

*Ulrich Matter*

---

# ***Data Handling Pocket Reference***

*To the students who have motivated and inspired this work.*

---

## *Contents*

---



---

## ***List of Figures***

---



---

## ***List of Tables***

---





---

## ***Preface***

---

In applied econometric research and business analytics alike, many steps in handling digital data are involved before running regression estimations and training machine learning algorithms. While often neglected in statistics and econometrics curricula, the steps of gathering, cleaning, storing, and filtering data for research/analytics purposes are of utmost importance to ensure accurate and reproducible analytic insights. This pocket reference first introduces and summarizes foundational and practically relevant data and data processing concepts, and then guides the reader through each step of how to get from the raw data to the final data analysis output.

---

### **Prerequisites and aims**

The book presupposes a basic knowledge of undergraduate economics and statistics and relates several case studies to practical questions in these fields. Finally, the aim is to give you firsthand practical insights into each part of the data science pipeline in the context of business and economics research.



**FIGURE 1:** Creative Commons License

The online version of this book is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License<sup>1</sup>.

---

<sup>1</sup><http://creativecommons.org/licenses/by-nc-sa/4.0/>

Ulrich Matter St. Gallen, Switzerland

# 1

---

## *Introduction*

---

Lower computing costs, a stark decrease in storage costs for digital data, as well as the diffusion of the Internet, have led to the development of new products (e.g., smartphones) and services (e.g., web search engines, cloud computing) over the last few decades. A side product of these developments is a strong increase in the availability of digital data describing all kinds of everyday human activities (??). As a consequence, new business models and economic structures are emerging with data as their core commodity (i.e., AI-related technological and economic change). For example, the current hype surrounding ‘Artificial Intelligence’ (AI) - largely fueled by the broad application of machine-learning techniques such as ‘deep learning’ (a form of artificial neural networks) - would not be conceivable without the increasing abundance of large amounts of digital data on all kind of socio-economic entities and activities. In short, without understanding and handling the underlying data streams properly, the AI-driven economy cannot function. The same rationale applies, of course, to other ways of making use of digital data. Be it traditional big data analytics or scientific research (e.g., applied econometrics).

The need for proper handling of large amounts of digital data has given rise to the interdisciplinary field of ‘Data Science’<sup>1</sup> and increasing demand for ‘Data Scientists’. While nothing within Data Science is particularly new on its own, the combination of skills and insights from different fields (particularly Computer Science and Statistics) has proven to be very productive in meeting new challenges posed by a data-driven economy. In that sense, Data Science is rather a craft than a scientific field. As such, it presupposes a more practical and broader understanding of the data than traditional Computer Science

---

<sup>1</sup>[https://en.wikipedia.org/wiki/Data\\_science](https://en.wikipedia.org/wiki/Data_science)

and Statistics from which Data Science borrows its methods. This book focuses on the skills necessary for *acquiring, cleaning, and manipulating* digital data for research/analytics purposes.

## 2

---

### *Programming with Data*

---

#### **2.1 Handling data programmatically**

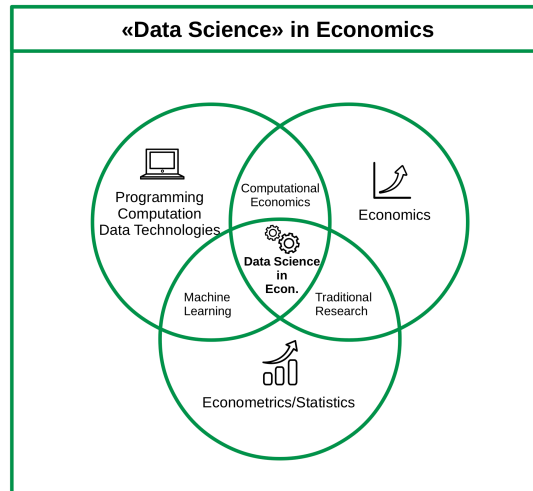
The need for proper handling of large amounts of digital data has given rise to the interdisciplinary field of ‘Data Science’<sup>1</sup> as well as an increasing demand for ‘Data Scientists’. While nothing within Data Science is particularly new on its own, it is the combination of skills and insights from different fields (particularly Computer Science and Statistics) that has proven to be very productive in meeting new challenges posed by a data-driven economy. The various facets of this new craft are often illustrated in the ‘Data Science’ Venn diagram, reflecting the combination of knowledge and skills from Mathematics/Statistics, substantive expertise in the particular scientific field in which Data Science is applied (here: Economics), and the skills for *acquiring, cleaning, and analyzing data programmatically*.

In the proposed framework of Data Science in Economics followed in this book, programming skills, basic knowledge about computing and in data technologies serve as a complement to engaging in modern economic analysis. They are necessary ingredients to both working new econometric approaches in machine learning (and the preceding feature engineering) as well as to solving complex problems in the domain of economic modeling. Moreover, programming (or ‘coding’) is the basis to better understand and engage with the handling of data for analytics purposes.

While industry-scale data science projects tend to include data processing frameworks and programming languages (such as Scala, Python, and SQL), we will see that the core steps of how to get from the

---

<sup>1</sup>[https://en.wikipedia.org/wiki/Data\\_science](https://en.wikipedia.org/wiki/Data_science)



**FIGURE 2.1:** Venn diagram illustrating the domains of Data Science in the context of Economics.

raw data to the final analysis report can for most simpler data projects be easily managed with just one programming language. In the case of this book, we choose R.

## 2.2 Why R?

### 2.2.1 The ‘data language’

The programming language and open-source statistical computing environment  $R^2$  has over the last decade become a core tool for data science in industry and academia. It was originally designed as a tool for statistical analysis. Many characteristics of the language make R particularly useful to work with data. R is increasingly used in various domains, going well beyond the traditional applications of academic research.

<sup>2</sup>[www.r-project.org](http://www.r-project.org)

### 2.2.2 High-level language, relatively easy to learn

R is a relatively easy computer language to learn for people with no previous programming experience. The syntax is rather intuitive and error messages are not too cryptic to understand (this facilitates learning by doing). Moreover, with R's recent stark rise in popularity, there are plenty of freely accessible resources online that help beginners to learn the language.

### 2.2.3 Free, open source, large community

Due to its vast base of contributors, R serves as a valuable tool for users in various fields related to data analysis and computation (economics/econometrics, biomedicine, business analytics, etc.). R users have direct access to thousands of freely available 'R-packages' (small software libraries written in R), covering diverse aspects of data analysis, statistics, data preparation, and data import.

Hence, a lot of people using R as a tool in their daily work do not actually 'write programs' (in the traditional sense of the word), but apply R packages. Applied econometrics with R is a good example of this. Almost any function a modern commercial computing environment with a focus on statistics and econometrics (such as STATA<sup>3</sup>) is offering, can also be found within the R environment. Furthermore, there are R packages covering all the areas of modern data analytics, including natural language processing, machine learning, big data analytics, etc. (see the CRAN Task Views<sup>4</sup> for an overview). We thus do not actually have to write a program for many tasks we perform with R. Instead, we can build on already existing and reliable packages.

---

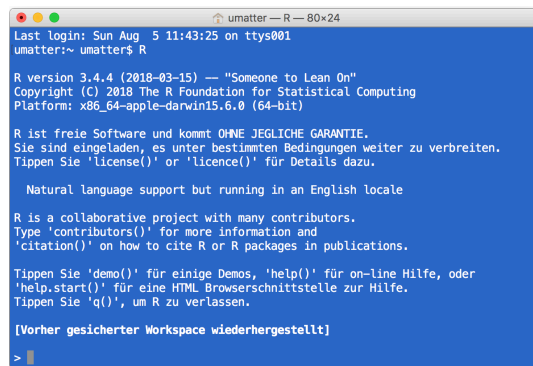
<sup>3</sup><http://www.stata.com/>

<sup>4</sup><https://cran.r-project.org/web/views/>

### 2.3 R/RStudio overview

R is the high-level (meaning ‘more user friendly’) programming language for statistical computing. Once we have installed R on our computer, we can run it...

- a. ...directly from the command line, by typing `R` and hit enter (here in the OSX terminal):



```
umatter - R - 80x24
Last login: Sun Aug 5 11:43:25 on ttys001
umatter:~ umatter$ R

R version 3.4.4 (2018-03-15) -- "Someone to Lean On"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin15.6.0 (64-bit)

R ist freie Software und kommt OHNE JEGLICHE GARANTIE.
Sie sind eingeladen, es unter bestimmten Bedingungen weiter zu verbreiten.
Tippen Sie 'license()' or 'licence()' für Details dazu.

Natural language support but running in an English locale

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.

Tippen Sie 'demo()' für einige Demos, 'help()' für on-line Hilfe, oder
'help.start()' für eine HTML Browserschnittstelle zur Hilfe.
Tippen Sie 'q()', um R zu verlassen.

[Vorher gesicherter Workspace wiederhergestellt]

>
```

**FIGURE 2.2:** Running R in the Mac/OSX terminal.

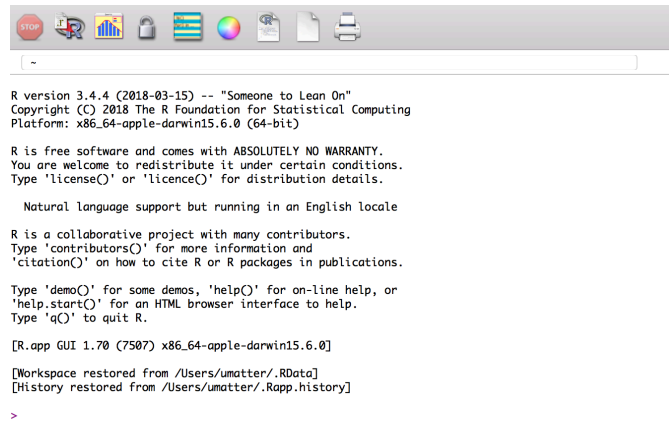
- b. ...with the simple Integrated Development Environment (IDE)<sup>5</sup> delivered with the basic R installation
- c. ...or with the more elaborated and user-friendly IDE called *RStudio* (either locally or in the cloud, see, for example *RStudio Cloud*<sup>6</sup>:

The latter is what we will do throughout this course. RStudio is a very helpful tool for simple data analysis with R, writing R scripts (short R programs), or even for developing R packages (software written in R),

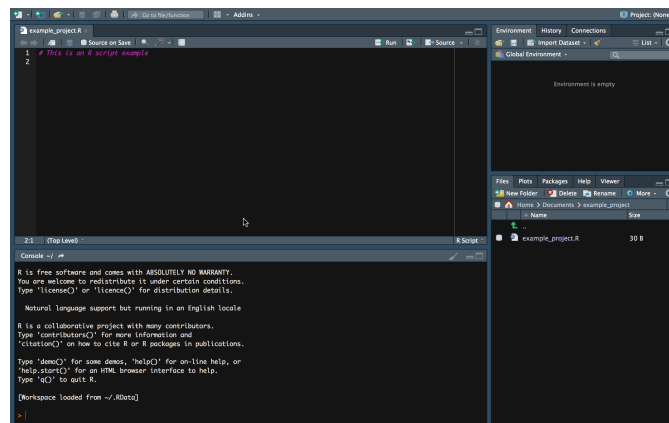
<sup>5</sup>[https://en.wikipedia.org/wiki/Integrated\\_development\\_environment](https://en.wikipedia.org/wiki/Integrated_development_environment)

<sup>6</sup><https://rstudio.cloud/>





**FIGURE 2.3:** Running R in the original R GUI/IDE.



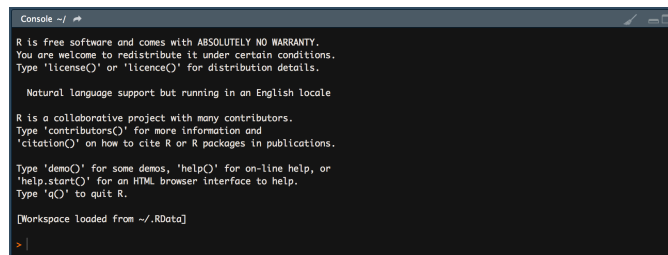
**FIGURE 2.4:** Running R in RStudio (IDE).

as well as building interactive documents, presentations, etc. Moreover, it offers many options to change its own appearance (Pane Layout, Code Highlighting, etc.).

In the following, we have a look at each of the main panels that will be relevant in this course.

### 2.3.1 The R-Console

When working in an interactive session, we simply type *R* commands directly into the *R* console. Typically, the output of executing a command this way is also directly printed to the console. Hence, we type a command on one line, hit enter, and the output is presented on the next line.



**FIGURE 2.5:** Running *R* in the Mac/OSX terminal.

For example, we can tell *R* to print the phrase `Hello world` to the console, by typing to following command in the console and hit enter:

```
print("Hello world")
```

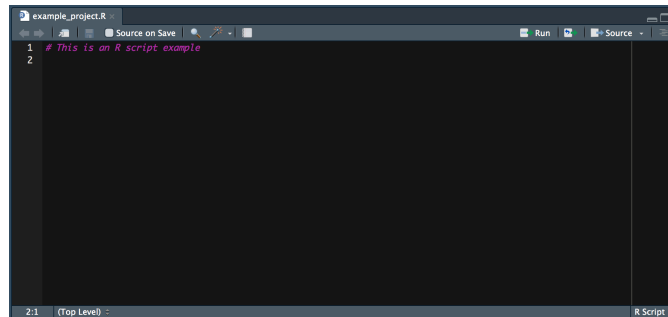
```
## [1] "Hello world"
```

### 2.3.2 R-Scripts

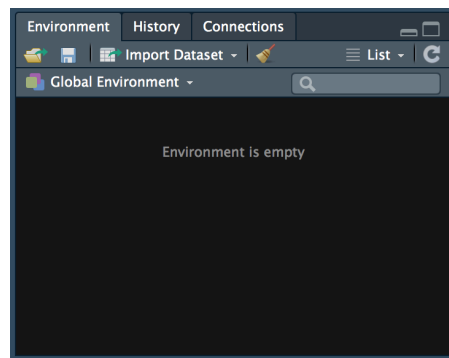
Apart from very short interactive sessions, it usually makes sense to write *R* code not directly in the command line but to an *R*-script in the script panel. This way, we can easily execute several lines at once, comment the code (to explain what it does), save it on our hard disk, and further develop the code later on.

### 2.3.3 R Environment

The environment pane shows what variables, objects, and data are loaded in our current *R* session. Moreover, it offers functions to open documents and import data.



**FIGURE 2.6:** The R Script window in RStudio.



**FIGURE 2.7:** The environment window in RStudio.

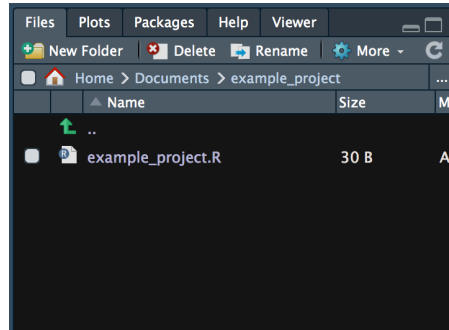
### 2.3.4 File Browser

With the file browser window we can navigate through the folder structure and files on our computer's hard disk, modify files, and set the working directory of our current R session. Moreover, it has a pane to show plots generated in R and a pane with help pages and R documentation.

---

## 2.4 First steps with R

Before introducing some of the key functions and packages for data handling and data analysis with R, we should understand how such programs basically work and how we can write them in R. Once we



**FIGURE 2.8:** The file browser window in RStudio.

understand the basics of the R language and how to write simple programs, understanding and applying already implemented programs is much easier.<sup>7</sup>

#### 2.4.1 Values, Vectors, and Variables

The simplest objects to work with in R are vectors. In fact, even a simple numeric value such as 5.5 or a string of characters (text) like "Hello" is considered a vector (a scalar).<sup>8</sup>

A first good step to get familiar with coding in R is to assign names to the objects/values you are working with. For example, when summing up two numeric values, you might want to store the result in a separate object and call this object `result`. This is done with the assignment operator `<-` (or `=`, which serves the same purpose).

```
# assign the variable name "result" to the sum of two numeric values
result <- 25.6 + 53.4
```

Whenever you want to re-use the just computed sum, you can directly call the object by its name (the variable `result`):

<sup>7</sup>In fact, since R is an open source environment, you can directly look at already implemented programs in order to learn how they work.

<sup>8</sup>You can try this out in the R console by typing `is.vector(5.5)` and `is.vector("Hello")`.

```
# check what "is stored in" result  
result
```

```
## [1] 79
```

```
# further work with the value in result  
result - 20
```

```
## [1] 59
```

With the combine-function (`c()`), you easily form vectors with several elements and name the elements in the vector. By doing so, you create a very simple dataset. For example, suppose you survey the age of a sample of persons. The age values (in years) gives you an integer vector. By naming each integer value (each vector element) after the corresponding person's name, you then have a simple dataset stored in a named R vector.

```
# a simple integer vector  
a <- c(10,22,33, 22, 40)  
  
# give names to vector elements  
names(a) <- c("Andy", "Betty", "Claire", "Daniel", "Eva")  
a
```

```
##   Andy  Betty Claire Daniel   Eva  
##    10    22    33    22    40
```

To retrieve specific values from this vector, you can either select the corresponding vector element with the element's index (the first, second, third, etc. element) or via its name.

```
# indexing either via a number of vector element (start count with 1)  
# or by element name  
a[3]  
  
## Claire
```

```
##      33
```

```
a["Claire"]
```

```
## Claire
```

```
##      33
```

When not sure what kind of object `a` is, the `str()` (structure) function, provides you with a short summary.

```
# inspect the object you are working with  
str(a) # returns the structure of the object ("what is in variable a?")
```

```
## Named num [1:5] 10 22 33 22 40
```

```
## - attr(*, "names")= chr [1:5] "Andy" "Betty" "Claire" "Daniel" ...
```

If you want to learn more about what the `c()` or `str()` functions (or any other pre-defined R functions) do and how they should be used, type `help(FUNCTION-NAME)` or `?FUNCTION-NAME` in the console and hit enter. A help-page with detailed explanations of what the function is for and how it can be used will appear in one of the R-Studio panels.

```
help(str)
```

```
?c
```

### 2.4.2 Math operators

Above, we have just in a side remark introduced the very intuitive syntax for two common math operators in R: `+` for the addition of numeric or integer values, and `-` for subtraction. R knows all basic math operators and has a variety of functions to handle more advanced mathematical problems. One basic practical application of R in academic life is to use it as a sophisticated (and programmable) calculator.

```
# basic arithmetic
```

```
2+2
```

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

```
sum_result <- 2+2  
sum_result
```

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

```
sum_result -2
```

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

```
4*5
```

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

```
20/5
```

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

```
# order of operations  
2+2*3
```

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

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

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

```
(5+5)/(2+3)
```

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

```
# work with variables  
a <- 20
```

```
b <- 10  
a/b
```

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

```
# arithmetics with vectors  
a <- c(1,4,6)  
a * 2
```

```
## [1] 2 8 12
```

```
b <- c(10,40,80)  
a * b
```

```
## [1] 10 160 480
```

```
a + b
```

```
## [1] 11 44 86
```

```
# other common math operators and functions  
4^2
```

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

```
sqrt(4^2)
```

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

```
log(2)
```

```
## [1] 0.6931
```



```
exp(10)
```

```
## [1] 22026
```

```
log(exp(10))
```

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

To look up the most common math operators in R and get more details about how to use them type

```
?`+`
```

in the R console and hit enter.

---

## 2.5 Basic programming concepts in R

In very simple terms, programming/coding is all about using a computer language to instruct a computer what to do. What reads very complex at first sight, is actually rather simple in (at least for a large array of basic programming problems). At the core of almost any R program is the right application and combination of just a handful of basic programming concepts: loops, logical statements, control statements, and functions. Once you a) conceptually understand what these concepts are for, and b) have learned the syntax of how to use these concepts when writing a program in R, addressing all kind of data handling problems efficiently with R, will simply become a matter of training/practice.

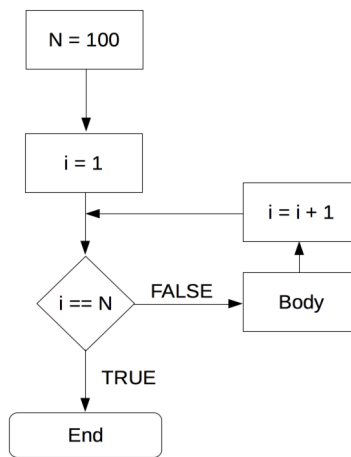
### 2.5.1 Loops

A loop is typically a sequence of statements executed a specific number of times. How often the code ‘inside’ the loop is executed depends on a clearly defined control statement. If we know in advance how often the code inside the loop has to be executed, we typically write a so-called

‘for-loop’. We typically write a so-called ‘while-loop’ if the number of iterations is not clearly known before executing the code. The following subsections illustrate both of these concepts in R.

### 2.5.2 For-loops

In simple terms, a for-loop tells the computer to execute a sequence of commands ‘for each case in a set of  $n$  cases’. The flowchart in Figure @ref(fig: for) illustrates the concept.



**FIGURE 2.9:** For-loop illustration.

For example, a for-loop could be used, to sum up each element in a numeric vector of fixed length (thus, the number of iterations is clearly defined). In plain English, the for-loop would state something like: “Start with 0 as the current total value, for each of the elements in the vector, add the value of this element to the current total value.” Note how this logically implies that the loop will ‘stop’ once the value of the last element in the vector is added to the total. Let’s illustrate this in R. Take the numeric vector `c(1,2,3,4,5)`. A for loop to sum up all elements can be implemented as follows:

```
# vector to be summed up
numbers <- c(1,2.1,3.5,4.8,5)
```

```
# initiate total
total_sum <- 0
# number of iterations
n <- length(numbers)
# start loop
for (i in 1:n) {
  total_sum <- total_sum + numbers[i]
}

# check result
total_sum
```

```
## [1] 16.4
```

```
# compare with the result of sum() function
sum(numbers)
```

```
## [1] 16.4
```

### 2.5.2.1 Nested for-loops

In some situations, a simple for-loop might not be sufficient. Within one sequence of commands, there might be another sequence of commands that also has to be executed for a number of times each time the first sequence of commands is executed. In such a case, we speak of a ‘nested for-loop’. We can illustrate this easily by extending the example of the numeric vector above to a matrix for which we want to sum up the values in each column. Building on the loop implemented above, we would say ‘for each column *j* of a given numeric matrix, execute the for-loop defined above’.

```
# matrix to be summed up
numbers_matrix <- matrix(1:20, ncol = 4)
numbers_matrix
```

```
##      [,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
```

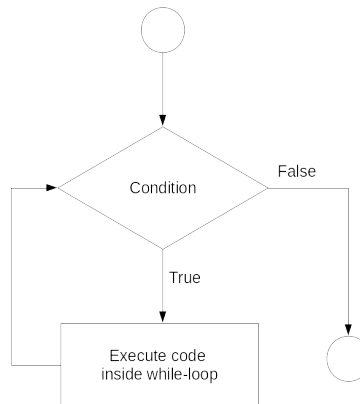
```
# number of iterations for the outer loop
m <- ncol(numbers_matrix)
# number of iterations for the inner loop
n <- nrow(numbers_matrix)
# start outer loop (loop over columns of the matrix)
for (j in 1:m) {
  # start inner loop
  # initiate total
  total_sum <- 0
  for (i in 1:n) {
    total_sum <- total_sum + numbers_matrix[i, j]
  }
  print(total_sum)
}
```

```
## [1] 15
## [1] 40
## [1] 65
## [1] 90
```

#### 2.5.2.2 While-loop

In a situation where a program has to repeatedly run a sequence of commands, but we don't know in advance how many iterations we need to reach the intended goal, a while-loop can help. In simple terms, a while loop keeps executing a sequence of commands as long as a certain logical statement is true. The flow chart in Figure @ref(fig:while) illustrates this point.

For example, a while-loop in plain English could state something like “start with 0 as the total, add 1.12 to the total until the total is larger than 20.” We can implement this in R as follows.

**FIGURE 2.10:** While-loop illustration.

```

# initiate starting value
total <- 0
# start loop
while (total <= 20) {
  total <- total + 1.12
}

# check the result
total

```

```
## [1] 20.16
```

### 2.5.3 Booleans and logical statements

Note that in order to write a meaningful while-loop we have to make use of a logical statement such as “the value stored in the variable `total` is smaller or equal to 20” (`total <= 20`). A logical statement results in a ‘Boolean’ data type. That is, a data type with the only two possible values `TRUE` or `FALSE` (1 or 0).

```
2+2 == 4
```

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

```
3+3 == 7
```

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

Logical statements play an important role in fundamental programming concepts. In particular, they are crucial to make conditional statements ('if-statements') that build the control structure of a program, controlling the 'direction' the program takes (given certain conditions).

```
condition <- TRUE
if (condition) {
  print("This is true!")
} else {
  print("This is false!")
}
```

```
## [1] "This is true!"
```

```
condition <- FALSE
if (condition) {
  print("This is true!")
} else {
  print("This is false!")
}
```

```
## [1] "This is false!"
```

#### 2.5.4 R functions

R programs heavily rely on functions. Conceptually, 'functions' in R are very similar to what we know as 'functions' in math (i.e.,  $f : X \rightarrow Y$ ). A function can thus, e.g., take a variable  $X$  as input and provide value  $Y$  as output. The actual calculation of  $Y$  based on  $X$  can be something as simple as  $2 \times X = Y$ . But it could also be a very complex algo-

rithm or an operation that does not directly have anything to do with numbers and arithmetic.<sup>9</sup>

In R—and many other programming languages—functions take ‘parameter values’ as input, process those values according to a predefined program, and ‘return’ the result. For example, a function could take a numeric vector as input and return the sum of all the individual numeric values in the input vector.

When we open RStudio, all basic functions are already loaded automatically. This means we can directly call them from the R-Console or by executing an R-Script. As R is made for data analysis and statistics, the basic functions loaded with R cover many aspects of tasks related to working with and analyzing data. Besides these basic functions, thousands of additional functions covering all kinds of topics related to data analysis can be loaded additionally by installing the respective R-packages (`install.packages("PACKAGE-NAME")`) and then loading the packages with `library(PACKAGE-NAME)`. In addition, it is straightforward to define our own functions.

#### 2.5.4.1 Case study: Compute the mean

To illustrate the point of how functions work in R and how we can write our own functions in R, the following code-example illustrates how to implement a function that computes the mean/average value, given a numeric vector.

First, we initiate a simple numeric vector which we then use as an example to test the function. Whenever you implement a function, it is very useful to first define a simple example of an input for which you know what the output should be.

---

<sup>9</sup>Of course, on the very low level, everything that happens in a microprocessor can, in the end, be expressed in some formal way using math. However, the point here is that at the level we work with R, a function could simply process different text strings (i.e., stack them together). Thus for us as programmers, R functions do not necessarily have to do anything with arithmetic and numbers but could serve all kinds of purposes, including the parsing of HTML code, etc.

```
# a simple integer vector, for which we want to compute the Mean
a <- c(5.5, 7.5)
# desired functionality and output:
# my_mean(a)
# 6.5
```

In this example, we would thus expect the output to be 6.5. Later, we will compare the output of our function with this in order to check whether our function works as desired.

In addition to defining a simple example and the desired output, it makes sense to also think about *how* the function is expected to produce this output. When implementing functions related to statistics (such as the mean), it usually makes sense to have a look at the mathematical definition:

$$\bar{x} = \frac{1}{n} \left( \sum_{i=1}^n x_i \right) = \frac{x_1 + x_2 + \dots + x_n}{n}.$$

Now, we can start thinking about implementing the function based on built-in R functions. From looking at the mathematical definition of the mean ( $\bar{x}$ ), we recognize that there are two main components to computing the mean:

- $\sum_{i=1}^n x_i$ : the sum of all the elements in vector  $x$
- $n$ : the number of elements in vector  $x$ .

Once we know how to get these two components, computing the mean is straightforward. In R, there are two built-in functions that deliver exactly these two components:

- `sum()` returns the sum of all the values in its arguments (i.e., if  $x$  is a numeric vector, `sum(x)` returns the sum of all elements in  $x$ ).
- `length()` returns the length of a given vector.

With the following short line of code, we thus get the mean of the elements in vector `a`.

```
sum(a)/length(a)
```

```
## [1] 6.5
```



All that is left to do is to pack all this into the function body of our newly defined `my_mean()` function:

```
# define our own function to compute the mean, given a numeric vector
my_mean <- function(x) {
  x_bar <- sum(x) / length(x)
  return(x_bar)
}
```

Now we can test it based on our example:

```
# test it
my_mean(a)
```

```
## [1] 6.5
```

Moreover, we can test it by comparing it with the built-in `mean()` function:

```
b <- c(4,5,2,5,5,7)
my_mean(b) # our own implementation
```

```
## [1] 4.667
```

```
mean(b) # the built_in function
```

```
## [1] 4.667
```



### 3

---

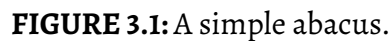
#### *Data*

---

To better understand the role of data in today's economy and society, we study the usage forms and purposes of data records in human history. In the second step, we look at how a computer processes digital data.

Throughout human history, the recording and storage of data have primarily been motivated by measuring, quantifying, and keeping records of both our social and natural environments. Early on, data recording was related to economic activity and scientific endeavors. The neolithic transition from hunter-gatherer societies to agriculture and settlements (the economic development sometimes referred to as the 'first industrial revolution') came along with a division of labor and more complex organizational structures of society. Because of the change in agricultural food production, more people could be fed. But it also implied that food production would need to follow a more careful planning (e.g., the right time to seed and harvest) and that the produced food (e.g., grains) would partly be stored and not consumed entirely on the spot. It is believed that partly due to these two practical problems, keeping track of time and keeping record of production quantities, neolithic societies started to use signs (numbers/letters) carved in stone or wood (?). Keeping record of the time and later measuring and keeping record of production quantities in order to store and trade likely led to the first 'data sets'. At the same time the development of mathematics, particularly geometry, took shape.

In order to keep track of calculations with large numbers, humans started to use mechanical aids such as pebbles or shells and later developed the counting frame ('abacus')<sup>1, 2</sup>. We can understand the counting frame as a simple mechanical tool to process numbers (data). In order to use the abacus properly, one must agree on a standard regarding what each column of the frame represents, as well as how many beads each column can contain. In other words, one has to define what the *base* of the frame's numeral system is. For example, the Roman numeral system is essentially of base 10, with 'I' representing one, 'X' representing ten, 'C' representing one hundred, and 'M' representing one thousand. Figure ?? illustrates how a counting frame based on this numeral system works (examples are written out in Arabic numbers). The first column on the right represents units of  $10^0 = 1$ , the second  $10^1 = 10$ , and so forth.



<sup>1</sup><https://en.wikipedia.org/wiki/Abacus>

<sup>2</sup>See ?, Chapter 1 for a detailed account of the abacus' origin and usage.

which the digit is multiplied:  $139 = (1 \times 10^2) + (3 \times 10^1) + (9 \times 10^0)$ . In addition, a base of 10 means that the system is based on 10 different signs (0, 1, ..., 9) with which we distinguish one-digit numbers. Further, we recognize that each column in the abacus has 9 beads (when we agree on the standard that a column without any bead represents the sign 0).

The numeral system with base 10 (the ‘decimal system’) is what we use to measure/count/quantify things in our everyday life. Moreover, it is what we normally work with in applied math and statistics. However, the simple but very useful concept of the counting frame also works well for numeral systems with other bases. Historically, numeral systems with other bases existed in various cultures. For example, ancient cultures in Mesopotamia used different forms of a sexagesimal system (base 60), consisting of 60 different signs to distinguish ‘one-digit’ numbers.

Note that this logic holds both ways: if a numeral system only consists of two different signs, it follows that the system has to be of base 2 (i.e., a ‘binary system’). As it turns out, this is exactly the kind of numeral system that becomes relevant once we use electronic tools, that is, digital computers, to calculate and process data.

### 3.1.1 The binary system

Anything related to electronically processing (and storing) *digital data* has to be built on a binary system. The reason is that a microprocessor (similar to a light switch) can only represent two signs (states): *on and off*. We usually refer to ‘off’ with the sign ‘0’ and to ‘on’ with the sign ‘1’. A numeral system consisting only of the signs 0 and 1 must, thus, be of base 2. It follows that an abacus of this binary system has columns  $2^0 = 1$ ,  $2^1 = 2$ ,  $2^2 = 4$ , and so forth. Moreover, each column has only one bead (1) or none (0). Nevertheless, we can express all (natural) numbers from the decimal system.<sup>3</sup> For example, the num-

<sup>3</sup>Representing fractions is much harder in a binary system than representing natural numbers. The reason is fractions such as  $1/3 = 0.333\dots$  actually constitute an infinite sequence of 0s and 1s. The representation of such numbers in a computer (via so-called ‘floating point’ numbers) is thus in reality not 100% accurate.

ber 139 which we have expressed with the ‘decimal abacus’, would be expressed as follows with a base 2 system:

$$(1 \times 2^7) + (1 \times 2^3) + (1 \times 2^1) + (1 \times 2^0) = 139.$$

More precisely, when including all columns of our binary system abacus (that is, including also the columns set to 0), we would get the following:

$$(1 \times 2^7) + (0 \times 2^6) + (0 \times 2^5) + (0 \times 2^4) + (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) = 139.$$

Now, compare this with the abacus in the decimal system above. There, we set the third column to 1, the second column to 3, and the first column (from the right) to 9, resulting in 139. If we do the same in the binary system (where we can set each column either to 0 or 1), we get 10001011. That is, the number 139 in the decimal system corresponds to 10001011 in the binary system.

How can a computer know that? To correctly print the three symbols 139 to our computer screen when dealing with the binary expression 10001011, the computer needs to rely on a predefined mapping between binary coded values and the symbols for (Arabic) decimal numbers like 3. Since a computer can only understand binary expressions, we have to define a *standard* of how 0s and 1s correspond to symbols, colors, etc. that we see on the screen.

Of course, this does not change the fact that any digital data processing is, in the end, happening in the binary system. But in order to avoid having to work with a keyboard consisting only of an on/off (1/0) switch, low-level standards that define how symbols like 3, A, #, etc. corresponding expressions in 0s and 1s help us to interact with the computer. It makes it easier to enter data and commands into the computer as well as understand the output (i.e., the result of a calculation performed on the computer) on the screen. A standard defining how our number symbols in the decimal system correspond to binary numbers (again, reflecting the idea of an abacus) can be illustrated in the following table:

Number	128	64	32	16	8	4	2	1
0 =	0	0	0	0	0	0	0	0
1 =	0	0	0	0	0	0	0	1
2 =	0	0	0	0	0	0	1	0
3 =	0	0	0	0	0	0	1	1
...								
139 =	1	0	0	0	1	0	1	1

### 3.1.2 The hexadecimal system

From the above table, we also recognize that binary numbers can become quite long rather quickly (have many digits). In Computer Science it is, therefore, quite common to use another numeral system to refer to binary numbers: the *hexadecimal* system. In this system, we have 16 symbols available, consisting of 0-9 (used like in the decimal system) and A-F (for the numbers 10 to 15). Because we have 16 symbols available, each digit represents an increasing power of 16 ( $16^0$ ,  $16^1$ , etc.). The decimal number 139 is expressed in the hexadecimal system as follows.

$$(8 \times 16^1) + (11 \times 16^0) = 139.$$

More precisely, following the convention of the hexadecimal system used in Computer Science (where B stands for 11), it is:

$$(8 \times 16^1) + (B \times 16^0) = 8B = 139.$$

Hence, 10001011 in the binary system is 8B in the hexadecimal system and 139 in the decimal system. The primary use of hexadecimal notation when dealing with binary numbers is the more 'human-friendly' representation of binary-coded values. First, it is shorter than the raw binary representation (as well as the decimal representation). Second, with a little bit of practice it is much easier for humans to translate forth and back between hexadecimal and binary notation than it is between decimal and binary. This is because each hexadecimal digit can directly be converted into its four-digit binary equivalent (from looking at the table above):  $8 = 1000$ ,  $B = 11 = 1011$ , thus 8B in hexadecimal notation corresponds to 10001011 (1000 1011) in binary coded values.

### 3.2 Character encoding

Computers are not only used to process numbers but in a wide array of applications, they are used to process text. Most fundamentally, when writing computer code, we type in commands in the form of text consisting of common alphabetical letters. How can a computer understand text if it only understands 0s and 1s? Well, again, we have to agree on a certain standard of how 0s and 1s correspond to characters/letters. While the conversions of integers between different numeral systems follow all essentially the same principle (~ the idea of a ‘digital’ abacus), the introduction of standards defining how 0s and 1s correspond to specific letters of different human languages is way more arbitrary.

Today, many standards define how computers translate 0s and 1s into more meaningful symbols like numbers, letters, and special characters in various natural (human) languages. These standards are called *character encodings*. They consist of what is called a ‘coded character set’, basically a mapping of unique numbers (in the end in binary coded values) to each character in the set. For example, the classical ASCII (American Standard Code for Information Interchange) assigns the following numbers to the respective characters:

Binary	Hexadecimal	Decimal	Character
0011 1111	3F	63	?
0100 0001	41	65	A
0110 0010	62	98	b

The convention that 0100 0001 corresponds to A is only true by definition of the ASCII character encoding.<sup>4</sup> It does not follow any law of nature or fundamental logic. Somebody simply defined this standard at some point. To have such standards (and widely accept them) is paramount to have functioning software and meaningful data sets that can be used and shared across many users and computers. If we

<sup>4</sup>Each character digit is expressed in 1 byte (8 0/1 digits). The ASCII character set thus consists of  $2^8 = 256$  different characters.



write a text and store it in a file based on the ASCII standard, and that text (which, under the hood, only consists of 0s and 1s) is read on another computer only capable of mapping the binary code to another character set, the output would likely be complete gibberish.

In practice, the software we use to write emails, browse the Web, or conduct statistical analyses all have some of these standards built into them. Usually, we do not have to worry about encoding issues when simply interacting with a computer through such programs. The practical relevance of such standards becomes much more relevant once we *store* or *read* previously stored data/computer code.

---

### 3.3 Computer code and text files

Understanding the fundamentals of what digital data is and how computers process them is the basis for approaching two core themes of this book: (I) How *data* can be *stored* digitally and be read by/imported to a computer (this will be the main subject of the next chapter), and (II) how we can give instructions to a computer by writing *computer code* (this will be the main topic of the first two exercises/workshops).

In both of these domains, we mainly work with one simple type of document: *text files*. Text files are a collection of characters (with a given character encoding). Following the logic of how binary code is translated forth and back into text, they are a straightforward representation of the underlying information (0s and 1s). They are used to store both structured data (e.g., tables), unstructured data (e.g., plain texts), or semi-structured data (such as websites). Similarly, they are used to write and store a collection of instructions to the computer (i.e., computer code) in any computer language.

From our everyday use of computers (notebooks, smartphones, etc.), we are accustomed to certain software packages to write text. Most prominently, we use programs like Microsoft Word or email clients (Outlook, Thunderbird, etc.). However, these programs are a poor choice for writing/editing plain text. Such programs tend to add all kinds of formatting to the text and use specific file formats to store

formatting information in addition to the raw text. In other words, when using this type of software, we actually do not only write plain text. However, any program to read data or execute computer code expects only plain text. Therefore, we must only use *text editors* to write and edit text files. For example, with Atom<sup>5</sup> or RStudio<sup>6</sup>.

---

### 3.4 Data processing basics

By now, we already have a much better understanding of what the 0s and 1s stand for, how they might enter the computer, where they might be stored on the computer, and where we might see the output of processing data on a computer.



**FIGURE 3.2:** The ‘blackbox’ of data processing.

This section briefly summarizes the key components of processing data with a computer, following ? (chapter 9, pages 199-204).

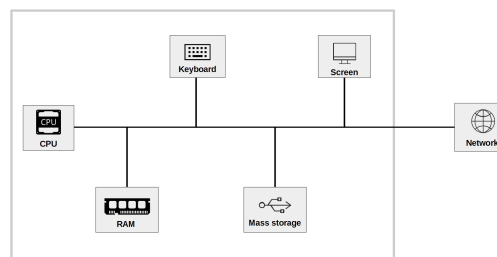
---

<sup>5</sup><https://atom.io/>

<sup>6</sup><https://www.rstudio.com/products/RStudio/>

### 3.4.1 Components of a standard computing environment

Figure ?? illustrates the key components of a standard computing environment (PC, notebook, tablet, etc.) with which we can process data. A programming language (such as R) allows us to work with these hardware components (i.e., control them) in order to perform our tasks (e.g., cleaning/analyzing data).



**FIGURE 3.3:** Basic components of a standard computing environment.

- The component actually *processing* data is the Central Processing Unit (CPU). When using R to process data, R commands are translated into complex combinations of a small set of basic operations which the CPU then executes.
- To work with data (e.g., in R), it first must be loaded into the *memory* of our computer. More specifically, into the Random Access Memory (RAM). Typically, data is only loaded in the RAM as long as we work with it.
- The *Keyboard* is the typical *input* hardware. It is the key tool we need to interact with the computer in this book (in contrast, in a book on graphic design this would rather be the mouse).
- *Mass Storage* refers to the type of computer memory we use to store data in the long run. Typically this is what we call the *hard drive* or *hard disk*. In these days, the relevant hard disk for storing and accessing data is actually often not the one physically built into our computer but a hard disk ‘in the cloud’ (built into a server to which we connect over the Internet).

- The *Screen* is our primary output hardware that is key to interact with the computer.
- Today almost all computers are connected to a local *network* which in turn is connected to the Internet. With the rise of ‘Big Data’ this component has become increasingly relevant to understand, as the key data sources might reside in this network.

Note how understanding the role of these components helps us to capture what happened in the initial example of this book (online survey of the class):

1. You have used your computers to access a website (over the Internet) and used your keyboards to enter data into this website (a Google sheet in that case).
2. My R program has accessed the data you have entered into the Google sheet (again over the Internet), downloaded the data, and loaded it into the RAM on my notebook.
3. There, the data has been processed in order to produce an output (in the form of statistics/plots), which in the end has been shown on the screen.

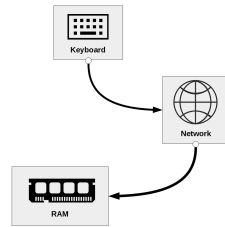
In the following, we look at a related example (taken from ?, Chapter 9), illustrating which hardware components are involved when extracting data from a website.

### 3.4.2 Illustration

#### 3.4.2.1 Downloading/accessing a website

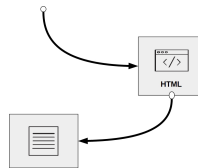
First, we use our keyboard to enter the address of a website in the address bar of our browser (i.e., Firefox). The browser then accesses the network and loads the webpage to the RAM. Figure ?? illustrates this point.

In this simple task of accessing/visiting a website, several parts of data are involved, as shown in ??:



**FIGURE 3.4:** Components involved in visiting a website.

1. The string of characters that we type into the address bar of the browser
2. The source code of the website (written in Hypertext markup language, HTML), being sent through the network
3. The HTML document (the website) stored in the RAM



**FIGURE 3.5:** Data involved in visiting a website.

What we just did with our browser can also be done via R. First, we tell R to download the webpage and read its source code into RAM:

```
clockHTML <- readLines("https://www.census.gov/popclock/")
```

And we can have a look at the first lines of the web page's source code:

```
head(clockHTML)
```

```
## [1] "<!DOCTYPE html>"  
## [2] "<html lang=\"en\">"
```

```
## [3] "<head>"
## [4] "\t<title>Population Clock</title>"
## [5] "      <meta charset=\"UTF-8\">"
## [6] "      <meta name=\"description\" content=\"Shows estimates of current USA Population overa
```

Note that what happened is essentially the same as displayed in the diagram above. The only difference is that here, we look at the raw code of the webpage, whereas in the example above, we looked at it through our browser. What the browser did, is to *render* the source code, meaning it translated it into the nicely designed webpage we see in the browser window.

#### 3.4.2.2 Searching/filtering the webpage

Having the webpage loaded into RAM, we can now process this data in different ways via R. For example; we can search the source code for the line that contains the part of the website with the world population counter:

```
line_number <- grep('id="world-pop-container"', clockHTML)
```

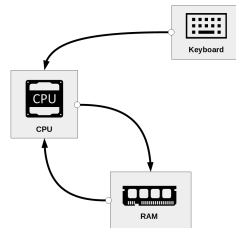
That is, we ask R on which line in `clockHTML` (the source code stored in RAM) the text `id="world-pop-container"` is and store the answer (the line number) in RAM under the variable name `line_number`.

We can now check on which line the text was found.

```
line_number
```

```
## [1] 444
```

Again, we can illustrate with a simple diagram which computer components were involved in this task. First, we entered an R command with the keyboard; the CPU is processing it and as a result accesses `clockHTML` in RAM, executes the search, returns the line number, and stores it in a new variable called `line_number` in RAM. In our current R session, we thus now have two ‘objects’ stored in RAM: `clockHTML` and `line_number`, which we can further process with R.

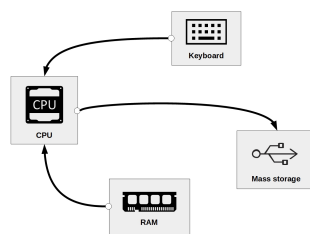


**FIGURE 3.6:** Accessing data in RAM, processing it, and storing the result in RAM.

Finally, we can tell R to store the source code of the downloaded web-page on our local hard drive.

```
writeLines(clockHTML, "clock.html")
```

That means we tell R to access the data stored in `clockHTML` in RAM and write this data to a text file called “clock.html” on our local hard drive (in the current working directory of our R session). The following diagram again illustrates the hardware components involved in these steps.



**FIGURE 3.7:** Writing data stored in RAM to a Mass Storage device (hard drive).





# 4

---

## *Data Storage and Data Structures*

---

A core part of the practical skills covered in this book has to do with writing, executing, and storing *computer code* (i.e., instructions to a computer in a language that it understands) as well as storing and reading *data*. The way data is typically stored on a computer follows quite naturally from the outlined principles of how computers process data (both technically speaking and in terms of practical applications). Based on a given standard of how 0s and 1s are translated into the more meaningful symbols we see on our keyboard, we simply write data (or computer code) to a *text file* and save this file on the hard-disk drive of our computer. Again, what is stored on disk is in the end only consisting of 0s and 1s. But, given the standards outlined in the previous chapter, these 0s and 1s properly map to characters that we can understand when reading the file again from the disk and looking at its content on our computer screen.

---

### 4.1 Unstructured data in text files

In the simplest case, we want to store a text literally. For example, we store the following phrase

```
Hello, World!
```

in a text file named `helloworld.txt`. Technically, this means that we allocate a block of computer memory to `helloworld.txt` (a ‘file’ is, in the end, a block of computer memory). The ending `.txt` indicates to the operating system of our computer what kind of data is stored in this file. When clicking on the file’s symbol, the operating system will open the file (read it into RAM) with the default program assigned to open

.txt-files (for example, Atom<sup>1</sup> or RStudio<sup>2</sup>). That is, the *format* of a file says something about how to interpret the 0s and 1s. If the text editor displays `Hello World!` as the content of this file, the program correctly interprets the 0s and 1s as representing plain text and uses the correct character encoding to convert the raw 0s and 1s yet again to symbols that we can read more easily. Similarly, we can show the content of the file in the command-line terminal (here OSX or Linux):

```
cat helloworld.txt
```

```
## Hello World!
```

Or, from the R-console:

```
system("cat helloworld.txt")
```

However, we can also use a command-line program (here, `xxd`) to display the content of `helloworld.txt` without actually ‘translating’ it into ASCII characters.<sup>3</sup> That is, we can directly look at how the content looks like as 0s and 1s:

```
xxd -b helloworld.txt
```

```
## 00000000: 01001000 01100101 01101100 01101100 01101111 00100000 Hello
## 00000006: 01010111 01101111 01110010 01101100 01100100 00100001 World!
```

Similarly, we can display the content in hexadecimal values:

```
xxd helloworld.txt
```

```
## 00000000: 4865 6c6c 6f20 576f 726c 6421 Hello World!
```

Next, consider the text file `hastamanana.txt`, which was written and stored with another character encoding. When looking at the content,

<sup>1</sup><https://atom.io/>

<sup>2</sup><https://www.rstudio.com/>

<sup>3</sup>To be precise, this program shows both the raw binary content as well as its ASCII representation.

assuming the now common UTF-8 encoding, the content seems a little odd:

```
cat hastamanana.txt
```

```
## Hasta Ma?ana!
```

We see that it is a short phrase written in Spanish. Strangely it contains the character `?` in the middle of a word. The occurrence of special characters in unusual places of text files is an indication of using the wrong character encoding to display the text. Let's check what the file's encoding is.

```
file -b hastamanana.txt
```

```
## ISO-8859 text
```

This tells us that the file is encoded with ISO-8859 ('Latin1'), a character set for several Latin languages (including Spanish). Knowing this, we can check how the content looks like when we change the encoding to UTF-8 (which then properly displays the content on the screen)<sup>4</sup>:

```
iconv -f iso-8859-1 -t utf-8 hastamanana.txt | cat
```

```
## Hasta Mañana!
```

When working with data, we must therefore ensure that we apply the proper standards to translate the binary coded values into a character/text representation that is easier to understand and work with. In recent years, much more general (or even 'universal') standards have been developed and adopted worldwide, particularly UTF-8. Thus, if we deal with recently generated data sets, we usually can expect that

---

<sup>4</sup>Note that changing the encoding in this way only works well if we know what the original encoding of the file is (i.e., the encoding used when the file was initially created). While the `file` command above simply tells us what the original encoding was, it has to make an educated guess in most cases (as the original encoding is often not stored as metadata describing a file). See, for example, the Mozilla Charset Detectors<sup>5</sup>.

they are encoded in UTF-8, independent of the data's country of origin. However, when working on a research project with slightly older data sets from different sources, encoding issues still occur quite frequently. It is thus crucial to understand the origin of the problem early on when the data seem to display weird characters. Encoding issues are among the most basic problems that can occur when importing data.

So far, we have only looked at very simple examples of data stored in text files, essentially only containing short phrases of text. Nevertheless, the most common formats of storing and transferring data (CSV, XML, JSON, etc.) build on exactly the same principle: characters in a text-file. The difference between such formats and the simple examples above is that their content is structured in a very specific way. That is, on top of the lower-level conversion of 0s and 1s into characters/text, we add another standard, a data format, giving the data more *structure*, making it even simpler to interpret and work with the data on a computer. Once we understand how data is represented at a low level (the topic of the previous chapter), it is much easier to understand and distinguish different forms of storing structured data.

---

## 4.2 Structured data formats

As nicely pointed out by ?, the most commonly used software today is designed in a very 'user-friendly' way, leading to situations in which we as users are 'told' by the computer what to do (rather than the other way around). A prominent symptom of this phenomenon is that specific software is automatically assigned to open files of specific formats, so we don't have to bother about what the file's format actually is but only have to click on the icon representing the file.

However, if we actually want to engage seriously with a world driven by data, we have to go beyond the habit of simply clicking on (data-)files and let the computer choose what to do with it. Since most common formats to store data are in essence text files (in research contexts usually in ASCII or UTF-8 encoding), a good way to start engaging with a

data set is to look at the raw text file containing the data in order to get an idea of how the data is structured.

For example, let's have a look at the file `ch_gdp.csv`. When opening the file in a text editor we see the following:

```
year,gdp_chfb
1980,184
1985,244
1990,331
1995,374
2000,422
2005,464
```

At first sight, the file contains a collection of numbers and characters. Having a closer look, certain structural features become apparent. The content is distributed over several rows, with each row containing a comma character (,). Moreover, the first row seems systematically different from the following rows: it is the only row containing alphabetical characters, all the other rows contain numbers. Rather intuitively, we recognize that the specific way in which the data in this file is structured might, in fact, represent a table with two columns: one with the variable `year` describing the year of each observation (row), the other with the variable `gdp_chfb` with numerical values describing another characteristic of each observation/row (we can guess that this variable is somehow related to GDP and CHF/Swiss francs).

The question arises, how do we work with this data? Recall that it is simply a text file. All the information stored in it is displayed above. How could we explain to the computer that this is a table with two columns containing two variables describing six observations? We would have to come up with a sequence of instructions/rules (i.e., an algorithm) of how to distinguish columns and rows when all that a computer can do is sequentially read one character after the other (more precisely one 0/1 after the other, representing characters).

This algorithm could be something along the lines of:

1. Start with an empty table consisting of one cell (1 row/column).

2. While the end of the input file is not yet reached, do the following: Read characters from the input file, and add them one-by-one to the current cell. If you encounter the character ',', ignore it, create a new field, and jump to the new field. If you encounter the end of the line, create a new row and jump to the new row.

Consider the usefulness of this algorithm. It would certainly work quite well for the particular file we are looking at. But what if another file we want to read data from does not contain any ',' but only ';' ? We would have to tweak the algorithm accordingly. Hence, again, it is extremely useful if we can agree on certain standards of how data structured as a table/matrix is stored in a text file.

#### 4.2.1 CSVs and fixed-width format

Incidentally, the example we are looking at here is in line with such a standard, i.e., Comma-Separated Values (CSV, therefore `.csv`). In technical terms, the simple algorithm outlined above is a CSV parser<sup>6</sup>. A CSV parser is a software component which takes a text file with CSV standard structure as input and builds a data structure in the form of a table (represented in RAM). If the input file does not follow the agreed-on CSV standard, the parser will likely fail to properly 'parse' (read/translate) the input.

CSV files are very often used to transfer and store data in a table-like format. They essentially are based on two rules defining the structure of the data: commas delimit values/fields in a row and the end of a line indicates the end of a row.<sup>7</sup>

The comma is clearly visible when looking at the raw text content of `ch_gdp.csv`. However, how does the computer know that a line is ending? By default most programs to work with text files do not show an explicit symbol for line endings but instead display the text on

---

<sup>6</sup>[https://en.wikipedia.org/wiki/Parsing#Computer\\_languages](https://en.wikipedia.org/wiki/Parsing#Computer_languages)

<sup>7</sup>In addition, if a field contains itself a comma (the comma is part of the data), the field needs to be surrounded by double-quotes ("). If a field contains double-quotes it also has to be surrounded by double-quotes, and the double quotes in the field must be preceded by an additional double-quote.

the next line. Under the hood, these line endings are, however, non-printable characters. We can see this when investigating `ch_gdp.csv` in the command-line terminal (via `xxd`):

```
xxd ch_gdp.csv
```

```
## 00000000: efbb bf79 6561 722c 6764 705f 6368 6662  ...year,gdp_chfb
## 00000010: 0d31 3938 302c 3138 340d 3139 3835 2c32  .1980,184.1985,2
## 00000020: 3434 0d31 3939 302c 3333 310d 3139 3935  44.1990,331.1995
## 00000030: 2c33 3734 0d32 3030 302c 3432 320d 3230  ,374.2000,422.20
## 00000040: 3035 2c34 3634                                05,464
```

When comparing the hexadecimal values with the characters they represent on the right side, we recognize that a full stop (.) is printed to the output right before every year. Moreover, when inspecting the hexadecimal code, we recognize that this . corresponds to `0d` in the hexadecimal code. `0d` is indeed the sequence of `0s` and `1` indicating the end of a line. Because this character does not actually correspond to a symbol printed on the screen, it is replaced by . in the printed output of the text.<sup>8</sup>

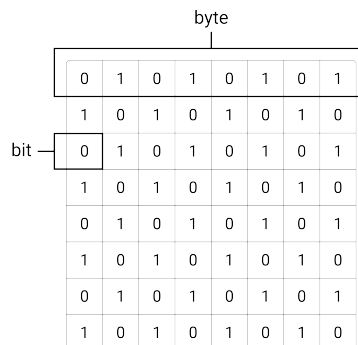
While CSV files have become a very common way to store ‘flat’/table-like data in plain text files, several similar formats can be encountered in practice. Most commonly they either use a different delimiter (for example, tabs/white space) to separate fields in a row, or fields are defined to consist of a fixed number of characters (so-called fixed-width formats). In addition, various more complex standards to store and transfer digital data are widely used to store data. Particularly in the context of web data (data stored and transferred online), files storing data in these formats (e.g., XML, JSON, YAML, etc.) are in the end, just plain text files. However, they contain a larger set of special characters/delimiters (or combinations of these) to indicate the structure of the data stored in them.

---

<sup>8</sup>The same applies to other sequences of `0s` and `1s` that do not correspond to a printable character.

### 4.3 Units of information/data storage

The question of how to store digital data on computers also raises the question of storage capacity. Because every type of digital data can, in the end, only be stored as 0s and 1s, it makes perfect sense to define the smallest unit of information in computing as consisting of either a 0 or a 1. We call this basic unit a *bit* (from *binary digit*; abbrev. 'b'). Recall that the decimal number 139 corresponds to the binary number 10001011. To store this number on a hard disk, we require a capacity of 8 bits or one *byte* (1 byte = 8 bits; abbrev. 'B'). Historically, one byte encoded a single character of text (i.e., in the ASCII character encoding system). 4 bytes (or 32 bits) are called a *word*. The following figure illustrates this point.



**FIGURE 4.1:** Byte and bit.

Bigger units for storage capacity usually build on bytes:

- 1 kilobyte (KB) =  $1000^1 \approx 2^{10}$  bytes
- 1 megabyte (MB) =  $1000^2 \approx 2^{20}$  bytes
- 1 gigabyte (GB) =  $1000^3 \approx 2^{30}$  bytes
- 1 terabyte (TB) =  $1000^4 \approx 2^{40}$  bytes
- 1 petabyte (PB) =  $1000^5 \approx 2^{50}$  bytes
- 1 exabyte (EB) =  $1000^6 \approx 2^{60}$  bytes
- 1 zettabyte (ZB) =  $1000^7 \approx 2^{70}$  bytes

1ZB = 1000000000000000000000 bytes = 1 billion terabytes = 1 trillion gigabytes.



---

#### 4.4 Data structures and data types in R

So far, we have focused on how data is stored on the hard disk. That is, we discovered how 'O's and 'I's correspond to characters (using specific standards: character encodings), how a sequence of characters in a text file is a useful way to store data on a hard disk, and how specific standards are used to structure the data in a meaningful way (using special characters). When thinking about how to store data on a hard disk, we are usually concerned with how much space (in bytes) the data set needs, how well it is transferable/understandable, and how well we can retrieve information from it. All of these aspects go into the decision of what structure/format to choose etc.

However, none of these considers how we actively work with the data. For example, when we want to sum up the `gdp_chfb` column in `ch_gdp.csv` (the table example above), do we actually work within the CSV data structure? The answer is usually no.

Recall from the basics of data processing that data stored on the hard disk is loaded into RAM to work with (analysis, manipulation, cleaning, etc.). The question arises as to how data is structured in RAM? That is, what data structures/formats are used to work with the data actively?

We distinguish two basic characteristics:

1. Data **types**: integers; real numbers ('numeric values', floating point numbers); text ('string', 'character values').
2. Basic **data structures** in RAM: - *Vectors* - *Factors* - *Arrays/matrices* - *Lists* - *Data frames* (very R-specific)

Depending on the data structure/format stored in on the hard disk, the data will be more or less usefully represented in one of the above structures in RAM. For example, in the R language it is quite common to represent data stored in CSVs on disk as *data frames* in RAM. Similarly, it is quite common to represent a more complex format on disk, such as JSON, as a nested list in RAM.

Importing data from the hard disk (or another mass storage device) into RAM in order to work with it in R essentially means reading the sequence of characters (in the end, of course, 0s and 1s) and mapping them, given the structure they are stored in, into one of these structures for representation in RAM.<sup>9</sup>.

#### 4.4.1 Data types

From the previous chapters, we know that digital data (0s and 1s) can be interpreted in different ways depending on the *format* it is stored in. Similarly, data loaded into RAM can be interpreted differently by R depending on the data *type*. Some operators or functions in R only accept data of a specific type as arguments. For example, we can store the numeric values 1.5 and 3 in the variables `a` and `b`, respectively.

```
a <- 1.5  
b <- 3
```

R interprets this data as type `double` (class 'numeric'; a 'double precision floating point number'):

```
typeof(a)
```

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

```
class(a)
```

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

Given that these bytes of data are interpreted as numeric, we can use operators (here: math operators) that can work with such types:

```
a + b
```

```
## [1] 4.5
```

---

<sup>9</sup>In the chapter on data gathering and data import, we will cover this crucial step in detail.

If we define `a` and `b` as follows, R will interpret the values stored in `a` and `b` as text (character).

```
a <- "1.5"  
b <- "3"
```

```
typeof(a)
```

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

```
class(a)
```

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

Now the same line of code as above will result in an error:

```
a + b
```

```
## Error in a + b: non-numeric argument to binary operator
```

The reason is that an operator `+` expects numeric or integer values as arguments. When importing data sets with many different variables (columns), it is thus necessary to ensure that each column is interpreted in the intended way. That is, we have to make sure R is assigning the right type to each of the imported variables. Usually, this is done automatically. However, with large and complex data sets, the automatic recognition of data types when importing data can fail when the data is not perfectly cleaned/prepared (in practice, this is very often the case).

#### 4.4.2 Data structures

For now, we have only looked at individual bytes of data. An entire data set can consist of gigabytes of data containing text and numeric values. How can such collections of data values be represented in R? Here, we look at the main data structures implemented in R. All of these structures will play a role in some of the hands-on exercises.

#### 4.4.2.1 Vectors

Vectors are collections of values of the same type. They can contain either all numeric values or all character values. We will use the following symbol to refer to vectors.



1
2
3

**FIGURE 4.2:** Illustration of a numeric vector.

For example, we can initiate a character vector containing the names of persons:

```
persons <- c("Andy", "Brian", "Claire")
persons
```

```
## [1] "Andy" "Brian" "Claire"
```

And we can initiate a numeric vector with the age of these persons:

```
ages <- c(24, 50, 30)
ages
```

```
## [1] 24 50 30
```

#### 4.4.2.2 Factors

Factors are sets of categories. Thus, the values come from a fixed set of possible values.



Male
Male
Female

**FIGURE 4.3:** Illustration of a factor.

For example, we might want to initiate a factor that indicates the gender of a number of people.

```
gender <- factor(c("Male", "Male", "Female"))
gender
```

```
## [1] Male   Male   Female
## Levels: Female Male
```

#### 4.4.2.3 Matrices/Arrays

Matrices are two-dimensional collections of values, arrays of higher-dimensional collections of values of the same type. For an illustration, consider the following  $3 \times 3$  integer matrix and the  $3 \times 3 \times 3$  integer array.

1	4	7
2	5	8
3	6	9

**FIGURE 4.4:** Illustration of an integer matrix.

5	16	0		
4	10	23	15	
8	2	4	10	6
1	17	0	12	
5	56	13		

**FIGURE 4.5:** Illustration of three-dimensional integer array.

In R, we can, for example, initiate a three-row/three-column integer matrix as follows.

```
my_matrix <- matrix(1:9, nrow = 3)
my_matrix
```

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

```
## [3,]    3    6    9
```

Similarly, we can initiate a three-dimensional array via the `array()`-function as shown below. Note that instead of defining only the numbers of rows (columns), you need to define the number of elements in each dimension via the `dim`-parameter.

```
my_array <- array(1:27, dim = c(3,3,3))
my_array
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]   10   13   16
## [2,]   11   14   17
## [3,]   12   15   18
##
## , , 3
##
##      [,1] [,2] [,3]
## [1,]   19   22   25
## [2,]   20   23   26
## [3,]   21   24   27
```

We can access parts of matrices and arrays via indexing implemented with the `[]`-syntax. For example, select a particular row, column, or individual element of a matrix.

```
# Select the first row
my_matrix[1,]
```

```
## [1] 1 4 7
```

```
# Select the second column
my_matrix[,2]
```

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

```
# Select the value in the second column, third row.
my_matrix[3,2]
```

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

Following the same syntax, we can access specific parts of an array. Note, though, that we need to consider the correct number of dimensions of the corresponding array object. In the example of above, we have generated three-dimensional array, hence there needs to be three parts within the `[]`.

```
# Select the first rows
my_array[1,,]
```

```
##      [,1] [,2] [,3]
## [1,]    1   10   19
## [2,]    4   13   22
## [3,]    7   16   25
```

```
# Select the "third matrix"
my_array[, ,3]
```

```
##      [,1] [,2] [,3]
## [1,]   19   22   25
## [2,]   20   23   26
## [3,]   21   24   27
```

```
# Select the "last" element in the array
my_array[3,3,3]
```

```
## [1] 27
```

#### 4.4.2.4 Data frames, tibbles, and data tables

Data frames are the typical representation of a (table-like) data set in R. Each column can contain a vector of a given data type (or a factor), but all columns need to be of identical length. Thus in the context of data analysis, we would say that each row of a data frame contains an observation, and each column contains a characteristic of this observation.

1	Male	Andy
2	Male	Brian
3	Female	Claire

**FIGURE 4.6:** Illustration of a data frame.

The historical implementation of data frames in R is not very appropriate to work with large datasets.<sup>10</sup> These days there are new implementations of the data frame concept in R provided by different packages, which aim at making data processing based on ‘data frames’ faster. One is called `tibbles`, implemented and used in the `tidyverse` packages. The other is called `data table`, implemented in the `data. table`-package. In this book, we will encounter primarily classical `data. frame` and `tibble` (however, there will also be some hints to tutorials with `data. table`). In any case, once we understand data frames in general, working with `tibble` and `data. table` is quite easy because functions that accept classical data frames as arguments also accept those newer implementations.

Here is how we define a data frame in R, based on the examples of vectors and factors shown above.

<sup>10</sup>In the early days of R, this was not an issue because datasets that are rather large by today’s standards (in the Gigabytes) could not have been handled properly by normal computers anyway (due to a lack of RAM).

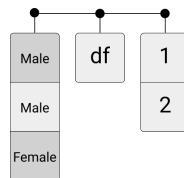


```
df <- data.frame(person = persons, age = ages, gender = gender)
df
```

```
##   person age gender
## 1  Andy  24   Male
## 2  Brian  50   Male
## 3 Claire  30 Female
```

#### 4.4.2.5 Lists

Unlike data frames, lists can contain different data types in each element. Moreover, they even can contain different data structures of *different dimensions* in each element. For example, a list could contain different other lists, data frames, and vectors with differing numbers of elements.



**FIGURE 4.7:** Illustration of a list.

This flexibility can easily be demonstrated by combining some of the data structures created in the examples above:

```
my_list <- list(my_array, my_matrix, df)
my_list
```

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

```
## , , 2
##
##      [,1] [,2] [,3]
## [1,]   10   13   16
## [2,]   11   14   17
## [3,]   12   15   18
##
## , , 3
##
##      [,1] [,2] [,3]
## [1,]   19   22   25
## [2,]   20   23   26
## [3,]   21   24   27
##
##
## [[2]]
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
##
## [[3]]
##   person age gender
## 1   Andy  24   Male
## 2  Brian  50   Male
## 3 Claire  30 Female
```

## 5

### *High-Dimensional Data*

So far, we have only looked at data structured in a flat/table-like representation (e.g. CSV files). In applied econometrics/statistics, it is common to only work with data sets stored in such format. Data manipulation, filtering, aggregation, etc. presupposes data in a table-like format. Hence, storing data in this format makes perfect sense.

As we observed in the previous chapter, the CSV structure has some disadvantages when representing more complex data in a text file. This is in particular true if the data contains nested observations (i.e., hierarchical structures). While a representation in a CSV file is theoretically possible, it is often far from practical to use other formats for such data. On the one hand, it is likely less intuitive to read the data correctly. On the other hand, storing the data in a CSV file might introduce a lot of redundancy. That is, the identical values of some variables would have to be repeated in the same column. The following code block illustrates this point for a data set on two families ((?), p. 116).

father	mother	name	age	gender
		John	33	male
		Julia	32	female
John	Julia	Jack	6	male
John	Julia	Jill	4	female
John	Julia	John jnr	2	male
		David	45	male
		Debbie	42	female
David	Debbie	Donald	16	male
David	Debbie	Dianne	12	female

From simply looking at the data, we can make the best guess which observations belong together (are one family). However, the implied

hierarchy is not apparent at first sight. While it might not matter too much that several values have to be repeated several times in this format, given that this data set is so small, the repeated values can become a problem when the data set is much larger. For each time `John` is repeated in the `father` column, we use 4 bytes of memory. Suppose millions of people are in this data set, and we have to transfer this data set very often over a computer network. In that case, these repetitions can become quite costly (as we would need more storage capacity and network resources).

Issues with complex/hierarchical data (with several observation types), intuitive human readability (self-describing), and efficiency in storage, as well as transfer, are all of great importance on the Web. In the context of web technologies, several data formats have been put forward to address these issues. Here, we discuss the two most prominent of these formats: Extensible Markup Language (XML)<sup>1</sup> and JavaScript Object Notation (JSON)<sup>2</sup>.

---

## 5.1 Deciphering XML

Before going into more technical details, let's try to figure out the basic logic behind the XML format by simply looking at some raw example data. For this, we turn to the Point Nemo case study in (?). The following code block shows the upper part of the data set downloaded from NASA's LAS server (here in a CSV-type format).

```
VARIABLE : Monthly Surface Clear-
sky Temperature (ISCCP) (Celsius)
FILENAME : ISCCPMonthly_avg.nc
FILEPATH : /usr/local/fer_data/data/
BAD FLAG : -1.E+34
SUBSET   : 48 points (TIME)
LONGITUDE: 123.8W(-123.8)
LATITUDE : 48.8S
```

---

<sup>1</sup><https://en.wikipedia.org/wiki/XML>

<sup>2</sup><https://en.wikipedia.org/wiki/JSON>

```

123.8W
16-JAN-1994 00    9.200012
16-FEB-1994 00    10.70001
16-MAR-1994 00    7.5
16-APR-1994 00    8.100006

```

Below, the same data is now displayed in XML format. Note that in both cases, the data is simply stored in a text file. However, it is stored in a format that imposes a different *structure* on the data.

```

<?xml version="1.0"?>
<temperatures>
<variable>Monthly Surface Clear-sky Temperature (ISCCP) (Celsius)</variable>
<filename>ISCCPMonthly_avg.nc</filename>
<filepath>/usr/local/fer_data/data/</filepath>
<badflag>-1.E+34</badflag>

<subset>48 points (TIME)</subset>
<longitude>123.8W(-123.8)</longitude>
<latitude>48.8S</latitude>
<case date="16-JAN-1994" temperature="9.200012" />
<case date="16-FEB-1994" temperature="10.70001" />
<case date="16-MAR-1994" temperature="7.5" />
<case date="16-APR-1994" temperature="8.100006" />

...
</temperatures>

```

What features does the format have? What is its logic? Is there room for improvement? The XML data structure becomes more apparent by using indentation and code highlighting.

```

<?xml version="1.0"?>
  <temperatures>
    <variable>Monthly Surface Clear-sky Temperature (ISCCP) (Celsius)</variable>
    <filename>ISCCPMonthly_avg.nc</filename>
    <filepath>/usr/local/fer_data/data/</filepath>
    <badflag>-1.E+34</badflag>
  </temperatures>

```

```

<subset>48 points (TIME)</subset>
<longitude>123.8W(-123.8)</longitude>
<latitude>48.8S</latitude>
<case date="16-JAN-1994" temperature="9.200012" />
<case date="16-FEB-1994" temperature="10.70001" />
<case date="16-MAR-1994" temperature="7.5" />
<case date="16-APR-1994" temperature="8.100006" />

...
</temperatures>

```

First, note how special characters are used to define the document's structure. We notice that < and >, containing some text labels seem to play a key role in defining the structure. These building blocks are called 'XML tags'. We are free to choose what tags we want to use. In essence, we can define them ourselves to most properly describe the data. Moreover, the example data reveals the flexibility of XML to depict hierarchical structures.

The actual content we know from the CSV-type example above is nested between the 'temperatures'-tags, indicating what the data is about.

```

<temperatures>
...
</temperatures>

```

Comparing the actual content between these tags with the CSV-type format above, we further recognize that there are two principal ways to link variable names to values.

```

<variable>Monthly Surface Clear-sky Temperature (ISCCP) (Celsius)</variable>
<filename>ISCCPMonthly_avg.nc</filename>
<filepath>/usr/local/fer_data/data/</filepath>
<badflag>-1.E+34</badflag>
<subset>48 points (TIME)</subset>

```

```

<longitude>123.8W(-123.8)</longitude>
<latitude>48.8S</latitude>
<case date="16-JAN-1994" temperature="9.200012" />
<case date="16-FEB-1994" temperature="10.70001" />
<case date="16-MAR-1994" temperature="7.5" />
<case date="16-APR-1994" temperature="8.100006" />

```

One way is to define opening and closing XML-tags with the variable name and surround the value with them, such as in `<filename>ISCCPMonthly_avg.nc</filename>`. Another way would be to encapsulate the values within one tag by defining tag attributes such as in `<case date="16-JAN-1994" temperature="9.200012" />`. In many situations, both approaches can make sense. For example, the way the temperature measurements are encoded in the example data set is based on the tag-attributes approach:

```

<case date="16-JAN-1994" temperature="9.200012" />
<case date="16-FEB-1994" temperature="10.70001" />
<case date="16-MAR-1994" temperature="7.5" />
<case date="16-APR-1994" temperature="8.100006" />

```

We could rewrite this by only using XML tags and no attributes:

```

<cases>
  <case>
    <date>16-JAN-1994</date>
    <temperature>9.200012</temperature>
  </case>
  <case>
    <date>16-FEB-1994</date>
    <temperature>10.70001</temperature>
  </case>
  <case>
    <date>16-MAR-1994</date>
    <temperature>7.5</temperature>
  </case>
</cases>

```

```
<case/>
<case>
  <date>16-APR-1994<date/>
  <temperature>8.100006<temperature/>
<case/>
<cases/>
```

As long as we follow the basic XML syntax, both versions are valid, and XML parsers can read them equally well.

Note the key differences between storing data in XML format in contrast to a flat, table-like format such as CSV:

- Storing the actual data and metadata in the same file is straightforward (as the above example illustrates). Storing metadata in the first lines of a CSV file (such as in the example above) is theoretically possible. However, by doing so, we break the CSV syntax, and a CSV-parser would likely break down when reading such a file (recall the simple CSV parsing algorithm). More generally, we can represent much more *complex (multi-dimensional)* data in XML files than what is possible in CSVs. In fact, the nesting structure can be arbitrarily complex as long as the XML syntax is valid.
- The XML syntax is largely self-explanatory and thus both *machine-readable and human-readable*. That is, not only can parsers/computers more easily handle complex data structures, but human readers can intuitively understand what the data is all about by looking at the raw XML file.

A potential drawback of storing data in XML format is that variable names (in tags) are repeated. Since each tag consists of a couple of bytes, this can be highly inefficient compared to a table-like format where variable names are only defined once. Typically, this means that if the data at hand is only two-dimensional (observations/variables), a CSV format makes more sense.



---

## 5.2 Deciphering JSON

In many web applications, JSON serves the same purpose as XML<sup>3</sup>. An obvious difference between the two conventions is that JSON does not use tags but attribute-value pairs to annotate data. The following code example shows how the same data can be represented in XML or in JSON (example code taken from <https://en.wikipedia.org/wiki/JSON>):

XML:

```
<person>
  <firstName>John</firstName>
  <lastName>Smith</lastName>
  <age>25</age>
  <address>
    <streetAddress>21 2nd Street</streetAddress>
    <city>New York</city>
    <state>NY</state>
    <postalCode>10021</postalCode>
  </address>
  <phoneNumber>
    <type>home</type>
    <number>212 555-1234</number>
  </phoneNumber>
  <phoneNumber>
    <type>fax</type>
    <number>646 555-4567</number>
  </phoneNumber>
  <gender>
    <type>male</type>
  </gender>
</person>
```

JSON:

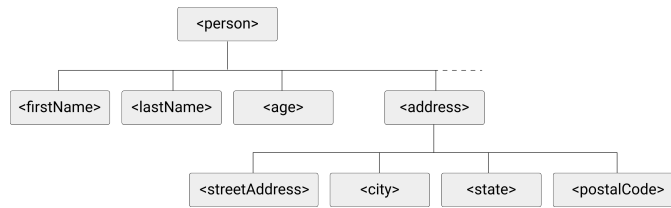
---

<sup>3</sup>Not that programs running on the server side are frequently capable of returning the same data in either format

```
{ "firstName": "John",  
  "lastName": "Smith",  
  "age": 25,  
  "address": {  
    "streetAddress": "21 2nd Street",  
    "city": "New York",  
    "state": "NY",  
    "postalCode": "10021"  
  },  
  "phoneNumber": [  
    {  
      "type": "home",  
      "number": "212 555-1234"  
    },  
    {  
      "type": "fax",  
      "number": "646 555-4567"  
    }  
  ],  
  "gender": {  
    "type": "male"  
  }  
}
```

Note that despite the syntax differences, the similarities regarding the nesting structure are visible in both formats. For example, `postalCode` is embedded in `address`, `firstName` and `lastName` are at the same nesting level, etc. The following figure, illustrating the nesting structure further illustrates this point. The basic logic of what element belongs to which higher-level element displayed in the tree diagram is encoded in both the JSON and the XML excerpts shown above.

Both XML and JSON are predominantly used to store rather complex, multi-dimensional data. Because they are both human- and machine-readable, these formats are often used to transfer data between applications and users on the Web. Therefore, we encounter these data formats particularly often when we collect data from online sources.

**FIGURE 5.1:** XML tree diagram.

Moreover, in the economics/social science context, large and complex data sets ('Big Data') often come along in these formats exactly because of the increasing importance of the Internet as a data source for empirical research in these disciplines.

### 5.3 Parsing XML and JSON in R

As in the case of CSV-like formats, there are several parsers in R that we can use to read XML and JSON data. The following examples are based on the example code shown above (the two text-files `persons.json` and `persons.xml`)

```

# load packages
library(xml2)

# parse XML, represent XML document as R object
xml_doc <- read_xml("persons.xml")
xml_doc

## {xml_document}
## <person>
## [1] <firstName>John</firstName>
## [2] <lastName>Smith</lastName>
## [3] <age>25</age>
## [4] <address>\n <streetAddress>21 2nd Street</streetAddress>\ ...
## [5] <phoneNumber>\n <type>home</type>\n <number>212 555-1234 ...
## [6] <phoneNumber>\n <type>fax</type>\n <number>646 555-4567< ...

```

```
## [7] <gender>\n  <type>male</type>\n</gender>
```

```
# load packages
library(jsonlite)

# parse the JSON-document shown in the example above
json_doc <- from JSON("persons.json")

# check the structure
str(json_doc)

## List of 6
## $ firstName : chr "John"
## $ lastName  : chr "Smith"
## $ age       : int 25
## $ address   :List of 4
## ..$ streetAddress: chr "21 2nd Street"
## ..$ city         : chr "New York"
## ..$ state        : chr "NY"
## ..$ postalCode   : chr "10021"
## $ phoneNumber:'data.frame': 2 obs. of 2 variables:
## ..$ type : chr [1:2] "home" "fax"
## ..$ number: chr [1:2] "212 555-1234" "646 555-4567"
## $ gender    :List of 1
## ..$ type: chr "male"
```

---

## 5.4 HTML

Recall the data processing example in which we investigated how a webpage can be downloaded, processed, and stored via the R command line. The code constituting a webpage is written in HyperText Markup Language (HTML)<sup>4</sup>, designed to be read in a web browser. Thus, HTML is predominantly designed for visual display in browsers,

---

<sup>4</sup><https://en.wikipedia.org/wiki/HTML>

not for storing data. HTML is used to annotate content and define the hierarchy of content in a document to tell the browser how to display ('render') this document on the computer screen. Interestingly for us, its structure is very close to that of XML. However, the tags that can be used are strictly pre-defined and are aimed at explaining the structure of a website.

While not intended as a format to store and transfer data, HTML documents (webpages) have de facto become a very important data source for many data science applications both in industry and academia.<sup>5</sup> Note how the two key concepts of computer code and digital data (both residing in a text file) are combined in an HTML document. From a web designer's perspective, HTML is a tool to design the layout of a webpage (and the resulting HTML document is rather seen as *code*). On the other hand, from a data scientist's perspective, HTML gives the data contained in a webpage (the actual content) a certain degree of structure which can be exploited to systematically extract the data from the webpage. In the context of HTML documents/webpages as data sources, we thus also speak of 'semi-structured data': a webpage can contain an HTML table (structured data) but likely also contains just raw text (unstructured data). In the following, we explore the basic syntax of HTML by building a simple webpage.

#### 5.4.1 Write a simple webpage with HTML

We start by opening a new text-file in RStudio (File->New File->TextFile). On the first line, we tell the browser what kind of document this is with the `<!DOCTYPE>` declaration set to `HTML`. In addition, the content of the whole HTML document must be put within `<html>` and `</html>`, which represents the 'root' of the HTML document. In this, you already recognize the typical (XML-like) annotation style in HTML with so-called HTML tags, starting with `<` and ending with `>` or `/>` in the case of the closing tag, respectively. What is defined between two tags is either another HTML tag or the actual content. An HTML document usually consists of two main components: the head (everything between `<head>` and `</head>`) and the body. The head typi-

---

<sup>5</sup>The systematic collection and extraction of data from such web sources (often referred to as Web Data Mining) goes well beyond the scope of this book.

cally contains metadata describing the whole document like, the title of the document: `<title>hello, world</title>`. The body (everything between `<body>` and `</body>`) contains all kinds of specific content: text, images, tables, links, etc. In our very simple example, we add a few words of plain text. We can now save this text document as `mysite.html` and open it in a web browser.

```
<!DOCTYPE html>

<html>
  <head>
    <title>hello, world</title>
  </head>
  <body>
    <h2> hello, world </h2>
  </body>
</html>
```

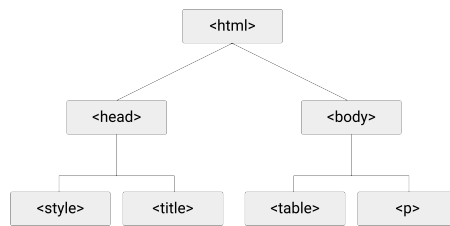
From this example, we can learn a few important characteristics of HTML:

1. It becomes apparent how HTML is used to annotate/'mark up' data/text (with tags) to define the document's content, structure, and hierarchy. Thus if we want to know what the title of this document is, we have to look for the `<title>`-tag.
2. This systematic structuring adheres to the nesting principle: 'head' and 'body' are nested within the 'HTML' document, at the same hierarchy level. The 'title' is defined within the 'head', one level lower. In other words, the 'title' is a component of the 'head,' which is a component of the 'html' document. This logic of encapsulating one part in another holds true in all correctly defined HTML documents. This is the type of 'data structure' mentioned above, which can be used to extract specific parts of data from HTML documents in a systematic manner.
3. We recognize that HTML code essentially expresses what is what in a document (note the similarity of the concept to

storing data in XML). HTML code does not contain explicit instructions like programming languages, telling the computer what to do. If an HTML document contains a link to another website, all that is needed, is to define (with HTML tag `<a href=...>`) that this is a link. We do not have to explicitly express something like “if the user clicks on a link, then execute this and that...”.

What to do with the HTML document is thus in the hands of the person who works with it. From the data scientist’s perspective, we need to know how to traverse and exploit the structure of an HTML document in order to systematically extract the specific content/data that we are interested in. We thus have to learn how to tell the computer (with R) to extract a specific part of an HTML document. And to do so, we have to acquire a basic understanding of the nesting structure implied by HTML (essentially the same logic as with XML).

One way to think about an HTML document is to imagine it as a tree-diagram, with `<html>...</html>` as the ‘root’, `<head>...</head>` and `<body>...</body>` as the ‘children’ of `<html>...</html>` (and ‘siblings’ of each other), `<title>...</title>` as the child of `<head>...</head>`, etc. Figure ?? illustrates this point.



**FIGURE 5.2:** HTML (DOM) tree diagram.

The illustration of the nested structure can help to understand how we can instruct the computer to find/extract a specific part of the data from such a document. In the following exercise, we revisit the example shown in the chapter on data processing to do exactly this.

### 5.4.2 Two ways to read a webpage into R

In this example, we look at Wikipedia's Economy of Switzerland page<sup>6</sup>, which contains the table depicted in Figure ??.

Year	GDP (billions of CHF)	US Dollar Exchange
1980	184	1.67 Francs
1985	244	2.43 Francs
1990	331	1.38 Francs
1995	374	1.18 Francs
2000	422	1.68 Francs
2005	464	1.24 Francs
2006	491	1.25 Francs
2007	521	1.20 Francs
2008	547	1.08 Francs
2009	535	1.09 Francs
2010	546	1.04 Francs
2011	659	0.89 Francs
2012	632	0.94 Francs
2013	635	0.93 Francs
2014	644	0.92 Francs
2015	646	0.96 Francs
2016	659	0.98 Francs
2017	668	1.01 Francs
2018	694	1.00 Francs

**FIGURE 5.3:** Source: [https://en.wikipedia.org/wiki/Economy\\_of\\_Switzerland](https://en.wikipedia.org/wiki/Economy_of_Switzerland).

As in the example shown in the data processing chapter ('world population clock'), we first tell R to read the lines of text from the HTML document that constitutes the webpage.

```
swiss_econ <- readLines("https://en.wikipedia.org/wiki/Economy_of_Switzerland")

## Warning in readLines("https://en.wikipedia.org/wiki/
## Economy_of_Switzerland"): incomplete final line found on
## 'https://en.wikipedia.org/wiki/Economy_of_Switzerland'
```

And we can check if everything worked out well by having a look at the first lines of the web page's HTML code:

<sup>6</sup>[https://en.wikipedia.org/wiki/Economy\\_of\\_Switzerland](https://en.wikipedia.org/wiki/Economy_of_Switzerland)



```
head(swiss_econ)
```

```
## [1] "<!DOCTYPE html>"
## [2] "<html class=\"client-nojs\" lang=\"en\" dir=\"ltr\">"
## [3] "<head>"
## [4] "<meta charset=\"UTF-8\"/>"
## [5] "<title>Economy of Switzerland - Wikipedia</title>"
## [6] "<script>document.documentElement.className=\"client-
js\";RLCONF={\"wgBreakFrames\":false,\"wgSeparatorTransformTable\":[\"\", \"\"],\"wgDigitTransl
3486-4c22-a5f8-689e1bbe3470\", \"wgCSPNonce\":false,\"wgCanonicalNamespace\":\"\", \"wgCanonical
language sources (de)\", \"Articles with German-
language sources (de)\", \"Webarchive template wayback links\", \"Articles with French-
language sources (fr)\", \"Articles with short description\", \"Short description is different from
```

The next thing we do is to look at how we can filter the webpage for certain information. For example, we search in the source code for the line that contains the part of the webpage with the table showing data on the Swiss GDP:

```
line_number <- grep('US Dollar Exchange', swiss_econ)
```

Recall: we ask R on which line in `swiss_econ` (the source code stored in RAM) the text `US Dollar Exchange` is and store the answer (the line number) in RAM under the variable name `line_number`.

Then, we check on which line the text was found.

```
line_number
```

```
## [1] 231
```

Knowing that the R object `swiss_econ` is a character vector (with each element containing one line of HTML code as a character string), we can look at this particular code:

```
swiss_econ[line_number]
```

```
## [1] "<th>US Dollar Exchange"
```

Note that this rather primitive approach is ok to extract a chunk of code from an HTML document, but it is far from practical when we want to extract specific parts of data (the actual content, not including HTML code). So far, we have completely ignored that the HTML tags give some structure to the data in this document. That is, we simply have read the entire document line by line, not making a difference between code and data. The approach to filter the document could have equally well been taken for a plain text file.

If we want to exploit the structure given by HTML, we need to *parse* the HTML when reading the webpage into R. We can do this with the help of functions provided in the *rvest* package:

```
# install package if not yet installed
# install.packages("rvest")

# load the package
library(rvest)
```

After loading the package, we read the webpage into R, but this time using a function that parses the HTML code:

```
# parse the webpage, show the content
swiss_econ_parsed <- read_html("https://en.wikipedia.org/wiki/Economy_of_Switzerland")
swiss_econ_parsed
```

```
## {html_document}
## <html class="client-nojs" lang="en" dir="ltr">
## [1] <head>\n<meta http-equiv="Content-Type" content="text/html ...
## [2] <body class="mediawiki ltr sitedir-ltr mw-hide-empty-elt n ...
```

Now we can easily separate the data/text from the HTML code. For ex-

ample, we can extract the HTML table containing the data we are interested in as data frames.

```
tab_node <- html_node(swiss_econ_parsed, xpath = "//*[@id='mw-content-text']/div/table[2]")
tab <- html_table(tab_node)
tab
```

```
## # A tibble: 19 x 3
##   Year `GDP (billions of CHF)` `US Dollar Exchange`
##   <int>          <int> <chr>
## 1  1980          184 1.67 Francs
## 2  1985          244 2.43 Francs
## 3  1990          331 1.38 Francs
## 4  1995          374 1.18 Francs
## 5  2000          422 1.68 Francs
## 6  2005          464 1.24 Francs
## 7  2006          491 1.25 Francs
## 8  2007          521 1.20 Francs
## 9  2008          547 1.08 Francs
## 10 2009          535 1.09 Francs
## 11 2010          546 1.04 Francs
## 12 2011          659 0.89 Francs
## 13 2012          632 0.94 Francs
## 14 2013          635 0.93 Francs
## 15 2014          644 0.92 Francs
## 16 2015          646 0.96 Francs
## 17 2016          659 0.98 Francs
## 18 2017          668 1.01 Francs
## 19 2018          694 1.00 Francs
```



# 6

## Text Data

*This chapter has been contributed by Aurélien Salling (ASallin<sup>1</sup>).*

### 6.1 Introduction

Text as data has become increasingly available in the past years, especially following the spread of the Internet and the numerisation of text sources. We are now able to make use of literary texts, analyses (such as financial analyses), reactions on social media (e.g., sentiment analysis to predict financial prices), political discourses, judicial decisions, central bank minutes, etc. in quantitative analyses.

The main challenge of working with text is that text is **unstructured**. Structured data are coded in such a way that variables can be directly used for analysis without too much loss of content. For instance, data are coded into binary or categorical variables in order to reduce and summarize the complexity of the information. In contrast, with unstructured text data, information is not easily summarized and requires a substantial amount of preparation in order to be used for analysis.

Working with text data can be broken down into eight steps, as depicted in the figure below. (1) Data acquisition: text data are collected from disparate sources. Examples are webpage scrapping, numerization of old administrative records, collection of tweets through an API, etc. (2) Text cleaning and (3) text preprocessing require the analyst to prepare the text in such a way that text information can be read by a statistical software. Most importantly, cleaning and preprocessing

---

<sup>1</sup><https://github.com/ASallin>