3

Data = Content + Structure

Before we delve into the practical challenges of data processing, let us take a closer look at some core concepts we need throughout the book. The concept of scientific "data" is obviously of key importance. We need to clarify what we mean by it, and how information can be represented digitally as data. We will also learn to separate the data *content* – which refers to the actual information – from the logical *structure* in which this information is contained. Tables are by far the most frequently used data structure in the social sciences, which is why we spend a great deal of this chapter discussing the tabular data representation and its limits. We also review some basic functions of R: What are data frames, and how do we use them to store information? As discussed at the beginning, the book does not give a comprehensive introduction to R, but the examples below will help you refresh your memory.

3.1 WHAT IS DATA?

In our research, we use scientific data, which is systematically coded information about the real world.[1] Thus, we represent particular aspects of the real world by using *codes* so that this information can be stored as part of our dataset and later be processed by the researchers themselves, or by computers. In most cases, we will use numbers as codes, which represent, for example, the population of countries, or the vote counts of parties in

[1] Note that *data* is the plural of the Latin word *datum*. However, it is increasingly being used also as a singular word. Throughout this book, we follow the same convention. See also the blog post by Izzo (2012) about this.

an election. In other cases, we can use words as codes. For example, a list of political parties will likely include the party name, encoded as text in a particular language. Also, much simpler codes are possible, for example if we represent the presence or absence (0/1) of a particular feature (for instance, if a country is considered to be democratic or not).

Most scientific datasets are created to help us conduct comparisons between different entities – countries, precincts, experimental subjects, etc. This is why a dataset typically contains data about many different, yet comparable, entities. A social science dataset can be generated in different ways. In a survey, for example, we simply record the answers that subjects give to the specific survey questions. Here, the coding is predetermined by the way we design our survey and the questions we include. Other datasets are created by human coders, for example most of the cross-national datasets on political regimes or violent conflict. Yet another type of dataset requires little to no additional coding; for example, if we are interested in communication on social media, we can obtain a dataset of tweets directly from the Twitter platform. Here, again, information about each tweet will be encoded in a particular way, for example the date and time it was sent, or the name of the Twitter handle it was sent from.

The process of assigning codes to represent particular characteristics of real-world entities is sometimes simple (e.g., Twitter data comes with a precise time stamp readily assigned to each tweet), while it is much more difficult in other cases: For example, coding whether a country has a democratic system is difficult and subject to a major debate in political science. The challenges to coding and measurement in the social sciences are typically characterized by the requirements of *validity* and *reliability* that most readers will be familiar with; the former means that the coding or the measurement in a dataset should correspond to the theoretical concept we aim to capture, while the latter demands that the assignment of codes in our data be transparent, replicable, and uniformly applied across all the different entities we cover. These challenges arise at the data *collection* stage, which is why they are not discussed in this book. What matters for us is *how* data of particular types is represented and processed, but not *where* this information comes from.

3.2 DATA CONTENT AND STRUCTURE

In the previous section, we defined data as systematically coded information about the real world. For this data to be useful for scientific analysis,

we need to make sure that it is kept in a format that can be stored, shared, and analyzed. In other words, we need to find a good *representation* for it. Consider the following example:[2]

```
sdb <-
  "Switzerland is a country with 8.3 million inhabitants,
  and its capital is Bern. Another country is Austria;
  its capital is Vienna and the population is 8.7 million."
```

The simple object `sdb` is essentially a database; it contains information about two countries, their capitals, and their population. This information is what we call the *content* of the data. However, the information contained in this text may be obvious and easy to extract for humans, but it is much more difficult to process computationally. In other words, this data comes without a clear *structure*; unless we understand human language (which computers usually do not), we do not know what entities are referred to in the text, nor is it straightforward to locate the information about these entities. Now compare this example with the following method to set up a database, where we use the same content but with a given structure:

```
tdb <- data.frame(
  country = c("Switzerland", "Austria"),
  population = c(8.3, 8.7),
  capital = c("Bern", "Vienna"))
```

In this example, we use R's default data structure for tables, a data frame, to create our database in a structured way. For each country contained in our tabular database `tdb`, we have different types of information, clearly labeled as such. In a table, each line typically refers to an observation, while the columns contain the different variables we have for the observations. This structure makes the second dataset much easier to understand and process as compared to the simple database `sdb` above. In short, while the two examples are the same in terms of content, they differ significantly when it comes to their structure. Almost all data we use in our work comes in tabular formats, and all statistical toolkits are designed to process data in tables. Despite the omnipresence of tables, it is, however, important to understand that a table is just *one type* of data structure; it is one that is very convenient for social science applications, but also has its limits, as we will see later.

[2] Population estimates for the following examples were obtained from the United Nations Department of Economic and Social Affairs (2019) and rounded.

3.3 TABLES, TABLES, TABLES

Tables (or so-called rectangular datasets) are the main type of data structure in the social sciences. They have *rows* and *columns*. In social science terminology, each row represents a *case* or an *observation*, and each column a *variable* in our dataset. Let us take a look at how R deals with tables, using again the data frame we created above. There are several standard operations we can perform on a table.

3.3.1 Accessing Data

R gives us several easy ways to access the information in our table. For example, we can access a single value by using the row and the column index. For example, Switzerland's (row 1) population (column 2) can be retrieved with

```
tdb[1,2]
```

```
[1] 8.3
```

Alternatively, we can filter out the entire record for Switzerland by omitting the column identifier, as in

```
tdb[1, ]
```

```
      country population capital
1 Switzerland        8.3    Bern
```

In general, the square brackets notation is used in R for subsetting. Here, we apply it to data frames, but it can also be used for simple vectors, matrices of numbers, etc. Note the comma in the expression, which indicates that the given number is a row and not a column index. Selecting particular columns can also be done by providing their indices (here, the range from 1 to 2) as follows:

```
tdb[1:2]
```

```
      country population
1 Switzerland        8.3
2     Austria        8.7
```

or simply by providing the name of the column:

```
tdb$population
```

```
[1] 8.3 8.7
```

The `$` operator extracts a column from the table as a vector. It is important to mention that R automatically keeps track of the kind of information that is contained in the columns of a data frame. In other words, it maintains *types* for the columns. In our above example, some information in our dataset is coded as text, for example the capitals of the two countries. These short pieces of text are also referred to as *strings* in computer science. Other variables contain *numbers*, such as the country populations. Let us check the types that R has assigned to our dataset:

```
typeof(tdb$capital)
```

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

```
typeof(tdb$population)
```

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

As you can see, the names of the capitals are stored in a column of type "character," while the population estimates are of the type "double," which is the default type for numeric information. There are several other data types for vectors in R (such as "logical" values that can be either `TRUE` or `FALSE`, or the "integer" type used for storing integer numbers).

Oftentimes, we want to extract only a subset of the table that satisfies a particular filtering criterion. For example, we can extract the records for Switzerland (which, in our case, is only one) using:

```
tdb[tdb$country == "Switzerland", ]
```

```
      country population capital
1 Switzerland        8.3    Bern
```

Here, the `tdb$country == "Switzerland"` expression internally calculates a set of indices for those rows where the country column contains `Switzerland`. As above, we need to use the comma operator to tell R that the filtering condition we apply (the specification of a particular country name) applies to the rows of the table. If you think this expression is too complicated, there is also a simpler way to subset tables using the `subset()` function:

```
subset(tdb, country == "Switzerland")
```

```
      country population capital
1 Switzerland        8.3    Bern
```

3.3.2 Updating Data

Updating the information in a table is also straightforward. We can use the indexing notation again to update particular values in the table, for example, Switzerland's population:

```
tdb[1,2] <- 8.4
tdb

      country population capital
1 Switzerland        8.4    Bern
2     Austria        8.7  Vienna
```

This, however, is not convenient, since we have to refer to a column using the index and not the name. Instead, we can do the following:

```
tdb[1, "population"] <- 8.3
tdb

      country population capital
1 Switzerland        8.3    Bern
2     Austria        8.7  Vienna
```

This is still not optimal, since we need to know Switzerland's row index. To identify the rows for Switzerland, we can again use the statement we introduced above:

```
tdb[tdb$country == "Switzerland", "population"] <- 8.2
tdb

      country population capital
1 Switzerland        8.2    Bern
2     Austria        8.7  Vienna
```

3.3.3 Adding Data

Adding new data to a table can be done by either (i) inserting new rows or (ii) adding new columns. The latter can be done by simply assigning values to the new column:

```
tdb$area <- c(41, 83)
tdb

      country population capital area
1 Switzerland        8.2    Bern   41
2     Austria        8.7  Vienna   83
```

Inserting rows to our table is done using the `rbind()` function, which "binds" rows together. You can use it to combine two tables into one

(provided they have the same structure), but here we use it to add a single line:

```
tdb <- rbind(tdb, c("Liechtenstein", 0.038 , "Vaduz", 0.16))
tdb
```

```
        country population capital area
1   Switzerland        8.2    Bern   41
2       Austria        8.7  Vienna   83
3 Liechtenstein      0.038   Vaduz 0.16
```

Note that `rbind()` creates a new data frame from the inputs it gets. Therefore, we need to store the newly created table again in the original variable, which essentially deletes the old `tdb`.

3.3.4 Deleting Data

Finally, we also need to demonstrate how to remove data from our table. Again, there are two possible operation for deletions, namely, those affecting the columns and those affecting the rows of the table. Deleting columns is simple:

```
tdb$area <- NULL
tdb
```

```
        country population capital
1   Switzerland        8.2    Bern
2       Austria        8.7  Vienna
3 Liechtenstein      0.038   Vaduz
```

The deletion of rows from an R data frame may not be completely intuitive, as you need to create a subset of the rows you would like to keep, and overwrite the old data frame. This can be done, for example, using the `subset()` function we have described above:

```
tdb <- subset(tdb, country != "Liechtenstein")
tdb
```

```
      country population capital
1 Switzerland        8.2    Bern
2     Austria        8.7  Vienna
```

In this statement, we subset our data frame to those rows where the country column does *not* equal `Liechtenstein`, and store the result in the `tdb` variable.

3.4 THE STRUCTURE OF TABLES MATTERS

Before we start digging into actual data using different tools, let us spend some more time thinking about tables and their structure. While for many applications, it is entirely obvious what columns you need in your table, in some cases finding a good structure for your table is not as straightforward as it seems. This is why we will take a closer look at a few more toy examples, so as to better understand why and how the structure of tables matters. The recommendations here constitute the traditional way to organize data, which applies to most applications and projects we deal with in the social sciences.

3.4.1 Tables Should Grow Down, Not Sideways

A general rule of thumb you should observe when defining a tabular structure is that the columns – that is, the variables in the table – should be independent from the observations it eventually contains. That is, you need to select columns that capture all the important aspects of your data, regardless of how many cases/rows you later add to the table. A common mistake we oftentimes see is the use of case-specific information in the column *names* rather than in the individual cells of the table. This happens frequently in cross-sectional time series data, which is data about different entities (e.g., countries) that are observed at multiple time points (e.g., years). Consider our example from above, now revised to record the country population in different years:

```
bad_table <- data.frame(
  country = c("Switzerland", "Austria"),
  pop1950 = c(4.7, 6.9),
  pop1960 = c(5.3, 7.1),
  pop1970 = c(6.2, 7.5))
bad_table
```

```
      country pop1950 pop1960 pop1970
1 Switzerland     4.7     5.3     6.2
2     Austria     6.9     7.1     7.5
```

This format is called a "wide" table. The setup of the table may be convenient for human readers, but it causes many issues when processing the data computationally. It obviously violates our requirement that the variables we record in the dataset (which constitute the columns in the table) should be independent from the set of entities we record these characteristics for. In the above example, when adding population estimates

for more recent years, we would have to add more columns, rather than rows, to the table. This is not an issue in itself, but this structure is difficult to work with if we want to perform simple calculations on our table. For example, suppose we want to compute the average population across different observations in our table. This is easy to do by year:

```
mean(bad_table$pop1950)
```

```
[1] 5.8
```

```
mean(bad_table$pop1960)
```

```
[1] 6.2
```

```
mean(bad_table$pop1970)
```

```
[1] 6.85
```

However, what if we are interested in the average across all countries and years? With the table above, this is more difficult:

```
mean(c(
  mean(bad_table$pop1950),
  mean(bad_table$pop1960),
  mean(bad_table$pop1970)))
```

```
[1] 6.283333
```

This still looks acceptable, but now imagine that we are adding observations for more years to our dataset. This will make the table grow sideways, not down. If we compute the average population from the table, the statement becomes longer and longer. And, even more problematic, we need to update the calculation of the average population *every time we add a new year* to our table, which is not very convenient. How, then, is it possible to fix this? Can we design a better table structure for time series data? Consider this example:

```
good_table <- data.frame(
  country = c(rep("Switzerland", 3), rep("Austria", 3)),
  year = c(rep(c(1950, 1960, 1970), 2)),
  population = c(4.7, 5.3, 6.2, 6.9, 7.1, 7.5))
good_table
```

```
      country year population
1 Switzerland 1950        4.7
2 Switzerland 1960        5.3
3 Switzerland 1970        6.2
4     Austria 1950        6.9
5     Austria 1960        7.1
6     Austria 1970        7.5
```

This format is called a *long* table. The main difference between the `bad_table` and the `good_table` is obvious: Rather than using table columns for different years, we now introduce a new column `year` to link population values not just to the respective country, but also to the year they refer to. This makes working with our table much easier: Adding observations for more years is simple in this table structure; we can just append more rows to the table. Also, computing the average population over all observations is now a simple operation:

```
mean(good_table$population)
```

```
[1] 6.283333
```

You will never have to change this statement, regardless of how many observations and years you are adding to the data frame. Readers may now wonder how we get the annual average out of this table, which was easy in the `bad_table` above. For the `good_table`, we do this by letting R compute averages over *groups* of data, rather than the entire set of observations. This is called *aggregation*. One way to perform an aggregation in R is by using the `summaryBy()` function in the `doBy` package:

```
library(doBy)
summaryBy(population ~ year, data = good_table, FUN = mean)
```

```
  year population.mean
1 1950            5.80
2 1960            6.20
3 1970            6.85
```

In the statement above, we need to specify which variable we would like to aggregate over (`population`), and which variable(s) we would like to use for grouping (`year`). Also, we need to tell the function what the data frame is for the aggregation (`good_table`), as well as the summary function we would like to use (`mean`). The function then combines all observations with the same values in the grouping variable, and applies the summary function to each of these groups. This is exactly what we need, and it returns the annual averages from our dataset. So overall, the structure in our `good_table` seems to be much easier to handle, at least when we process our data computationally. You still see many examples similar to the `bad_table`, which may be due to the fact that they can be easier to understand for human readers. As we will see later, spreadsheets such as Excel are useful when humans interact manually with data, but not when we try to push the automation of data processing for maximum efficiency and transparency, which is our aim in this book.

Published online by Cambridge University Press

3.4.2 One or Multiple Tables?

The above example showed us that there are good and bad ways to structure individual tables. We now turn to the question of *how many* tables we need for a good representation of our data. Again, let us consider the `good_table`. Let us assume that, in addition to the yearly population estimates, we would like to store information about national capitals, like we did in the examples above. The simplest way to do this is to add the names of the capitals in a new column:

```
good_table2 <- good_table # create a copy to keep the original one
good_table2$capital <- c(rep("Bern", 3), rep("Vienna", 3))
good_table2
```

```
      country year population capital
1 Switzerland 1950        4.7    Bern
2 Switzerland 1960        5.3    Bern
3 Switzerland 1970        6.2    Bern
4     Austria 1950        6.9  Vienna
5     Austria 1960        7.1  Vienna
6     Austria 1970        7.5  Vienna
```

Since national capitals rarely change, the information in the capitals column is essentially constant over the years in our dataset, and we need to repeat it for every single year in the dataset. From a data representation point of view, this is clearly not optimal, as we have *redundant* information in our dataset. This makes data maintenance more difficult and error-prone. First, inserting the information in the first place is cumbersome, since we have to copy and paste the name of the capital of a given country for each year this country is listed in the dataset. This may be easy in our toy example, but quickly becomes infeasible when we deal with a much longer time series. Also, updating the data is equally difficult, for example, if we decide to refer to the capitals not in English, but in the respective national language (which would require us to replace `Vienna` with `Wien`). Redundant information also means that we can have inconsistencies in our data; for instance, if we forget to update all instances of `Vienna`, we may end up with a dataset that sometimes refers to the capital of Austria as `Wien`, while in other cases it uses the English name.

The problem of data redundancy always comes up if we store information about different entities that refer to each other in a single table. In our example, we have two types of entities: the countries (each of which has a capital), and the country-years (each of which has a population estimate). This data structure should better be stored in two tables that link to each other. For example, rather than adding a new column to `good_table`, we

could *add a new table* that contains only the information on the national capitals:

```
populations <- data.frame(
  country=c(rep("Switzerland", 3), rep("Austria", 3)),
  year=c(rep(c(1950, 1960, 1970), 2)),
  population=c(4.7, 5.3, 6.2, 6.9, 7.1, 7.5))
populations
```

```
      country year population
1 Switzerland 1950        4.7
2 Switzerland 1960        5.3
3 Switzerland 1970        6.2
4     Austria 1950        6.9
5     Austria 1960        7.1
6     Austria 1970        7.5
```

```
capitals <- data.frame(
  country=c("Switzerland", "Austria"),
  capital=c("Bern", "Vienna"))
capitals
```

```
      country capital
1 Switzerland    Bern
2     Austria  Vienna
```

As a result, our database now consists of two tables: a `capitals` table with country-level information (in our case, only the capitals) and a `populations` table with information at the country-year level (in our case, population estimates). In this setup, each piece of information is contained *only once* in the dataset; in other words, we have eliminated redundant data. This makes data maintenance extremely easy. For example, if we want to adjust the name of the Swiss capital, we do this in exactly one place:

```
capitals[capitals$country == "Switzerland", "capital"] <- "Berne"
capitals
```

```
      country capital
1 Switzerland   Berne
2     Austria  Vienna
```

The split of data into several tables is clearly something that may be desirable from a data management point of view, as it reduces (and, ideally, eliminates) redundancies in our data. At the same time, it is likely not a good way to interface with software for statistical analysis, most of which requires the data to be nicely arranged in a single rectangular table. What can we do about it? The solution to this is what we alluded to in

Chapter 1: the need to separate (i) data processing and management and (ii) data analysis into different stages of our workflow, potentially using different software tools supporting these stages. Recall that in Chapter 1, I recommended that you create "analysis datasets," which are tailored to the respective analysis and the software you use at the analysis stage. For our example, if we need information from the `populations` and the `capitals` tables in a single, rectangular format, we can simply merge the two tables:

```
merge(populations, capitals, by = "country")
```

```
      country year population capital
1     Austria 1950        6.9  Vienna
2     Austria 1960        7.1  Vienna
3     Austria 1970        7.5  Vienna
4 Switzerland 1950        4.7   Berne
5 Switzerland 1960        5.3   Berne
6 Switzerland 1970        6.2   Berne
```

Of course, we would only do this once we have finished the processing of our data, since we introduce redundancy in the merged dataset. Later in this book, we will deal with relational databases, which are designed to work with many tables at the same time, thus providing a suitable way to manage even complex datasets.

3.5 Summary and Outlook

In this chapter, our main focus was the distinction between the content and the structure of data. Data without structure (such as human speech, for example) can be difficult to process computationally, since it is difficult for computers to locate the important bits of information. In the social sciences, research data is usually collected and stored in tabular data structures. Tables are omnipresent, and they constitute the main way in which most statistical packages import and process data. In its simplest form, a tabular data structure is very easy to handle. It only requires us to specify

- A set of columns and their names (which correspond to the variables in our dataset)
- The types of each of these columns (a number, or a string of text)

We can then insert rows into the table, which represent the different observations in our dataset. Of course, these rows need to conform with the table definition, such that the columns contain the correct type of information. Note that while most software toolkits (such as R) keep

track of the type of information stored in the columns of a table, they cannot check for other sorts of errors. For example, if you record the age of respondents in a numeric column and mistakenly enter the value `200`, R will not complain. Therefore, it is up to you to identify semantic errors in your data and correct them.

Because of the importance of data structure, working with research data usually requires us to think about data content *and* structure. Before we can populate a dataset with information about survey responses, country-level indicators, or conflict events, we need to define what our dataset should look like, or, in other words, what its structure should be. This is usually referred to as *data definition*. Once we have a structure for our data, we can fill it with new information, update existing information, or delete parts of it. Together, these operations are referred to as *data manipulation*. Last, we use our dataset for scientific analyses, which is why eventually we need to output it in some way that is suitable for processing with other tools. This is called *data extraction*.

In this chapter, we also took a closer look at the structure of tables. In particular, I showed that it is beneficial to choose a table structure that lets your table grow down, not sideways, as you add more data. Also, I demonstrated that you may be better off splitting your data into separate tables, in particular if you deal with different types of entities. You may wonder why we spend so much time thinking about table structure, as this question is entirely straightforward to solve for many applications. This is true, but table structure matters a lot as soon as we deal with more complex scenarios. In particular, as soon as our observations vary along more than one dimension (e.g., countries and years), choosing a sub-optimal table structure can make your life difficult. By introducing some important considerations about tables and their design, we pave the way for later topics we cover in this book, in particular relational databases. In short, it pays off to think about the table structure before you start collecting your data. If you rely on existing data, you may benefit from transforming a given table to a more suitable design, such that you can optimize your research workflow down the road.

Now that we have completed a basic introduction using some toy examples, it is time to do some real work. In the next chapter, we will start with several tools that rely on *file-based* data storage. This means that your data is contained in files; you temporarily open these to process your data, and later the save the result again to a file. In later parts of the book, we discuss an alternative approach, where your data is stored in a database.