R Bootcamp - How to use R for Data Science

Yigit Aydede

2022-10-12

Contents

About **5**
Why R? 6
License 7

1 Introduction **9**
1.1 Installation of R, RStudio and R Packages 9
1.2 RStudio 11
1.3 Packages 15
1.4 Working directory 16
1.5 Hints 17
1.6 Console or Script? 17
1.7 R as a calculator 18
1.8 Data & Object Types 18

2 Vectors **21**
2.1 One type, same type 21
2.2 Patterns 23
2.3 Attributes 24
2.4 Character operators 25
2.5 Sort, rank, and order 26
2.6 Simple descriptive measures 26
2.7 Subsetting Vectors 28
2.8 Vectorization or vector operations 31
2.9 Set operations 32
2.10 Missing values 32
2.11 Factors 33

3 Matrices **35**
3.1 Matrix Operations 36
3.2 Subsetting Matrix 38
3.3 R-Style Guide 39

4 Cross-references **41**
4.1 Chapters and sub-chapters 41

4.2 Captioned figures and tables 41

5 Parts 45

6 Footnotes and citations 47

6.1 Footnotes 47

6.2 Citations 47

7 Blocks 49

7.1 Equations 49

7.2 Theorems and proofs 49

7.3 Callout blocks 49

8 Sharing your book 51

8.1 Publishing 51

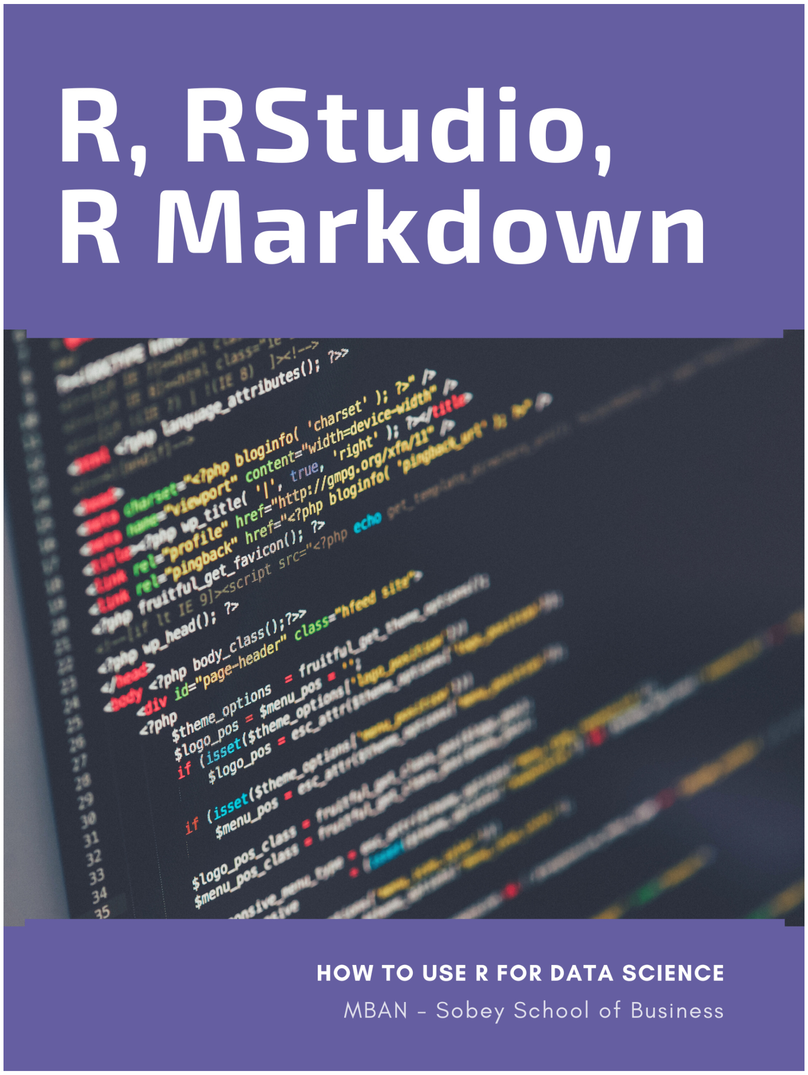
8.2 404 pages 51

8.3 Metadata for sharing 51

About

This book covers basics to learn R for Data Science. It is designed for MBAN students.

We also have a companion R package named `RBootcamp`, containing the data sets used as well as interactive exercises for each part.



R, RStudio, R Markdown

HOW TO USE R FOR DATA SCIENCE

MBAN - Sobey School of Business

Why R?

R is both a programming language and software environment for statistical computing, which is free and open-source.

With ever increasing availability of large amounts of data, it is critical to have the ability to analyze the data and learn from it for making informed decisions. Familiarity with software such as R allows users to visualize data, run statistical tests, and apply machine learning algorithms. Even if you already know other software, there are still good reasons to learn R:

1. **R is free.** If your future employer does not already have R installed, you can always download it for free, unlike other proprietary software packages that require expensive licenses. You can always have access to R on your computer.

2. **R gives you access to cutting-edge technology.** Top researchers develop statistical learning methods in R, and new algorithms are constantly added to the list of packages you can download.

3. **R is a useful skill.** Employers that value analytics recognize R as useful and important. If for no other reason, learning R is worthwhile to help **boost your resume.**

Here is a very good article about R and Programming that everybody should read: 7 Reasons for policy professionals to get into R programming in 2019 [**?**].

License

Figure 1: This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

Chapter 1

Introduction

The following sections will serve as an introduction to the R basics that could be used in data analytics. At the beginning, these introductory R subjects may feel like an overwhelming amount of information. The leaning curve will be steeper as practice more. You should try all of the codes from these examples and solve the practice exercises.

R is used both for software development and data analysis. We will not use it for software development but apply some concepts in that area. Our main goal will be to analyze data, but we will also perform programming exercises that help illustrate certain algorithmic concepts.

What we will review in this Chapter:

1. **R, RStudio, and R Packages**,
2. **Starting with RStudio**,
3. **Working Directory**,
4. **Data Types and Structures (Vectors and Matrices)**,
5. **R-Style Guide**

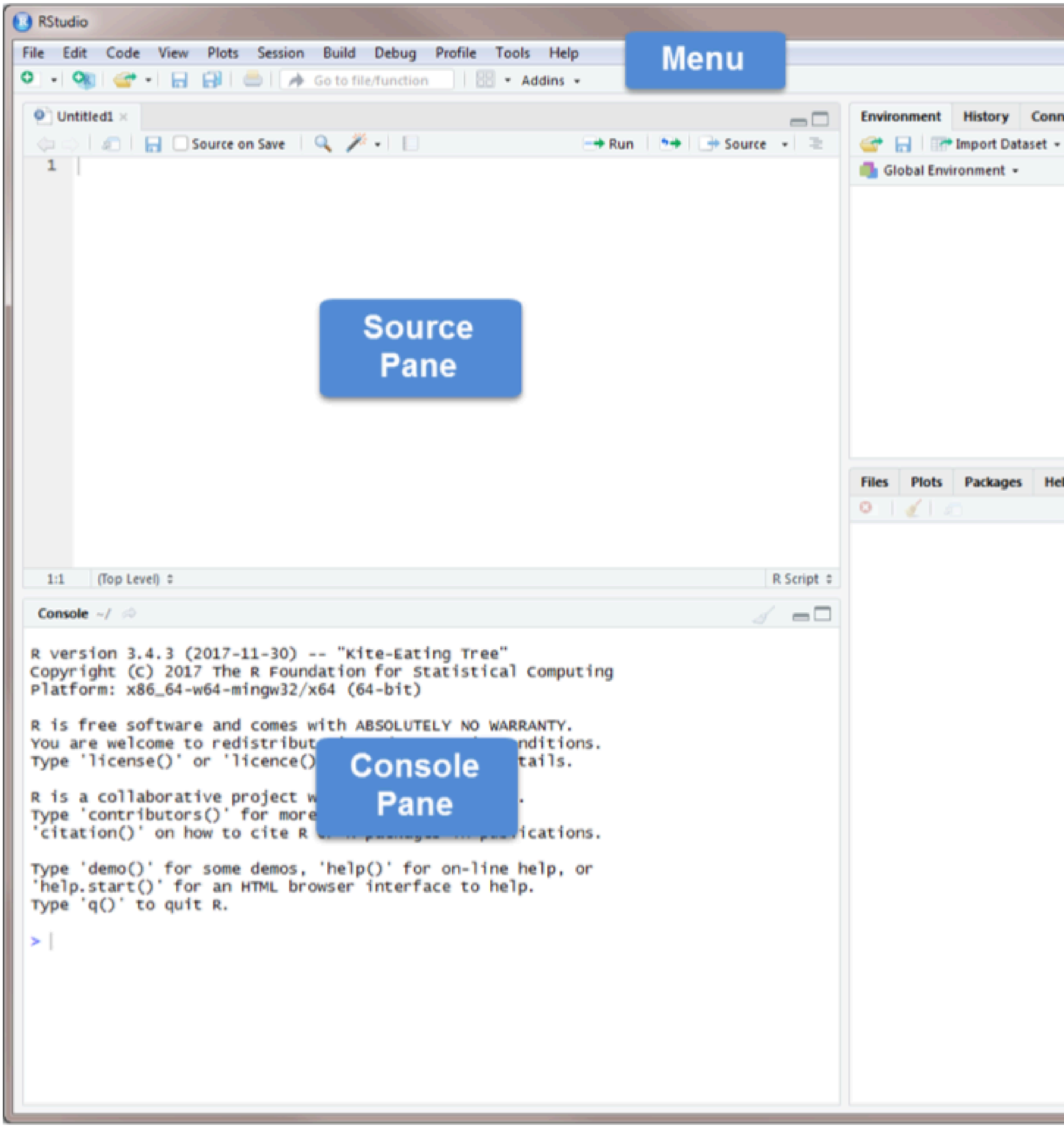
1.1 Installation of R, RStudio and R Packages

To get started, you will need to install two pieces of software:

R, the actual programming language: **Download it from here.** – Chose your operating system, and select the most recent version.

RStudio, an excellent integrated development environment (IDE) for working with R, an interface used to interact with R: **Download it from here.** Throughout this book, you will use RStudio instead of R to learn R programming.

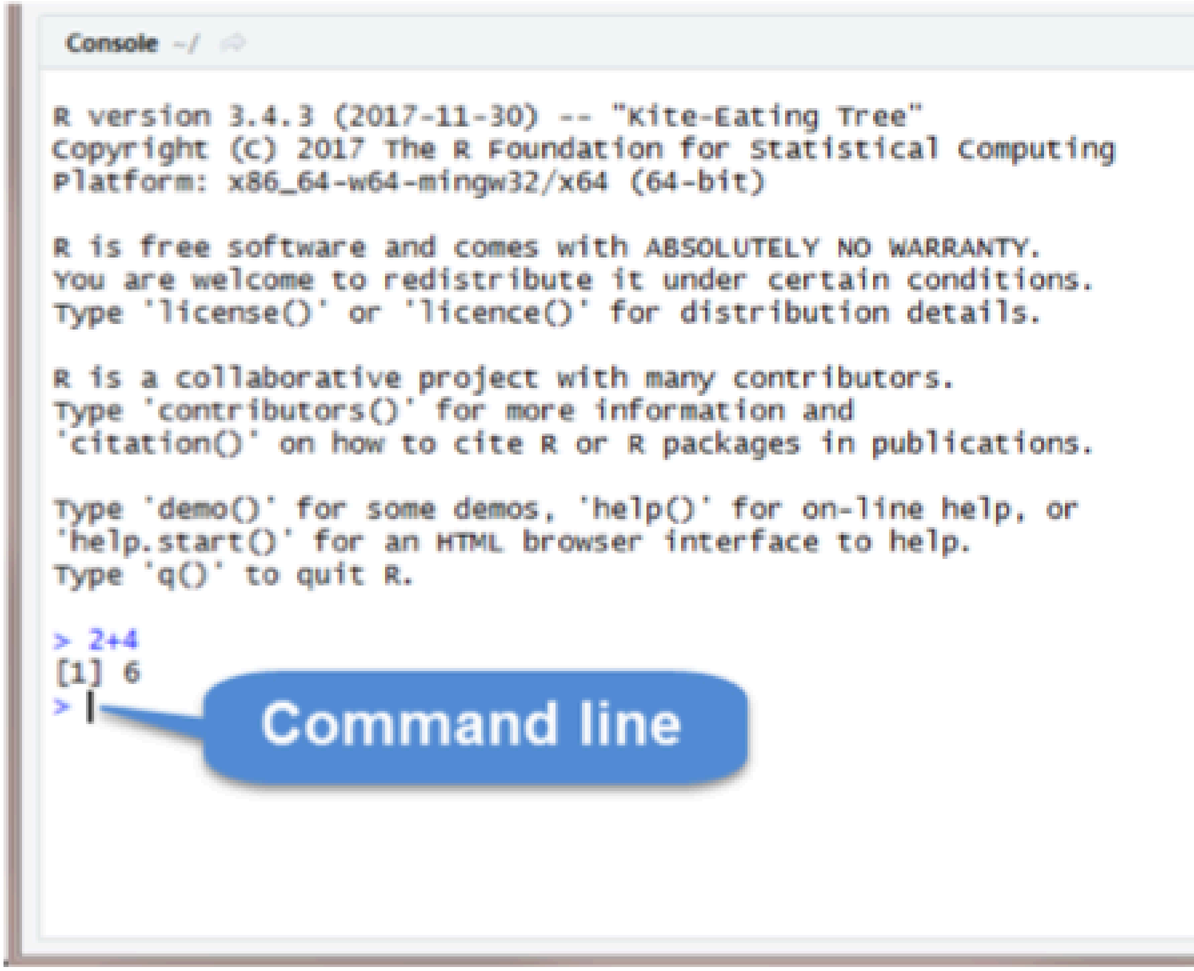
1.2 RStudio



```
R version 3.4.3 (2017-11-30) -- "Kite-Eating Tree"
Copyright (C) 2017 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribut                nditions.
Type 'license()' or 'licence()                tails.

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

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

> 
```

Source Pane, click on the plus sign in the top left corner. From the drop-down menu, select **R Script** . As shown in that dropdown menu, you can also open an R Script by pressing **Ctrl+Shift+N**. You should now see the screen above.

The **Console Pane** is the interface to R. If you opened R directly instead of opening RStudio, you would see just this console. You can type commands directly in the console. The console displays the results of any command you run. For example, type 2+4 in the command line and press enter. You should see the command you typed, the result of the command, and a new command line.



```
R version 3.4.3 (2017-11-30) -- "Kite-Eating Tree"
Copyright (C) 2017 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

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

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

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

> 2+4
[1] 6
> 
```

To clear the console, you press **Ctrl+L** or type **cat("\014")** in the command line.

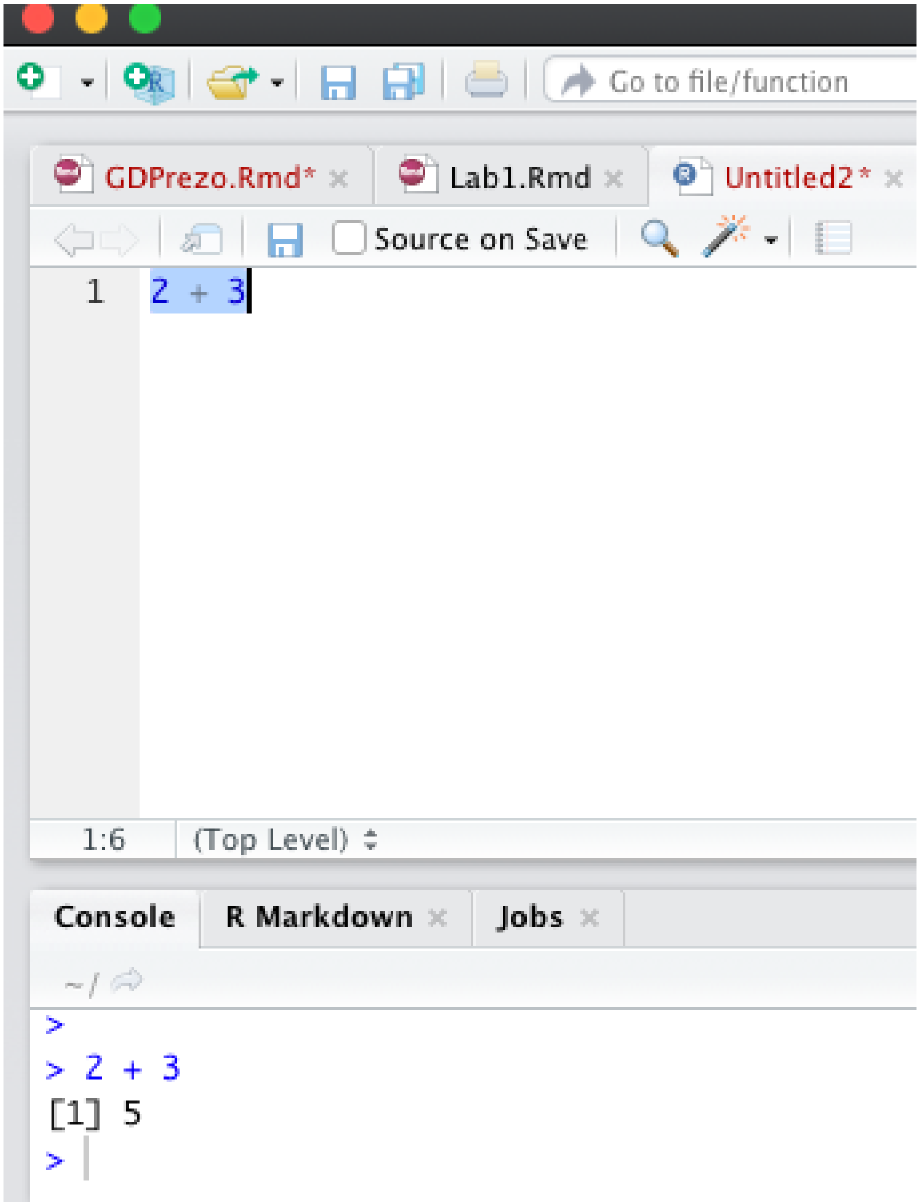
R code can be entered into the command line directly (in Console Pane) or saved to a script (Source Pane).

Let's try some coding.

```
2 + 3 #write this on the command line and hit Enter
```

```
## [1] 5
```

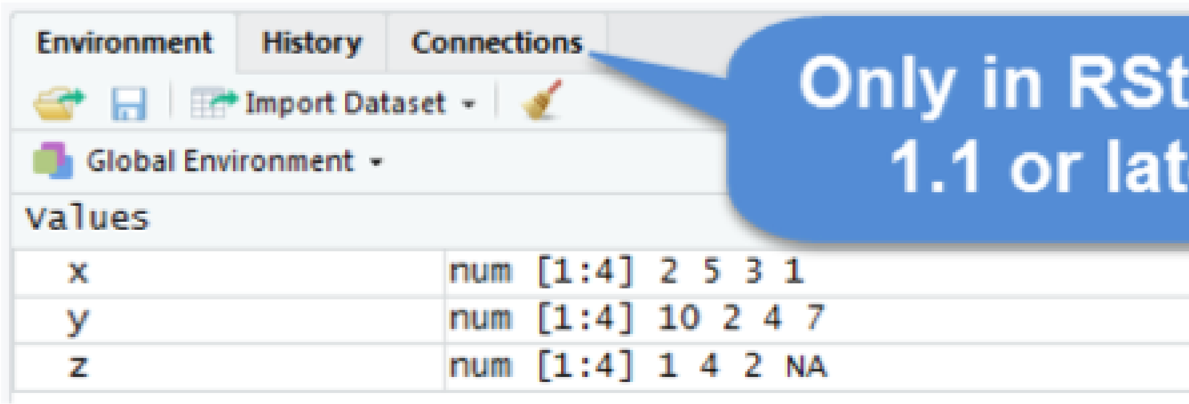
Now write the same line into the script in Source Pane and **run** it



```
1  2 + 3
```

```
>
> 2 + 3
[1] 5
>
```

The **Source Pane** is a text editor where you can type your code before running it. You can save your code in a text file called a script. Scripts have typically file names with the extension **.R**. Any text shown in green is a comment in the script. You write a comment by adding a # to an RScript. Anything to the right of a **#** is considered a comment and is thus ignored by R when running code. Place your cursor anywhere on the first few lines of code and click **Run**. You can also run code by pressing **Ctrl+Enter**.



Values

x	num [1:4] 2 5 3 1
y	num [1:4] 10 2 4 7
z	num [1:4] 1 4 2 NA

The **Environment Pane** includes an Environment, a History tab, and a Connections tab. The Connections tab makes it easy to connect to any data source on your system.

The Environment tab displays any objects that you have created during your R session. For example, if we create three variables: x, y, and z, R stored those variables as objects so that you can see them in the Environment pane.

To do object assignments, you need to assign value(s) to a name via the assignment operator, which will create a new object with a name.

```
x <- 5
y <- x*1.5
z <- x - y*3
ls()
```

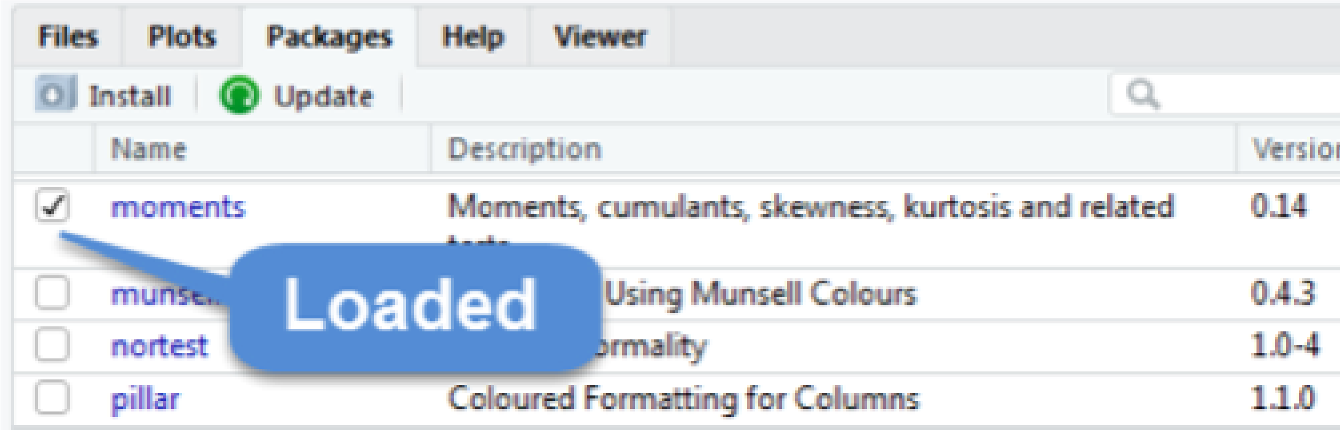
```
## [1] "x" "y" "z"
```

We will discuss R objects in more detail later. If you want to see a list of all objects in the current session, type `ls()` in the command line. You can remove an individual object from the environment with the `rm()` command. For example, remove x by typing `rm(x)` in the command line. You can remove all objects from the environment by clicking or typing `rm(list=ls())` in the command line. The History tab keeps a record of all the commands you have run. To copy a command from the history into the console, select the command and press Enter.

The **Files Pane** includes several tabs that provide useful information. The Files tab displays the contents of your working directory. The Plot tab shows all graphs that you have created. The Packages tab displays the R packages that you have installed in your System Library.

1.3 Packages

An R package typically includes code, data, documentation for the package and functions inside, and tests to check everything works as it should. Check to see if the package `moments` has been installed. If you cannot find it, you need to install it by using the command `install.packages("moments")`. Once you have installed the package, you need to load it using the command `library(moments)`. Or you can use install tab and follow the instructions and the go to package to check it to activate as shown below.



	Name	Description	Versio
☑	moments	Moments, cumulants, skewness, kurtosis and related tests	0.14
□	munse	Using Munsell Colours	0.4.3
□	nortest	ormality	1.0-4
□	pillar	Coloured Formatting for Columns	1.1.0

The help tab has built-in documentation for packages and functions in R. The help is automatically available for any loaded packages. You can access that file by typing `help(mean)` or `?mean` in the command line. You can also use the search bar in the help tab.

The packages can be installed from sources other than CRAN. For example, in this book we will use `RBootcamp` which is not located on CRAN

```
#install.packages("remotes")
#remotes::install_github("yaydede/RBootcamp")
```

One of the most difficult things to do when learning R is to know how to find help. Your very first helper should be **Google** where you post your error message or a short description of your issue. The ability to solve problems using this method is quickly becoming an extremely valuable skill.

Do not be discouraged by running into errors and difficulties when learning R. It is simply part of the learning process.

The Viewer tab displays HTML output. R packages such as R Markdown and Shiny create HTML outputs that you can view in the Viewer tab. We'll see it later.

1.4 Working directory

Without further specification, files will be loaded from and saved to the working directory. The functions `getwd()` and `setwd()` will get and set the working directory, respectively.

```
getwd()
```

```
## [1] "/Users/yigitaydede/Dropbox/Documents/Courses/MBAN/RBootcamps/Bootcamp_book"
```

```
#setwd("Book2022")

#List all the objects in your local workspace using
ls()
```

```
## [1] "x" "y" "z"
```

```
#List all the files in your working directory using list.files() or
dir()
```

```
##  [1] "_book"                     "_bookdown_files"         "_bookdown.yml"
##  [4] "_main_files"               "_output.yml"             "01-intro.Rmd"
##  [7] "02-cross-refs_files"       "02-cross-refs.Rmd"       "03-parts.Rmd"
## [10] "04-citations.Rmd"          "05-blocks.Rmd"           "06-share.Rmd"
## [13] "07-references.Rmd"         "Book_for_RBootcamp.log" "Book_for_RBootcamp.rds"
## [16] "book.bib"                  "Bootcamp_book.Rproj"     "docs"
## [19] "index.md"                  "index.Rmd"               "packages.bib"
## [22] "png"                       "preamble.tex"            "README.md"
## [25] "render1a5727895a64.rds" "style.css"
```

```
#As we go through this lesson, you should be examining the help page
#for each new function. Check out the help page for list.files with the
#command
?list.files
#or
help("list.files")

#Using the args() function on a function name is also a handy way to
#see what arguments a function can take.
args(list.files)
```

```
## function (path = ".", pattern = NULL, all.files = FALSE, full.names = FALSE,
##     recursive = FALSE, ignore.case = FALSE, include.dirs = FALSE,
##     no.. = FALSE)
## NULL
```

For this bootcamp, I would suggest to create a RStudio Project in your local driver. RStudio projects make it straightforward to divide your work into multiple contexts, each with their own working directory, workspace, history, and source documents.

RStudio projects are associated with R working directories. You can create an RStudio project:

- In a brand new directory
- In an existing directory where you already have R code and data
- By cloning a version control (Git or Subversion) repository

To create a new project in the RStudio, use the Create Project command (available on the Projects menu and on the global toolbar)

1.5 Hints

- R distinguishes upper case from lower case letters. Thus a variable named `X` differs from another variable named `x`.
- The way to learn programming is through practice. The learning curve to R is not bad. You may struggle a bit in the process, but the skills learned will be invaluable for you in the future.
- There are many ways to write a code to solve the same thing. You can develop your own style. But, if you see nother and better code, try to learn from others!
- There are many option that you can customize R Studio. Check out Tools > Global Options > General tab in the menu bar of RStudio.
- Before you start coding, draft a plan to address the question at hand.
- There is convention in writing codes. Try to adhere these accepted styles.
- Comment your code properly (using a # sign at the beginning of each line). Good documentation is a great reminder what you have done. Believe me you will forget later the lines in the your own script.
- The character `>` in the R Console indicates that R is ready for you to enter a command.

- Do not overwrite the original data set and variables. Create new data sets just to be sure, especially when taking a subset from that data set.

1.6 Console or Script?

The Script Window is the place to enter and run your code so that it is easily edited and saved for future use. You create new R Script by clicking on File > New File > R Script in the RStudio menu bar.

To execute your code in the R script, you can either highlight the code and click on Run, or you can highlight the code and press CTRL + Enter on your keyboard.

If you prefer, you can enter code directly in the Console Window and click Enter. The commands that you run will be shown in the History Window on the top right of RStudio. You can save these commands for future use.

Or, to find the older commands in the console, use the upper arrow to get them again …

Make sure you save your code. With your code, you can always regenerate your workspace but it could take a little time. Or you can save your workspace which allows you to start where you left off, with all of the variables you created and renamed saved.

Your code (in your script window) is saved by clicking the SAVE button in the RStudio menu bar. The code will be saved in the working directory. See 2.1 for setting and getting the working directory.

1.7 R as a calculator

At a very basic level, we can use R as a calculator.

See `Lesson1` in our package `Rbootcamp`:

```
library(RBootcamp)
#learnr::run_tutorial("Lesson1", "RBootcamp")
```

1.8 Data & Object Types

R has a number of basic data types.

Numeric: Also known as Double. The default type when dealing with numbers. 1,1.0,42.5
Integer: 1L,2L,42L
Complex: 4 + 2i
Logical: Two possible values: TRUE and FALSE. NA is also considered logical.
Character:“a”,“Statistics”,“1plus2.”

R also has a number of basic data structures. A data structure is either **homogeneous** (all elements are of the same data type) or **heterogeneous** (elements can be of more than one data type): You can think each data structure as **data container** (object types) where you data is stored. Here are the main “container” or data structures. Think it as Stata or Excel spread-sheets.

Vector: 1 dimension (column OR row) and homogeneous. That is every element of the vector has to be the same type. Each vector can be thought of as a variable.
Matrix: 2 dimensions (column AND row) and homogeneous. That is every

element of the matrix has to be the same type.
Data Frame: 2 dimensions (column AND row) and heterogeneous. That is every element of the data frame doesn't have to be the same type. This is the main difference between a matrix and a data frame. Data frames are the most common data structure in any data analysis.
List: 1 dimension and heterogeneous. Data can be multiple data structures.
Array: 3+ dimensions and homogeneous.

Chapter 2

Vectors

Many operations in R make heavy use of vectors. Possibly the most common way to create a vector in R is using the `c()` function, which is short for "combine." As the name suggests, it combines a list of elements separated by commas.

```
c(1, 5, 0, -1)
```

```
## [1]  1  5  0 -1
```

If we would like to store this vector in a **variable** we can do so with the assignment operator `<-` or `=`. But the convention is `<-`

```
x <- c(1, 5, 0, -1)
z = c(1, 5, 0, -1)
x
```

```
## [1]  1  5  0 -1
```

```
z
```

```
## [1]  1  5  0 -1
```

Note that scalars do not exists in R. They are simply vectors of length 1.

```
y <- 24  #this a vector with 1 element, 24
```

2.1 One type, same type

Because vectors must contain elements that are all the same type, R will automatically coerce to a single type when attempting to create a vector that combines multiple types.

```
c(10, "Machine Learning", FALSE)
```

```
## [1] "10"               "Machine Learning" "FALSE"
```

```
c(10, FALSE)
```

```
## [1] 10  0
```

```
c(10, TRUE)
```

```
## [1] 10  1
```

```
x <- c(10, "Machine Learning", FALSE)
str(x) #this tells us the structure of the object
```

```
##  chr [1:3] "10" "Machine Learning" "FALSE"
```

```
class(x)
```

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

```
y <- c(10, FALSE)
str(y)
```

```
##  num [1:2] 10 0
```

```
class(y)
```

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

We know that vectors are objects that have values of the same type. If you combine them into a vector, R will unify all values into the most complex one, which is usually called the coercion rule.

```
m <- c(TRUE, 5, -2, FALSE)
m
```

```
## [1]  1  5 -2  0
```

```
class(m)
```

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

And,

```
m_2 <- c(8, "Joe", 21, "Mustang")
m_2
```

```
## [1] "8"       "Joe"     "21"      "Mustang"
```

```
class(m_2)
```

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

You can also manually convert the vectors

```
n <- c(8, 3, 21, 2)
n
```

```
## [1]  8  3 21  2
```

```
nc <- as.character(n)
nc
```

```
## [1] "8"  "3"  "21" "2"
```

```
n <- as.numeric(nc)
n
```

```
## [1]  8  3 21  2
```

And be careful:

```
m_2 <- c(8, "Joe", 21, "Mustang")
as.numeric(m_2)
```

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

```
## [1]  8 NA 21 NA
```

2.2 Patterns

If you want to create a vector based on a sequence of numbers, you can do it easily with an operator, which creates a sequence of integers between two specified integers.

```
y <- c(1:15)
y
```

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

```
#or
y <- 1:15
y
```

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

If you want to create a vector based on a specific sequence of numbers increasing or decreasing, you can use `seq()`

```
y <- seq(from = 1.5, to = 13, by = 0.9) #increasing
y
```

```
##  [1]  1.5  2.4  3.3  4.2  5.1  6.0  6.9  7.8  8.7  9.6 10.5 11.4 12.3
```

```
y <- seq(1.5, -13, -0.9) #decreasing.  Note that you can ignore the argument labels
y
```

```
##  [1]   1.5   0.6  -0.3  -1.2  -2.1  -3.0  -3.9  -4.8  -5.7  -6.6  -7.5  -8.4
## [13]  -9.3 -10.2 -11.1 -12.0 -12.9
```

The other useful tool is `rep()`

```
rep("ML", times = 10)
```

```
##  [1] "ML" "ML" "ML" "ML" "ML" "ML" "ML" "ML" "ML" "ML"
```

```
#or

x <- c(1, 5, 0, -1)
rep(x, times = 2)
```

```
## [1]  1  5  0 -1  1  5  0 -1
```

And we can use them as follows.

```
wow <- c(x, rep(seq(1, 9, 2), 3), c(1, 2, 3), 42, 2:4)
wow
```

```
##  [1]  1  5  0 -1  1  3  5  7  9  1  3  5  7  9  1  3  5  7  9  1  2  3 42  2  3
## [26]  4
```

Another one, which can be used to create equal intervals.

```
g <- seq(6, 60, length = 4)
g
```

```
## [1]  6 24 42 60
```

See this

```
unique(wow)
```

```
##  [1]  1  5  0 -1  3  7  9  2 42  4
```

2.3 Attributes

We can calculate the number of elements in a vector:

```
length(wow)
```

```
## [1] 26
```

There is set of functions starting with `is.***()`. For example: `is.numeric()`, which checks whether a vector is of numeric type,

```
is.numeric(g)
```

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

```
is.character(g)
```

```
## [1] FALSE
```

In addition to storing the values of a vector, you can also create named vectors.

```
x <- c(165, 60, 22)
x
```

```
## [1] 165  60  22
```

```
x_n <- c(height = 125, weight = 56, BMI = 21)
x_n
```

```
## height weight    BMI
##    125     56     21
```

And,

```
attributes(x_n)
```

```
## $names
## [1] "height" "weight" "BMI"
```

2.4 Character operators

```
animals <- c("dog", "cat", "donkey")
nchar(animals)
```

```
## [1] 3 3 6
```

We can concatenate several strings into a single string.

```
wrong <- c("we have", "dogs", "cats", "and, donkey")
wrong
```

```
## [1] "we have"     "dogs"        "cats"        "and, donkey"
```

```
right <-paste("we have ", "dogs, ", "cats, ", "and, donkey")
right
```

```
## [1] "we have  dogs,  cats,  and, donkey"
```

You can check `paste0()`

```
hah <- toupper(right)
hah
```

```
## [1] "WE HAVE  DOGS,  CATS,  AND, DONKEY"
```

```
haha <-tolower(hah)
haha
```

```
## [1] "we have  dogs,  cats,  and, donkey"
```

2.5 Sort, rank, and order

```
x <- c(2, 3, 2, 0, 4, 7)
x
```

```
## [1] 2 3 2 0 4 7
```

By default, the `sort()` function sorts elements in vector in the increasing order.

```
sort(x)
```

```
## [1] 0 2 2 3 4 7
```

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

```
## [1] 7 4 3 2 2 0
```

The `rank()` function gives the corresponding positions in the ascending order.

```
rank(x)
```

```
## [1] 2.5 4.0 2.5 1.0 5.0 6.0
```

You can see that the smallest value of `x` is 0, which corresponds to the fourth element. Thus, the fourth element has rank 1.

As for the `order()` function, it is confusing and a very different function from `sort()`.

```
order(x)
```

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

We can see that the `order()` function returns indices for the elements in the ascending order.

2.6 Simple descriptive measures

Let's have a numeric vector:

```
h <- c(x, rep(seq(1, 9, 2), 3), c(1, 2, 3), 42, 2:4)
h
```

```
##  [1]  2  3  2  0  4  7  1  3  5  7  9  1  3  5  7  9  1  3  5  7  9  1  2  3 42
## [26]  2  3  4
```

```
summary(h)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
##   0.000   2.000   3.000   5.357   7.000  42.000 
```

The set of statistical measures:

```
min(h)
```

```
## [1] 0
```

```
max(h)
```

```
## [1] 42
```

```
mean(h)
```

```
## [1] 5.357143
```

```
median(h)
```

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

```
sd(h)
```

```
## [1] 7.650777
```

```
range(h)
```

```
## [1]  0 42
```

```
sum(h)
```

```
## [1] 150
```

```
cumsum(h)
```

```
##  [1]   2   5   7   7  11  18  19  22  27  34  43  44  47  52  59  68  69  72  77
## [20]  84  93  94  96  99 141 143 146 150
```

```
prod(h)
```

```
## [1] 0
```

```
quantile(h)
```

```
##   0%  25%  50%  75% 100% 
##    0    2    3    7   42
```

```
IQR(h) #interquartile range
```

```
## [1] 5
```

We can also find the index number of maximum and minimum numbers

```
which.max(h)
```

```
## [1] 25
```

```
which.min(h)
```

```
## [1] 4
```

2.7 Subsetting Vectors

One of the most confusing subjects in R is subsetting the data containers. It's an important part in data management and if it is done in 2 steps, the whole operation becomes quite easy:

1. Identifying the index of the element that satisfies the required condition,
2. Calling the index to subset the vector.

But before we start, lets see a simple subsetting. (Note the square brackets)

```
#Suppose we have the following vector
myvector <- c(1, 2, 3, 4, 5, 8, 4, 10, 12)

#I can call each element with its index number:
myvector[c(1,6)]
```

```
## [1] 1 8
```

```
myvector[4:7]
```

```
## [1] 4 5 8 4
```

```
myvector[-6]
```

```
## [1]  1  2  3  4  5  4 10 12
```

Okay, let's see commonly used operators for doing comparisons:

```
x <- 3
```

```
x < 2      #less
```

```
## [1] FALSE
```

```
x <= 2     #less or equal to
```

```
## [1] FALSE
```

```
x > 1      #bigger
```

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

```
x >= 1     #bigger or equal to
```

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

```
x == 3     #equal to
```

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

```
#x = 3     #Note that this an assignment operator
x != 3     #not equal to
```

```
## [1] FALSE
```

```
#Let's look at this vector
myvector <- c(1, 2, 3, 4, 5, 8, 4, 10, 12)

#We want to subset only those less than 5

#Step 1: use a logical operator to identify the elements
#meeting the condition.
logi <- myvector < 5
logi
```

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

```
#logi is a logical vector
class(logi)
```

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

```
#Step 2: use it for subsetting
newvector <- myvector[logi==TRUE]
newvector
```

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

or better:

```
newvector <- myvector[logi]
newvector
```

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

This is good as it shows those 2 steps. Perhaps, we can combine these 2 steps as follows:

```
newvector <- myvector[myvector < 5]
newvector
```

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

Another way to do this is to use of `which()`, which gives us the index of each element that satisfies the condition.

```
ind <- which(myvector < 5)  # Step 1
ind
```

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

```
newvector <- myvector[ind]  # Step 2
newvector
```

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

Or we can combine these 2 steps:

```r
newvector <- myvector[which(myvector < 5)]
newvector
```

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

Last one: find the 4's in `myvector` make them 8 (I know hard, but after a couple of tries it will seem easier):

```r
myvector <- c(1, 2, 3, 4, 5, 8, 4, 10, 12)
#I'll show you 3 ways to do that.

#1st way to show the steps
ind <- which(myvector==4) #identifying the index with 4
newvector <- myvector[ind] + 4 # adding them 4
myvector[ind] <- newvector #replacing those with the new values
myvector
```

```
## [1]  1  2  3  8  5  8  8 10 12
```

```r
#2nd and easier way
myvector[which(myvector==4)] <- myvector[which(myvector==4)] + 4
myvector
```

```
## [1]  1  2  3  8  5  8  8 10 12
```

```r
#3nd and easiest way
myvector[myvector==4] <- myvector[myvector==4] + 4
myvector
```

```
## [1]  1  2  3  8  5  8  8 10 12
```

What happens if the vector is a character vector? How can we subset it? We can use `grep()` as shown below:

```r
m <- c("about", "aboard", "board", "bus", "cat", "abandon")

#Now suppose that we need to pick the elements that contain "ab"

#Same steps again
a <- grep("ab", m) #similar to which() that gives us index numbers
a
```

```
## [1] 1 2 6
```

```r
newvector <- m[a]
newvector
```

```
## [1] "about"   "aboard"  "abandon"
```

2.8 Vectorization or vector operations

One of the biggest strengths of R is its use of vectorized operations. Lets see it in action!

```
x <- 1:10
x
```

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

```
x+1
```

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

```
2 * x
```

```
##  [1]  2  4  6  8 10 12 14 16 18 20
```

```
2 ^ x
```

```
##  [1]    2    4    8   16   32   64  128  256  512 1024
```

```
x ^ 2
```

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

```
sqrt(x)
```

```
##  [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427
##  [9] 3.000000 3.162278
```

```
log(x)
```

```
##  [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.7917595 1.9459101
##  [8] 2.0794415 2.1972246 2.3025851
```

Its like a calculator!

```
y <- 1:10
y
```

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

```
x + y
```

```
##  [1]  2  4  6  8 10 12 14 16 18 20
```

How about this:

```
y <- 1:11
x + y
```

```
## Warning in x + y: longer object length is not a multiple of shorter object
## length
```

```
##  [1]  2  4  6  8 10 12 14 16 18 20 12
```

OK, the warning is self-explanatory. But what's "12" at the end?
It's the sum of the first element of x, which is 1 and the last element of y, which is 11.

2.9 Set operations

```
x <- c(1, 2, 1, 3, 1)
y <- c(1, 1, 3, 6, 6, 5)
```

You can use the intersect() function to get values in both x and y:

```
intersect(x, y)
```

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

To get values in either x or y

```
union(x, y)
```

```
## [1] 1 2 3 6 5
```

To get values in x but not in y

```
setdiff(x, y)
```

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

```
setdiff(y, x)
```

```
## [1] 6 5
```

To check whether each element of x is inside y

```
is.element(x, y)
```

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

```
# or

x %in% y
```

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

2.10 Missing values

R uses NA to represent missing values indicating they are not available. In a data file, NA's are very common and have to be dealt with properly. Why?

```
x <- c(1, NA, 2, NA, 3)
mean(x)
```

```
## [1] NA
```

```r
sum(x)
```

```
## [1] NA
```

And it's contagious

```r
y <- 1:5
x+y
```

```
## [1]  2 NA  5 NA  8
```

To deal with NA's, we need to know how to find indices with missing values

```r
# Do we have and NA?
anyNA(x)
```

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

```r
# Which ones?
is.na(x)
```

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

```r
# Or
which(is.na(x))
```

```
## [1] 2 4
```

How to remove? But before removing them:

```r
mean(x, na.rm = TRUE)
```

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

So we may skip removing them from the data as many functions has built-in arguments

```r
x2 <- x[!is.na(x)]
x2
```

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

```r
# Or
x[complete.cases(x)]
```

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

2.11 Factors

Factor type is known as an "indicator" variable.

```
set.seed((123))
anim <- sample(animals, 100, replace = TRUE)
anim
```

```
##   [1] "donkey" "donkey" "donkey" "cat"    "donkey" "cat"    "cat"    "cat"   
##   [9] "donkey" "dog"    "cat"    "cat"    "dog"    "cat"    "donkey" "dog"   
##  [17] "donkey" "donkey" "dog"    "dog"    "dog"    "dog"    "donkey" "cat"   
##  [25] "donkey" "cat"    "dog"    "cat"    "donkey" "cat"    "dog"    "donkey"
##  [33] "donkey" "dog"    "donkey" "cat"    "dog"    "donkey" "dog"    "dog"   
##  [41] "cat"    "donkey" "donkey" "dog"    "donkey" "dog"    "donkey" "cat"   
##  [49] "dog"    "cat"    "dog"    "dog"    "donkey" "dog"    "cat"    "dog"   
##  [57] "dog"    "donkey" "dog"    "cat"    "dog"    "donkey" "dog"    "donkey"
##  [65] "cat"    "donkey" "cat"    "cat"    "donkey" "cat"    "cat"    "donkey"
##  [73] "donkey" "dog"    "cat"    "cat"    "dog"    "cat"    "dog"    "dog"   
##  [81] "cat"    "donkey" "donkey" "dog"    "cat"    "dog"    "cat"    "dog"   
##  [89] "donkey" "donkey" "cat"    "donkey" "dog"    "cat"    "cat"    "donkey"
##  [97] "cat"    "dog"    "donkey" "donkey"
```

```
table(anim)
```

```
## anim
##    cat    dog donkey 
##     32     33     35
```

Let's define `anim` vector as factor variable:

```
animf <-as.factor(anim)
levels(animf)
```

```
## [1] "cat"    "dog"    "donkey"
```

Chapter 3

Matrices

R stores matrices and arrays in a similar manner as vectors, but with the attribute called dimension. A matrix is an array that has two dimensions. Data in a matrix are organized into rows and columns. Matrices are commonly used while arrays are rare. We will not see arrays in this book. Matrices are homogeneous data structures, just like atomic vectors, but they can have 2 dimensions, rows and columns, unlike vectors.

Matrices can be created using the **matrix** function.

```
#Let's create 5 x 4 numeric matrix containing numbers from 1 to 20
mymatrix <- matrix(1:20, nrow = 5, ncol = 4)  #Here we order the number by columns
mymatrix
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    6   11   16
## [2,]    2    7   12   17
## [3,]    3    8   13   18
## [4,]    4    9   14   19
## [5,]    5   10   15   20
```

```
class(mymatrix)
```

```
## [1] "matrix" "array"
```

```
dim(mymatrix)
```

```
## [1] 5 4
```

```
mymatrix <- matrix(1:20, nrow = 5, ncol = 4, byrow = TRUE)
mymatrix
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12
## [4,]   13   14   15   16
## [5,]   17   18   19   20
```

We will be using two different variables. Following the usual mathematical convention, lower-case x (or any other letter), which stores a vector and capital `X`, which stores a matrix. We can do this because R is case sensitive.

3.1 Matrix Operations

Now some key matrix operations:

```
X <- matrix(1:9, nrow = 3, ncol = 3)
Y <- matrix(11:19, nrow = 3, ncol = 3)

A <- X + Y
A
```

```
##      [,1] [,2] [,3]
## [1,]   12   18   24
## [2,]   14   20   26
## [3,]   16   22   28
```

```
B <- X * Y
B
```

```
##      [,1] [,2] [,3]
## [1,]   11   56  119
## [2,]   24   75  144
## [3,]   39   96  171
```

```
#The symbol %*% is called pipe operator.
#And it carries out a matrix multiplication
#different than a simple multiplication.

C <- X%*%Y
C
```

```
##      [,1] [,2] [,3]
## [1,]  150  186  222
## [2,]  186  231  276
## [3,]  222  276  330
```

Note that X * Y is not a matrix multiplication. It is element by element multiplication. (Same for X / Y). Instead, matrix multiplication uses `%*%`. Other matrix functions include `t()` which gives the transpose of a matrix and `solve()` which returns the inverse of a square matrix if it is invertible.

`matrix()` function is not the only way to create a matrix. Matrices can also be created by combining vectors as columns, using `cbind()`, or combining vectors as rows, using `rbind()`. Look at this:

```
#Let's create 2 vectors.
x <- rev(c(1:9))  #this can be done by c(9:1). I wanted to show rev()
x
```

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

```
y <- rep(2, 9)
y
```

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

```
A <- rbind(x, y)
A
```

```
##   [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
## x    9    8    7    6    5    4    3    2    1
## y    2    2    2    2    2    2    2    2    2
```

```
B <- cbind(x, y)
B
```

```
##       x y
##  [1,] 9 2
##  [2,] 8 2
##  [3,] 7 2
##  [4,] 6 2
##  [5,] 5 2
##  [6,] 4 2
##  [7,] 3 2
##  [8,] 2 2
##  [9,] 1 2
```

```
#You can label each column and row
colnames(B) <- c("column1", "column2")
B
```

```
##       column1 column2
##  [1,]       9       2
##  [2,]       8       2
##  [3,]       7       2
##  [4,]       6       2
##  [5,]       5       2
##  [6,]       4       2
##  [7,]       3       2
##  [8,]       2       2
##  [9,]       1       2
```

Here are some operations very useful when using matrices:

```
rowMeans(A)
```

```
## x y
## 5 2
```

```
colMeans(B)
```

```
## column1 column2
##       5       2
```

```
rowSums(B)
```

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

```
colSums(A)
```

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

Last thing: When vectors are coerced to become matrices, they are column vectors. So a vector of length n becomes an $n \times 1$ matrix after coercion.

```
x
```

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

```
X <- as.matrix(x)
X
```

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

3.2 Subsetting Matrix

Like vectors, matrices can be subsetted using square brackets, `[ ]`. However, since matrices are two-dimensional, we need to specify both row and column indices when subsetting.

```
Y
```

```
##      [,1] [,2] [,3]
## [1,]   11   14   17
## [2,]   12   15   18
## [3,]   13   16   19
```

```
Y[1,3]
```

```
## [1] 17
```

```
Y[,3]
```

```
## [1] 17 18 19
```

```
Y[2,]
```

```
## [1] 12 15 18
```

```
Y[2, c(1, 3)] # If we need more than a column (row), we use c()
```

```
## [1] 12 18
```

Conditional subsetting is the same as before in vectors.
Let's solve this problem: what's the number in column 1 in Y when the number in column 3 is 18?

```
Y
```

```
##      [,1] [,2] [,3]
## [1,]   11   14   17
## [2,]   12   15   18
## [3,]   13   16   19
```

```
Y[Y[,3]==18, 1]
```

```
## [1] 12
```

```
#What are the numbers in a row when the number in column 3 is 18?
Y[Y[,3]==19, ]
```

```
## [1] 13 16 19
```

```
#Print the rows in Y when the number in column 3 is more than 17?
Y[Y[,3] > 17, ]
```

```
##      [,1] [,2] [,3]
## [1,]   12   15   18
## [2,]   13   16   19
```

We will see later how these conditional subsetting can be done much smoother with data frames.

3.3 R-Style Guide

The idea is simple: your R code, or any other code in different languages, should be written in a readable and maintainable style. Here is a blog by Roman Pahl that may help you develop a better styling in your codes. (You may find in some chapters and labs that my codes are not following the “good” styling practices. I am trying to improve!)

Next: Lists and data frames

Chapter 4

Cross-references

Cross-references make it easier for your readers to find and link to elements in your book.

4.1 Chapters and sub-chapters

There are two steps to cross-reference any heading:

1. Label the heading: `# Hello world {#nice-label}`.
   - Leave the label off if you like the automated heading generated based on your heading title: for example, `# Hello world` = `# Hello world {#hello-world}`.
   - To label an un-numbered heading, use: `# Hello world {-#nice-label}` or `{# Hello world .unnumbered}`.
2. Next, reference the labeled heading anywhere in the text using `\@ref(nice-label)`; for example, please see Chapter 4.
   - If you prefer text as the link instead of a numbered reference use: any text you want can go here.

4.2 Captioned figures and tables

Figures and tables *with captions* can also be cross-referenced from elsewhere in your book using `\@ref(fig:chunk-label)` and `\@ref(tab:chunk-label)`, respectively.

See Figure 4.1.

```r
par(mar = c(4, 4, .1, .1))
plot(pressure, type = 'b', pch = 19)
```

Don't miss Table 4.1.

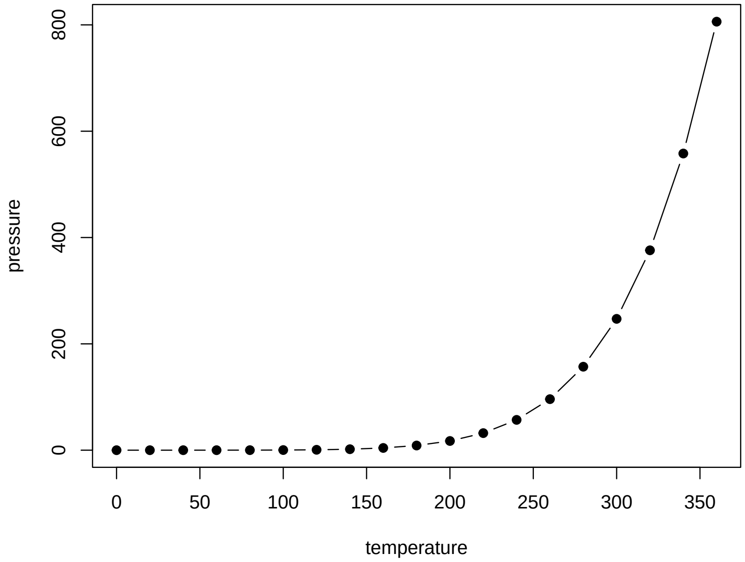


Figure 4.1: Here is a nice figure!

```
knitr::kable(
  head(pressure, 10), caption = 'Here is a nice table!',
  booktabs = TRUE
)
```

Table 4.1: Here is a nice table!

temperature	pressure
0	0.0002
20	0.0012
40	0.0060
60	0.0300
80	0.0900
100	0.2700
120	0.7500
140	1.8500
160	4.2000
180	8.8000

Chapter 5

Parts

You can add parts to organize one or more book chapters together. Parts can be inserted at the top of an .Rmd file, before the first-level chapter heading in that same file.

Add a numbered part: `# (PART) Act one {-}` (followed by `# A chapter`)

Add an unnumbered part: `# (PART\*) Act one {-}` (followed by `# A chapter`)

Add an appendix as a special kind of un-numbered part: `# (APPENDIX) Other stuff {-}` (followed by `# A chapter`). Chapters in an appendix are prepended with letters instead of numbers.

Chapter 6

Footnotes and citations

6.1 Footnotes

Footnotes are put inside the square brackets after a caret `^[]`. Like this one [1].

6.2 Citations

Reference items in your bibliography file(s) using `@key`.

For example, we are using the **bookdown** package [Xie, 2022] (check out the last code chunk in index.Rmd to see how this citation key was added) in this sample book, which was built on top of R Markdown and **knitr** [Xie, 2015] (this citation was added manually in an external file book.bib). Note that the `.bib` files need to be listed in the index.Rmd with the YAML `bibliography` key.

The RStudio Visual Markdown Editor can also make it easier to insert citations: https://rstudio.github.io/visual-markdown-editing/#/citations

[1] This is a footnote.

Chapter 7

Blocks

7.1 Equations

Here is an equation.

$$f\left(k\right) = \binom{n}{k} p^k \left(1 - p\right)^{n-k} \tag{7.1}$$

You may refer to using `\@ref(eq:binom)`, like see Equation (7.1).

7.2 Theorems and proofs

Labeled theorems can be referenced in text using `\@ref(thm:tri)`, for example, check out this smart theorem 7.1.

Theorem 7.1. *For a right triangle, if c denotes the* length *of the hypotenuse and a and b denote the lengths of the* ***other*** *two sides, we have*

$$a^2 + b^2 = c^2$$

Read more here https://bookdown.org/yihui/bookdown/markdown-extensions-by-bookdown.html.

7.3 Callout blocks

The R Markdown Cookbook provides more help on how to use custom blocks to design your own callouts: https://bookdown.org/yihui/rmarkdown-cookbook/custom-blocks.html

Chapter 8

Sharing your book

8.1 Publishing

HTML books can be published online, see: https://bookdown.org/yihui/bookdown/publishing.html

8.2 404 pages

By default, users will be directed to a 404 page if they try to access a webpage that cannot be found. If you'd like to customize your 404 page instead of using the default, you may add either a `_404.Rmd` or `_404.md` file to your project root and use code and/or Markdown syntax.

8.3 Metadata for sharing

Bookdown HTML books will provide HTML metadata for social sharing on platforms like Twitter, Facebook, and LinkedIn, using information you provide in the `index.Rmd` YAML. To setup, set the `url` for your book and the path to your `cover-image` file. Your book's `title` and `description` are also used.

This `gitbook` uses the same social sharing data across all chapters in your book-all links shared will look the same.

Specify your book's source repository on GitHub using the `edit` key under the configuration options in the `_output.yml` file, which allows users to suggest an edit by linking to a chapter's source file.

Read more about the features of this output format here:

https://pkgs.rstudio.com/bookdown/reference/gitbook.html

Or use:

```
?bookdown::gitbook
```

Bibliography

Yihui Xie. *Dynamic Documents with R and knitr*. Chapman and Hall/CRC, Boca Raton, Florida, 2nd edition, 2015. URL http://yihui.org/knitr/. ISBN 978-1498716963.

Yihui Xie. *bookdown: Authoring Books and Technical Documents with R Markdown*, 2022. URL https://CRAN.R-project.org/package=bookdown. R package version 0.28.