6

Basic Data Management in R

As we saw in the previous chapter, using spreadsheets to prepare data for analysis may be convenient at first, but entails a number of major drawbacks. In this chapter, we introduce the basics of data management using the R statistical toolkit. R is one of the most popular software tools for data analysis – it has numerous features and extension packages for statistics, visualization, machine learning, etc. Therefore, it is convenient to also use it to prepare your data *before* you actually analyze it. This way, you can stick to a single software package and one language to implement your entire research workflow from beginning to end.

As you know, you interact with R not by pointing and clicking with your mouse, but by entering commands in the R programming language. This way, you can have R run statistical analyses for you, visualize your data, but also perform data management operations. While cumbersome at first, this mode of interaction is extremely powerful and has a number of advantages. Most importantly, the set of commands you send to R (which is typically called a "script") can be saved, such that you can later return to it, fix potential problems, or simply replicate the steps you carried out to arrive at a particular result. This resolves one of the main drawbacks in the spreadsheet-based data management approach we discussed in the previous chapter, where it is difficult – if not impossible – to keep track of the different modifications you made to your data.

In this chapter, we focus on "base R," which is the set of commands and functions that are part of R's core functionality. We do this with a particular emphasis on R's features for data storage and processing, and how we can get data into R and back out. In the next chapter, I describe the `tidyverse`, an R extension that follows a different approach for data processing. While many consider the `tidyverse` as superior, I believe that it is still necessary to be familiar with R's basic functions and syntax for data management.

6.1 APPLICATION: INEQUALITY AND ECONOMIC PERFORMANCE IN THE US

Inequality remains a global issue of major concern (Piketty, 2014). In the practical example for this chapter, we focus on the historic development of inequality in the US, which former president Barack Obama considered to be a "defining challenge of our time" (The White House, 2013). How does inequality during Obama's presidency compare to other presidents? To what extent does inequality depend on the size of the US economy overall?

To find out, we use data from different sources. Data on inequality comes from the World Inequality Database (WID, 2020). The dataset contains time series for several measures related to inequality for many countries, and therefore allows for systematic, historical research into the determinants and consequences of inequality. In our example, we use one of the many indicators for income inequality: the share of the pre-tax income received by the top 10% of all individuals with the highest income in a country. Higher values of this measure indicate higher levels of inequality. The WID has a powerful web interface at https://wid.world/data/, where users can select the indicators, the countries and the years of observation they are interested in. The data file in the repository, however, was created using the bulk download function for the entire database, selecting the US and only the variable we are interested in. The resulting table was saved as a CSV file, which you can find in the data repository for this chapter in the file `us-inequality.csv`.

Data on US economic performance can be obtained from the FRED data portal of the US Federal Reserve Bank St. Louis (2020). The *real gross domestic product per capita series* was selected and downloaded in CSV format. The dataset is available in the data repository for this book in the file `us-gdp-pc.csv`. In addition, we combine the inequality estimates from the WID and the GDP data with data on US presidents, available online from the US Library of Congress (2020). For your convenience, the latter data is available in a shortened version in the file `us-presidents.csv` in the data repository.

6.2 LOADING THE DATA

We start by importing the three data files into R. Let us take a closer look at the inequality data from the WID first. If you open this file in RStudio's editor, you will see the following first three lines:

```
country,variable,percentile,year,value,age,pop
US,sptincj992,p90p100,1913,0.4231,992,j
US,sptincj992,p90p100,1914,0.4295,992,j
```

The structure of this file is straightforward; the first line contains the variable names, and the data start in the second row. A comma is used to separate the different fields in a row.[1] In our data, `variable` refers to the particular variable we are using from the WID, in our case the share (thus "s") of the pre-tax income ("ptinc"). `percentile` specifies the percentile range of the distribution we are looking at: `p90p100` is the range between the 90th and the 100th percentile, and thus corresponds to the top 10% of earners. We can use R's standard functions to read the data from the CSV file:

```
wid <- read.csv(file.path("ch06", "us-inequality.csv"))
```

A quick summary of the data shows that the import worked correctly:

```
summary(wid)

   country            variable           percentile             year     
 Length:100         Length:100         Length:100         Min.   :1913  
 Class :character   Class :character   Class :character   1st Qu.:1938  
 Mode  :character   Mode  :character   Mode  :character   Median :1963  
                                                          Mean   :1963  
                                                          3rd Qu.:1989  
                                                          Max.   :2014  
     value             age           pop           
 Min.   :0.3384   Min.   :992   Length:100        
 1st Qu.:0.3604   1st Qu.:992   Class :character  
 Median :0.4026   Median :992   Mode  :character  
 Mean   :0.4071   Mean   :992                     
 3rd Qu.:0.4536   3rd Qu.:992                     
 Max.   :0.4803   Max.   :992                     
```

One issue we should fix is the presence of unnecessary data in our data frame. Many of the variables such as `country` or `variable` are constant, and are only included because our data is a subset of the entire WID

[1] If you choose to download a custom-defined data file from the WID yourself, the import is not straightforward, since these files contain a header that does not conform to a regular CSV format.

(which contains many more countries and variables). We therefore retain only the data in columns 4 and 5 (`year` and `value`), and give them more meaningful names:

```
wid <- wid[4:5]
colnames(wid) <- c("year", "p90p100")
summary(wid)

      year         p90p100
 Min.   :1913   Min.   :0.3384
 1st Qu.:1938   1st Qu.:0.3604
 Median :1963   Median :0.4026
 Mean   :1963   Mean   :0.4071
 3rd Qu.:1989   3rd Qu.:0.4536
 Max.   :2014   Max.   :0.4803
```

Next, we need to import the GDP per capita estimates. The CSV data file is formatted according to standard conventions, using a comma as field separator and a header with the column names, which is why the import is straightforward. However, we again adjust the column names to something meaningful and change the type of the first column such that it properly reflects the dates:

```
gdp <- read.csv(file.path("ch06", "us-gdp-pc.csv"))
colnames(gdp) <- c("date", "gdppc")
gdp$date <- as.Date(gdp$date)
summary(gdp)

      date                gdppc
 Min.   :1947-01-01   Min.   :13999
 1st Qu.:1965-03-09   1st Qu.:21153
 Median :1983-05-16   Median :30482
 Mean   :1983-05-17   Mean   :33533
 3rd Qu.:2001-07-24   3rd Qu.:46691
 Max.   :2019-10-01   Max.   :58392
```

As you can see, the `gdp` data frame contains quarterly estimates of GDP per capita. We only need one estimate per year, which is why we retain only the observations for July:

```
gdp <- subset(gdp, as.numeric(format(date, "%m")) == 7)
```

Finally, let us import the dataset with the US presidents. This dataset was exported from a spreadsheet, which is why a semicolon is used as a field separator. This requires us to set the `sep` parameter of the `read.csv()` function accordingly. R again obtains the column names from the first line in the file. It replaces the whitespaces in the names with dots, since R does not accept column names with spaces. We again set the column names to lower case and retain only those columns we need later.

```
presidents <- read.csv(file.path("ch06", "us-presidents.csv"), sep = ";")
colnames(presidents) <- tolower(colnames(presidents))
presidents <- subset(presidents, select = c(inoffice, president))
summary(presidents)
```

```
   inoffice          president        
 Length:15          Length:15         
 Class :character   Class :character  
 Mode  :character   Mode  :character  
```

All three datasets – `wid`, `gdp`, and `presidents` – are data frames, which, as you know, is the main data structure for tables in R. In RStudio, you can view data frames just like spreadsheet tables using the `View()` command or by clicking on the data frame in the “Environment” tab in the top right panel. Note that, unlike in a spreadsheet program, you cannot manually edit the data – this would have to be done using R commands. As we have discussed above, there is no fixed standard for storing data in text files (CSV and similar formats). This is why you need to be careful when importing these data and make sure that the import was successful. Above, we checked some of our imported datasets simply by printing a summary. Another way to achieve this is the `str()` function:

```
str(wid)
```

```
'data.frame':	100 obs. of  2 variables:
 $ year   : int  1913 1914 1915 1916 1917 1918 1919 1920 1921 1922 ...
 $ p90p100: num  0.423 0.429 0.422 0.444 0.449 ...
```

In particular, this allows you to check:

- Whether all rows have been imported. If you load the CSV file in a text editor, you can easily count the rows in the original file. This file typically has one more row than the data frame in R (the header in the first line). Some CSV files have empty lines at the end; these can be excluded from the import using the `nrows` parameter in `read.csv()`, which restricts the number of rows to import.
- Whether all columns have been imported. Usually, inconsistencies in the number of cells between different lines will trigger errors and the file will not be read, but even if there are no error messages, it is still useful to check whether all columns were imported successfully and have the right names.

- Whether the columns have the right data type. The `str()` function shows us the type of data contained in the columns (e.g., `int` and `num` for the WID data frame). Since standard CSV files explicitly specify only the names – but not the types – of the table columns, the default behavior in R is to *infer* the variable types from the data it encounters in the respective columns. That is, if a column contains only numeric values and properly coded missing values – as is the case for the `p90p100` column in the `wid` data frame – R will correctly use a numeric type for it. If, however, we were to denote missing values with the string `n.a.` in the original CSV dataset, R would convert the entire column to a character variable, and you could not use it for any type of analysis that requires numeric input.

In general, if you encounter text files that deviate from common standards and cause issues during the import, I recommend that you try to address these problems using R code rather than fixing the data file manually. For example, you can skip a given number of lines at the beginning of a CSV file with `read.csv()`'s `skip` parameter, which is useful for some CSV files that have a header of more than a single line. Fixing import issues using R's functions rather than manually editing the files has a number of advantages. You could easily replace the old version of the data file with a newly downloaded one, for example, if a new version of the data has been released. Also, you avoid making undocumented modifications to a raw data file, which is something I recommended against at the beginning of the book.

6.3 MERGING TABLES

For comparing inequality and economic performance in the US over time, it is convenient to merge the two tables with each other. Both contain annual observations, so this is straightforward. However, before we can do this, we need to make sure that both tables have columns we can use to join them. The `wid` data frame already has a column containing the year of the observation; for the `gdp` data frame, we still need to create such a column by extracting the year from the `date` column:

```
gdp$year <- as.numeric(format(gdp$date, "%Y"))
```

We use again the `format()` function for this and convert the result to a number. Now, we are ready to merge the WID and GDP tables and store the result in a new data frame:

```
data_annual <- merge(wid, gdp, by = "year")
summary(data_annual)

      year         p90p100             date                 gdppc
 Min.   :1947   Min.   :0.3384   Min.   :1947-07-01   Min.   :14008
 1st Qu.:1964   1st Qu.:0.3549   1st Qu.:1964-12-30   1st Qu.:21088
 Median :1982   Median :0.3703   Median :1981-12-30   Median :30232
 Mean   :1981   Mean   :0.3872   Mean   :1981-06-30   Mean   :32314
 3rd Qu.:1998   3rd Qu.:0.4254   3rd Qu.:1998-03-31   3rd Qu.:43412
 Max.   :2014   Max.   :0.4714   Max.   :2014-07-01   Max.   :53452
```

What does the merge() function do? It takes two data frames and joins them line by line, for all lines that have the same values in the year column. This is why our resulting data frame will have all the columns from the first and the second data frame combined, as well as the column(s) used for merging. In our case, the merge column has the same name in both datasets, but merge() can also deal with merge columns of different names (you would use the by.x and by.y parameters instead of by). The function can also deal with applications where you merge not just on a single column, but on multiple ones (e.g., if you merge annual observations for different countries).

Now, take a closer look at the number of observations in the original and the merged datasets:

```
nrow(gdp)

[1] 73

nrow(data_annual)

[1] 66
```

The merged data frame contains fewer observations. The reasons is that our WID data do not start until 1962, while the GDP data are available from 1947 onwards. merge() retains only lines with at least one match in the other dataset, so we lose those observations from gdp that do not have a match in the WID. If we wanted to keep all observations from gdp, we could use the all.y = T parameter setting. However, in the merged table, the corresponding fields for the WID values would remain empty (NA).

As a final step, we need to merge the information about the US presidents to our data_annual dataset. However, the presidents data frame contains time periods, each with a start and an end year. This is why we cannot use it directly in the merge() function, because it requires a common attribute. Therefore, we need to make the time periods in the presidents data frame (which is currently a character variable) compatible with the year variable (which is a number) in the WID. A first step for doing this is to extract the start year and the end year of each president and to add them to the data frame. You probably noticed that the periods given in the data are overlapping; for every presidency, the first year is the same as the last year of the previous one. To avoid confusion in our dataset with annual observations, we therefore reduce the end year given in the data by one, such that we have exactly one president per year:

```
presidents$startyear <- as.numeric(substr(presidents$inoffice, 1, 4))
presidents$endyear <- as.numeric(substr(presidents$inoffice, 6, 9)) - 1
```

Since the information in startyear and endyear is now redundant in the table, we can remove the old variable:

```
presidents$inoffice <- NULL
```

We now need to merge the two data frames based on the corresponding years; so for each entry in the data_annual data frame, we need to look up the corresponding president based on the start and the end year of his tenure. This would be simple if we had a dataset with annual observations of US presidents. We do not have this, so we need to use a simple trick. We first create all possible combinations of rows from data_annual and from presidents. The merge() function does this if we set the all = T parameter:

```
data_annual <- merge(data_annual, presidents, all = T)
data_annual[1:5, ]

  year p90p100       date gdppc       president startyear endyear
1 1947  0.3708 1947-07-01 14008 Harry S. Truman      1945    1948
2 1948  0.3891 1948-07-01 14515 Harry S. Truman      1945    1948
3 1949  0.3836 1949-07-01 14182 Harry S. Truman      1945    1948
4 1950  0.3899 1950-07-01 15388 Harry S. Truman      1945    1948
5 1951  0.3771 1951-07-01 16223 Harry S. Truman      1945    1948
```

The result of this operation is called the *Cartesian product* of the two tables. Obviously, it yields many useless combinations. For example, the first line contains the inequality and GDP values for 1962, combined with the information on President Truman, who was in office during 1945–1949 and 1949–1953, which makes little sense. This result is not surprising, since we specify no condition whatsoever about which rows are supposed to match. However, in an additional step, we can now use simple filtering to get rid of the lines with non-matching information.

Specifically, we retain those lines where the current year (indicated by the year variable) is larger than the first year of the respective president's term (as specified in the startyear column), and smaller or equal to the last year (as contained in the endyear column):

```
data_annual <- subset(data_annual, year >= startyear & year <= endyear)
data_annual[15:20, ]

    year p90p100       date gdppc        president startyear endyear
213 1961  0.3583 1961-07-01 18319   John F. Kennedy      1961    1962
214 1962  0.3609 1962-07-01 19126   John F. Kennedy      1961    1962
281 1964  0.3698 1964-07-01 20567 Lyndon B. Johnson      1963    1964
348 1966  0.3629 1966-07-01 22650 Lyndon B. Johnson      1965    1968
349 1967  0.3529 1967-07-01 23020 Lyndon B. Johnson      1965    1968
350 1968  0.3551 1968-07-01 24009 Lyndon B. Johnson      1965    1968
```

This gives us exactly what we want: a table with GDP and WID information, combined with information about the US president in office during the respective year. The above approach for merging tables by creating the Cartesian product and then retaining only the matching lines is a recipe you should remember for later parts of this book.

6.4 AGGREGATING DATA FROM A TABLE

We now have a complete data frame with all the data we need for our simple analysis. Before we present the final result of our analysis, let us take a look at how we aggregate the data in different ways to show descriptive statistics for the different presidencies. As we have already discussed in Chapter 3, "aggregate" statistics are computed over *groups* of rows. In our example, we may be interested in the average level of inequality and GDP for each president's term(s). This can be done using the doBy package, where we specify the variables to be aggregated as well as the grouping variable(s), as follows:

```
library(doBy)
summaryBy(p90p100 + gdppc ~ president, data = data_annual)

            president p90p100.mean gdppc.mean
1        Barack Obama    0.4608167   51305.17
2        Bill Clinton    0.4172000   42081.50
3 Dwight D. Eisenhower   0.3586750   17377.50
4         George Bush    0.3893000   37397.00
5       George W. Bush    0.4431250   49458.38
6       Gerald R. Ford    0.3424667   26642.67
7      Harry S. Truman    0.3792667   15103.67
8        Jimmy Carter    0.3463500   29470.75
```

```
9        John F. Kennedy    0.3596000   18722.50
10     Lyndon B. Johnson    0.3601750   22561.50
11      Richard M. Nixon    0.3439000   25195.60
12         Ronald Reagan    0.3642375   32733.25
```

This simple aggregation groups the rows in our data frame by presidents. For each group, it applies the `mean()` function to each of the specified variables, `p90p100` and `gdppc`. Note the naming of the aggregated columns: The default behavior of the `summaryBy()` function is that it uses the name of the original variable and appends the name of the function applied to it. For example, the `p90p100.mean` variable contains the averages of the `p90p100` values for each president.

Computing averages is the default, but we can also specify other aggregation functions. For example, we can count the number of years that the respective president was in office. To do this, we simply add the `length()` function as an additional one to be applied to each group of rows:

```
summaryBy(p90p100 + gdppc ~ president,
  data = data_annual,
  FUN = c(length, mean)
)

              president p90p100.length gdppc.length p90p100.mean gdppc.mean
1          Barack Obama              6            6    0.4608167   51305.17
2          Bill Clinton              8            8    0.4172000   42081.50
3   Dwight D. Eisenhower             8            8    0.3586750   17377.50
4           George Bush              4            4    0.3893000   37397.00
5        George W. Bush              8            8    0.4431250   49458.38
6        Gerald R. Ford              3            3    0.3424667   26642.67
7       Harry S. Truman              6            6    0.3792667   15103.67
8          Jimmy Carter              4            4    0.3463500   29470.75
9       John F. Kennedy              2            2    0.3596000   18722.50
10    Lyndon B. Johnson              4            4    0.3601750   22561.50
11     Richard M. Nixon              5            5    0.3439000   25195.60
12        Ronald Reagan              8            8    0.3642375   32733.25
```

The `summaryBy()` function returns the result of the aggregation as a new data frame. This is useful if we want to continue working with this result; for example, we may want to order the entries in the aggregation table temporally by the time of each president's term. We can do this by adding the year as an aggregation variable, and the minimum as an aggregation function. This way, for each president, we obtain the first year this president shows up in our dataset, and can use this for ordering our aggregated data frame. You will see that by adding more variables and aggregation functions, the result of the aggregation becomes rather difficult to work with. This is because the `summaryBy()` function applies each of the aggregation functions to each of the aggregation variables, even though this is not what we want. This is why we simply drop the aggregated columns we do not need, to avoid confusion. In later chapters, I will present better ways for doing this.

```
data_term <- summaryBy(gdppc + p90p100 + year ~ president,
  data = data_annual,
  FUN = c(length, mean, min)
)
data_term <- subset(data_term,
  select = c(
    president,
    gdppc.mean,
    p90p100.mean,
    year.length,
    year.min
  )
)
data_term <- data_term[order(data_term$year.min), ]
print(data_term[1:3, ])
```

```
             president gdppc.mean p90p100.mean year.length year.min
7      Harry S. Truman   15103.67    0.3792667           6     1947
3 Dwight D. Eisenhower   17377.50    0.3586750           8     1953
9      John F. Kennedy   18722.50    0.3596000           2     1961
```

6.5 RESULTS: INEQUALITY AND ECONOMIC PERFORMANCE IN THE US

In the plot in Figure 6.1, we see the development of economic performance and inequality by presidency. Overall, economic performance has been steadily increasing over time in the US, and there are no particular differences observable by presidency. At the same time, inequality does not seem to be tracking this trend closely, until we get to Jimmy Carter's presidency in the late 1970s. This time is seen as the beginning of the American deindustrialization, where inequality and poverty rose due to the increased off-shoring of jobs primarily in the manufacturing sector (Strong, 2021). Since then, inequality in the US has been increasing steadily, until it reached a level that is about 50% higher as compared to the first time periods in our sample. During Barack Obama's presidency, according to our statistics, almost half of the pre-tax national income went to the top 10% of earners.

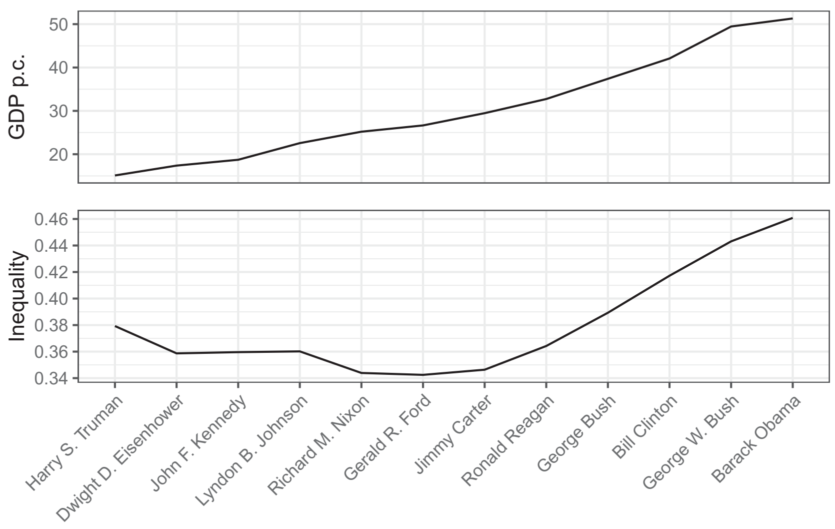

FIGURE 6.1. US GDP per capita (in 1,000 USD) and inequality by presidency.

6.6 SUMMARY AND OUTLOOK

In this chapter, we processed data from three sources to analyze trends in inequality and economic performance in the US across different presidencies. We did this using R's core functionality (with one exception: the `doBy` package). In particular, we imported data from text files such that they are available as data frames in R. This requires some caution, since data in text files may not be formatted according to standard conventions, and import errors can occur. Also, many standard text file formats do not explicitly specify the type of variables contained in a table, which is why R can only infer them (and this can go wrong). We merged our three datasets using R's `merge()` function, but also encountered the limitations of this process when dealing with more complex merges. The process of first creating the Cartesian product of the two tables, and then retaining the desired combinations is one way to bypass these limitations. Finally, we aggregated the data in R, applying a set of aggregation functions over groups of data.

Data processing using CSV files, data frames, and the functions I have presented so far is the standard workflow in R, and something you need to be familiar with. What we covered in this chapter is already a great improvement beyond a spreadsheet-based workflow: In R, you specify all your data operations in code. This way, you can correct, amend, and re-run this code, but also share it with others. Still, R's basic data processing features do not necessarily constitute the optimal and most intuitive way to handle data in R. We therefore discuss an easier and, in several ways, better way to process your data in the next chapter: the `tidyverse`. Still, there are several lessons you should remember from this chapter:

- *Knowing base R is important:* Even though there are now several extensions of R's core data wrangling features (the `tidyverse` being the most prominent one), you still need to know your way around base R. Many important packages are not compatible with the `tidyverse`, and you often will have to work with R's core data structures.
- *Data frames as R's main tabular data structure:* For us as social scientists who mainly work with tables, it is essential to know the features and pitfalls of data frames. The syntax to extract rows or columns may often seem strange, but it corresponds to R's vector-based programming approach. As we saw in the chapter, R does maintain types for the columns in a data frame, but they can change dynamically as you add new data. This is something to watch out for, and it can make explicit type conversions (casts) necessary.
- *Make sure that imports work correctly:* Due to the implicit type conversions that can occur in data frames, it is necessary to check imported data carefully. Most text-based data files such as CSV do not preserve the column types of your data, which is why they must be inferred (or explicitly specified) during the import.
- *A few simple packages add standard data manipulation features:* Base R can do most basic operations on tabular data, but for some tasks, it is necessary to rely on external packages. In this chapter, we used the `doBy` package, which is one way to run basic aggregation operations on tabular data.