

Chapter 2

Loading and Manipulating Data

We now turn to using R to conduct data analysis. Our first basic steps are simply to load data into R and to clean the data to suit our purposes. Data cleaning and recoding are an often tedious task of data analysis, but nevertheless are essential because miscoded data will yield erroneous results when a model is estimated using them. (In other words, garbage in, garbage out.) In this chapter, we will load various types of data using differing commands, view our data to understand their features, practice recoding data in order to clean them as we need to, merge data sets, and reshape data sets.

Our working example in this chapter will be a subset of Poe et al.'s (1999) Political Terror Scale data on human rights, which is an update of the data in Poe and Tate (1994). Whereas their complete data cover 1976–1993, we focus solely on the year 1993. The eight variables this dataset contains are:

country: A character variable listing the country by name.

democ: The country's score on the Polity III democracy scale. Scores range from 0 (least democratic) to 10 (most democratic).

sdnew: The U.S. State Department scale of political terror. Scores range from 1 (low state terrorism, fewest violations of personal integrity) to 5 (highest violations of personal integrity).

military: A dummy variable coded 1 for a military regime, 0 otherwise.

gnpcats: Level of per capita GNP in five categories: 1 = under \$1000, 2 = \$1000–\$1999, 3 = \$2000–\$2999, 4 = \$3000–\$3999, 5 = over \$4000.

lpop: Logarithm of national population.

civ_war: A dummy variable coded 1 if involved in a civil war, 0 otherwise.

int_war: A dummy variable coded 1 if involved in an international war, 0 otherwise.

Electronic supplementary material: The online version of this chapter (doi: [10.1007/978-3-319-23446-5_2](https://doi.org/10.1007/978-3-319-23446-5_2)) contains supplementary material, which is available to authorized users.

2.1 Reading in Data

Getting data into R is quite easy. There are three primary ways to import data: Inputting the data manually (perhaps written in a script file), reading data from a text-formatted file, and importing data from another program. Since it is a bit less common in Political Science, inputting data manually is illustrated in the examples of Chap. 10, in the context of creating vectors, matrices, and data frames. In this section, we focus on importing data from saved files.

First, we consider how to read in a delimited text file with the `read.table` command. R will read in a variety of delimited files. (For all the options associated with this command type `?read.table` in R.) In text-based data, typically each line represents a unique observation and some designated delimiter separates each variable on the line. The default for `read.table` is a space-delimited file wherein any blank space designates differing variables. Since our Poe et al. data file, named `hmnrghts.txt`, is space-separated, we can read our file into R using the following line of code. This data file is available from the Dataverse named on page vii or the chapter content link on page 13. You may need to use `setwd` command introduced in Chap. 1 to point R to the folder where you have saved the data. After this, run the following code:

```
hmnrghts<-read.table("hmnrghts.txt",
  header=TRUE, na="NA")
```

Note: As was mentioned in the previous chapter, R allows the user to split a single command across multiple lines, which we have done here. Throughout this book, commands that span multiple lines will be distinguished with hanging indentation. Moving to the code itself, observe a few features: One, as was noted in the previous chapter, the left arrow symbol (`<-`) assigns our input file to an object. Hence, `hmnrghts` is the name we allocated to our data file, but we could have called it any number of things. Second, the first argument of the `read.table` command calls the name of the text file `hmnrghts.txt`. We could have preceded this argument with the `file=` option—and we would have needed to do so if we had not listed this as the first argument—but R recognizes that the file itself is normally the first argument this command takes. Third, we specified `header=TRUE`, which conveys that the first row of our text file lists the names of our variables. It is essential that this argument be correctly identified, or else variable names may be erroneously assigned as data or data as variable names.¹ Finally, within the text file, the characters NA are written whenever an observation of a variable is missing. The option `na="NA"` conveys to R that this is the data set's symbol for a missing value. (Other common symbols of missingness are a period (`.`) or the number `-9999`.)

The command `read.table` also has other important options. If your text file uses a delimiter other than a space, then this can be conveyed to R using the `sep` option. For instance, including `sep="\t"` in the previous command would have

¹A closer look at the file itself will show that our header line of variable names actually has one fewer element than each line of data. When this is the case, R assumes that the first item on each line is an observation index. Since that is true in this case, our data are read correctly.

allowed us to read in a tab-separated text file, while `sep=","` would have allowed a comma-separated file. The commands `read.csv` and `read.delim` are alternate versions of `read.table` that merely have differing default values. (Particularly, `read.csv` is geared to read comma-separated values files and `read.delim` is geared to read tab-delimited files, though a few other defaults also change.) Another important option for these commands is `quote`. The defaults vary across these commands for which characters designate string-based variables that use alphabetic text as values, such as the name of the observation (e.g., country, state, candidate). The `read.table` command, by default, uses either single or double quotation marks around the entry in the text file. This would be a problem if double quotes were used to designate text, but apostrophes were in the text. To compensate, simply specify the option `quote = "\""` to only allow double quotes. (Notice the backslash to designate that the double-quote is an argument.) Alternatively, `read.csv` and `read.delim` both only allow double quotes by default, so specifying `quote = "\"'"` would allow either single or double quotes, or `quote = "\'"` would switch to single quotes. Authors also can specify other characters in this option, if need be.

Once you download a file, you have the option of specifying the full path directory in the command to open the file. Suppose we had saved `hmnrghts.txt` into the path directory `C:/temp/`, then we could load the file as follows:

```
hmnrghts <- read.table("C:/temp/hmnrghts.txt",
  header=TRUE, na="NA")
```

As was mentioned when we first loaded the file, another option would have been to use the `setwd` command to set the working directory, meaning we would not have to list the entire file path in the call to `read.table`. (If we do this, all input and output files will go to this directory, unless we specify otherwise.) Finally, in any GUI-based system (e.g., non-terminal), we could instead type:

```
hmnrghts<-read.table(file.choose(),header=TRUE,na="NA")
```

The `file.choose()` option will open a file browser allowing the user to locate and select the desired data file, which R will then assign to the named object in memory (`hmnrghts` in this case). This browser option is useful in interactive analysis, but less useful for automated programs.

Another format of text-based data is a *fixed width file*. Files of this format do not use a character to delimit variables within an observation. Instead, certain columns of text are consistently dedicated to a variable. Thus, R needs to know which columns define each variable to read in the data. The command for reading a fixed width file is `read.fwf`. As a quick illustration of how to load this kind of data, we will load a different dataset—roll call votes from the 113th United States Senate, the term running from 2013 to 2015.² This dataset will be revisited in a practice problem

²These data were gathered by Jeff Lewis and Keith Poole. For more information, see <http://www.voteview.com/senate113.htm>.

in Chap. 8. To open these data, start by downloading the file `sen113kh.ord` from the Dataverse listed on page vii or the chapter content link on page 13. Then type:

```
senate.113<-read.fwf("sen113kh.ord",
  widths=c(3,5,2,2,8,3,1,1,11,rep(1,657)))
```

The first argument of `read.fwf` is the name of the file, which we draw from a URL. (The file extension is `.ord`, but the format is plain text. Try opening the file in Notepad or TextEdit just to get a feel for the formatting.) The second argument, `widths`, is essential. For each variable, we must enter the number of characters allocated to that variable. In other words, the first variable has three characters, the second has five characters, and so on. This procedure should make it clear that we must have a codebook for a fixed width file, or inputting the data is a hopeless endeavor. Notice that the last component in the `widths` argument is `rep(1,657)`. This means that our data set ends with 657 variables that are one character long. These are the 657 votes that the Senate cast during that term of Congress, with each variable recording whether each senator voted yea, nay, present, or did not vote.

With any kind of data file, including fixed width, if the file itself does not have names of the variables, we can add these in R. (Again, a good codebook is useful here.) The commands `read.table`, `read.csv`, and `read.fwf` all include an option called `col.names` that allows the user to name every variable in the dataset when reading in the file. In the case of the Senate roll calls, though, it is easier for us to name the variables afterwards as follows:

```
colnames(senate.113)[1:9]<-c("congress","icpsr","state.code",
  "cd","state.name","party","occupancy","attaining","name")
for(i in 1:657){colnames(senate.113)[i+9]<-paste("RC",i,sep="")}
```

In this case, we use the `colnames` command to set the names of the variables. To the left of the arrow, by specifying `[1:9]`, we indicate that we are only naming the first nine variables. To the right of the arrow, we use one of the *most fundamental* commands in R: the `combine` command (`c`), which combines several elements into a vector. In this case, our vector includes the names of the variables in text. On the second line, we proceed to name the 657 roll call votes `RC1`, `RC2`, ..., `RC657`. To save typing we make this assignment using a `for` loop, which is described in more detail in Chap. 11. Within the `for` loop, we use the `paste` command, which simply prints our text (`"RC"`) and the index number `i`, separated by nothing (hence the empty quotes at the end). Of course, by default, R assigns generic variable names (`V1`, `V2`, etc.), so a reader who is content to use generic names can skip this step, if preferred. (Bear in mind, though, that if we name the first nine variables like we did, the first roll call vote would be named `V10` without our applying a new name.)

2.1.1 Reading Data from Other Programs

Turning back to our human rights example, you also can import data from many other statistical programs. One of the most important libraries in R is the `foreign` package, which makes it very easy to bring in data from other statistical packages, such as SPSS, Stata, and Minitab.³ As an alternative to the text version of the human rights data, we also could load a Stata-formatted data file, `hmnrghts.dta`.

Stata files generally have a file extension of `dta`, which is what the `read.dta` command refers to. (Similarly, `read.spss` will **read** an **SPSS**-formatted file with the `.sav` file extension.) To open our data in Stata format, we need to download the file `hmnrghts.dta` from the Dataverse linked on page vii or the chapter content linked on page 13. Once we save it to our hard drive, we can either set our working directory, list the full file path, or use the `file.choose()` command to access our data. For example, if we downloaded the file, which is named `hmnrghts.dta`, into our `C:\temp\` folder, we could open it by typing:

```
library(foreign)
setwd("C:/temp/")
hmnrghts.2 <- read.dta("hmnrghts.dta")
```

Any data in Stata format that you select will be converted to R format. One word of warning, by default if there are value labels on Stata-formatted data, R will import the labels as a string-formatted variable. If this is an issue for you, try importing the data without value labels to save the variables using the numeric codes. See the beginning of Chap. 7 for an example of the `convert.factors=FALSE` option. (One option for data sets that are not excessively large is to load two copies of a Stata dataset—one with the labels as text to serve as a codebook and another with numerical codes for analysis.) It is always good to see exactly how the data are formatted by inspecting the spreadsheet after importing with the tools described in Sect. 2.2.

2.1.2 Data Frames in R

R distinguishes between *vectors*, *lists*, *data frames*, and *matrices*. Each of these is an object of a different class in R. Vectors are indexed by length, and matrices are indexed by rows and columns. Lists are not pervasive to basic analyses, but are handy for complex storage and are often thought of as *generic* vectors where each element can be any class of object. (For example, a list could be a vector of

³The `foreign` package is so commonly used that it now downloads with any new R installation. In the unlikely event, though, that the package will not load with the `library` command, simply type `install.packages("foreign")` in the command prompt to download it. Alternatively, for users wishing to import data from *Excel*, two options are present: One is to save the Excel file in comma-separated values format and then use the `read.csv` command. The other is to install the `XLConnect` library and use the `readWorksheetFromFile` command.

model results, or a mix of data frames and maps.) A data frame is a matrix that **R** designates as a data set. With a data frame, the columns of the matrix can be referred to as variables. After reading in a data set, **R** will treat your data as a data frame, which means that you can refer to any variable within a data frame by adding `$VARIABLENAME` to the name of the data frame.⁴ For example, in our human rights data we can print out the variable `country` in order to see which countries are in the dataset:

```
hmnrghts$country
```

Another option for calling variables, though an *inadvisable* one, is to use the `attach` command. **R** allows the user to load multiple data sets at once (in contrast to some of the commercially available data analysis programs). The `attach` command places one dataset at the forefront and allows the user to call directly the names of the variables without referring the name of the data frame. For example:

```
attach(hmnrghts)
country
```

With this code, **R** would recognize `country` in isolation as part of the attached dataset and print it just as in the prior example. The problem with this approach is that **R** may store objects in memory with the *same name* as some of the variables in the dataset. In fact, when recoding data the user should *always* refer to the data frame by name, otherwise **R** confusingly will create a copy of the variable in memory that is distinct from the copy in the data frame. For this reason, I generally recommend against attaching data. If, for some circumstance, a user feels attaching a data frame is unavoidable, then the user can conduct what needs to be done with the attached data and then use the `detach` command as soon as possible. This command works as would be expected, removing the designation of a working data frame and no longer allowing the user to call variables in isolation:

```
detach(hmnrghts)
```

2.1.3 Writing Out Data

To export data you are using in **R** to a text file, use the functions `write.table` or `write.csv`. Within the `foreign` library, `write.dta` allows the user to write out a Stata-formatted file. As a simple example, we can generate a matrix with four observations and six variables, counting from 1 to 24. Then we can write this to a comma-separated values file, a space-delimited text file, and a Stata file:

```
x <- matrix(1:24, nrow=4)
write.csv(x, file="sample.csv")
write.table(x, file="sample.txt")
write.dta(as.data.frame(x), file="sample.dta")
```

⁴More technically, data frames are objects of the **S3** class. For all **S3** objects, attributes of the object (such as variables) can be called with the dollar sign (`$`).

Note that the command `as.data.frame` converts matrices to data frames, a distinction described in the prior section. The command `write.dta` expects the object to be of the `data.frame` class. Making this conversion is unnecessary if the object is already formatted as data, rather than a matrix. To check your effort in saving the files, try removing `x` from memory with the command `rm(x)` and then restoring the data from one of the saved files.

To keep up with where the saved data files are going, the files we write out will be saved in our *working directory*. To **get** the working directory (that is, have **R** tell us what it is), simply type: `getwd()`. To change the working directory where output files will be written, we can **set** the working directory, using the same `setwd` command that we considered when opening files earlier. All subsequently saved files will be output into the specified directory, unless **R** is explicitly told otherwise.

2.2 Viewing Attributes of the Data

Once data are input into **R**, the first task should be to inspect the data and make sure they have loaded properly. With a relatively small dataset, we can simply print the whole data frame to the screen:

```
hmnrghts
```

Printing the entire dataset, of course, is not recommended or useful with large datasets. Another option is to look at the names of the variables and the first few lines of data to see if the data are structured correctly through a few observations. This is done with the `head` command:

```
head(hmnrghts)
```

For a quick list of the names of the variables in our dataset (which can also be useful if exact spelling or capitalization of variable names is forgotten) type:

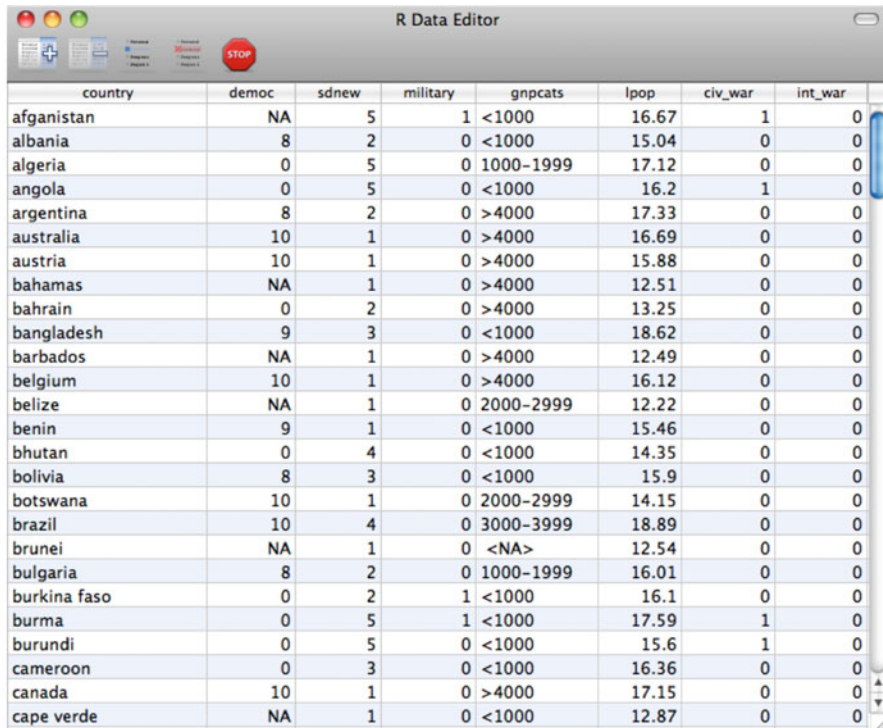
```
names(hmnrghts)
```

A route to getting a comprehensive look at the data is to use the `fix` command:

```
fix(hmnrghts)
```

This presents the data in a spreadsheet allowing for a quick view of observations or variables of interest, as well as a chance to see that the data matrix loaded properly. An example of this data editor window that `fix` opens is presented in Fig. 2.1. The user has the option of editing data within the spreadsheet window that `fix` creates, though unless the revised data are written to a new file, there will be no permanent record of these changes.⁵ Also, it is key to note that before continuing an **R** session

⁵The `View` command is similar to `fix`, but does not allow editing of observations. If you prefer to only be able to see the data without editing values (perhaps even by accidentally leaning on your keyboard), then `View` might be preferable.



country	democ	sdnew	military	gnpcats	lpop	civ_war	int_war
afghanistan	NA	5	1	<1000	16.67	1	0
albania	8	2	0	<1000	15.04	0	0
algeria	0	5	0	1000-1999	17.12	0	0
angola	0	5	0	<1000	16.2	1	0
argentina	8	2	0	>4000	17.33	0	0
australia	10	1	0	>4000	16.69	0	0
austria	10	1	0	>4000	15.88	0	0
bahamas	NA	1	0	>4000	12.51	0	0
bahrain	0	2	0	>4000	13.25	0	0
bangladesh	9	3	0	<1000	18.62	0	0
barbados	NA	1	0	>4000	12.49	0	0
belgium	10	1	0	>4000	16.12	0	0
belize	NA	1	0	2000-2999	12.22	0	0
benin	9	1	0	<1000	15.46	0	0
bhutan	0	4	0	<1000	14.35	0	0
bolivia	8	3	0	<1000	15.9	0	0
botswana	10	1	0	2000-2999	14.15	0	0
brazil	10	4	0	3000-3999	18.89	0	0
brunei	NA	1	0	<NA>	12.54	0	0
bulgaria	8	2	0	1000-1999	16.01	0	0
burkina faso	0	2	1	<1000	16.1	0	0
burma	0	5	1	<1000	17.59	1	0
burundi	0	5	0	<1000	15.6	1	0
cameroon	0	3	0	<1000	16.36	0	0
canada	10	1	0	>4000	17.15	0	0
cape verde	NA	1	0	<1000	12.87	0	0

Fig. 2.1 R data editor opened with the `fix` command

with more commands, you must close the data editor window. The console is frozen as long as the `fix` window is open.

We will talk more about descriptive statistics in Chap. 4. In the meantime, though, it can be informative to see some of the basic descriptive statistics (including the mean, median, minimum, and maximum) as well as a count of the number of missing observations for each variable:

```
summary(hmnrghts)
```

Alternatively, this information can be gleaned for only a single variable, such as logged population:

```
summary(hmnrghts$lpop)
```

2.3 Logical Statements and Variable Generation

As we turn to cleaning data that are loaded in R, an essential toolset is the group of logical statements. Logical (or Boolean) statements in R are evaluated as to whether they are TRUE or FALSE. Table 2.1 summarizes the common logical operators in R.

Table 2.1 Logical operators in R

Operator	Means
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equal to
!=	Not equal to
&	And
	Or

Note that the Boolean statement “is equal to” is designated by two equals signs (==), whereas a single equals sign (=) instead serves as an assignment operator.

To apply some of these Boolean operators from Table 2.1 in practice, suppose, for example, we wanted to know which countries were in a civil war and had an above average democracy score in 1993. We could generate a new variable in our working dataset, which I will call `dem.civ` (though the user may choose the name). Then we can view a table of our new variable and list all of the countries that fit these criteria:

```
hmnrghts$dem.civ <- as.numeric(hmnrghts$civ_war==1 &
                               hmnrghts$democ>5.3)
table(hmnrghts$dem.civ)
hmnrghts$country[hmnrghts$dem.civ==1]
```

On the first line, `hmnrghts$dem.civ` defines our new variable within the human rights dataset.⁶ On the right, we have a two-part Boolean statement: The first asks whether the country is in a civil war, and the second asks if the country’s democracy score is higher than the average of 5.3. The ampersand (&) requires that both statements must simultaneously be true for the whole statement to be true. All of this is embedded within the `as.numeric` command, which encodes our Boolean output **as** a **numeric** variable. Specifically, all values of TRUE are set to 1 and FALSE values are set to 0. Such a coding is usually more convenient for modeling purposes. The next line gives us a table of the relative frequencies of 0s and 1s. It turns out that only four countries had above-average democracy levels and were involved in a civil war in 1993. To see which countries, the last line asks R to print the names of countries, but the square braces following the vector indicate which observations to print: Only those scoring 1 on this new variable.⁷

⁶Note, though, that any new variables we create, observations we drop, or variables we recode only change the data in *working memory*. Hence, our original data file on disk remains unchanged and therefore safe for recovery. Again, we must use one of the commands from Sect. 2.1.3 if we want to save a second copy of the data including all of our changes.

⁷The output prints the four country names, and four values of NA. This means in four cases, one of the two component statements was TRUE but the other statement could not be evaluated because the variable was missing.

Another sort of logical statement in R that can be useful is the `is` statement. These statements ask whether an observation or object meets some criterion. For example, `is.na` is a special case that asks whether an observation is missing or not. Alternatively, statements such as `is.matrix` or `is.data.frame` ask whether an object is of a certain class. Consider three examples:

```
table(is.na(hmnrghs$democ))
is.matrix(hmnrghs)
is.data.frame(hmnrghs)
```

The first statement asks for each observation whether the value of democracy is missing. The `table` command then aggregates this and informs us that 31 observations are missing. The next two statements ask whether our dataset `hmnrghs` is saved as a matrix, then as a data frame. The `is.matrix` statement returns `FALSE`, indicating that matrix-based commands will not work on our data, and the `is.data.frame` statement returns `TRUE`, which indicates that it is stored as a data frame. With a sense of logical statements in R, we can now apply these to the task of cleaning data.

2.4 Cleaning Data

One of the first tasks of data cleaning is deciding how to deal with *missing data*. R designates missing values with `NA`. It translates missing values from other statistics packages into the `NA` missing format. However a scholar deals with missing data, it is important to be mindful of the relative proportion of unobserved values in the data and what information may be lost. One (somewhat crude) option to deal with missingness would be to prune the dataset through listwise deletion, or removing every observation for which a single variable is not recorded. To create a new data set that prunes in this way, type:

```
hmnrghs.trim <- na.omit(hmnrghs)
```

This diminishes the number of observations from 158 to 127, so a tangible amount of information has been lost.

Most modeling commands in R give users the option of estimating the model over complete observations only, implementing listwise deletion on the fly. As a warning, listwise deletion is actually the default in the base commands for linear and generalized linear models, so data loss can fly under the radar if the user is not careful. Users with a solid background on regression-based modeling are urged to consider alternative methods for dealing with missing data that are superior to listwise deletion. In particular, the `mice` and `Amelia` libraries implement the useful technique of multiple imputation (for more information see King et al. 2001; Little and Rubin 1987; Rubin 1987).

If, for some reason, the user needs to redesignate missing values as having some numeric value, the `is.na` command can be useful. For example, if it were beneficial to list missing values as `-9999`, then these could be coded as:

```
hmnrghs$democ[is.na(hmnrghs$democ)] <- -9999
```

In other words, all values of democracy for which the value is missing will take on the value of -9999 . *Be careful*, though, as R and all of its modeling commands will now regard the formerly missing value as a valid observation and will insert the misleading value of -9999 into any analysis. This sort of action should only be taken if it is required for data management, a special kind of model where strange values can be dummied-out, or the rare case where a missing observation actually can take on a meaningful value (e.g., a budget dataset where missing items represent a \$0 expenditure).

2.4.1 Subsetting Data

In many cases, it is convenient to subset our data. This may mean that we only want observations of a certain type, or it may mean we wish to winnow-down the number of variables in the data frame, perhaps because the data include many variables that are of no interest. If, in our human rights data, we only wanted to focus on countries that had a democracy score from 6–10, we could call this subset `dem.rights` and create it as follows:

```
dem.rights <- subset(hmnrghs, subset=democ>5)
```

This creates a 73 observation subset of our original data. Note that observations with a *missing* (NA) value of **democ** will not be included in the subset. Missing observations also would be excluded if we made a greater than or equal to statement.⁸

As an example of variable selection, if we wanted to focus only on democracy and wealth, we could keep only these two variables and an index for all observations:

```
dem.wealth<-subset(hmnrghs,select=c(country, democ, gnpcats))
```

An alternative means of selecting which variables we wish to keep is to use a minus sign after the `select` option and list only the columns we wish to drop. For example, if we wanted all variables except the two indicators of whether a country was at war, we could write:

```
no.war <- subset(hmnrghs,select=-c(civ_war,int_war))
```

Additionally, users have the option of calling both the `subset` and `select` options if they wish to choose a subset of variables over a specific set of observations.

⁸This contrasts from programs like Stata, which treat missing values as positive infinity. In Stata, whether missing observations are included depends on the kind of Boolean statement being made. R is more consistent in that missing cases are always excluded.

2.4.2 Recoding Variables

A final aspect of data cleaning that often arises is the need to recode variables. This may emerge because the functional form of a model requires a transformation of a variable, such as a logarithm or square. Alternately, some of the values of the data may be misleading and thereby need to be recoded as missing or another value. Yet another possibility is that the variables from two datasets need to be coded on the same scale: For instance, if an analyst fits a model with survey data and then makes forecasts using Census data, then the survey and Census variables need to be coded the same way.

For mathematical transformations of variables, the syntax is straightforward and follows the form of the example below. Suppose we want the actual population of each country instead of its logarithm:

```
hmnrghts$pop <- exp(hmnrghts$lpop)
```

Quite simply, we are applying the exponential function (`exp`) to a logged value to recover the original value. Yet any type of mathematical operator could be substituted in for `exp`. A variable could be squared (`^2`), logged (`log()`), have the square root taken (`sqrt()`), etc. Addition, subtraction, multiplication, and division are also valid—either with a scalar of interest or with another variable. Suppose we wanted to create an ordinal variable coded 2 if a country was in both a civil war and an international war, 1 if it was involved in either, and 0 if it was not involved in any wars. We could create this by adding the civil war and international war variables:

```
hmnrghts$war.ord<-hmnrghts$civ_war+hmnrghts$int_war
```

A quick table of our new variable, however, reveals that no nations had both kinds of conflict going in 1993.

Another common issue to address is when data are presented in an undesirable format. Our variable **gnpcats** is actually coded as a text variable. However, we may wish to recode this as a numeric ordinal variable. There are two means of accomplishing this. The first, though taking several lines of code, can be completed quickly with a fair amount of copy-and-paste:

```
hmnrghts$gnp.ord <- NA
hmnrghts$gnp.ord[hmnrghts$gnpcats=="<1000"] <-1
hmnrghts$gnp.ord[hmnrghts$gnpcats=="1000-1999"] <-2
hmnrghts$gnp.ord[hmnrghts$gnpcats=="2000-2999"] <-3
hmnrghts$gnp.ord[hmnrghts$gnpcats=="3000-3999"] <-4
hmnrghts$gnp.ord[hmnrghts$gnpcats==">4000"] <-5
```

Here, a blank variable was created, and then the values of the new variable filled-in contingent on the values of the old using Boolean statements.

A second option for recoding the GNP data can be accomplished through John Fox's companion to applied regression (`car`) library. As a user-written library, we must download and install it before the first use. The installation of a library is straightforward. First, type:

```
install.packages("car")
```

Once the library is installed (again, a step which need not be repeated unless R is reinstalled), the following lines will generate our recoded per capita GNP measure:

```
library(car)
hmnrghts$gnp.ord.2<-recode(hmnrghts$gnpcats, "'<1000'=1;
    '1000-1999'=2;'2000-2999'=3;'3000-3999'=4;'>4000'=5')
```

Be careful that the `recode` command is delicate. Between the apostrophes, all of the reassignments from old values to new are defined separated by semicolons. A single space between the apostrophes will generate an error. Despite this, `recode` can save users substantial time on data cleaning. The basic syntax of `recode`, of course, could be used to create dummy variables, ordinal variables, or a variety of other recoded variables. So now two methods have created a new variable, each coded 1 to 5, with 5 representing the highest per capita GNP.

Another standard type of recoding we might want to do is to create a dummy variable that is coded as 1 if the observation meets certain conditions and 0 otherwise. For example, suppose instead of having categories of GNP, we just want to compare the highest category of GNP to all the others:

```
hmnrghts$gnp.dummy<-as.numeric(hmnrghts$gnpcats==">4000")
```

As with our earlier example of finding democracies involved in a civil war, here we use a logical statement and modify it with the `as.numeric` statement, which turns each TRUE into a 1 and each FALSE into a 0.

Categorical variables in R can be given a special designation as *factors*. If you designate a categorical variable as a factor, R will treat it as such in statistical operation and create dummy variables for each level when it is used in a regression. If you import a variable with no numeric coding, R will automatically call the variable a *character* vector, and convert the character vector into a factor in most analysis commands. If we prefer, though, we can designate that a variable is a factor ahead of time and open up a variety of useful commands. For example, we can designate `country` as a factor:

```
hmnrghts$country <- as.factor(hmnrghts$country)
levels(hmnrghts$country)
```

Notice that R allows the user to put the same quantity (in this case, the variable **country**) on both sides of an assignment operator. This recursive assignment takes the old values of a quantity, makes the right-hand side change, and then replaces the new values into the same place in memory. The `levels` command reveals to us the different recorded values of the factor.

To change which level is the first level (e.g., to change which category R will use as the reference category in a regression) use the `relevel` command. The following code sets “united states” as the reference category for **country**:

```
hmnrghts$country<-relevel(hmnrghts$country,"united states")
levels(hmnrghts$country)
```

Now when we view the levels of the factor, “united states” is listed as the first level, and the first level is always our reference group.

2.5 Merging and Reshaping Data

Two final tasks that are common to data management are merging data sets and reshaping panel data. As we consider examples of these two tasks, let us consider an update of Poe et al.'s (1999) data: Specifically, Gibney et al. (2013) have continued to code data for the Political Terror Scale. We will use the 1994 and 1995 waves of the updated data. The variables in these waves are:

Country: A character variable listing the country by name.

COWAlpha: Three-character country abbreviation from the Correlates of War dataset.

COW: Numeric country identification variable from the Correlates of War dataset.

WorldBank: Three-character country abbreviation used by the World Bank.

Amnesty.1994/Amnesty.1995: Amnesty International's scale of political terror. Scores range from 1 (low state terrorism, fewest violations of personal integrity) to 5 (highest violations of personal integrity).

StateDept.1994/StateDept.1995: The U.S. State Department scale of political terror. Scores range from 1 (low state terrorism, fewest violations of personal integrity) to 5 (highest violations of personal integrity).

For the last two variables, the name of the variable depends on which wave of data is being studied, with the suffix indicating the year. Notice that these data have four identification variables: This is designed explicitly to make these data easier for researchers to use. Each index makes it easy for a researcher to link these political terror measures to information provided by either the World Bank or the Correlates of War dataset. This should show how ubiquitous the act of merging data is to Political Science research.

To this end, let us practice *merging data*. In general, merging data is useful when the analyst has two separate data frames that contain information about the same observations. For example, if a Political Scientist had one data frame with economic data by country and a second data frame containing election returns by country, the scholar might want to merge the two data sets to link economic and political factors within each country. In our case, suppose we simply wanted to link each country's political terror score from 1994 to its political terror score from 1995. First, download the data sets `pts1994.csv` and `pts1995.csv` from the Dataverse on page vii or the chapter content link on page 13. As before, you may need to use `setwd` to point R to the folder where you have saved the data. After this, run the following code to load the relevant data:

```
hmnrghts.94<-read.csv("pts1994.csv")
hmnrghts.95<-read.csv("pts1995.csv")
```

These data are comma separated, so `read.csv` is the best command in this case.

If we wanted to take a look at the first few observations of our 1994 wave, we could type `head(hmnrghts.94)`. This will print the following:

	Country	COWAlpha	COW	WorldBank
1	United States	USA	2	USA
2	Canada	CAN	20	CAN
3	Bahamas	BHM	31	BHS
4	Cuba	CUB	40	CUB
5	Haiti	HAI	41	HTI
6	Dominican Republic	DOM	42	DOM

	Amnesty.1994	StateDept.1994
1	1	NA
2	1	1
3	1	2
4	3	3
5	5	4
6	2	2

Similarly, `head(hmnrghs.95)` will print the first few observations of our 1995 wave:

	Country	COWAlpha	COW	WorldBank
1	United States	USA	2	USA
2	Canada	CAN	20	CAN
3	Bahamas	BHM	31	BHS
4	Cuba	CUB	40	CUB
5	Haiti	HAI	41	HTI
6	Dominican Republic	DOM	42	DOM

	Amnesty.1995	StateDept.1995
1	1	NA
2	NA	1
3	1	1
4	4	3
5	2	3
6	2	2

As we can see from our look at the top of each data frame, the data are ordered similarly in each case, and our four index variables have the same name in each respective data set. We only need one index to merge our data, and the other three are redundant. For this reason, we can drop three of the index variables from the 1995 data:

```
hmnrghs.95<-subset(hmnrghs.95,
  select=c(COW,Amnesty.1995,StateDept.1995))
```

We opted to delete the three text indices in order to merge on a numeric index. This choice is arbitrary, however, because **R** has no problem merging on a character variable either. (Try replicating this exercise by merging on COWAlpha, for example.)

To combine our 1994 and 1995 data, we now turn to the `merge` command.⁹ We type:

```
hmnrghts.wide<-merge(x=hmnrghs.94,y=hmnrghs.95,by=c("COW"))
```

Within this command, the option `x` refers to one dataset, while `y` is the other. Next to the `by` option, we name an identification variable that uniquely identifies each observation. The `by` command actually allows users to name multiple variables if several are needed to uniquely identify each observation: For example, if a researcher was merging data where the unit of analysis was a country-year, a country variable and a year variable might be essential to identify each row uniquely. In such a case the syntax might read, `by=c("COW", "year")`. As yet another option, if the two datasets had the same index, but the variables were named differently, R allows syntax such as, `by.x=c("COW")`, `by.y=c("cowCode")`, which conveys that the differently named index variables are the name.

Once we have merged our data, we can preview the finished product by typing `head(hmnrghts.wide)`. This prints:

	COW	Country	COWAlpha	WorldBank	Amnesty.1994
1	2	United States	USA	USA	1
2	20	Canada	CAN	CAN	1
3	31	Bahamas	BHM	BHS	1
4	40	Cuba	CUB	CUB	3
5	41	Haiti	HAI	HTI	5
6	42	Dominican Republic	DOM	DOM	2

	StateDept.1994	Amnesty.1995	StateDept.1995
1	NA	1	NA
2	1	NA	1
3	2	1	1
4	3	4	3
5	4	2	3
6	2	2	2

As we can see, the 1994 and 1995 scores for Amnesty and StateDept are recorded in one place for each country. Hence, our merge was successful. By default, R excludes any observation from either dataset that does not have a *linked* observation (e.g., equivalent value) from the other data set. So if you use the defaults and the new dataset includes the same number of rows as the two old datasets, then all observations were linked and included. For instance, we could type:

```
dim(hmnrghts.94); dim(hmnrghts.95); dim(hmnrghts.wide)
```

This would quickly tell us that we have 179 observations in both of the inputs, as well as the output dataset, showing we did not lose any observations. Other options within `merge` are `all.x`, `all.y`, and `all`, which allow you to specify whether to force

⁹Besides the `merge`, the `dplyr` package offers several data-joining commands that you also may find of use, depending on your needs.

the inclusion of all observations from the dataset `x`, the dataset `y`, and from either dataset, respectively. In this case, `R` would encode `NA` values for observations that did not have a linked case in the other dataset.

As a final point of data management, sometimes we need to *reshape* our data. In the case of our merged data set, `hmnrghts.wide`, we have created a panel data set (e.g., a data set consisting of repeated observations of the same individuals) that is in *wide format*. Wide format means that each row in our data defines an individual of study (a country) while our repeated observations are stored in separate variables (e.g., `Amnesty.1994` and `Amnesty.1995` record Amnesty International scores for two separate years). In most models of panel data, we need our data to be in *long format*, or stacked format. Long format means that we need two index variables to identify each row, one for the individual (e.g., country) and one for time of observation (e.g., year). Meanwhile, each variable (e.g., `Amnesty`) will only use one column. `R` allows us to reshape our data from wide to long, or from long to wide. Hence, whatever the format of our data, we can reshape it to our needs.

To reshape our political terror data from wide format to long, we use the `reshape` command:

```
hmnrghts.long<-reshape(hmnrghts.wide,varying=c("Amnesty.1994",
"StateDept.1994","Amnesty.1995","StateDept.1995"),
timevar="year",idvar="COW",direction="long",sep=".")
```

Within the command, the first argument is the name of the data frame we wish to reshape. The `varying` term lists all of the variables that represent repeated observations over time. *Tip:* Be sure that repeated observations of the same variable have the same *prefix* name (e.g., `Amnesty` or `StateDept`) and then the *suffix* (e.g., `1994` or `1995`) consistently reports time. The `timevar` term allows us to specify the name of our new time index, which we call `year`. The `idvar` term lists the variable that uniquely identifies individuals (countries, in our case). With `direction` we specify that we want to convert our data into long format. Lastly, the `sep` command offers `R` a cue of what character separates our prefixes and suffixes in the repeated observation variables: Since a period (.) separates these terms in each of our `Amnesty` and `StateDept` variables, we denote that here.

A preview of the result can be seen by typing `head(hmnrghts.long)`. This prints:

	COW	Country	COWAlpha	WorldBank	year
2.1994	2	United States	USA	USA	1994
20.1994	20	Canada	CAN	CAN	1994
31.1994	31	Bahamas	BHM	BHS	1994
40.1994	40	Cuba	CUB	CUB	1994
41.1994	41	Haiti	HAI	HTI	1994
42.1994	42	Dominican Republic	DOM	DOM	1994
		Amnesty	StateDept		
2.1994	1	NA			
20.1994	1	1			

31.1994	1	2
40.1994	3	3
41.1994	5	4
42.1994	2	2

Notice that we now have only one variable for Amnesty and one for StateDept. We now have a new variable named `year`, so between `COW` and `year`, each row uniquely identifies each country-year. Since the data are naturally sorted, the top of our data only show 1994 observations. Typing `head(hmnrghs.long[hmnrghs.long$year==1995,])` shows us the first few 1995 observations:

	COW	Country	COWAlpha	WorldBank	year
2.1995	2	United States	USA	USA	1995
20.1995	20	Canada	CAN	CAN	1995
31.1995	31	Bahamas	BHM	BHS	1995
40.1995	40	Cuba	CUB	CUB	1995
41.1995	41	Haiti	HAI	HTI	1995
42.1995	42	Dominican Republic	DOM	DOM	1995

	Amnesty	StateDept
2.1995	1	NA
20.1995	NA	1
31.1995	1	1
40.1995	4	3
41.1995	2	3
42.1995	2	2

As we can see, all of the information is preserved, now in long (or stacked) format.

As a final illustration, suppose we had started with a data set that was in long format and wanted one in wide format. To try this, we will reshape `hmnrghs.long` and try to recreate our original wide data set. To do this, we type:

```
hmnrghs.wide.2<-reshape(hmnrghs.long,
  v.names=c("Amnesty","StateDept"),
  timevar="year",idvar="COW",direction="wide",sep=".")
```

A few options have now changed: We now use the `v.names` command to indicate the variables that include repeated observations. The `timevar` parameter now needs to be a variable within the dataset, just as `idvar` is, in order to separate individuals from repeated time points. Our `direction` term is now `wide` because we want to convert these data into wide format. Lastly, the `sep` command specifies the character that R will use to separate prefixes from suffixes in the final form. By typing `head(hmnrghs.wide.2)` into the console, you will now see that this new dataset recreates the original wide dataset.

This chapter has covered the variety of means of importing and exporting data in R. It also has discussed data management issues such as missing values, subsetting, recoding data, merging data, and reshaping data. With the capacity to clean and manage data, we now are ready to start analyzing our data. We next proceed to data visualization.

2.6 Practice Problems

As a practice dataset, we will download and open a subset of the 2004 American National Election Study used by Hanmer and Kalkan (2013). This dataset is named `hanmerKalkanANES.dta`, and it is available from the Dataverse referenced on page vii or in the chapter content link on page 13. These data are in Stata format, so be sure to load the correct library and use the correct command when opening. (*Hint:* When using the proper command, be sure to specify the `convert.factors=F` option within it to get an easier-to-read output.) The variables in this dataset all relate to the 2004 U.S. presidential election, and they are: a respondent identification number (**caseid**), retrospective economic evaluations (**retecon**), assessment of George W. Bush's handling of the war in Iraq (**bushiraq**), an indicator for whether the respondent voted for Bush (**presvote**), partisanship on a seven-point scale (**partyid**), ideology on a seven-point scale (**ideol7b**), an indicator of whether the respondent is white (**white**), an indicator of whether the respondent is female (**female**), age of the respondent (**age**), level of education on a seven-point scale (**educ1_7**), and income on a 23-point scale (**income**). (The variable **exptrnout2** can be ignored.)

1. Once you have loaded the data, do the following to check your work:
 - (a) If you ask **R** to return the variable names, what does the list say? Is it correct?
 - (b) Using the `head` command, what do the first few lines look like?
 - (c) If you use the `fix` command, do the data look like an appropriate spreadsheet?
2. Use the `summary` command on the whole data set. What can you learn immediately? How many missing observations do you have?
3. Try subsetting the data in a few ways:
 - (a) Create a copy of the dataset that removes all missing observations with listwise deletion. How many observations remain in this version?
 - (b) Create a second copy that only includes the respondent identification number, retrospective economic evaluations, and evaluation of Bush's handling of Iraq.
4. Create a few new variables:
 - (a) The seven-point partisanship scale (**partyid**) is coded as follows: 0 = Strong Democrat, 1 = Weak Democrat, 2 = Independent Leaning Democrat, 3 = Independent No Leaning, 4 = Independent Leaning Republican, 5 = Weak Republican, and 6 = Strong Republican. Create two new indicator variables. The first should be coded 1 if the person identifies as Democrat in any way (including independents who lean Democratic), and 0 otherwise. The second new variable should be coded 1 if the person identifies as Republican in any way (including independents who lean Republican), and 0 otherwise. For each of these two new variables, what does the `summary` command return for them?
 - (b) Create a new variable that is the squared value of the respondent's age in years. What does the `summary` command return for this new variable?
 - (c) Create a new version of the income variable that has only four categories. The first category should include all values of **income** that range from 1–12,

the second from 13–17, the third from 18–20, and the last from 21–23. Use the `table` command to see the frequency of each category.

- (d) Bonus: Use the `table` command to compare the 23-category version of income to the four-category version of income. Did you code the new version correctly?