# 4

# Storing Data in Files

In this and the next chapters of this book, we will focus on tools for data contained in files: Your data resides in physical files on your hard disk, from where it is opened with a software of your choice, processed in various ways, and then stored again in a file. This is by far the most common workflow used in social science projects. Why do we need files at all? The answer is very simple: We use files for permanent data storage. When you work with a dataset in R (or in some other software, such as Excel or Stata), the table(s) – for example, the data frames in R – are temporarily stored in your computer's main memory. This is the part of your computer where data and programs are kept for fast access during the actual operation of your system. The problem is that this volatile memory does not function anymore when you turn off your system, and the entire content (and, thus, your data) disappears. Therefore, every computer has another type of data storage that remains *persistent* even when the system is shut down. This is usually your hard disk drive, but it can also be a network drive or a cloud storage folder.

When we save tabular data contained in the computer's main memory to files, we need to make sure that the tabular structure is preserved. Recall our discussion of the importance of data structure in the initial chapters of this book – a persistent storage of data in files would be useless if the actual structure of the data were lost. Therefore, there are different ways in which tabular data can be stored as files, such that the tabular structure is preserved. For a given file, the *file type* typically indicates if it contains a data table and how this table is stored in the file. You are probably familiar with file types for text documents (e.g., the Word format indicated by the `.docx` extension) or for graphics (e.g., the JPEG format using the `.jpg`



https://doi.org/10.1017/9781108990424.007 Published online by Cambridge University Press

or `.jpeg` extension). Similarly, there are different file types to store data tables. These file types are designed to keep the logical structure of our data table as a set of columns of particular types, and a set of rows. These file types constitute the main focus of this chapter.

## 4.1 TEXT AND BINARY FILES

Before we go through the list of the most commonly used file formats for data in the social sciences, we need to make a basic distinction between text and binary files. As the name suggests, *text* files contain information stored as plain text, such as program code for R and other programming languages. This is why you can open them with any text editor (such as the one built into RStudio) and view the contents. In contrast, *binary* files can be used and processed only by particular software tools – they essentially contain only 0s and 1s that make little sense to humans (but can be understood by the software tools designed for them). The term "binary" applies to all files that are not text and is used for many different file types, not just those that contain tabular data. For example, images, video, and sound are typically stored in binary files. To illustrate the difference, Figure 4.1 shows the contents of a binary file, viewed with the Unix `hexdump` command.

As you can see, the information in a binary file is not human-readable – the contents are completely cryptic and can only be processed by software designed for this file type. In contrast, the content of a text file can be understood by humans. You can create and open text files even with RStudio: Just choose File 〉 New File 〉 Text File, and you get a new editor pane, where you can start adding content to your text file and save it (see Figure 4.2).

The screenshot shows that text files contain text and numbers, but also various other invisible characters that are usually hidden in the text editor. Under Tools 〉 Global Options, you can turn on/off the display of these characters in the "Code" section in RStudio's preferences menu, in the "Display" pane. Just tick the box for "Show whitespace characters," and your text file will look similar to the one in Figure 4.2. There are

```
00000000  d0 cf 11 e0 a1 b1 1a e1  00 00 00 00 00 00 00 00  |................|
00000010  00 00 00 00 00 00 00 00  3e 00 03 00 fe ff 09 00  |........>.......|
00000020  06 00 00 00 00 00 00 00  00 00 00 00 01 00 00 00  |................|
00000030  27 00 00 00 00 00 00 00  00 10 00 00 29 00 00 00  |'...........)...|
00000040  01 00 00 00 fe ff ff ff  00 00 00 00 26 00 00 00  |............&...|
00000050  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|
```

FIGURE 4.1. Contents of a binary file.

https://doi.org/10.1017/9781108990424.007 Published online by Cambridge University Press

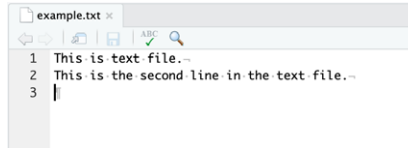


FIGURE 4.2. Example of a text file viewed in RStudio.

different invisible characters in the text file above; for example, white spaces (denoted as gray dots in the editor) are used to separate words, and there is a linebreak character at the end of each line. The end of the entire file is again marked with a specific character.

The set of characters you can use in a text file is defined by the file encoding. A file encoding is a mapping of numbers (which the computer stores internally, so nothing you need to worry about) to actual characters. There are lots of different file encodings for computers, partly because there was a need to encode different human languages and their special characters. Luckily, however, most conversion issues can now be avoided due to the Unicode standard, which accommodates most languages and special characters worldwide. Still, you may encounter other encoding standards, so watch out if you use (or if your data contains) special characters. To demonstrate what can go wrong if you choose the wrong file encoding, open the `un-secretaries.txt` file in the RStudio editor. You should see a list of UN Secretaries General, ordered by the year they served. Note that there are two special characters in this list: Dag Hammarskjöld's name contains an "ö" (an o-umlaut), and António Guterres's first name is spelled with an "ó" (an o-acute). This file is encoded in Unicode, which is the default for macOS and other current systems.

Before we start the conversion to a different encoding scheme, save the file under a different name with File 〉 Save As..., so that we do not overwrite the original version. Now, let's save the file in a different encoding. Go to File 〉 Save with Encoding, which brings up the dialogue box in Figure 4.3.

The current coding of the file is *UTF-8*, which refers to the *Unicode* standard. This is also the default for my current operating system (macOS) and therefore labeled as such in the list. Now, select *ASCII* and click OK. This will transform the file to the *American Standard Code for Information Interchange* (ASCII) standard, which is an old encoding standard developed in the USA to encode text in English (this is the file

https://doi.org/10.1017/9781108990424.007 Published online by Cambridge University Press

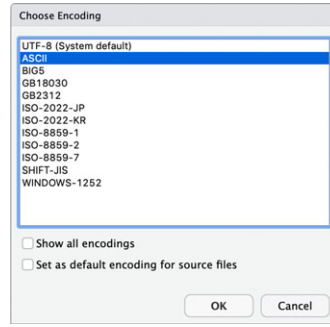


FIGURE 4.3. Choosing the file encoding in RStudio.

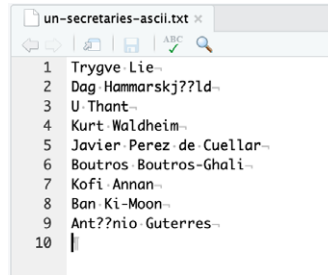


FIGURE 4.4. Viewing a file encoding in ASCII.

`un-secretaries-ascii.txt` included in the supplementary material for this chapter). If you now close the file and open it again, Figure 4.4 shows what you get.

This looks almost the same as the old file, but there are differences in two places: The special characters are missing. It is not difficult to understand why: Since the ASCII encoding does not include characters such as "ö" and "ó", they are simply replaced in the converted file (in our case, with two question marks). This example illustrates that if you leave the Unicode world and deal with text files in other encodings, you need to be careful, since special characters and symbols can be transformed in unexpected ways or simply disappear.

What if you encounter a file and are not sure about its encoding? Unfortunately, it is not straightforward to recognize the encoding. The `readr` package offers a function that guesses the encoding of a given file together with a confidence estimate. The following example demonstrates this:

https://doi.org/10.1017/9781108990424.007 Published online by Cambridge University Press

```
library(readr)
guess_encoding(file.path("ch04", "un-secretaries.txt"))$encoding
```

```
[1] "UTF-8"
```

```
guess_encoding(file.path("ch04", "un-secretaries-ascii.txt"))$encoding
```

```
[1] "ASCII"
```

The function detects the encoding of the original UN secretaries file correctly, and also for the ASCII version we created. This may be helpful if you encounter conversion errors during the import that could be due to encoding (such as garbled characters) – in this case, you can try to set the encoding manually (e.g., by using the `fileEncoding` parameter in R's `read.csv()` function) to fix these problems.

## 4.2 File Formats for Tabular Data

There is a variety of file types designed for storing tabular data. The discussion below takes you through the most important ones, many of which you will already be familiar with. In the remainder of the book, we will encounter several other file formats that can be used for tables, but also for other types of data.

The format of a file is typically indicated by the file extension, which is the dot and the letters at the end of a file name. For example, MS Excel files use the extension `.xlsx` (or `.xls` for the legacy Excel format), while simple text files are usually marked with `.txt`. It is important to note, however, that the file extension is no guarantee that the file actually conforms to a particular format. For example, you can easily rename an Excel document such that it ends in `.docx` (the file extension for Word documents). If you then double-click your file, your operating system calls Word to open it, since it believes that this is a Word file because of the file extension. Word, however, cannot open the file, since internally it uses the Excel format.

On some operating systems such as macOS and Windows, file extensions are hidden by default and you might be wondering what we are talking about. To show the file extensions on all files on your computer, follow these steps. On Windows:

- Open Windows Explorer
- Expand the Ribbon menu (Shortcut: Ctrl + F1)
- Click on the "View" tab
- Check the box that says "File name extensions"

https://doi.org/10.1017/9781108990424.007 Published online by Cambridge University Press

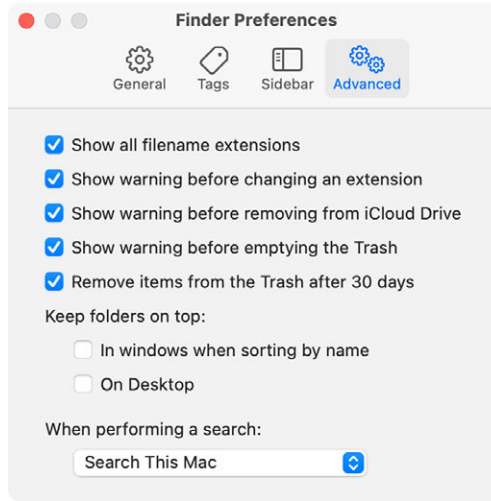


FIGURE 4.5. Check the first box to display all filename extensions on macOS.

On a Mac:

- Open the Finder app
- Click on Finder > Preferences
- Click on Advanced
- Check "Show all filename extensions" as shown in Figure 4.5

Linux usually does not use file extensions to determine how to open a file and just considers them part of the filename. You will not need to change any settings if your computer runs Linux.

In the following sections of this chapter, I will briefly introduce the most common file formats you are likely to encounter when working with social science data. For each of these formats, we cover some general features, as well as how to open and save it in R. The discussion starts with several *text* file formats typically used for storing tables. As we have seen above, the advantage of using text files is that you can manually check the content of a file. Also, almost any software tool for data analysis can read and write text-based files with tabular data. At the same time, however, there is no real standardization: This means that the file import can go wrong, and you need to check that the imported table actually corresponds to what is in the file and what you expect.

https://doi.org/10.1017/9781108990424.007 Published online by Cambridge University Press

### 4.2.1 CSV Files

The format most often used for storing tabular data in files is the *Comma-Separated Values (CSV)* format. In a CSV file, each line in the text represents a row from the table, and the cells in that line are separated by a special character such as a comma (hence the name). As in any proper tabular structure, each line must have the same number of cells for the table to be perfectly rectangular. When I use the term "CSV" in this book, I mean any kind of text file that stores tables in the same way (possibly using characters other than the comma as field separator). These files sometimes use the `.dat` file extension, but also others. Let us take a look at an example of a CSV dataset. Open the file `csv-example.csv` that is part of the data repository for this chapter in RStudio's text editor. The file contains distances between all national capitals worldwide, compiled by Gleditsch (2020). For simplicity, we use only a subset of it – the distances between Washington, DC and other countries' capitals. This is what you should see in the first three lines:

```
numa,ida,numb,idb,kmdist,midist
2,USA,20,CAN,738.31,460.56
2,USA,31,BHM,1639.23,1022.12
2,USA,40,CUB,1831.13,1141.3
```

The first line is the header of the table and contains the names of the six columns. Each distance is measured between the capital of one country, which has an identifier and a name (`numa` and `ida`), and a second country, also referenced with an identifier and a name (`numb` and `idb`). Finally, the distances are provided in kilometers (`kmdist`) and miles (`midist`). The data start in the second row. The cells are separated with a comma (the *field separator* character), and each line ends with an (invisible) newline character. Let us now import this file as an R data frame and take a look at the first three lines:

```
csv <- read.csv(file.path("ch04", "csv-example.csv"))
csv[1:3, ]
```

```
  numa ida numb idb  kmdist  midist
1    2 USA   20 CAN  738.31  460.56
2    2 USA   31 BHM 1639.23 1022.12
3    2 USA   40 CUB 1831.13 1141.30
```

There is no single standard for CSV files, which is why they come in many different forms. One issue you may encounter is that a `.csv` file

https://doi.org/10.1017/9781108990424.007 Published online by Cambridge University Press

uses a separator other than a comma as the separator. Open `csv-example-semicolon.csv` in RStudio's text editor, which contains a slightly modified version of the original data. This is what you should see:

```
numa;ida;numb;idb;kmdist;midist
2;USA; 20;CAN;  738,31;  460,56
2;USA; 31;BHM; 1639,23; 1022,12
2;USA; 40;CUB; 1831,13; 1141,30
```

In this file, the cells are separated by a semicolon instead of a comma, and the comma is used as a decimal indicator (which is the standard in many countries in Europe and elsewhere). This clearly illustrates the problems that can arise when using CSV and related file types in the absence of a fixed definition of a file format: is the comma or the semicolon used as field separator? We can clearly see this when eyeballing the file, but it is not straightforward for the software we use. So when you try to import `csv-example-semicolon.csv` in the way we showed you above, this does not work:

```
csv_semicolon <- read.csv(file.path("ch04", "csv-example-semicolon.csv"))
```

```
Error in read.table(file = file, header = header, sep = sep, quote = quote, : more
columns than column names
```

The problem is that R's `read.csv()` function by default assumes a comma as the separator. This results in a mismatch between the header of the file – which is treated as only one column name, since it does not contain a comma – and the actual data, which, when split at the comma characters, has *three* fields per row. To correctly import this dataset, you have to specify the separator explicitly:

```
csv_semicolon <- read.csv(file.path("ch04", "csv-example-semicolon.csv"),
   sep = ";")
csv_semicolon[1:3,]
```

```
  numa ida numb idb   kmdist   midist
1    2 USA   20 CAN   738,31   460,56
2    2 USA   31 BHM  1639,23  1022,12
3    2 USA   40 CUB  1831,13  1141,30
```

A similar issue arises when dealing with strings that contain the field separator. For example, we may want to add the name of the first country's capital, `Washington, DC`, to our capital distances data. This can lead to confusion when importing the dataset, since R (or other tools for that matter) will interpret the comma in the capital name as

https://doi.org/10.1017/9781108990424.007 Published online by Cambridge University Press

a field separator. For that reason, CSV files often enclose the affected strings with double-quotation marks ("), which basically means: "Treat the entire content between the quotes as a single string, regardless of what it contains." To see how this works, open the file `csv-example-quotes.csv` in RStudio's text editor, and take a look at the first line below the header:

```
2,USA,20,CAN,738.31,460.56,"Washington, DC"
```

The dataset now has a seventh column with the name of the capital of the first state. Since this name contains a comma, the entire string is enclosed in double quotes. But as you may have guessed, this again does not fully solve the issue. What if your string variable contains " characters that are *not* used for quotation? The standard way of dealing with this is to replace them with "" (two double-quotation marks next to each other). When you export files as CSV, the software usually takes care of string quotation. However, when you import CSV files, it might still be the case that quotes are not handled properly and errors occur, so you need to be careful and double-check that the import works correctly. If it does not, in many cases your only option is to open the file in a text editor, identify the source of the error, and fix it manually.

Saving CSV files from R is simple. R provides the `write.csv()` function for doing this, which is part of R's basic set of functions. By default, `write.csv()` uses comma as the field separator, and string quotation is enabled by default. This is all fine, but by default, the function produces a file that looks like this:

```
write.csv(csv, file.path("ch04", "output.csv"))
```

```
"","numa","ida","numb","idb","kmdist","midist"
"1",2,"USA",20,"CAN",738.31,460.56
"2",2,"USA",31,"BHM",1639.23,1022.12
"3",2,"USA",40,"CUB",1831.13,1141.3
```

First, note that the function by default quotes all strings, regardless of whether this is necessary. In our case, none of the field names or the data contain a comma, so we could actually omit the quotes in the header and in the string variables in the data. Second, R adds a new (unnamed) column to the data. This column contains the row numbers, which is what R uses to preserve the order of the data in the file. In practice, however, the ordering of rows in a data frame oftentimes does not (and should not) matter, which is why we recommend that you disable this feature:

https://doi.org/10.1017/9781108990424.007 Published online by Cambridge University Press

```
write.csv(csv, file.path("ch04", "output.csv"), row.names = F)
```

There are a number of other useful parameters for the `write.csv()` function, most importantly the `sep` parameter that lets you define a field separator to be used for the file. The `col.names` parameter allows you to disable the inclusion of a header should you wish to do so (although this is generally not recommended). In addition, there is one important feature of R read/write functions that is very useful when dealing with large files: You can use it to *compress* files. File compression ("zipping") is a technique where text files are saved such that they reduce the space they need on disk. For example, our (uncompressed) dataset of capital distances needs about 6 kilobytes of disk space. However, if we use R file compression, we can reduce the size considerably. This is done by using R's functionality to create "gzipped" files, a frequently used algorithm for compressing files:

```
write.csv(csv, gzfile("csv-example.csv.gz"), row.names = F)
```

The compressed file now only uses around 3 kilobytes, which is about 50% of the size of the original file. For larger files, the size reduction is usually much higher. Compression works particularly well if your data contain long sequences of repeated characters, which is typically the case for tables with a lot of text. Of course, R can also read the zipped CSV files again – there is no need for using additional functions, and you can simply provide the name when importing it, as in `read.csv("csv-example.csv.gz")`.

In this section, we only covered the basic features of CSV files, and the standard way to process them in R. There are various other packages and functions for this, some of which we will introduce in the next chapters. As regards the CSV format in general, it is important to emphasize again that it is only a *convention* for using text files to store tabular data rather than a fixed standard. While there actually exists a standard for CSV files (Shafranovich, 2005), it is not widely known and most tools (including Excel) do not conform to it, so you can safely ignore it. This means that you need to be aware of the potential pitfalls when using CSV files. We discussed the most common ones, which include the file encoding, the definition of the field separator (a comma, a semicolon or the invisible tab character `\t` are the typical choices), and the quotation of strings. Also, unlike in our examples above, CSV files sometimes do not contain the headers of the table in the first line, in which case you would have to set them manually in your code.

https://doi.org/10.1017/9781108990424.007 Published online by Cambridge University Press

Still, the CSV format has a number of advantages that explain why it is so widely used: It is an open format that is completely transparent, since you can open a CSV file on just about any computer system and inspect its contents. For that reason, the CSV format is compatible with most data processing and data analysis tools, and belongs to the most commonly used file types for data storage. One important downside is the lack of meta-information (such as column types or documentation information), so this must be provided in associated data such as codebooks or Readme files.

### 4.2.2 Excel

In the previous section, I described how to use text files to store tabular data. Now, we are turning to a number of binary data formats for the same purpose. However, as we discussed above, binary files are oftentimes designed to be used with a particular software and cannot be inspected manually with a simple text editor. A well-known example of this kind is the MS Excel file format, which, due to the popularity of the MS Excel spreadsheet software, is still a widely used format also for social science data. Excel files come in one of two formats: the legacy `.xls` format (which is a truly binary format), and the current `.xlsx` format that is actually a zipped collection of different text files, which together contain the data.

Different packages allow you to read and write Excel files in R. I recommend the `readxl` library for this, since it installs and runs without any additional configuration and is nicely integrated into the `tidyverse` environment that we cover in Chapter 7. As an example, let us use the data in the file `unsc-membership.xls`, which contains information on UN Security Council membership from Dreher et al. (2009). If you try to open this file with a regular text editor, you will see that it is a binary file, the contents of which are not human-readable. In R, we can open the file as follows:

```
library(readxl)
xls <- read_excel(file.path("ch04", "unsc-membership.xls"),
  sheet = 2,
  na = ".")
```

The R function that does all the work is `read_excel()`, and you need to specify the name of the input file (with a complete path if necessary), as well as the number or the name of the sheet you are importing. In our case, this is the second sheet, since the first one only contains metadata

https://doi.org/10.1017/9781108990424.007 Published online by Cambridge University Press

about the dataset. We also set the `na` option to `"."`, so that during the import process, fields that contain this character are interpreted as missing values. Writing data to an Excel file is equally simple with the `openxslx` package:

```
library(openxlsx)
write.xlsx(xls, file.path("ch04", "output.xlsx"))
```

According to my experience, however, importing data directly from a spreadsheet can be tricky. As we discuss in more detail in the next chapter, the problem is that spreadsheets do not impose a strict tabular structure, while almost all statistical software tools do. This means that, for example, numeric columns can contain text, or data can even be placed in the spreadsheet outside the area where the regular dataset is kept. This is why you often encounter problems and errors during the import process. As the next chapter will make clear, I generally recommend against using MS Excel (or any other spreadsheet software) for data management, if you can avoid it. However, since many datasets are still distributed in spreadsheet formats such Excel or LibreOffice/OpenOffice, it is hard to avoid them completely. This is why we spend an entire chapter on MS Excel (see Chapter 5), where we cover the various issues that can arise when managing research data with spreadsheet software.

### 4.2.3 Stata

Stata is one of the major tools for statistical analysis in economics and political science. It uses its own binary `.dta` format for data storage. Unlike Excel files, Stata's data files only contain a single table. This mirrors Stata's workflow well. It allows users to keep only a single table at a time in their working environment, which serves as input to all the analyses and visualizations the user carries out – this is very different from R, where you can have several data frames in your workspace. `.dta` files contain variable names and data, but optionally also short labels for the variables in the dataset.

Due to the fact that each Stata data file only contains a single, rectangular table, its import and use in R is typically much less problematic compared to spreadsheet files. As an example, we use the data on the targets of terrorism compiled by Polo (2020). The import function is provided by the `haven` package, and is straightforward to use:

```
library(haven)
dta <- read_dta(file.path("ch04", "terrorism-targets.dta"))
```

https://doi.org/10.1017/9781108990424.007 Published online by Cambridge University Press

When importing Stata files, the short variable labels are preserved and shown when you click on the data frame in the *Environment* tab of RStudio. Alternatively, you can display, set, or change the variable labels with the `labelled` package, which can be very helpful when inspecting a data frame for the first time:

```
library(labelled)
var_label(dta$attacksum)

[1] "Number of attacks"
```

Similar to reading Stata files, `haven` can also write R data frames in the `.dta` file format:

```
write_dta(dta, file.path("ch04", "terrorism.dta"))
```

Stata has introduced several versions of its file format over time. `haven` can read and write all versions that Stata has used so far, although you may have to set the `version` parameter manually (the version refers to the Stata version used to create the file). One important fact to keep in mind, however, is that before version 14, Stata did not use a fixed string encoding, which means that you can run into the encoding problems we discussed above. Since Stata 14, text is saved in UTF-8 format and can therefore contain characters in any language and a wide variety of other symbols. Stata as well as SPSS (below) also differ from R in how they handle labeled data and missing values. It is beyond the scope of this book to discuss these differences in detail, especially because the `haven` documentation is very detailed in explaining these issues.

### 4.2.4 SPSS

SPSS (now called *IBM SPSS Statistics*) is another commercial software package that is frequently used in the social sciences. Similar to Stata, SPSS also comes with its own file format for data files, identified by the `.sav` file extension. These files are also binary, which means that they cannot be opened with a text editor and inspected manually. `.sav` files also contain only one table or list, with variable names, the data and (optionally) labels and documentation for the data. As an example for a dataset in SPSS format, we use the 2012 – 2016 version of the Worlds of Journalism Study (WJS, 2019), which assesses the state of journalism around the world. We again import the data with the `haven` package and summarize the first three columns:

https://doi.org/10.1017/9781108990424.007 Published online by Cambridge University Press

```
library(haven)
sav <- read_sav(file.path("ch04", "journalism.sav"))
```

The original WJS data are based on interviews with journalists in different countries and cover topics such as editorial independence or how journalists see their role. The data we have here are national-level aggregates over all respondents. The WJS SPSS file also allows variables to be labeled, which we can use to find out what the variable names really mean:

```
library(labelled)
var_label(sav$C9)
```

```
[1] "Editorial autonomy: selecting stories (means)"
```

```
var_label(sav$C10)
```

```
[1] "Editorial autonomy: aspects emphasized (means)"
```

Similar to Stata files, the `haven` package can also write R data frames to SPSS files:

```
write_sav(sav, file.path("ch04", "wjs.sav"))
```

There are various other conventions and potential pitfalls when working with Stata and SPSS files. If you need more information on this, the documentation of the `haven` package is a good place to start.

### 4.2.5 R Data

R cannot only read and write files in other formats, but has its own file formats for preserving data. There is an important difference between the file formats I described above and R data files. While the above formats were all designed to store tabular data, R's data files can be used to store *any* kind of R object. So, for example, if you have a single vector, a list, or a data frame, they could all be permanently stored on disk using R's own file formats.

There are two types of R data formats: R data files, which have the extension `.RData` or `.rda`, and "serialized" R data files with the extension `.rds`. Both file types store the data in binary format (the default behavior). The difference is that `.rds` files save only a single object, and *without the name* the object was previously given. This means that when you load the object again from the file, you need to assign a new name. This is different for `.rda` files: A single file can contain *several* R objects, and will save each of them *with its name*. So when you load your data again, each

https://doi.org/10.1017/9781108990424.007 Published online by Cambridge University Press

object (data frame, list, vector, etc.) will be available in your workspace under the name it was given previously.

To demonstrate this, let us use the replication data for Barberá (2015), which analyzes Twitter behavior of world leaders. In the replication data you will find a file called `leaders-twitter.RData`. This file contains data derived from Twitter accounts of political actors in six countries (US, Spain, Netherlands, UK, Italy, and Germany). The data set also indicates the party to which each actor belongs (if applicable). In our example, we first clear all objects in our workspace with `rm()`, import the data with `load()` and then show the objects in the workspace with `ls()`:

```
rm(list = ls())
load(file.path("ch04", "leaders-twitter.Rdata"))
ls()
```

```
[1] "elites.data"
```

As you can see, you now have an object called `elites.data` in your environment, even though we did not specify a name. Rather, `elites.data` was created by the author of the dataset, and then saved to the file. What type of object is this? A data frame? Let us check:

```
class(elites.data)
```

```
[1] "list"
```

```
summary(elites.data)
```

```
        Length Class      Mode
US      23     data.frame list
UK       4     data.frame list
spain    4     data.frame list
NL       4     data.frame list
germany  4     data.frame list
italy    4     data.frame list
```

`elites.data` is a list with six entries, each of which is a data frame. This means that we have one table for each country, which can be accessed using the country name (e.g., `elites.data$germany`). For a simple demonstration of how to save `.RData` files, let's extract the data for Germany and Italy as two separate objects, and `save()` them:

```
elites_germany <- elites.data$germany
elites_italy <- elites.data$italy
save(elites_germany, elites_italy, file = "elites-germany-italy.rda")
```

Loading this file (after wiping our environment with `rm()`) makes the two objects available again:

https://doi.org/10.1017/9781108990424.007 Published online by Cambridge University Press

```
rm(list = ls())
load("elites-germany-italy.rda")
ls()
```

```
[1] "elites_germany" "elites_italy"
```

Finally, let us do a quick comparison with the `.rds` format. Remember that we can only save one object at a time; in our case, we use `elites_italy`. We save this object using `saveRDS()`, and load it again with `readRDS()` under a different name:

```
saveRDS(elites_italy, "elites-italy.rds")
italy <- readRDS("elites-italy.rds")
```

This example shows that the original (`elites_italy`) and the newly loaded (`italy`) datasets can exist in the same workspace, but with different names (since we can adjust this during the loading). A simple check reveals that they contain identical data:

```
identical(elites_italy, italy)
```

```
[1] TRUE
```

Due to their ability to store any kind of R objects, R data files are extremely flexible, as long as you do not want to exchange data with other software tools. Both data formats can only be processed with R, and users of other statistical packages will *not* be able to use your data. Using our above example, you are now able to import `.RData` and `.rds` files from other sources; however, you should think about whether distributing your own data in one of these formats is a good idea. In particular when dealing with tabular data, I rather recommend a text-based CSV format, which most statistical packages and programming languages are able to read.

## 4.3 Transparent and Efficient Use of Files

Over the course of your research projects you are likely to accumulate a large number of data files: data from different sources and data you create yourself, using different naming schemes and file types. While you are working on a project, it is often possible to keep track of what these files contain, where they come from, and what you need them for. But experience shows that once you take a break from a project, it can be difficult to make sense of the different files in your project. I therefore provide you with some simple guidance on how to effectively organize

https://doi.org/10.1017/9781108990424.007 Published online by Cambridge University Press

your research projects to minimize headaches and make your life easier. Many of these suggestions come from Jennifer Bryan's (2015) excellent talk on the matter, combined with my own experience.

### 4.3.1 Directory Structure

Good file organization starts at the directory structure of your project, that is, the folders in which your files are stored. In particular if your project involves many data files, I recommend that you create three subfolders in your project folder:

1. `/raw` contains all the raw data you collected yourself or that comes from other sources. You should consider this folder *read only*! This is uncleaned data that your R scripts should never change, only read.
2. `/analysis` This folder contains the output of all your data cleaning and processing, ready for analysis and structured however is best for your project. If you remember our recommended workflow from Chapter 1, this folder contains the analysis datasets. Importantly, you should consider the contents of this folder as *transitory*, and there should never be any data in this folder that cannot be recreated by running your scripts again. You should be able to delete *everything in this folder* and still arrive at the same data (and analysis results) after re-running your scripts that process the raw data.
3. `/replication` This folder should be populated at the end of your project with all the data necessary to replicate your results. It should contain only properly anonymized, cleaned data that is ready to be shared with others.

Usually, your R scripts will be located in the main working directory. To easily see what each R script does, consider using informative and consistent file names.

### 4.3.2 File Names

Once your directory structure is set up, you should also consider sticking to some conventions regarding the names of the files you use. While files in the `raw` folder should not be changed after you download them, it is up to you to give useful names to all the ones you create. Jennifer Bryan (2015) gives three principles for naming files that you should stick to, a

https://doi.org/10.1017/9781108990424.007 Published online by Cambridge University Press

recommendation I fully support: file names should be *machine readable*, *human readable* and *play well with the default ordering* of files on computers. What does this mean in practice?

1. *Machine-readable file names*: The great benefit of machine readable file names is that they make your life much easier when you process files automatically. Also, you can computationally extract information from the names that would be cumbersome to store and retrieve otherwise. To make files machine readable, avoid spaces, punctuation, and non-ASCII characters in your filenames and make sure you avoid case sensitivity (you should not have two different files called `myData.csv` and `mydata.csv`). Sticking to these rules makes using the default search function of your computer much more powerful, but also to retrieve and process the files using your script.
2. *Human readable file names* means that you should be able to tell from the name of a file what is in it. Giving your files names such as `01_clean-data.R` is vastly superior to just calling the file `01.R` or `data1.R`. By using delimiters such as the underscore and the hyphen consistently, you can also encode metadata about your file in the filename. For example, you can encode the order in which to run the files by starting with a numeral, and what the files do. Use underscores to separate these elements of metadata in your files and hyphens to separate words within the meta data.
3. *File names that sort well*: Starting your file names with a numeral allows for proper ordering when shown on your computer. You should always left-pad your numbers with a leading zero (otherwise on many systems, `10_analysis.R` will be sorted before `1_analysis.R`). When you use dates in your filenames, they should follow the ISO 8601 format YYYY-MM-DD and preferably be put at the beginning. This results in proper chronological ordering and prevents confusion from the different ordering of days and months in Europe and North America. While it is tempting to insert dates into filenames to denote different versions of a file over time (such as `20190312_data.R` and `20190313_data.R`), this oftentimes results in large, confusing numbers of files. If you want to preserve earlier versions of your code, consider using a version control system, which is particularly useful when collaborating with others. In Chapter 14, we cover these systems briefly.

https://doi.org/10.1017/9781108990424.007 Published online by Cambridge University Press

## 4.4 SUMMARY AND OUTLOOK

Most social science research data is contained and distributed in files, and there are many file formats that can be used for tabular data. In this chapter, I provided a general introduction to the most common file formats you are likely to encounter in your work. The most convenient and flexible way is to use simple text files for tabular data, as for example the CSV format. Reading and writing is possible with almost any software package, and we can check contents manually with a simple text editor. However, there is no established standardization for these files; important features such as the choice of the field separator or the inclusion of a header can vary, all of which requires some caution when working with CSV files.

There also exist a number of binary file formats for tabular data, most of which can be processed with R. Among the most frequently used ones are spreadsheet files, most importantly Excel. Stata and SPSS also have their proprietary data formats, designed to store individual research datasets along with some documentation (e.g., labels of variables and values). The `haven` package in R offers a lot of functionality to work with these files. R objects (which includes data frames) can also be stored as files in R's own formats. However, exchanging data with other software tools using these formats is impossible, which is why you should use these file formats only when you really need them (e.g., if your data does not follow a tabular format and therefore cannot be easily stored in a CSV).

There are lots of other file formats, many of which are also used for social science data. As a general overview of file formats, the comprehensive file format guide by the US Library of Congress (2019) may be helpful. In case you cannot open a file with the R libraries we used in this chapter, I recommend you take a look at the `rio` package, which is able to read a large number of file formats in the fastest and most efficient way possible. In sum, here is a list of recommendations based on the discussion in this chapter:

- *Understand the basics of file-based storage:* When working with data stored in files, it is important to understand how files work, irrespective of whether they contain research data or not. We discussed the important difference between text and binary files. For the former, you need to be aware of the fact that there are different encoding schemes for text, and choosing the wrong one can lead to strange characters and errors in your data. Luckily, Unicode has emerged as the standard on many operating systems, which means that conversion issues can largely be avoided, at least when working with more recent files.

https://doi.org/10.1017/9781108990424.007 Published online by Cambridge University Press

- *Familiarize yourself with different file formats:* There are few established conventions when it comes to storing tabular data in files. This means that for quantitative social scientists, there is a need to be familiar with different file formats as well as their strengths and weaknesses. Datasets for social science projects are distributed in many different formats, and it is likely that you will encounter a rarely used, legacy format in your work. Using the concepts and tools introduced in this chapter, you should be able to work even with the more difficult ones.
- *Organize your directories and files consistently:* To make the organization of your data and code as transparent as possible, try to stick to a consistent naming of files and folders. This is not only useful for others as they replicate your work, but it also helps you when you return to your project after some time. File and directory names should clearly indicate their content, and they should be constructed in a consistent way, such that they can be processed both by humans and computers.
- *For your data, choose a simple, well-known file format:* When you think about how to store your own data, it is advisable to prefer generic, software-independent file formats. For example, (correctly formatted) CSV files can be imported by almost any type of statistical software. Since they are text files, they also permit inspection by humans. It is generally recommended to avoid proprietary file formats such as Excel or SPSS. This also applies to R's custom file formats (`.RData` and `.rds`), which other software cannot process.

https://doi.org/10.1017/9781108990424.007 Published online by Cambridge University Press