sweden_travel$years))
6e+06
2e+06
−2e+06
−8e+06
3e+06
4e+06
5e+06
6e+06
7e+06
8e+06
17
18
26


Normal Q−Q
Standardized residuals
Theoretical Quantiles
lm(arrivals ~ as.numeric(sweden_travel$years))
3
2
1
0
−1
−2
−3
17
18
26


Scale−Location
√|Standardized residuals|
0.0
0.5
1.0
1.5
26
18
17
3e+06
4e+06
5e+06
6e+06
7e+06
8e+06
Fitted values
lm(arrivals ~ as.numeric(sweden_travel$years))


Residuals vs Leverage
Standardized residuals
3
2
1
0
−1
−2
−3
17
18
26
0.5
0.5
1
Cook's distance
0.00
0.05
0.10
0.15
Leverage
lm(arrivals ~ as.numeric(sweden_travel$years))

Chapter 9

Other capabilties

We can create and perform other tasks using R. We will go quickly go through some R's capabilities and things you can do with R including

- Machine learning
- R-bookdown
- R-Shiny

9.1 Machine learning

We can perform machine learning tasks in R using the **caret** and **factoextra** packages. This includes models like

- Random forests
- Decisions Trees
- XGBoost
- K-means
- Neural Networks

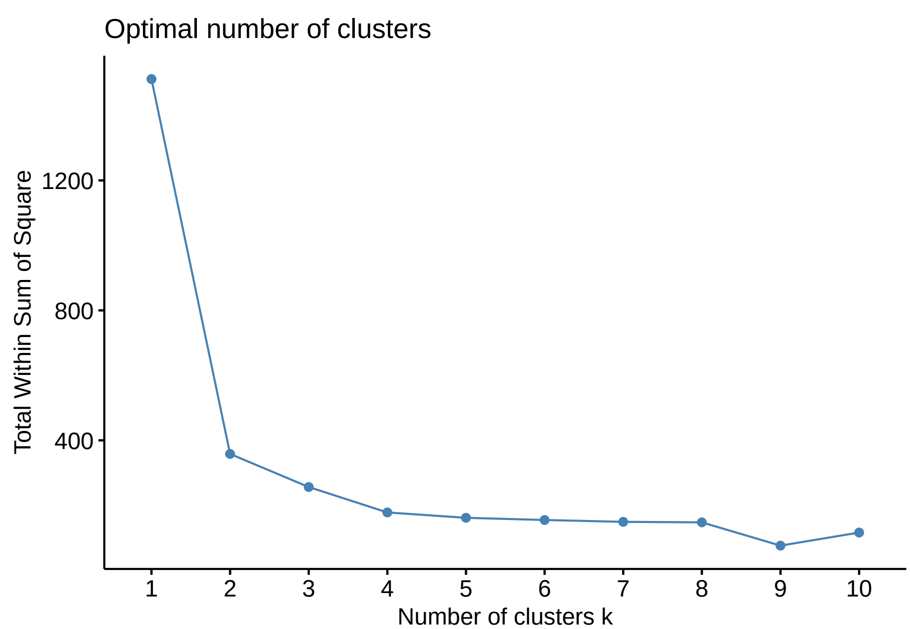
We will attempt to use K-means clustering on out dataset to see which countries are simliar. First we will need to install and load our packages and prepare our data.

```
library(factoextra)
```

```
## Welcome! Want to learn more? See two factoextra-related books at https://goo.gl/ve3W
```

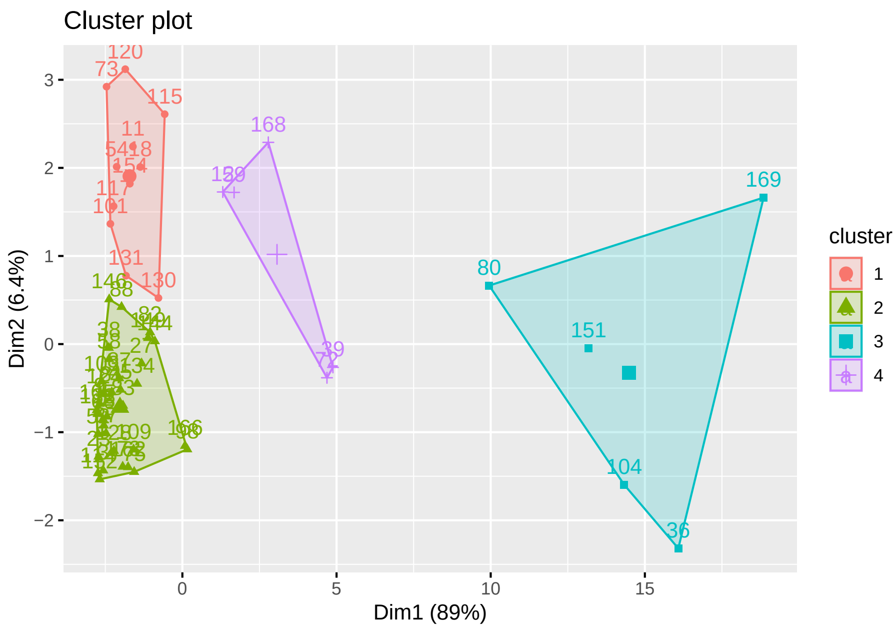
```
kmeans_travel <- travel_full_clean_subset %>%
  select(where(is.numeric))

scale_kmeans <- scale(kmeans_travel)
```

```
set.seed(123)
fviz_nbclust(scale_kmeans,kmeans,method="wss")
```



From looking at the elbow method, it looks like the optimal number of clusters is 2.

```
kmeans_travel_1 <- kmeans(scale_kmeans,centers=4,nstart=25)

fviz_cluster(kmeans_travel_1,data=scale_kmeans)
```



9.2 R-bookdown

9.3 R-Shiny