1 Introduction to R

1.1. Introduction

Statistics, conceived broadly, is the process of "getting meaning from data".[1] We perform statistical analyses on datasets to further our understanding. As such, statistics is a fundamentally human process that makes large sets of numbers embedded in complex datasets amenable to human cognition.

Some people think of statistics as being only the very last step of the empirical process. You design your study, you collect your data, *then* you perform a statistical analysis of the data. This is a narrow view of statistics.

This book assumes a broad view. In particular, I view the process of getting the data in shape for an analysis as part of your actual analysis. Thus, what people talk of as 'preprocessing' or 'data wrangling' is an integral part of statistics. In fact, most of your time during an actual analysis will be spent on wrangling with the data. The first two chapters focus on teaching you the skills for doing this. I will also teach you the first steps towards an efficient workflow, as well as how to do data processing in a reproducible fashion.

R makes all of this easy—once you get used to it. There's absolutely no way around a command-line-based tool if you want to be efficient with data. You need the ability to type in programming commands, rather than dealing with the data exclusively via some graphical user interface. Using a point-and-click-based software tool such as Excel slows you down and is prone to error. More importantly, it makes it more difficult for others to reproduce your analysis. Pointing and clicking yourself through some interface means that another researcher will have a difficult time tracing your steps. You need a record of your analysis in the form of programming code.

As a telling example of what can go wrong with processing your data, consider the case of 'austerity's spreadsheet error', which has been widely covered in the news:[2] Reinhart and Rogoff (2010) published an influential paper which showed that, on average, economic growth was diminished when a country's debt exceeds a certain limit. Many policy makers used this finding as an argument for austerity politics. However, the MIT graduate student Thomas Herndon discovered that the results were

1 This phrase is used by Michael Starbird in his introduction to statistics for *The Great Courses*.

2 For example [accessed, October 12, 2018]:
www.theguardian.com/politics/2013/apr/18/uncovered-error-george-osborne-austerity
www.bbc.co.uk/news/magazine-22223190
www.aeaweb.org/articles?id=10.1257/aer.100.2.573

based on a spreadsheet error: certain rows were accidentally omitted in their analysis. Including these rows led to drastically different results, with different implications for policy makers. The European Spreadsheet Risks Interest Group curates a long list of spreadsheet "horror stories".[3] The length of this list is testament to the fact that it is difficult *not* to make errors when using software such as Excel.

The upshot of this discussion is that there's no way around learning a bit of R. The fact that this involves typing in commands rather than clicking yourself through some graphical interface may at first sight seem daunting. But don't panic—this book will be your guide. Those readers who are already experienced with R may skim through the next two chapters or skip them altogether.

1.2. Baby Steps: Simple Math with R

You should have installed R and RStudio by now. R is the actual programming language that you will use throughout this book. RStudio makes managing projects easier, thus facilitating your workflow. However, it is R embedded within RStudio that is the actual workhorse of your analysis.

When you open up RStudio, the first thing you see is the console, which is your window into the world of R. The console is where you type in commands, which R will then execute. Inside the console, the command line starts with the symbol '>'. Next to it, you will see a blinking cursor '|'. The blinking is R's way of telling you that it's ready for you to enter some commands.

One way to think about R is that it's just an overblown calculator. Type in '`2 + 2`' and press ENTER:

```
2 + 2
```

```
[1] 4
```

This is addition. What about subtraction?

```
3 - 2
```

```
[1] 1
```

What happens if you supply an incomplete command, such as '`3 -`', and then hit ENTER? The console displays a plus sign. Hitting ENTER multiple times yields even more plus signs.

```
3 -
```

```
+
+
+
+
```

3 www.eusprig.org/horror-stories.htm [accessed August 26, 2019]

You are stuck. In this context, the plus sign has nothing to do with addition. It's R's way of showing you that the last command is incomplete. There are two ways out of this: either supplying the second number, or aborting by pressing ESC. Remember this for whenever you see a plus sign instead of a '>' in the console.

When you are in the console and the cursor is blinking, you can press the up and down arrows to toggle through the history of executed commands. This may save you some typing in case you want to re-execute a command.

Let's do some division, some multiplication and taking a number to the power of another number:

```
3 / 2 # division
```

```
[1] 1.5
```

```
3 * 2 # multiplication
```

```
[1] 6
```

```
2 ^ 2 # two squared
```

```
[1] 4
```

```
2 ^ 3 # two to the power of three
```

```
[1] 8
```

You can stack mathematical operations and use brackets to overcome the default order of operations. Let's compare the output of the following two commands:

```
(2 + 3) * 3
```

```
[1] 15
```

```
2 + (3 * 3)
```

```
[1] 11
```

The first is 2 + 3 = 5, multiplied by 3, which yields 15. The second is 3 * 3 = 9 plus 2, which yields 11. Simple mathematical operations have the structure 'A operation B', just as in mathematics. However, most 'functions' in R look different from that. The general structure of an R function is as follows:

```
function(argument1, argument2, ...)
```

A function can be thought of as a verb, or an action. Arguments are the inputs to functions—they are what functions act on. Most functions have at least one argument.

If a function has multiple arguments, they are separated by commas. Some arguments are obligatory (the function won't run without being supplied a specific argument). Other arguments are optional.

This is all quite abstract, so let's demonstrate this with the square root function `sqrt()`:

```
sqrt(4)
```

```
[1] 2
```

This function only has one obligatory argument. It needs a number to take the square root of. If you fail to supply the corresponding argument, you will get an error message.

```
sqrt()
```

```
Error in sqrt() : 0 arguments passed to 'sqrt' which
requires 1
```

Another example of a simple function is the absolute value function `abs()`. This function makes negative numbers positive and leaves positive numbers unchanged, as demonstrated by the following two examples.

```
abs(-2)
```

```
[1] 2
```

```
abs(2)
```

```
[1] 2
```

1.3. Your First R Script

So far, you have typed things straight into the console. However, this is exactly what you *don't* want to do in an actual analysis. Instead, you prepare an R script, which contains everything needed to reproduce your analysis. The file extension .R is used for script files. Go to RStudio and click on 'File' in the menu tab, then click on 'New File' in the pop-down menu, then 'R Script'.

Once you have opened up a new script file, your RStudio screen is split into two halves. The top half is your R script; the bottom half is the console. Think of your R script as the recipe, and the R console as the kitchen that cooks according to your recipe. An alternative metaphor is that your R script is the steering wheel, and the console is the engine.

Type in the following command into your R script (*not* into the console) and press ENTER.

```
2 * 3
```

Nothing happens. The above command is only in your script—it hasn't been executed yet. To make something happen, you need to position your cursor in the line of the command and press the green arrow in the top right of the R script window. This will 'send' the instructions from the script down to the console, where R will execute the command. However, rather than using your mouse to click the green button, I strongly encourage you to learn the keyboard shortcut for running a command, which is COMMAND + ENTER on a Mac and ALT + ENTER on a PC.

When working with R, try to work as much as possible in the script, which should contain everything that is needed to reproduce your analysis. Scripts also allow you to comment your code, which you can do with the hashtag symbol '#'. Everything to the right of the hashtag will be ignored by R. Here's how you could comment the above command:

```
# Let's multiply two times three:
2 * 3
```

Alternatively, you can have the comment in the same line. Everything up to the comment is executed; everything after the comment is ignored.

```
2 * 3  # Multiply two times three
```

Commenting is crucial. Imagine getting back to your analysis after a two-year break, which happens surprisingly often. For a reasonably complex data analysis, it may take you hours to figure out what's going on. Your future self will thank you for leaving helpful comments. Perhaps even more importantly, comprehensive commenting facilitates reproducibility since other researchers will have an easier time reading your code.

Different coders have different approaches, but I developed the personal habit of 'commenting before coding'. That is, I write what I am going to do next in plain language such as '`# Load in data:`'. I supply the corresponding R code only after having written the comment. As a result of this practice, my scripts are annotated from the get-go. It also helps my thinking, because each comment states a clear goal before I start hacking away at the keyboard.

Maybe you don't want to adopt this habit, and that's OK. Programming styles are deeply personal and it will take time to figure out what works for you. However, regardless of how you write your scripts, write them with a future audience in mind.

1.4. Assigning Variables

Let's use the R script to assign variables. Write the following command into your R script (not into the console).

```
x <- 2 * 3
```

If you send this command in the console (remember: COMMAND + ENTER or CTRL + ENTER), nothing happens. The leftwards pointing arrow '<−' is the 'assign

operator'. It assigns whatever is to the right of the operator to an 'object' that bears the name on the left. You decide the name yourself. In this case, the object is called `x`, and it stores the output of the operation '`2 * 3`'. I like to think of the arrow '`<-`' as metaphorically putting something into a container. Imagine a container with '`x`' written on it that contains the number 6. By typing in the container's name, you retrieve its content.

```
x
```

```
[1] 6
```

You will also see code that uses a different assignment operator, namely '`=`' as opposed to '`<-`'.

```
x = 2 * 3
```

```
[1] 6
```

There is a subtle difference between the two assignment operators that I won't go into here. For now, it's best if you stick to '`<-`', which is also what most R style guides recommend. As you will be using the '`<-`' assign operator constantly, make sure to learn its shortcut: ALT + minus.

The `x` can be used in further mathematical operations as if it's a number.

```
x / 2
```

```
[1] 3
```

Crucially, R is case-sensitive. Typing in capital '`X`' yields an error message because the object capital '`X`' does not exist in your 'working environment'.

```
X
```

```
Error: object 'X' not found
```

To retrieve a list of all objects in your current working environment, type `ls()` (this function's name stands for 'list').

```
ls()
```

```
[1] "x"
```

Since you just started a new session and only defined one object, your working environment only contains the object `x`. Notice one curiosity about the `ls()` function: it is one of the few functions in R that doesn't need any arguments, which is why running the function without an argument didn't produce an error message.

1.5. Numeric Vectors

Up to this point, the object `x` only contained one number. One handy function to create multi-number objects is the concatenate function `c()`. The following code uses this function to put the numbers 2.3, 1, and 5 into one object. As before, typing the object's name reveals its content. This command overrides the previous `x`.

```
x <- c(2.3, 1, 5)

x
```

```
[1] 2.3 1.0 5.0
```

The object `x` is what is called a vector. In R, a 'vector' simply is a list of numbers. You can check how long a vector is with the `length()` function.

```
length(x)
```

```
[1] 3
```

As you will see shortly, there are different types of vectors. The vector `x` contains numbers, so it is a vector of type 'numeric'. The `mode()` and `class()` function can be used to assess vector types.[4] People often either talk of a vector's 'atomic mode' or 'atomic class'.

```
mode(x)
```

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

```
class(x)
```

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

It's important to know what type of vector you are dealing with, as certain mathematical operations can only be applied to numeric vectors.

Let's create a sequence of integers from 10 to 1 using the colon function. In R, the colon is a sequence operator that creates an integer sequence from the first number to the last number.

```
mynums <- 10:1

mynums
```

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

4 These functions are equivalent for simple vectors, but exhibit different behavior for more complex objects (not covered here).

Given that `mynums` is a numeric vector, it is possible to perform all kinds of new mathematical operations on it. The following code showcases some useful summary functions.

```
sum(mynums) # sum
```

```
[1] 55
```

```
min(mynums) # smallest value (minimum)
```

```
[1] 1
```

```
max(mynums) # largest value (maximum)
```

```
[1] 10
```

```
range(mynums)  # minimum and maximum together
```

```
[1] 1 10
```

```
diff(range(mynums))  # range: difference between min and max
```

```
[1] 9
```

```
mean(mynums)# arithmetic mean, see Ch. 3
```

```
[1] 5.5
```

```
sd(mynums)  # standard deviation, see Ch. 3
```

```
[1] 3.02765
```

```
median(mynums)     # median, see Ch. 3
```

```
[1] 5.5
```

If you use a function such as subtraction or division on a numeric vector, the function is repeated for all entries of the vector.

```
mynums - 5  # subtract 5 from every number
```

```
[1] 5 4 3 2 1 0 -1 -2 -3 -4
```

```
mynums / 2  # divide every number by two
```

```
[1] 5.0 4.5 4.0 3.5 3.0 2.5 2.0 1.5 1.0 0.5
```

1.6. Indexing

Often, you need to operate on specific subsets of data. Vectors can be indexed by position. Conceptually, it is important to separate a vector's position from the value that's stored at said position. Because each vector in R is ordered, it is possible to use indices to ask for the first data point, the second data point, and so on.

```
mynums[1]   # retrieve value at first position
```

```
[1] 10
```

```
mynums[2]   # retrieve value at second position
```

```
[1] 9
```

```
mynums[1:4] # retrieve first four values
```

```
[1] 10 9 8 7
```

Putting a minus in front of an index spits out everything inside a vector *except for that index.*

```
mynums[-2]  # retrieve everything except second position
```

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

Now that you know the basic of indexing, you can also understand why there's a '`[1]`' in front of each of line of R output you've seen so far. Creating a longer integer sequence will help you wrap your head around this.

```
1:100
```

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

The '`[1]`' simply means 'first position'. Whenever there's a line break, R will show the position of the first value that starts a new row. The numbers in the square brackets to the left may be different on your screen: this will depend on your screen resolution and the resolution of the font printed in the R console.

1.7. Logical Vectors

Calling data by position is impractical for large datasets. If you had, for example, a dataset with 10,000 rows, you wouldn't necessarily know in advance that a particular data point of interest is at the 7,384th position. You need to be able to ask for specific values, rather than having to know the position in advance. For this, logical statements are useful.

```
mynums > 3  # Which values are larger than 3?
```

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

The statement `mynums > 3` uses the 'greater than' sign '>'. This line of code is essentially the same as asking: 'Is it the case that `mynums` is larger than 3?' Because the vector `mynums` contains multiple entries, this question is repeated for each position, each time returning a `TRUE` value if the number is actually larger than 3, or a `FALSE` if the number is smaller than 3.

The logical operator '>=' translates to 'larger than or equal to'. The operators '<' and '<=' mean 'smaller than' and 'smaller than or equal to'. Have a look at what the following commands do, keeping in mind that the `mynums` vector contains the integer sequence `10:1`.

```
mynums >= 3  # Larger than or equal to 3?
```

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

```
mynums < 4  # Smaller than 4?
```

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

```
mynums <= 4  # Smaller than or equal to 4?
```

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

```
mynums == 4  # Equal to 4?
```

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

```
mynums != 4  # Not equal to 4?
```

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

The result of performing a logical operation is actually a vector itself. To illustrate this, the following code stores the output of a logical operation in the object `mylog`. The `class()` function shows that `mylog` is 'logical'.

```
mylog <- mynums >= 3

class(mylog)
```

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

Logical vectors can be used for indexing. The following code only returns those values that are larger than or equal to 3.

```
mynums[mylog]
```

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

Perhaps it is more transparent to put everything into one line of code rather than defining separate vectors:

```
mynums[mynums >= 3]
```

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

It may help to paraphrase this command as if directly talking to R: 'Of the vector `mynums`, please retrieve those numbers for which the statement `mynums >= 3` is `TRUE`'.

1.8. Character Vectors

Almost all analysis projects involve some vectors that contain text, such information about a participant's age, gender, dialect, and so on. For this, 'character' vectors are used.

The code below uses quotation marks to tell R that the labels `'F'` and `'M'` are character strings rather than object names or numbers. You can use either single quotes or double quotes, but you should not mix the two.[5]

```
gender <- c('F', 'M', 'M', 'F', 'F')
```

The character-nature of the `gender` vector is revealed when printing the vector into the console, which shows quotation marks.

5 I use single quotes because it makes the code look 'lighter' than double quotes, and because it saves me one additional key stroke on my keyboard.

```
gender
```

```
[1] "F" "M" "M" "F" "F"
```

As before, the type of vector can be verified with `class()`.

```
class(gender)
```

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

As before, you can index this vector by position or using logical statements.

```
gender[2]
```

```
[1] "M"
```

```
gender[gender == 'F']
```

```
[1] "F" "F" "F"
```

However, it is impossible to perform mathematical functions on this vector, and doing so will spit out a warning message.

```
mean(gender)
```

```
[1] NA
Warning message:
In mean.default(gender) : argument is not numeric or logi-
cal: returning NA
```

1.9. Factor Vectors

A fourth common type of vector is the 'factor' vector. The following code overrides the original `gender` vector with a new version that has been converted to a factor using `as.factor()`.[6]

```
gender <- as.factor(gender)

gender
```

```
[1] F M M F F
Levels: F M
```

The output shows text, but, unlike the character vector, there are no quotation marks. The 'levels' listed below the factor are the unique categories in the vector. In

6 There are also `as.numeric()`, `as.logical()`, and `as.character()`. Perhaps play around with these functions to see what happens (and what can go wrong) when you convert a vector of one type into another type. For example, what happens when you apply `as.numeric()` to a logical vector? (This may actually be useful in some circumstances.)

this case, the vector `gender` contains 5 data points, which are all tokens of the types `"F"` and `"M"`. The levels can be accessed like this:

```
levels(gender)
```

```
[1] "F" "M"
```

The issue with factor vectors is that the levels are fixed. Let's see what happens when you attempt to insert a new value `'not_declared'` into the third position of the `gender` vector.

```
gender[3] <- 'not_declared'
```

```
Warning message:
In '[<-.factor'('*tmp*', 3, value = "not_declared") :
 invalid factor level, NA generated
```

```
gender
```

```
[1] F  M  <NA> F  F
Levels: F M
```

The third position is now set to `NA`, a missing value. This happened because the only two levels allowed are `'F'` and `'M'`. To insert the new value `'not_declared'`, you first need to change the levels.

```
levels(gender) <- c('F', 'M', 'not_declared')
```

Let's re-execute the insertion statement.

```
gender[3] <- 'not_declared'
```

This time around, there's no error message, because `'not_declared'` is now a valid level of the `gender` vector. Let's check whether the assignment operation achieved the expected outcome:

```
gender
```

```
[1] M            F            not_declared M           M
Levels: M F not_declared
```

1.10. Data Frames

Data frames are basically R's version of a spreadsheet. A data frame is a two-dimensional object, with rows and columns. Each column contains a vector.

Let's build a data frame. The following command concatenates three names into one vector.

```
participant <- c('louis', 'paula', 'vincenzo')
```

Next, the `data.frame()` function is used to create a data frame by hand. Each argument of this function becomes a column. Here, the `participant` vector will be the first column. The second column is named `score`, and a vector of three numbers is supplied.

```
mydf <- data.frame(participant, score = c(67, 85, 32))

mydf
```

```
participant score
1   louis      67
2   paula      85
3 vincenzo     32
```

Because a data frame is two-dimensional, you can ask for the number of rows or columns.

```
nrow(mydf)
```

```
[1] 3
```

```
ncol(mydf)
```

```
[1] 2
```

The column names can be retrieved like this:

```
colnames(mydf)
```

```
[1] "participant" "score"
```

Data frames can be indexed via the name of the column by using the dollar sign operator '$'.

```
mydf$score
```

```
[1] 67 85 32
```

This results in a numeric vector. You can then apply summary functions to this vector, such as computing the mean:

```
mean(mydf$score)
```

```
[1] 61.33333
```

You can check the structure of the data frame with the `str()` function.

```
str(mydf)
```

```
'data.frame':  3 obs. of 2 variables:
 $ participant: Factor w/ 3 levels "louis","paula",..: 1 2 3
 $ resp       : num 67 85 32
```

This function lists all the columns and their vector types. Notice one curiosity: the `participant` column is indicated to be a factor vector, even though you only supplied a character vector! The `data.frame()` function secretly converted your character vector into factor vector.

The `summary()` function provides a useful summary, listing the number of data points for each participant, as well as what is called a 'five number summary' of the `score` column (see Chapter 3).

```
summary(mydf)
```

```
 participant   score
louis   :1  Min.   :32.00
paula   :1  1st Qu.:49.50
vincenzo:1  Median :67.00
            Mean   :61.33
            3rd Qu.:76.00
            Max.   :85.00
```

To index rows or columns by position, you can use square brackets. However, due to data frames being two-dimensional, this time around you need to supply identifiers for rows and columns, which are separated by comma, with rows listed first.

```
mydf[1,]  # first row
```

```
 participant score
1      louis    67
```

```
mydf[, 2]  # second column
```

```
[1] 67 85 32
```

```
mydf[1:2,]  # first two rows
```

```
 participant score
1      louis    67
2      paula    85
```

And these operations can be stacked, such as:

```
mydf[, 1][2]# first column, second entry
```

```
[1] paula
Levels: louis paula vincenzo
```

The last statement can be unpacked as follows: the first indexing operation extracts the first column. The output of this operation is itself a unidimensional vector, to which you can then apply another indexing operation.

What if you wanted to extract the row for the participant called Vincenzo? For this, logical statements can be used.

```
mydf[mydf$participant == 'vincenzo',]
```

```
  participant score
3  vincenzo     32
```

Let me paraphrase this command into plain English: 'Using the data frame `mydf`, extract only those rows for which the statement `mydf$participant == 'vincenzo'` returns a `TRUE` value.' Notice how the result of this indexing operation is a data frame with one row. Because the result is a data frame, you can use the dollar sign operator to further index a specific column, as in the following command:

```
mydf[mydf$participant == 'vincenzo',] $score
```

```
[1] 32
```

1.11. Loading in Files

When loading in files into your current working environment, R needs to know which folder on your computer to look at, what is called the 'working directory'. Use `getwd()` to check the current working directory.

```
getwd() # output specific to one's computer
```

```
[1] "/Users/bodo"
```

This is the folder on your machine where R currently 'looks at', and it is where it expects your files. You can look at the folder's content from within R, using the `list.files()` function. This should remind you of the `ls()` function. Whereas `ls()` displays the R-internal objects, `list.files()` displays R-external files.

```
list.files() # output not shown (specific to your machine)
```

To change your working directory to where your files are, you *can* use `setwd()`. Crucially, this command will be computer-specific, and it will differ between Mac/Linux and Windows. Rather than explaining all of this, I recommend you to set your working directory in RStudio, where you can find the menu item 'Set Working Directory' under the drop-down menu item 'Session'. Once you've clicked on 'Set Working Directory', navigate to the folder where the files for this book are (if you haven't downloaded those files yet, now is your chance!). It is OK if the files in the folder are not displayed or grayed out. You can still click 'Open' here as your goal right now is not to select the file, but the folder where the file is located at. Once the working directory has been set, you load the 'nettle_1999_climate.csv' file using

`read.csv()` as follows. The .csv extension means that this is a comma-separated file (internally, columns are separated by commas). This dataset is taken from Nettle's (1999) book *Linguistic Diversity* and will be explained in more detail in later chapters.

```
nettle <- read.csv('nettle_1999_climate.csv')
```

If there's no error message, you have successfully loaded the file. If there *is* an error message, check whether you have typed the file name correctly. If that is the case, use `list.files()` to check whether the file is actually in the folder. If that is not the case, you may not have set the working directory successfully, which you can assess using `getwd()`.

Whenever you load a file into R, the next step should be to check its content. The `head()` function shows the first six rows of the `nettle` data frame (the 'head' of the data frame).

```
head(nettle)
```

```
     Country Population Area  MGS Langs
1    Algeria       4.41 6.38 6.60    18
2     Angola       4.01 6.10 6.22    42
3  Australia       4.24 6.89 6.00   234
4 Bangladesh       5.07 5.16 7.40    37
5      Benin       3.69 5.05 7.14    52
6    Bolivia       3.88 6.04 6.92    38
```

The `tail()` function displays *last* six rows (the 'tail' of the data frame).

```
tail(nettle)
```

```
     Country Population Area  MGS Langs
69 Venezuela       4.31 5.96 7.98    40
70   Vietnam       4.83 5.52 8.80    88
71     Yemen       4.09 5.72 0.00     6
72     Zaire       4.56 6.37 9.44   219
73    Zambia       3.94 5.88 5.43    38
74  Zimbabwe       4.00 5.59 5.29    18
```

It is important to discuss a few more points regarding file management for data analysis projects. First, when you quit RStudio, a 'Quit R session' question will pop up, asking you whether you want to save the content of your workspace. Click 'No'. Each time you open up R or RStudio, your new R session should open a new working environment and load in the required files. You don't want objects from previous analysis projects to be floating around, which may slow things down and cause naming conflicts (two objects having the same name). Instead, you want to keep all your data external to R.

There's also absolutely no problem if you override or messed up an R object within a session. Let's say you accidentally override the `Langs` column with NAs (missing values).

```
nettle$Langs <- NA
```

This is not a problem at all. Simply re-execute the entirety of your script up to the point where the mistake happened, and you will have everything back to where it was. For most simple analyses, there's not really any purpose for 'backing up' R objects.[7]

It is a good idea to structure your workflow around .csv files, as these are quite easy to deal with in R. Of course, R supports many other file types. For example, the 'example_file.txt' in the book's folder is a tab-separated file; that is, columns are separated by tabs (which are written '`\t`' computer-internally). You can use `read.table()` to load in this file as follows (ignore the warning message):[8]

```
mydf <- read.table('example_file.txt',
                   sep = '\t', header = TRUE)
```

```
Warning message:
In read.table("example_file.txt", sep = "\t", header = TRUE):
 incomplete final line found by readTableHeader on
'example_file.txt'
```

```
mydf
```

```
  amanda jeannette gerardo
1      3         1       2
2      4         5       6
```

The `read.table()` function requires you to specify a separator (the argument `sep`), which is `'\t'` for tabs in this case. The `header = TRUE` argument is required when the first row of the table contains column names. Sometimes, column name information is stored in a separate file, in which case you need `header = FALSE`.

There are too many file types to cover in this book. Often, Google will be your friend. Alternatively, you may resort to Excel to open up a file and save it in a format that you can easily read into R, such as .csv. However, let me tell you about a general strategy for dealing with file types where you don't know the internal structure. Load the file into R as text, using `readLines()`.

```
x <- readLines('example_file.txt', n = 2)

x
```

```
[1] "amanda\tjeannette\tgerardo" "3\t1\t2"
```

7 The `save()` function allows you to save R objects in .RData files, as demonstrated here for the object `mydf`. You can use `load()` to load an .RData file into your current R session. Knowing about this comes in handy when you have to deal with computations that take a long time.

```
save(mydf, file = 'mydataframe.RData')
```

8 Warning messages differ from error messages. A warning message happens when a command was executed but the function wants to warn you of something. An error message means that a command was aborted.

The `n` argument specifies the number of lines to be read. You often do not need more than two lines in order to understand the structure of an object. In this case, you can see that the first line contains column names, which tells you that `header = TRUE` is necessary. In addition, the second row contains tab delimiters `'\t'`, which tells you that `sep = '\t'` is necessary. You can use this information to provide the right arguments to the `read.table()` function.

If you want to load in Excel files (.xls, .xlsx), there are various packages available. Other R packages exist for loading in SPSS or STATA files. However, all of these file types are proprietary, including .xls and .xlsx. This means that some company owns these file types. Whenever possible, try to avoid proprietary file types and use simple comma- or tab-separated files to manage your projects.

A note on how to use data in this book. For each chapter, new data will be loaded. It is perfectly fine to leave R open and work within one session for the entirety of this book. Alternatively, you can close R after working on a specific chapter (don't forget to *not* save your workspace). While it is OK for this book to work within one continued R session, you will always want to start a new R session in an actual data analysis, which helps to keep things neat.

1.12. Plotting

Let's begin by creating one of the most useful plots in all of statistics, the histogram. Figure 1.1 shows a histogram of the number of languages per country. The height of each rectangle (called 'bin') indicates the number of data points contained within the range covered by the rectangle, what is called the 'bin width'. In this case, there are

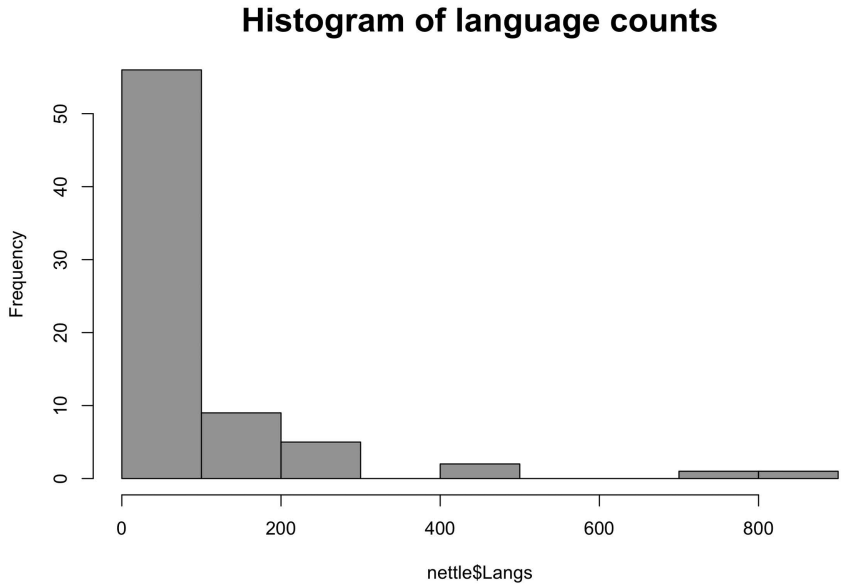

Figure 1.1. Histogram of the number of languages per country; data taken from Nettle (1999)

more than 50 countries that have between 0 and 100 languages. There are about 10 countries that have between 100 and 200 languages, and so on. There seems to be only a handful of countries with more than 700 languages.

The following command creates this plot, using the `Langs` column of the Nettle (1999) dataset.[9]

```
hist(nettle$Langs)
```

Let's rerun the histogram command and make the bins have a certain color, say, salmon.

```
hist(nettle$Langs, col = 'salmon')
```

The `col` argument is an optional argument of the `hist()` function. See what happens if you change the color to, say, `'steelblue'`. If you wanted to have a look at what colors are pre-specified in R, type in `colors()` to the console. This is another function that doesn't require any arguments, similar to `ls()`, `list.files()`, or `getwd()`. The code below shows only the first six colors.

```
head(colors())
```

```
[1] "white"          "aliceblue"      "antiquewhite"
[4] "antiquewhite1" "antiquewhite2" "antiquewhite3"
```

These colors are the *named* colors of R. You can also specify hexadecimal color codes. Check what happens if you supply `'#DD4433'` to the `col` argument.

1.13. Installing, Loading, and Citing Packages

R is a community project with a massive amount of content made available by its active user base. New functions are assembled into libraries called 'packages', which can be installed using the `install.packages()` function. The following code installs the `car` package (Fox & Weisberg, 2011).

```
install.packages('car')
```

Once a package is installed,[10] you can load it like this:

```
library(car)
```

9 These language counts are a considerable abstraction, as it's not always clear whether a linguistic variety is best considered as a dialect of a language, or whether it should be counted as its own language.

10 There are many reasons why the installation of a package may fail. Installation problems can usually be fixed with the help of Google searches. A lot of the time, installation problems result from having an outdated R version. Alternatively, there may have been a new R version released recently, and the package developers are lagging behind in updating their package to the new version.

This makes the functions from the `car` package available in the current R session. If you close R and reopen it, you will have to reload the package. That is, packages are only loaded for a single session.

For reproducibility and to acknowledge the valuable work of the respective developers, it's important to cite each package and report its package version, which is demonstrated here for `car`:

```
citation('car')$textVersion
```

```
[1] "John Fox and Sanford Weisberg (2011). An {R} Companion to Applied Regression, Second Edition. Thousand Oaks CA: Sage. URL: http://socserv.socsci.mcmaster.ca/jfox/Books/Companion"
```

```
packageVersion('car')
```

```
[1] '3.0.0'
```

Speaking of citing, this is how you can retrieve the citation and version information for R itself. These should also be reported. The commands below provide abbreviated output thanks to the indexing statements `'$textVersion'` and `'$version.string'`. You can also simply run the `citation()` and `R.Version()` functions without these statements, which returns more extensive outputs.

```
citation()$textVersion
```

```
[1] "R Core Team (2018). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/."
```

```
R.Version()$version.string
```

```
[1] "R version 3.5.0 (2018-04-23)"
```

1.14. Seeking Help

There's a help file for any R function. To access it, just put a question mark in front of a function's name, as demonstrated here for the `seq()` function, which is for generating number sequences.

```
?seq
```

I haven't introduced you to this function yet, but perhaps you can figure out how it works from the examples listed at the bottom of the help file (you will use this function in later chapters).

If you forgot a function name and wanted to search for it from within R, you can use the handy `apropos()` function. For example, running `apropos('test')`

will display all functions that have the string `'test'` in their name. In the following, I only show you the first six of these:

```
head(apropos('test'))
```

```
[1] ".valueClassTest" "ansari.test"
[3] "bartlett.test"   "binom.test"
[5] "Box.test"        "chisq.test"
```

Often, copy-and-pasting warning or error messages from the console into Google will immediately direct you to a solution to any problems you may encounter. If that doesn't help, you can ask others for help, such as via stackoverflow.com. However, asking good questions is not easy and it's essential that you perform extensive Google searches before asking others.

Perhaps most importantly, when you encounter an error or warning message, you should never think that you are stupid. R is quite quirky and learning how to program isn't easy. If you encounter a problem, rest assured that there are many others who ran into the same problem. You should be aware of the fact that even very advanced R users constantly encounter error and warning messages. For example, the very experienced R programmers Wickham and Grolemund (2007: 7) write: "I have been writing R code for years, and every day I still write code that doesn't work!" This should show you that there's no reason to feel stupid when you run into a problem.

1.15. A Note on Keyboard Shortcuts

You are heavily advised to spend some time learning R/RStudio keyboard shortcuts. When wanting to be efficient with data, the mouse is your enemy! Your future self is going to thank you (yet again!) for the countless hours saved thanks to keyboard shortcuts. Here are some very handy shortcuts that I use frequently:

Shortcut	Action
Ctrl/Command + N	open new script
Ctrl/Command + Enter	run current line (send from script to console)
Alt/Option + Minus '-'	insert '<–'
Ctrl/Command + Alt/Option + I	insert code chunk (R markdown, see Chapter 2)
Ctrl/Command + Shift + M	insert pipe (tidyverse, see Chapter 2)

These shortcuts are specific to R/RStudio. On top of that, I hope that you already are accustomed to using general text editing shortcuts. If not, here are some very useful shortcuts:

Shortcut	Action
Shift + Left/Right	highlight subsection of text
Alt/Option + Left/Right (Mac)	move cursor by a word

Ctrl + Left/Right (Windows)	move cursor by a word
Command + Left/Right (Mac)	move cursor to beginning/end of line
Home/End (Windows & Linux)	move cursor to beginning/end of line
Ctrl + K	delete line from position of cursor onwards
Ctrl/Command + C	copy
Ctrl/Command + X	cut
Ctrl/Command + V	paste
Ctrl/Command + Z	undo
Ctrl/Command + A	select all

Incorporating these and other shortcuts into your workflow will save you valuable time and mental energy. In the long run, knowing lots of shortcuts will help you experience 'flow' when coding.

1.16. Your R Journey: The Road Ahead

I have just taught you the very basics of R. Think about learning R as learning a foreign language. Obviously, you cannot learn a whole language in a single chapter! And, of course, you're bound to forget things, which is OK! It's important that you persevere through your mistakes. When you get stuck, take a break and resume later. Let's talk about the most common problems you will encounter:

- You wrote something in your script, but you have forgotten to execute the command in R (that is, it hasn't been sent to the console yet).
- If you get an error message which says "object not found", you likely have mistyped the name of the object, or you forgot to execute some assignment command.
- If you get a "function not found" message, you either mistyped the function name, or the relevant package hasn't been loaded yet.
- Warning messages also frequently arise from not knowing what type of object you are working with: What's the object's dimensionality (rows, columns)? What type of vector are you dealing with (e.g., character, factor, numeric, logical)?
- Sometimes you may run into syntax issues, such as having forgotten a comma between arguments inside a function, or having forgotten a closing bracket.
- Many or most errors result from some sort of typo. Extensive use of the copy-and-paste shortcuts prevents many typos.

As a general rule of thumb, you should never believe R does as intended (Burns, 2011). It is good to develop a habit of constantly checking the objects in your working environment. You should ask yourself questions such as 'Did the function I just executed actually produce the intended result?' or 'What is actually contained in the object that I'm currently working with?'

From now on, it's just learning-by-doing and trial-and-error. I will teach you some more R in the next chapter. After that, the focus will be on the stats side of things. As you go along, many commands will be repeated again and again. The same way it helps you to repeat words when learning the vocab of a foreign language, R 'words' need to be used extensively before they sink in. And, also like learning a foreign language, there's no way around continuous practice. I'll help you along the way.

1.17. Exercises

1.17.1. Exercise 1: Familiarizing Yourself with Base Plotting

Type the following commands into a script and then execute them together:

```
plot(x = 1, y = 1, type = 'n',
     xlim = c(-2, 2), ylim = c(-2, 2))
points(x = -1, y = 1)
segments(x0 = -0.5, y0 = -1, x1 = 0.5, y1 = -1)
```

The first line opens up an empty plot with a point at the coordinates `x = 1` and `y = 1`. The `type = 'n'` argument means that this point is not actually displayed. `xlim` and `ylim` specify the plot margins.

The `points()` function plots a single point at the specified coordinates. The `segments()` function plots a line segment. The beginning of the line segment is given by the arguments `x0` and `y0`. The end points are given by `x1` and `y1`.

What is displayed in your plotting window is actually a one-eyed smiley. By adding additional `points()` and `segments()`, can you create a more elaborate smiley? This exercise will help to wrap your head around working with coordinate systems.

1.17.2. Exercise 2: Swirl

The interactive `swirl` package teaches you R inside R.

```
install.packages('swirl')
library(swirl)
swirl()
```

Complete the first four modules of the R programming course. If you have extra time, complete the first five courses of the Exploratory Data Analysis course. Throughout your R journey, you can come back to `swirl` at any time and complete more courses.

1.17.3. Exercise 3: Spot-the-Error #1

Type the following two lines of code into your R script, exactly as they are printed here on the page. Then execute them. This will result in two error messages.

```
x_values <- c(1, 2 3, 4, 5, 6, 7, 8, 9)
mean_x <- mean(X_values)
```

Each line contains one error. Can you find them and correct them?[11]

11 I thank Márton Sóskuthy for this exercise idea.

1.17.4. Exercise 4: Spot-the-Error #2

Why does the following command return an `NA` value?

```
x <- c(2, 3, 4, '4')
mean(x)
```

Can you use the function `as.numeric()` to solve this problem?

1.17.5. Exercise 5: Spot-the-Error: #3

The following line of code tries to extract the row from the `nettle` data frame that contains information on the country Yemen. Why does this return an error and can you fix this?

```
nettle[nettle$Country = 'Yemen',  ]
```

1.17.6. Exercise 6: Indexing Data Frames

Gillespie and Lovelace (2017: 4) say that "R is notorious for allowing users to solve problems in many ways". In this section, you will learn a bunch of different ways of extracting information from the same data frame. Some of these ways are redundant, but knowing multiple paths to the same goal gives you flexibility in how to approach data analysis problems. This exercise also teaches you how indexing statements can be used recursively, stacked on top of each other.

```
head(nettle)  # display first 6 rows
```

```
      Country Population Area  MGS Langs
1     Algeria       4.41 6.38 6.60    18
2      Angola       4.01 6.10 6.22    42
3   Australia       4.24 6.89 6.00   234
4  Bangladesh       5.07 5.16 7.40    37
5       Benin       3.69 5.05 7.14    52
6     Bolivia       3.88 6.04 6.92    38
```

Next, the following statements extract information from this data frame. You haven't been taught all of these ways of indexing yet. However, try to understand what the corresponding code achieves and I'm sure you'll be able to figure it out.

Importantly, think about what is being extracted *first*, only *then* type in the command to see whether the output matches your expectations.

```
nettle[2, 5]
```

```
nettle[1:4, ]

nettle[1:4, 1:2]

nettle[nettle$Country == 'Bangladesh', ]

nettle[nettle$Country == 'Bangladesh', 5]

nettle[nettle$Country == 'Bangladesh', ] [, 5]

nettle[nettle$Country == 'Bangladesh', ] $Langs

nettle[nettle$Country == 'Bangladesh', 'Langs']

nettle[1:4, ] $Langs[2]

nettle[1:4, c('Country', 'Langs')]

head(nettle[,])
```