7

R and the tidyverse

In the previous chapter, we did some basic data processing with base R. Is this not enough to solve most of our tasks? Sure, but we can do better. Working with data in base R is oftentimes limited. Handling data frames can be difficult, and additional functions for data management must sometimes be added by importing external packages (e.g., the aggregation functions in the doBy package). Here, a set of packages, together referred to as the tidyverse, provides a much better and fully integrated way to work with data. It reflects our basic understanding of data handling very closely and also relies primarily on tables as the main data structure (variables as columns, rows as observations). The syntax, however, is much easier to remember, since the tidyverse uses natural names wherever possible and relies on verbs for actual operations to be carried out. Overall, this means that your code becomes more readable, not just for yourself but also for others trying to replicate it.

This chapter walks you through a simple application that demonstrates the use of the tidyverse's data management features. Rather than a single R library, the tidyverse is actually a collection of several R packages for different purposes, which, however, use a common underlying logic and syntax. You may be familiar with the ggplot2 package for producing graphics, but there are also several other extremely powerful packages that are part of the tidyverse. You can learn more about the entire tidyverse suite of packages at https://www.tidyverse.org.

All tidyverse packages are carefully designed, provide a wealth of useful features and are therefore highly recommended. In keeping with our focus on data management, however, we will only focus

https://doi.org/10.1017/9781108990424.010 Published online by Cambridge University Press

on two of them. `tidyr` provides basic functionality to store data in rectangular (tabular) data structures, and `dplyr` offers powerful functions to manipulate data. To make these (and other) packages available in R, you need to install the entire `tidyverse` as described in Chapter 2 (unless you are using the pre-configured project environment) and then load it with:

```r
library(tidyverse)
```

7.1 APPLICATION: GLOBAL PATTERNS OF INEQUALITY ACROSS REGIME TYPES

In the example for this chapter, we continue to explore the political determinants of inequality, but expand the scope of the analysis. While the previous chapter focused exclusively on the US, we now adopt a comparative perspective and study a global sample of countries over several decades. In particular, existing scholarship has suggested that inequality and regime type may be closely related (Acemoglu and Robinson, 2005; Houle, 2009). Our aim here is to create a dataset for analysis that allows us to track how patterns of inequality have developed over time in different political regimes.

Our main data source on inequality is again the World Inequality Database (WID, 2020), from which we obtain a cross-sectional time series dataset of income inequality estimates for many countries. We again rely on the full dataset, downloaded as a set of CSV files and merged into a single file for the exercises in this chapter. Again, we use the share of pre-tax income that goes to the richest 10% as our indicator for inequality (`p90p100`), which is available for many years and countries. The resulting data is available in the file `inequality.csv` in the data repository for this chapter.

In addition to the inequality data from the WID, we require data on the type of political regime that exists in a given country. Scholarship in political science has made different attempts to measure regime type along the dimension of autocracy vs. democracy. Our example in this chapter relies on the well-known Polity IV project (Marshall et al., 2015), which codes political regimes along a continuous dimension from -10 (full autocracy) to 10 (full democracy). Since political regimes change over time (e.g., by becoming more democratic or more autocratic), the Polity scores are provided as annual observations at the country level.

7.2 A NEW OPERATOR: THE PIPE

In data management, we often have to apply a series of operations to our data. We add and recode variables and filter selected cases, and merge our data set with others. The standard way of doing this in R is to apply a series of functions, creating intermediate datasets that are used as input at later stages of the process. Consider the following example of two artificial datasets with 20 annual observations, each of which contains a single additional variable (randomly assigned for simplicity):

```r
dataset1 <- data.frame(year = 2000:2019, var1 = runif(20))
dataset2 <- data.frame(year = 2000:2019, var2 = runif(20))
```

Let us assume we want to subset `dataset1` to observations that occurred after the year 2007 and merge it with `dataset2`. This is how to do this in base R:

```r
dataset1_subset <- subset(dataset1, year > 2007)
final_dataset <- merge(dataset1_subset, dataset2)
```

When we add more operations on our data, each of them generates a new intermediate result such as `dataset1_subset` and adds a new line of code. The `tidyverse` introduces a new operator that facilitates this process: The pipe `%>%` allows you to write your code in a more natural fashion, from left-to-right. What does this mean? In the following example, we again subset and merge the two datasets, but in a single line of code, and without intermediate results. Here, I demonstrate the use of the pipe with the same base R functions we used above – later, we will replace them with the appropriate ones from the `tidyverse`:

```r
final_dataset <- dataset1 %>% subset(year > 2007) %>% merge(dataset2)
```

The idea of the pipe is straightforward: It takes a given dataset and sticks it into a new operation. As you can see in the example, we can chain several pipe operations and specify a complete "pipeline" in a single line of code. This code essentially says: "take `dataset1`, filter it such that it only contains the years after 2007, and merge the result with `dataset2`." This code is easier to read, expresses the aim behind it more clearly, and the flow of the data is much more apparent. We also have to type less boilerplate code such as `subset(dataset1,...)` or `merge(dataset1_subset,...)`, because we are passing data directly from one operation to the next.

You may have noticed that when using the pipe operator, the input it sends to the next function becomes the first argument of that function;

for example, rather than writing subset(dataset1, year > 2007), we can simply say subset(year > 2007) and the first input to the subset() function – the data that should be filtered – will be provided by the pipe. In the tidyverse, all functions are designed for this intuitive use of pipes, while many functions from outside the tidyverse are incompatible. So, the pipe is not a generic new operator in R; rather, it works only with the functions designed for it. If you would like to learn more, I recommend the chapter on pipes in Wickham and Grolemund (2016) and the documentation of the magrittr package.

7.3 LOADING THE DATA

As always, we need to load our data into R before we can start processing it. When working with the tidyverse, we use the read_csv() function for this. This is a new implementation of R's basic import function read.csv(), and you can use it in a similar way:

```
wid <- read_csv(file.path("ch07", "inequality.csv"), na = "")
```

The read_csv() function assumes that the fields in a row are separated with a comma – similar to the base R functions, you can use read_delim() for files with a different separator, which allows you to manually specify the field separator. For the WID dataset, it is important to specify the empty string ("") to indicate NA values, otherwise the function would interpret Namibia's two-letter code "NA" as NA. Let us now drop again the unnecessary columns in our data, such that we retain only the ones we need – the country identifier (a two-letter ISO country code), the year, and the value of the inequality indicator for the respective country and year:

```
wid <- wid %>% select(country, year, value)
```

Here you can see the pipe %>% in action: We take the original wid dataset and pass it to the select() function, which we ask to retain three columns. We store the result in wid, overwriting its original content. What is the result of this import? Let us take a closer look at the wid object by simply printing the first three entries. We do so again using the pipe operator, but stick the wid into a different function: slice(), which is used for subsetting datasets according to a row range provided:

```
wid %>% slice(1:3)

# A tibble: 3 x 3
  country  year value
  <chr>   <dbl> <dbl>
1 AE       1990 0.593
2 AE       1991 0.595
3 AE       1992 0.597
```

When we print the dataset, we see that we are not dealing with a conventional data frame. Rather, the `read_csv()` creates something similar, called a "tibble." A tibble is a modern version of a data frame. Tibbles serve the same purpose (which is to store tabular data), but with several tweaks that streamline and improve their use in practical applications. They have a much nicer default `print()` method (which is invoked when simply typing the name of the tibble): It reports the overall dimensions of the table, the names of the columns, and their types. More about tibbles can be found in the documentation of the `tibble` package.

Before we proceed, let us explore a bit more on how to work with tibbles in practice. Tibbles support all the basic operations you can do with data frames. For example, you can rename columns (which in fact is done much more elegantly as compared to data frames in base R):

```
wid <- wid %>% rename(p90p100 = value)
```

Apart from a nicer way to print, tibbles come with very useful features that make working with data much easier. In certain cases, however, you may have to explicitly convert a tibble to a proper data frame; this can be done using `as.data.frame()`. While you can use the `[]` and `$` syntax for extracting data from tibbles in a similar way as for standard data frames, I strongly recommend that you use the corresponding functions provided by the `tidyverse` for this if possible. They are designed to work with the pipe operator and improve readability of the code. We have already used the `select()` and the `slice()` functions, as well as the `filter()` function to extract rows based on a search condition. It is important to emphasize that these functions always return tibbles; this reduces confusion in comparison to the corresponding functions for data frames, which sometimes return a vector rather than a data frame (e.g., the `$` operator).

In addition to the data from the WID, we also require data on political regimes from the Polity IV project. This data is distributed both in Excel and SPSS format, and we choose the former. Since the Excel file contains a properly formatted data table, the import does not cause any issues. We use the `readxl` package, which is also part of the `tidyverse` and hence creates a tibble:

```
library(readxl)
polity <- read_excel(file.path("ch07", "polity.xls"))
```

Since the Polity IV database contains many variables we do not need for our analysis, we only keep the main Polity indicator (`polity2`) in addition to the country (`ccode`) and year (`year`) identifiers:

```
polity <- polity %>% select(ccode, year, polity2)
```

We now have all the necessary data in two tibbles, `wid` and `polity`, which we use to generate a single dataset for our analysis.

7.4 MERGING THE WID AND POLITY IV DATASETS

Our next task is to merge the `wid` and `polity` datasets. Both contain annual observations at the country level, but merging them is complicated by the fact that there is no common country identifier yet. The WID refers to countries with a two-letter code, while the Polity database includes *Correlates of War* (COW) country codes, a system widely used in international relations and conflict research. Hence, we need to match the two-letter country codes in the WID to the COW codes. This task is greatly facilitated by the excellent `countrycode` library for R, which can translate between different codes and names for states. We can use the main translation function `countrycode()` from this package, which needs to know in which column the country identifier is stored that we want to translate (in our case, this is the `country` column). Also, it requires us to specify the coding system from which we want to translate (the ISO two-letter country code used in the WID, "iso2c"), and what coding system we want as output ("cown" is used to denote the numeric COW coding system). To store the result of the translation in a new column named `ccode`, we use the `mutate()` function:

```
library(countrycode)
wid <- wid %>% mutate(ccode = countrycode(country, "iso2c", "cown"))
```

Note that we get a warning from the `countrycode` function that two countries could not be merged. One of them is Palestine (two-letter code PS), which is not contained in the COW list of independent states. The second one is Serbia, where `countrycode` uses the old two-letter code for

Yugoslavia and therefore does not produce the correct COW code (345). We can fix the second issue by manually inserting the correct COW code for Serbia, using again the `mutate()` function to change the `ccode` variable:

```r
wid <- wid %>% mutate(ccode = if_else(country == "RS", 345, ccode))
```

The `if_else()` function in the statement has three parts. It basically says: If the two-letter code is `RS`, use code `345` as the new value for `ccode`, otherwise use the existing `ccode` value as the new one. With a common country identifier, it is now straightforward to merge the two datasets based on `ccode` and `year`. Functions for merging in the `tidyverse` are called *join* functions, which is the technical term for combining tables in relational databases (we will learn more about joins in the next chapters). You may recall from the previous chapter that the default mechanism for joining datasets is to keep only those observations that have at least one match in the other dataset. This is called an *inner* join. Let us first try to use this function for merging Polity and the WID based on the `ccode` and `year` variables:

```r
dataset <- polity %>% inner_join(wid, by = c("ccode", "year"))
```

The entire `polity` table has 17,562 observations, while the merged `dataset` has only 3,142. This is due to the fact that the WID only covers a subset of countries – if we now retain only observations from Polity with a match in the WID, all the countries that are contained in Polity but not in the WID are removed from the merged dataset. This is the standard behavior of all inner joins. If you need to retain all records from the first (the left) or the second (the right) dataset – similar to the `all.x` and `all.y` parameters of the `merge()` function in Chapter 6 – you could use a "left" or a "right" join, which can be executed with the `left_join()` and the `right_join()` function.

7.5 GROUPING AND AGGREGATION

The WID only covers a subset of all countries worldwide, and even for these, the inequality measure (`p90p100`) contains many missing values. We should first get a better overview of our dataset as regards the countries and time periods it covers, but also the countries/years for which we have valid observations from the WID. To generate some useful statistics to answer these questions, we use grouping and aggregation. Recall from Chapter 3 that data aggregation is the definition of different groups or subsets of data, with an aggregation function applied to each of these groups separately. As we have seen in the previous chapter, in a conventional data frame these groups can be dynamically defined in the summaryBy() function.

In a tibble, however, this mechanism is slightly different. Tibbles allow you to define the grouping as a *feature of the tibble*, which is then used whenever grouping functions are applied to it. Grouping is enabled with the group_by() function:

```
dataset <- dataset %>% group_by(country)
dataset

# A tibble: 3,142 x 5
# Groups:   country [107]
  ccode  year polity2 country p90p100
  <dbl> <dbl>   <dbl> <chr>     <dbl>
1     2  1913      10 US        0.423
2     2  1914      10 US        0.430
3     2  1915      10 US        0.422
# ... with 3,139 more rows
```

You can see in the output that the tibble now has the grouping by country enabled, and that there are 107 different groups (countries). We can now summarize the tibble, which will automatically be done separately for each of the groups. To see how many observations we have per country, we use the aggregation function n() that counts the number of cases in each group:

```
dataset %>% summarize(count_obs = n())

# A tibble: 107 x 2
  country count_obs
  <chr>       <int>
1 AE             27
2 AL             17
3 AO             28
# ... with 104 more rows
```

The output of this function creates a new tibble containing the summary statistics we computed. While we have 100 or more years' worth of data for countries such as France and the US, for many others the coverage is much more limited. For our analyses below, it would be useful to know since what year particular countries are covered in the WID, such that we can adjust our period of analysis accordingly. Therefore, we expand our summary such that it outputs the first year with an inequality estimate during the observation period. To do that, we use the minimum as an aggregation function:

```
dataset %>% summarize(firstyear = min(year))
```

```
# A tibble: 107 x 2
  country firstyear
  <chr>       <dbl>
1 AE           1990
2 AL           1996
3 AO           1990
# ... with 104 more rows
```

The example shows that data aggregation in the `tidyverse` is very elegant, and is a considerable improvement over the mechanism we used in the previous chapter. One of the main advantages is that we can define aggregation functions only for particular columns they should be applied to, and have full control over the naming of the columns holding the aggregated values. As you can see in the output, for many countries, there are few inequality estimates for years earlier than 1990, which is why we restrict our analysis below to the years 1990 and later. Before we do that, however, we disable grouping of our main dataset with `ungroup()`, since the next operations on the dataset do not need grouping:

```
dataset <- dataset %>% ungroup() %>% filter(year >= 1990)
```

To track patterns of inequality by regime type, we need a dataset with average annual values of inequality, computed separately for democracies and autocracies. As a first step, let us introduce a new binary variable `democracy`, which identifies those countries that are democracies in a given year. Following the standard convention, we code country-years with `polity2 >= 6` as democracies. Since we have missing values in the `polity2` variable, we drop these observations before computing the aggregation:

```
dataset <- dataset %>%
  filter(!is.na(polity2)) %>%
  mutate(democracy = if_else(polity2 >= 6, T, F))
```

Since we need average inequality values for each year and separately for democracies and autocracies, we need two levels of grouping. We therefore enable grouping again with:

```
dataset <- dataset %>% group_by(year, democracy)
```

Now, we can summarize our data as introduced above and compute the average level of inequality, separately for the democracies/autocracies and each year in our sample:

```
data_agg <- dataset %>%
  summarize(mean_ineq = mean(p90p100))
```

The result of the aggregation is stored in a new dataset, `data_agg`, which we use in the next section.

7.6 RESULTS: GLOBAL PATTERNS OF INEQUALITY ACROSS REGIME TYPES

Figure 7.1 plots the aggregated values for democracies and autocracies over time. Keep in mind that the WID does not cover all countries worldwide, so this result must be treated with some caution. The plot shows there are notable differences in the level of inequality between democratic and autocratic countries. In democracies, around 40% of the income go to the richest 10% of the population, which is a large share. In autocracies, however, this share is even higher, with values of more than 50%. So clearly, democracies seem to be doing better than autocracies in creating a more equal society. However, the figure also shows that in democracies, the level of inequality is increasing over time, while it is slightly decreasing in democratic countries. We should mention though that by simply averaging over all countries, our simple comparison hides much variation *within* each of the two categories. In particular, there are considerable differences among democratic countries when it comes to inequality in the population.

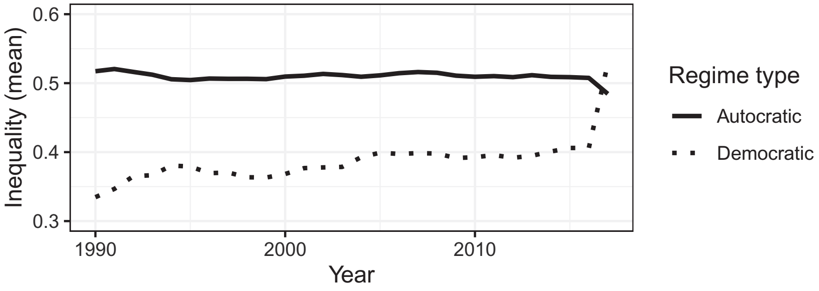

FIGURE 7.1. Trends in inequality over time, for democracies and autocracies.

7.7 OTHER USEFUL FUNCTIONS IN THE TIDYVERSE

Before we conclude this chapter, let us briefly review some other functions that can be really helpful to quantitative work in the social sciences.

7.7.1 Lags of Variables

The first of the functions help us create a lag of a variable (e.g., the value of that variable in the previous time period). Continuing with our example above, we may want to do a simple regression analysis of how democracy affects inequality. For analyses of this type, it is common to lag the main independent variable – in other words, this means that we predict the level of inequality with the *previous year's* democracy score for the respective country. For this, we need to extend our dataset such that in addition to the contemporary democracy scores in the `polity2` variable, we also have a new variable with the democracy scores from the previous year.

To create this variable, we use the `mutate()` function that you already know from above. In this function, we use the `lag()` function applied to the `polity2` variable, which is the variable we want to lag. This function also needs to know which variable specifies the temporal order of the data, in our case the `year`. Since we want to compute the lags separately for each country, we `group()` our dataset first, and `ungroup()` it after the operation is complete:

```
dataset <- dataset %>%
  group_by(ccode) %>%
  mutate(polity2_lag = lag(polity2, order_by = year)) %>%
  ungroup()
```

With the lagged predictor `polity2_lag` now being a new variable in our dataset, we can, for example, run a simple linear regression to test again our above result that democracies tend to have lower values of inequality.

7.7.2 Converting between Wide and Long Tables

The `tidyverse` also contains functions to convert between "long" and "wide" tables. What was this again? Recall our discussion in Chapter 3, where we talked about the features of a well-designed table. Good tables are those where you can add data by adding more rows to the table. Our `dataset` above is such a table: If additional data about more countries and/or years became available, we could just add a new row for each country-year we want to insert. This type of table is also called a "long"

https://doi.org/10.1017/9781108990424.010 Published online by Cambridge University Press

table. However, some of the data we use in social science projects comes in poorly designed tables. For example, for country-level data with annual observations, you sometimes encounter tables with one row per country, and a column for each year covered in the dataset. This format is called a "wide" table.

Let us use the data from the WID to illustrate how we can convert tables between long and wide formats. The original data is contained in `wid`, which is a "long" table. For illustration purposes, we simplify the table a bit and retain only three countries with observations from three years, and we also drop the COW code:

```
wid_simple<- wid %>%
  select(-ccode) %>%
  filter(year >= 2000 & year <= 2002) %>%
  filter(country %in% c("FR", "US", "DE"))
```

We can convert the table to a "wide" format with the `pivot_wider()` function from `tidyverse`. You need to specify the variable in the table that contains the values for the new header names (in our case, this is the `year` column), as well as the variable that contains the values you want in the converted table (in our case, the inequality levels in the `p90p100` column). It is useful to sort the table with `arrange()` beforehand, such that the new columns are properly ordered:

```
wid_wide <- wid_simple %>%
  arrange(year) %>%
  pivot_wider(names_from = year, values_from = p90p100)
```

This is what our new table looks like:

```
wid_wide

# A tibble: 3 x 4
  country `2000` `2001` `2002`
  <chr>    <dbl>  <dbl>  <dbl>
1 DE       0.316  0.316  0.317
2 FR       0.331  0.334  0.328
3 US       0.439  0.428  0.427
```

Since "wide" tables are usually difficult to deal with, we usually need to convert them to a "long" format rather than vice versa. This works with the `pivot_longer()` function:

```
wid_long <- wid_wide %>%
  pivot_longer(-country, names_to = "year", values_to = "p90p100")
```

This returns the data again in a long table, the format that should be preferred for most of the work we do in the social sciences:

```
wid_long

# A tibble: 9 x 3
  country year  p90p100
  <chr>   <chr>   <dbl>
1 DE      2000    0.316
2 DE      2001    0.316
3 DE      2002    0.317
# ... with 6 more rows
```

7.8 SUMMARY AND OUTLOOK

This chapter introduced the `tidyverse` framework for R, a collection of different R packages that integrate well with each other and use a consistent grammar. Although we have used the `tidyverse` only for simple data management operations here, its functionality goes well beyond this (e.g., with the `ggplot2` package for graphics). Much work in the `tidyverse` is done using a new operator, the pipe, which allows you to write code that is simple and intuitive to understand. I demonstrated how to work with `tibbles`, an extended version of the usual R data frame. The `tidyverse` offers a number of functions to perform standard data operations, such as selection, aggregation and merging of tables. It also has new and improved functions to import and export data from various different file types (see also the examples in Chapter 4).

In general, it is highly recommended to perform your data work with the `tidyverse` and its associated packages. It is elegant, powerful, and efficient, and allows your code to be easily understood and replicated by others. Although for some specialized types of data (e.g., spatial data), the integration with the `tidyverse` is not without pitfalls, all common tables with numbers and/or text can easily and conveniently be processed with it. It even interfaces well with relational databases, which we cover in the next chapters. Nevertheless, as an apt user of R, you should know both "worlds" well – base R, and how it differs from the `tidyverse`. You can then decide which one is the best choice for a given project. From this chapter, there are a number of recommendations for your work:

- *Use the pipe wherever possible:* The pipe operator allows for an improved, much more logical workflow for most data management operations. For example, in base R, users tend to create new R objects for every intermediate step of a data processing sequence. This can be confusing and error-prone. Arranged as a pipeline with the pipe operator connecting the different steps, there are no intermediate results we need to deal with – all that matters is how we get from the input to the final result.

- *Try not to mix:* Being fluent both in base R and the `tidyverse`, it is possible to switch back and forth between the different approaches in a single script. You should try to avoid this. If you opt for the `tidyverse` in your script, try to stick with it and implement the entire workflow using its functions. This makes your code consistent and easier to follow for others.
- *Watch out for potential issues with the* `tidyverse`*:* Despite the considerable advantages that the `tidyverse` has for most data management tasks, there are some potential drawbacks you should keep in mind. The `tidyverse` includes a wealth of functions, which means that conflicts can occur if other packages include functions with the same name. You see a message alerting you to (usually uncritical) conflicts with base R functions when you load the `tidyverse`. Once other packages are loaded, these conflicts can be problematic. Also, due to its size, the `tidyverse` depends on a large number of other packages, so your R installation will grow considerably and installation issues can arise.
- *Remember the conversions between long and wide tables:* As we have seen in the chapter, the `tidyverse` offers a convenient way to convert between wide and long tables. This is a task you may encounter from time to time, since existing tables are often formatted for humans to look at (and may therefore be distributed in a wide format). You should resist the temptation to manually convert them, and instead rely on R to do this.

https://doi.org/10.1017/9781108990424.010 Published online by Cambridge University Press