Chapter 1

Basics of R

1.15 Exercises

In these exercises, we use the following colour codes:

> ■ **Easy**: make sure you complete some of these before moving on. These exercises will follow examples in the text very closely.
>
> ♦ **Intermediate**: a bit harder. You will often have to combine functions to solve the exercise in two steps.
>
> ▲ **Hard**: difficult exercises! These exercises will require multiple steps, and significant departure from examples in the text.

We suggest you complete these exercises in an **R** markdown file. This will allow you to combine code chunks, graphical output, and written answers in a single, easy-to-read file.

1.15.1 Calculating

Calculate the following quantities:

1. ■ The sum of `100.1`, `234.9` and `12.01`

```
100.1 + 234.9 + 12.01
```

```
## [1] 347.01
```

2. ■ The square root of `256`

```
sqrt(256)
```

```
## [1] 16
```

3. ■ Calculate the `10`-based logarithm of `100`, and multiply the result with the cosine of π. *Hint:* see `?log` and `?pi`.

```
log10(100)*cos(pi)
```

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

4. ■ Calculate the cumulative sum ('running total') of the numbers 2,3,4,5,6.

```
cumsum(c(2,3,4,5,6))
```

```
## [1]  2  5  9 14 20
```

5. ♦ Calculate the cumulative sum of those numbers, but in reverse order. *Hint:* use the `rev` function.

```
cumsum(rev(c(2,3,4,5,6)))
```

```
## [1]  6 11 15 18 20
```

6. ♦ Find 10 random numbers between 0 and 100, rounded to the nearest whole number (*Hint:* you can use either `sample` or a combination of `round` and `runif`).

```
# sample
sample(0:100,10)
```

```
##  [1] 42 66 73 71 48 30 63 91  1 13
```

```
# runif and round
round(runif(10, 0, 100), 0)
```

```
##  [1] 23 65 30 24 92  6 38 73 71 36
```

1.15.2 Simple objects

Type the following code, which assigns numbers to objects `x` and `y`.

```
x <- 10
y <- 20
```

1. ■ Calculate the product of `x` and `y`

```
x * y
```

```
## [1] 200
```

2. ■ Store the result in a new object called `z`

```
z <- x * y
z
```

```
## [1] 200
```

3. ■ Inspect your workspace by typing `ls()`, and by clicking the `Environment` tab in Rstudio, and find the three objects you created.
4. ■ Make a vector of the objects `x`, `y` and `z`. Use this command,

   ```
   myvec <- c(x,y,z)
   ```

5. ■ Find the minimum, maximum, length, and variance of `myvec`.

```
min(myvec)
```

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

```
max(myvec)
```

```
## [1] 200
```

```
length(myvec)
```

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

```
var(myvec)
```

```
## [1] 11433.33
```

6. ■ Remove the `myvec` object from your workspace.

1.15.3 Working with a single vector

1. ■ The numbers below are the first ten days of rainfall amounts in 1996. Read them into a vector using the `c()` function (recall Section **??** on p. **??**).

```
 0.1  0.6 33.8  1.9  9.6  4.3 33.7  0.3  0.0  0.1
```

```
rainfall <- c(0.1,0.6, 33.8,  1.9,  9.6,  4.3, 33.7,  0.3,  0.0,  0.1)
```

Inspect Table **??** on page **??**, and answer the following questions:

2. ■ What was the mean rainfall, how about the standard deviation?

```
mean(rainfall)
```

```
## [1] 8.44
```

```
sd(rainfall)
```

```
## [1] 13.66473
```

3. ■ Calculate the cumulative rainfall ('running total') over these ten days. Confirm that the last value of the vector that this produces is equal to the total sum of the rainfall.

```
cumsum(rainfall)
```

```
##  [1]  0.1  0.7 34.5 36.4 46.0 50.3 84.0 84.3 84.3 84.4
```

4. ■ Which day saw the highest rainfall (write code to get the answer)?

```
which.max(rainfall)
```

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

1.15.4 Scripts

This exercise will make sure you are able to make a 'reproducable script', that is, a script that will allow you to repeat an analysis without having to start over from scratch. First, set up an **R** script (see Section **??** on page **??**), and save it in your current working directory.

1. ■ Find the `History` tab in Rstudio. Copy a few lines of history that you would like to keep to the script you just opened, by selecting the line with the mouse and clicking `To Source`.
2. ■ Tidy up your R script by writing a few comments starting with `#`.
3. ■ Now make sure your script works completely (that is, it is entirely *reproducible*). First clear the workspace (`rm(list=ls())` or click `Clear` from the `Environment` tab). Then, run the entire script (by clicking `Source` in the script window, top-right).

1.15.5 To quote or not to quote

This short exercise points out the use of quotes in **R**.

1. ■ Run the following code, which makes two numeric objects.

```
one <- 1
two <- 2
```

2. ♦ Run the following two lines of code, and look at the resulting two vectors. The first line makes a character vector, the second line a numeric vector by recalling the objects you just constructed. Make sure you understand the difference.

```
vector1 <- c("one","two")
vector2 <- c(one, two)
```

```
vector1 <- c("one","two")
vector2 <- c(one, two)
vector1
```

```
## [1] "one" "two"
```

```
vector2
```

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

3. ♦ The following lines of code contain some common errors that prevent them from being evaluated properly or result in error messages. Look at the code without running it and see if you can identify the errors and correct them all. Also execute the faulty code by copying and pasting the text into the console (not typing it, R studio will attempt to avoid these errors by default) so you get to know some common error messages (but not all of these result in errors!).

```
vector1 <- c('one', 'two', 'three', 'four, 'five', 'seven')

vec.var <- var(c(1, 3, 5, 3, 5, 1)
vec.mean <- mean(c(1, 3, 5, 3, 5, 1))

vec.Min <- Min(c(1, 3, 5, 3, 5, 1))

Vector2 <- c('a', 'b', 'f', 'g')
vector2
```

```
# vector1 <- c('one', 'two', 'three', 'four, 'five', 'seven')
# missing apostrophe after 'four'
vector1 <- c('one', 'two', 'three', 'four', 'five', 'seven')

# vec.var <- var(c(1, 3, 5, 3, 5, 1)
# missing closing parenthesis
# vec.mean <- mean(c(1, 3, 5, 3, 5, 1))
vec.var <- var(c(1, 3, 5, 3, 5, 1))
vec.mean <- mean(c(1, 3, 5, 3, 5, 1))

# vec.Min <- Min(c(1, 3, 5, 3, 5, 1))
# the 'min' function should have a lower-case 'm'
vec.Min <- min(c(1, 3, 5, 3, 5, 1))

# Vector2 <- c('a', 'b', 'f', 'g')
# vector2
# lower-case 'v' used here,
```

```
# upper-case 'V' used when defining variable in line above
vector2 <- c('a', 'b', 'f', 'g')
vector2
```

```
## [1] "a" "b" "f" "g"
```

1.15.6 Working with two vectors

1. ■ You have measured five cylinders, their lengths are:

   `2.1, 3.4, 2.5, 2.7, 2.9`

   and the diameters are :

   `0.3, 0.5, 0.6, 0.9, 1.1`

   Read these data into two vectors (give the vectors appropriate names).

```
lengths <- c(2.1, 3.4, 2.5, 2.7, 2.9)
diameters <- c(0.3, 0.5, 0.6, 0.9, 1.1)
```

2. ■ Calculate the correlation between lengths and diameters (use the `cor` function).

```
cor(lengths, diameters)
```

```
## [1] 0.3282822
```

3. ■ Calculate the volume of each cylinder (V = length * pi * (diameter / 2)^2).

```
# Calculate volumes and store in new vector
volumes <- lengths * pi * (diameters / 2)^2

# Look at the volumes
volumes
```

```
## [1] 0.1484403 0.6675884 0.7068583 1.7176658 2.7559622
```

4. ■ Calculate the mean, standard deviation, and coefficient of variation of the volumes.

```
mean(volumes)
```

```
## [1] 1.199303
```

```
sd(volumes)
```

```
## [1] 1.039402
```

```
sd(volumes) / mean(volumes)
```

```
## [1] 0.8666714
```

5. ♦ Assume your measurements are in centimetres. Recalculate the volumes so that their units are in cubic millimetres. Calculate the mean, standard deviation, and coefficient of variation of these new volumes.

```
volumes.mm <- 10 * lengths * pi * (10 * diameters / 2)^2

mean(volumes.mm)
```

```
## [1] 1199.303
```

```
sd(volumes.mm)
```

```
## [1] 1039.402
```

```
sd(volumes.mm) / mean(volumes.mm)
```

```
## [1] 0.8666714
```

6. ♦ You have measured the same five cylinders, but this time were distracted and wrote one of the measurements down twice:

   `2.1, 3.4, 2.5, 2.7, 2.9`

   and the diameters are :

   `0.3, 0.5, 0.6, 0.6, 0.9, 1.1`

   Read these data into two vectors (give the vectors appropriate names). As above, calculate the correlation between the vectors and store in a new vector. Also generate a vector of volumes based on these vectors and then calculate the mean and standard deviations of the volumes. Note that some steps result in errors, others in warnings, and some run perfectly fine. Why were some vectors created and others were not?

```
lengths1 <- c(2.1, 3.4, 2.5, 2.7, 2.9)
diameters1 <- c(0.3, 0.5, 0.6, 0.6, 0.9, 1.1)

# Calculate the correlation and store in a new vector
# (results in an error, new object not generated)
cor1 <- cor(lengths1, diameters1)
cor1

# Calculate volumes and store in new vector
# (results in a warning, new object is generated)
volumes1 <- lengths1 * pi * (diameters1 / 2)^2
volumes1

# Look at the volumes and calculate summary statistics
volumes1
mean(volumes1)
sd(volumes1)
```

1.15.7 Alphabet aerobics 1

For the second question, you need to know that the 26 letters of the Roman alphabet are conveniently accessible in R via `letters` and `LETTERS`. These are not functions, but vectors that are always loaded.

1. ♦ Read in a vector that contains "A", "B", "C" and "D" (use the `c()` function). Using `rep`, produce this:

   `"A" "A" "A" "B" "B" "B" "C" "C" "C" "D" "D" "D"`

   and this:

   `"A" "B" "C" "D" "A" "B" "C" "D" "A" "B" "C" "D"`

```
lets <- c("A", "B", "C", "D")

rep(lets, each = 3)
```

```
##  [1] "A" "A" "A" "B" "B" "B" "C" "C" "C" "D" "D" "D"
```

```
rep(lets, times = 3)
```

```
##  [1] "A" "B" "C" "D" "A" "B" "C" "D" "A" "B" "C" "D"
```

```
# The times argument can be omitted,
rep(lets, 3)
```

```
##  [1] "A" "B" "C" "D" "A" "B" "C" "D" "A" "B" "C" "D"
```

2. ♦ Draw 10 random letters from the lowercase alphabet, and sort them alphabetically (*Hint:* use `sample` and `sort`). The solution can be one line of code.

```
# First inspect letters, it is a vector:
letters
```

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

```
# Sample ten random letters (optionally, set replace=TRUE for sampling with replacement)
sample(letters, 10)
```

```
##  [1] "d" "m" "v" "q" "l" "j" "i" "r" "c" "g"
```

```
# And, in one line of code, sort them alphabetically
sort(sample(letters, 10))
```

```
##  [1] "a" "e" "g" "i" "j" "o" "r" "x" "y" "z"
```

3. ♦ Draw 5 random letters from each of the lowercase and uppercase alphabets, incorporating both into a single vector, and sort it alphabetically.

```
# If you like, you can store both sets of 5 letters in a vector, then combine:
low <- sample(letters, 5)
upp <- sample(LETTERS, 5)

# And then sort the combination of these vectors (use c() for that)
sort(c(low,upp))
```

```
##  [1] "a" "D" "q" "r" "R" "s" "t" "V" "W" "X"
```

```
# All of this can be achieved in one line of code
sort(c(sample(letters, 5),sample(LETTERS, 5)))
```

```
##  [1] "A" "c" "E" "f" "G" "i" "k" "R" "y" "Y"
```

4. ♦ Repeat the above exercise but sort the vector alphabetically in descending order.

```
# Inspect the help page ?sort, to find,
sort(c(sample(letters, 5),sample(LETTERS, 5)), decreasing = TRUE)
```

```
##  [1] "z" "S" "s" "r" "Q" "o" "N" "L" "d" "A"
```

1.15.8 Comparing and combining vectors

Inspect the help page `union`, and note the useful functions `union`, `setdiff` and `intersect`. These can be used to compare and combine two vectors. Make two vectors :

```
x <- c(1,2,5,9,11)
y <- c(2,5,1,0,23)
```

Experiment with the three functions to find solutions to these questions.

1. ♦ Find values that are contained in both `x` and `y`

```
# First read the vectors
x <- c(1,2,5,9,11)
y <- c(2,5,1,0,23)

# Having read the help page, it seems we need to use intersect()
intersect(x,y)
```

```
## [1] 1 2 5
```

2. ♦ Find values that are in `x` but not `y` (and vice versa).

```
# Note difference between,
setdiff(x,y)
```

```
## [1]  9 11
```

```
# and
setdiff(y,x)
```

```
## [1]  0 23
```

3. ♦ Construct a vector that contains all values contained in either `x` or `y`, and compare this vector to `c(x,y)`.

```
# union() finds values that are either in x or y,
union(x,y)
```

```
## [1]  1  2  5  9 11  0 23
```

```
# ... whereas c(x,y) simply concatenates (glues together) the two vectors
c(x,y)
```

```
##  [1]  1  2  5  9 11  2  5  1  0 23
```

1.15.9 Into the matrix

In this exercise you will practice some basic skills with matrices. Recall Section **??** on p. **??**.

1. ■ Construct a matrix with 10 columns and 10 rows, all filled with random numbers between 0 and 1 (see Section **??** on p. **??**).

```
m <- matrix(runif(100), ncol=10)
```

2. ■ Calculate the row means of this matrix (*Hint:* use `rowMeans`). Also calculate the standard deviation across the row means (now also use `sd`).

```
rowMeans(m)
```

```
##  [1] 0.5005640 0.4432295 0.5537734 0.3884319 0.3731112 0.5677393 0.3882833
##  [8] 0.4138591 0.6386653 0.5795310
```

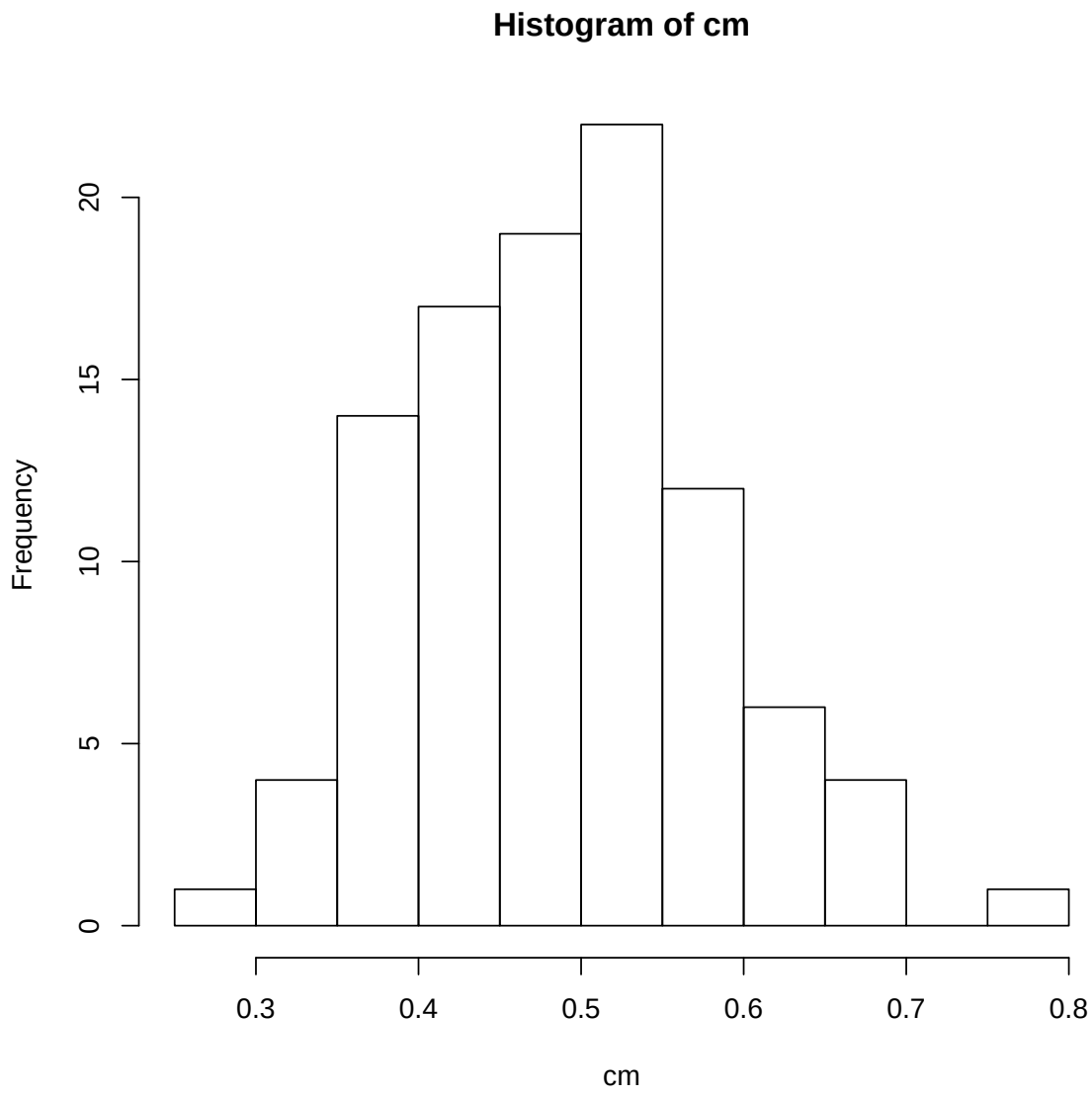
```
sd(rowMeans(m))
```

```
## [1] 0.0956676
```

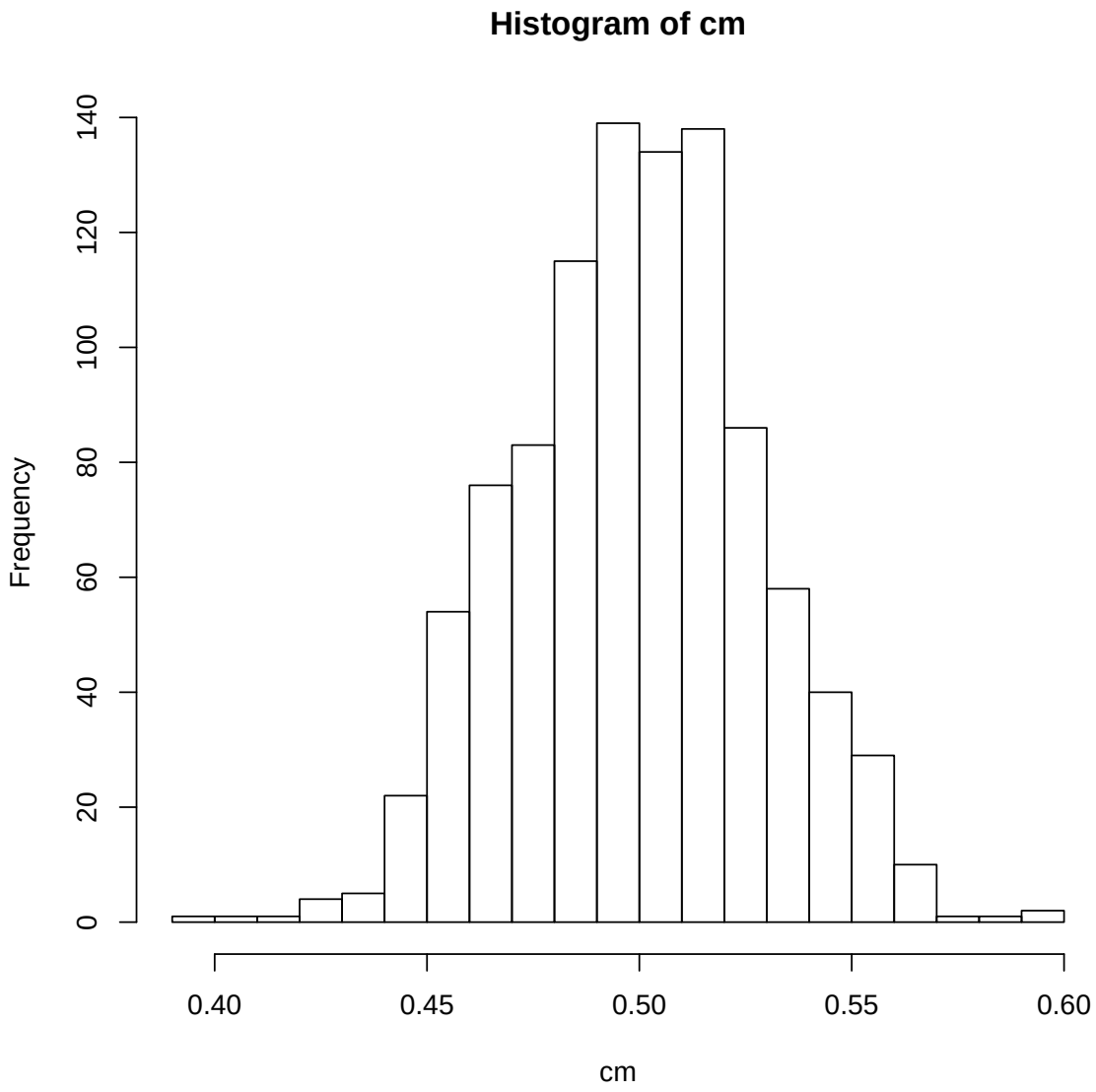
3. ▲ Now remake the above matrix with 100 columns, and 10 rows. Then calculate the column means (using, of course, `colMeans`), and plot a frequency diagram (a 'histogram') using `hist`. We will see this function in more detail in a later chapter, but it is easy enough to use as you just do `hist(myvector)`, where `myvector` is any numeric vector (like the column means). What sort

of shape of the histogram do you expect? Now repeat the above with more rows, and more columns.

```
# This resembles a normal distribution, a little bit?
m <- matrix(runif(100*10), ncol=100)
cm <- colMeans(m)
hist(cm)
```




```
# Yes, it does! This is the central limit theorem at work,
# the sum or mean of a bunch of random numbers follow a normal distribution.
m <- matrix(runif(1000*100), ncol=1000)
cm <- colMeans(m)
hist(cm, breaks=20)  # breaks argument to make more bins
```



1.15.10 Packages

This exercise makes sure you know how to install packages, and load them. First, read Section **??** (p. **??**).

1. ■ Install the `car` package (you only have to do this once for any computer).

```
install.packages("car")
```

2. ■ Load the `car` package (you have to do this every time you open Rstudio).

```
library(car)
```

3. ■ Look at the help file for `densityPlot` (Read Section **??** on p. **??**)

```
?densityPlot
```

4. ■ Run the example for `densityPlot` (at the bottom of the help file), by copy-pasting the example

into a script, and then executing it.

```
# Scroll down in the help file, and simply copy-paste the code into the R
# console.
```

5. ■ Run the example for `densityPlot` again, but this time use the `example` function:

   `example(densityPlot)`

   Follow the instructions to cycle through the different steps.

   ```
   example(densityPlot)
   ```

6. ♦ Explore the contents of the `car` package by clicking first on the Packages tab, then finding the `car` package, and clicking on that. This way, you can find out about all functions a package contains (which, normally, you hope to avoid, but sometimes it is the only way to find what you are looking for). The same list can be obtained with the command `library(help=car)`, but that displays a list that is not clickable, so probably not as useful.

1.15.11 Save your work

We strongly encourage the use of R markdown files or scripts that contain your entire workflow. In most cases, you can just rerun the script to reproduce all outputs of the analysis. Sometimes, however, it is useful to save all resulting objects to a file, for later use. First read Section **??** on p. **??**.

Before you start this exercise, first make sure you have a reproducable script as recommended in the third exercise to this chapter.

1. ■ Run the script, save the workspace, give the file an appropriate name (it may be especially useful to use the date as part of the filename, for example 'results_2015-02-27.RData'.
2. ■ Now close and reopen Rstudio. If you are in the correct working directory (it should become a habit to check this with the `getwd` function, do it now!), you can load the file into the workspace with the `load` function. Alternatively, in Rstudio, go to `File/Open File...` and load the workspace that way.

Chapter 2

Generating, reading and extracting data

2.6 Exercises

In these exercises, we use the following colour codes:

- ■ **Easy**: make sure you complete some of these before moving on. These exercises will follow examples in the text very closely.
- ♦ **Intermediate**: a bit harder. You will often have to combine functions to solve the exercise in two steps.
- ▲ **Hard**: difficult exercises! These exercises will require multiple steps, and significant departure from examples in the text.

We suggest you complete these exercises in an **R** markdown file. This will allow you to combine code chunks, graphical output, and written answers in a single, easy-to-read file.

2.6.1 Working with a single vector 2

Recall Exercise 1.15.3 on p. 5. Read in the `rainfall` data once more. We now practice subsetting a vector (see Section **??**, p. **??**).

1. ■ Take a subset of the rainfall data where rain is larger than 20.

```
rainfall[rainfall > 20]
```

```
## [1] 33.8 33.7
```

2. ■ What is the mean rainfall for days where the rainfall was at least 4?

```
mean(rainfall[rainfall >= 4])
```

```
## [1] 20.35
```

3. ■ Subset the vector where it is either exactly zero, or exactly 0.6.

```
rainfall[rainfall == 0 | rainfall == 0.6]
```

```
## [1] 0.6 0.0
```

```
# Alternative solution,
rainfall[rainfall %in% c(0, 0.6)]
```

```
## [1] 0.6 0.0
```

2.6.2 Alphabet aerobics 2

The 26 letters of the Roman alphabet are conveniently accessible in R via `letters` and `LETTERS`. These are not functions, but vectors that are always loaded.

1. ■ What is the 18th letter of the alphabet?

```
# You could have guessed,
LETTERS[18]

## [1] "R"

# or letters[18]
```

2. ■ What is the last letter of the alphabet (don't guess, write code)?

```
# Use length() to index the vector,
letters[length(letters)]

## [1] "z"

# You could use you knowledge that the alphabet contains 26 letters, but the above
# solution is more general.
```

3. ♦ Use `?sample` to figure out how to sample with replacement. Generate a random subset of fifteen letters. Are any letters in the subset duplicated? *Hint:* use the `any` and `duplicated` functions. Which letters?

```
# 1. Random subset of 15 letters
let15 <- sample(letters, 15, replace = TRUE)

# 2. Any duplicated?
any(duplicated(let15))

## [1] TRUE

# 3. Which are duplicated?
# This tells us the index of the letter that is replicated,
which(duplicated(let15))

## [1]  6  9 11

# We can use it to index letters to find the actual letters
let15[which(duplicated(let15))]

## [1] "h" "s" "c"
```

2.6.3 Basic operations with the Cereal data

For this exercise, we will use the Cereal data, see Section **??** (p. **??**) for a description of the dataset.

1. ■ Read in the dataset, look at the first few rows with `head` and inspect the data types of the variables in the dataframe with `str`.

```
cereals <- read.csv("Cereals.csv")
str(cereals)
```