

Chapter 1

Obtaining R and Downloading Packages

This chapter is written for the user who has never downloaded R onto his or her computer, much less opened the program. The chapter offers some brief background on what the program is, then proceeds to describe how R can be downloaded and installed completely free of charge. Additionally, the chapter lists some internet-based resources on R that users may wish to consult whenever they have questions not addressed in this book.

1.1 Background and Installation

R is a platform for the object-oriented statistical programming language S. S was initially developed by John Chambers at Bell Labs, while R was created by Ross Ihaka and Robert Gentleman. R is widely used in statistics and has become quite popular in Political Science over the last decade. The program also has become more widely used in the business world and in government work, so training as an R user has become a marketable skill. R, which is shareware, is similar to S-plus, which is the commercial platform for S. Essentially R can be used as either a matrix-based programming language or as a standard statistical package that operates much like the commercially sold programs Stata, SAS, and SPSS.

Electronic supplementary material: The online version of this chapter (doi: [10.1007/978-3-319-23446-5_1](https://doi.org/10.1007/978-3-319-23446-5_1)) contains supplementary material, which is available to authorized users.

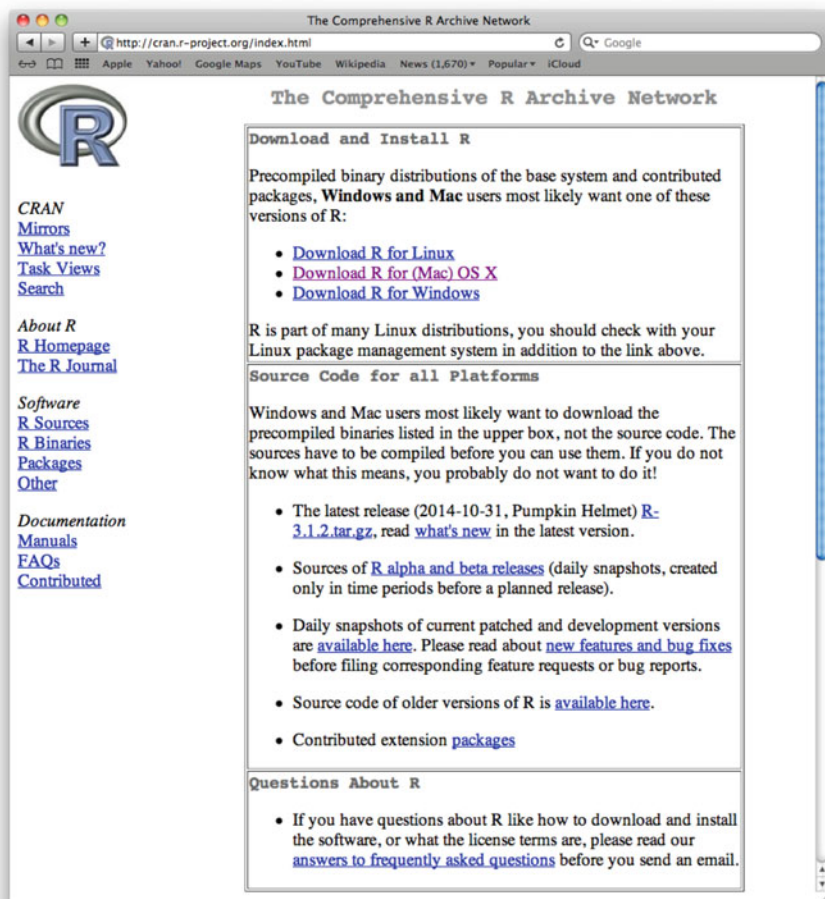


Fig. 1.1 The comprehensive R archive network (CRAN) homepage. The top box, “Download and Install R,” offers installation links for the three major operating systems

1.1.1 Where Can I Get R?

The beauty of R is that it is shareware, so it is free to anyone. To obtain R for Windows, Mac, or Linux, simply visit the comprehensive R archive network (CRAN) at <http://www.cran.r-project.org/>. Figure 1.1 shows the homepage of this website. As can be seen, at the top of the page is an inset labeled *Download and Install R*. Within this are links for installation using the Linux, Mac OS X, and Windows operating systems. In the case of Mac, clicking the link will bring up a downloadable file with the `pkg` suffix that will install the latest version. For Windows, a link named `base` will be presented, which leads to an `exe` file for

download and installation. In each operating system, opening the respective file will guide the user through the automated installation process.¹ In this simple procedure, a user can install R on his or her personal machine within five minutes.

As months and years pass, users will observe the release of new versions of R. There are not update patches for R, so as new versions are released, you must completely install a new version whenever you would like to upgrade to the latest edition. Users need not reinstall every single version that is released. However, as time passes, add-on libraries (discussed later in this chapter) will cease to support older versions of R. One potential guide on this point is to upgrade to the newest version whenever a library of interest cannot be installed due to lack of support. The only major inconvenience that complete reinstallation poses is that user-created add-on libraries will have to be reinstalled, but this can be done on an as-needed basis.

1.2 Getting Started: A First Session in R

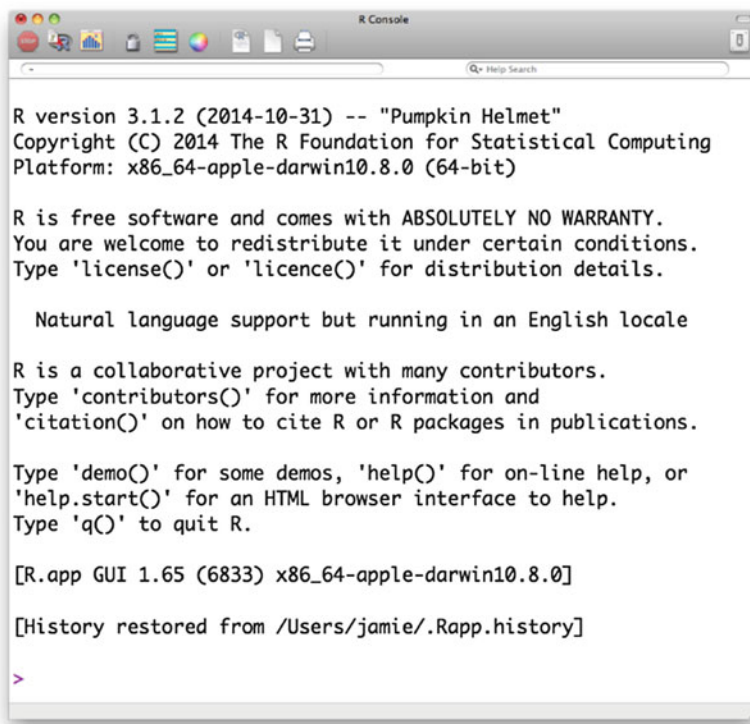
Once you have installed R, there will be an icon either under the Windows Start menu (with an option of placing a shortcut on the Desktop) or in the Mac Applications folder (with an option of keeping an icon in the workspace Dock). Clicking or double-clicking the icon will start R. Figure 1.2 shows the window associated with the Mac version of the software. You will notice that R does have a few push-button options at the top of the window. Within Mac, the menu bar at the top of the workspace will also feature a few pull-down menus. In Windows, pull down menus also will be presented within the R window. With only a handful of menus and buttons, however, commands in R are entered primarily through user code. Users desiring the fallback option of having more menus and buttons available may wish to install RStudio or a similar program that adds a point-and-click front end to R, but a knowledge of the syntax is essential. Figure 1.3 shows the window associated with the Mac version of RStudio.²

Users can submit their code either through script files (the recommended choice, described in Sect. 1.3) or on the command line displayed at the bottom of the R console. In Fig. 1.2, the prompt looks like this:

>

¹The names of these files change as new versions of R are released. As of this printing, the respective files are `R-3.1.2-snowleopard.pkg` or `R-3.1.2-mavericks.pkg` for various versions of Mac OS X and `R-3.1.2-win.exe` for Windows. Linux users will find it easier to install from a terminal. Terminal code is available by following the *Download R for Linux* link on the CRAN page, then choosing a Linux distribution on the next page, and using the terminal code listed on the resulting page. At the time of printing, Debian, various forms of Red Hat, OpenSUSE, and Ubuntu are all supported.

²RStudio is available at <http://www.rstudio.com>.



```
R version 3.1.2 (2014-10-31) -- "Pumpkin Helmet"
Copyright (C) 2014 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin10.8.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[R.app GUI 1.65 (6833) x86_64-apple-darwin10.8.0]

[History restored from /Users/jamie/.Rapp.history]

>
```

Fig. 1.2 R Console for a new session

Whenever typing code directly into the command prompt, if a user types a single command that spans multiple lines, then the command prompt turns into a plus sign (+) to indicate that the command is not complete. The plus sign does not indicate any problem or error, but just reminds the user that the previous command is not yet complete. The cursor automatically places itself there so the user can enter commands.

In the electronic edition of this book, input syntax and output printouts from R will be color coded to help distinguish what the user should type in a script file from what results to expect. Input code will be written in **blue teletype font**. R output will be written in **black teletype font**. Error messages that R returns will be written in **red teletype font**. These colors correspond to the color coding R uses for input and output text. While the colors may not be visible in the print edition, the book's text also will distinguish inputs from outputs. Additionally, names of variables will be written in **bold**. Conceptual keywords from statistics and programming, as well as emphasized text, will be written in *italics*. Finally, when the meaning of a command is not readily apparent, identifying initials will be underlined and bolded in the text. For instance, the **lm** command stands for **linear model**.

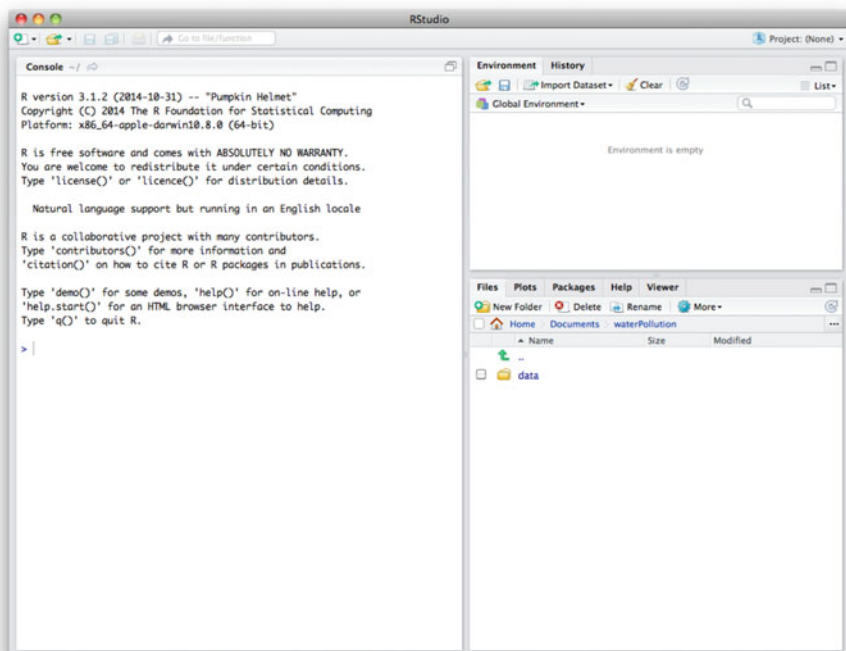


Fig. 1.3 Open window from an RStudio session

When writing R syntax on the command line or in a script file, users should bear a few important preliminaries in mind:

- Expressions and commands in R are case-sensitive. For example, the function `var` returns the variance of a variable: a simple function discussed in Chap. 4. By contrast, the `VAR` command from the `vars` library estimates a Vector Autoregression model—an advanced technique discussed in Chap. 9. Similarly, if a user names a dataset `mydata`, then it cannot be called with the names `MyData`, `MyData`, `MYDATA`, or `myData`. R would assume each of these names indicates a different meaning.
- Command lines do not need to be separated by any special character like a semicolon as in Limdep, SAS, or Gauss. A simple hard return will do.
- R ignores anything following the pound character (`#`) as a comment. This applies when using the command line or script files, but is especially useful when saving notes in script files for later use.
- An object name must start with an alphabetical character, but may contain numeric characters thereafter. A period may also form part of the name of an object. For example, `x.1` is a valid name for an object in R.
- You can use the arrow keys on the keyboard to scroll back to previous commands. One push of the up arrow recalls the previously entered command and places it in

the command line. Each additional push of the arrow moves to a command prior to the one listed in the command line, while the down arrow calls the command following the one listed in the command line.

Aside from this handful of important rules, the command prompt in R tends to behave in an intuitive way, returning responses to input commands that could be easily guessed. For instance, at its most basic level R functions as a high-end calculator. Some of the key *arithmetic* commands are: addition (+), subtraction (-), multiplication (*), division (/), exponentiation (^), the modulo function (%%), and integer division (%/%). Parentheses () specify the order of operations. For example, if we type the following input:

```
(3+5/78)^3*7
```

Then R prints the following output:

```
[1] 201.3761
```

As another example, we could ask R what the remainder is when dividing 89 by 13 using the modulo function:

```
89%%13
```

R then provides the following answer:

```
[1] 11
```

If we wanted R to perform integer division, we could type:

```
89%/13
```

Our output answer to this is:

```
[1] 6
```

The `options` command allows the user to tweak attributes of the output. For example, the `digits` argument offers the option to adjust how many digits are displayed. This is useful, for instance, when considering how precisely you wish to present results on a table. Other useful built-in functions from algebra and trigonometry include: `sin(x)`, `cos(x)`, `tan(x)`, `exp(x)`, `log(x)`, `sqrt(x)`, and `pi`. To apply a few of these functions, first we can expand the number of digits printed out, and then ask for the value of the constant π :

```
options(digits=16)
pi
```

R accordingly prints out the value of π to 16 digits:

```
[1] 3.141592653589793
```

We also may use commands such as `pi` to insert the value of such a constant into a function. For example, if we wanted to compute the sine of a $\frac{\pi}{2}$ radians (or 90°) angle, we could type:

```
sin(pi/2)
```

R correctly prints that $\sin(\frac{\pi}{2}) = 1$:

```
[1] 1
```

1.3 Saving Input and Output

When analyzing data or programming in R, a user will never get into serious trouble provided he or she follows two basic rules:

1. Always leave the original datafiles intact. Any revised version of data should be written into a *new* file. If you are working with a particularly large and unwieldy dataset, then write a short program that winnows-down to what you need, save the cleaned file separately, and then write code that works with the new file.
2. Write all input code in a script that is saved. Users should usually avoid writing code directly into the console. This includes code for cleaning, recoding, and reshaping data as well as for conducting analysis or developing new programs.

If these two rules are followed, then the user can always recover his or her work up to the point of some error or omission. So even if, in data management, some essential information is dropped or lost, or even if a journal reviewer names a predictor that a model should add, the user can always retrace his or her steps. By calling the original dataset with the saved program, the user can make *minor* tweaks to the code to incorporate a new feature of analysis or recover some lost information. By contrast, if the original data are overwritten or the input code is not saved, then the user likely will have to restart the whole project from the beginning, which is a waste of time.

A *script file* in R is simply plain text, usually saved with the suffix `.R`. To create a new script file in R, simply choose *File→New Document* in the drop down menu to open the document. Alternatively, the console window shown in Fig. 1.2 shows an icon that looks like a blank page at the top of the screen (second icon from the right). Clicking on this will also create a new R script file. Once open, the normal *Save* and *Save As* commands from the *File* menu apply. To open an existing R script, choose *File→Open Document* in the drop down menu, or click the icon at the top of the screen that looks like a page with writing on it (third icon from the right in Fig. 1.2). When working with a script file, any code within the file can be executed in the console by simply highlighting the code of interest, and typing the keyboard shortcut `Ctrl+R` in Windows or `Cmd+Return` in Mac. Besides the default script file editor, more sophisticated text editors such as Emacs and RWinEdt also are available.

The product of any R session is saved in the *working directory*. The working directory is the default file path for all files the user wants to read in or write out to. The command `getwd` (meaning **get** **w**orking **d**irectory) will print R's current working directory, while `setwd` (**set** **w**orking **d**irectory) allows you to change the working directory as desired. Within a Windows machine the syntax for checking, and then setting, the working directory would look like this:

```
getwd()
setwd("C:/temp/")
```

This now writes any output files, be they data sets, figures, or printed output to the folder `temp` in the `C:` drive. Observe that **R** expects forward slashes to designate subdirectories, which contrasts from Windows's typical use of backslashes. Hence, specifying `C:/temp/` as the working directory points to `C:\temp\` in normal Windows syntax. Meanwhile for Mac or Unix, setting a working directory would be similar, and the path directory is printed exactly as these operating systems designate them with forward slashes:

```
setwd("/Volumes/flashdisk/temp")
```

Note that `setwd` can be called multiple times in a session, as needed. Also, specifying the full path for any file overrides the working directory.

To *save output* from your session in **R**, try the `sink` command. As a general computing term, a **sink** is an output point for a program where data or results are written out. In **R**, this term accordingly refers to a file that records all of our printed output. To save your session's output to the file `Rintro.txt` within the working directory type:

```
sink("Rintro.txt")
```

Alternatively, if we wanted to override the working directory, in Windows for instance, we could have instead typed:

```
sink("C:/myproject/code/Rintro.txt")
```

Now that we have created an output file, any output that normally would print to the console will instead print to the file `Rintro.txt`. (For this reason, in a first run of new code, it is usually advisable to allow output to print to the screen and then rerun the code later to print to a file.) The `print` command is useful for creating output that can be easily followed. For instance, the command:

```
print("The mean of variable x is...")
```

will print the following in the file `Rintro.txt`:

```
[1] "The mean of variable x is..."
```

Another useful printing command is the `cat` command (short for **catenate**, to connect things together), which lets you mix objects in **R** with text. As a preview of simulation tools described in Chap. 11, let us create a variable named `x` by means of simulation:

```
x <- rnorm(1000)
```

By way of explanation: this syntax draws randomly 1000 times from a standard normal distribution and assigns the values to the vector `x`. Observe the arrow (`<-`), formed with a *less than* sign and a *hyphen*, which is **R**'s assignment operator. Any time we assign something with the arrow (`<-`) the name on the left (`x` in this case) allows us to recall the result of the operation on the right (`rnorm(1000)`)

in this case).³ Now we can print the mean of these 1000 draws (which should be close to 0 in this case) to our output file as follows:

```
cat("The mean of variable x is...", mean(x), "\n")
```

With this syntax, objects from R can be embedded into the statement you print. The character `\n` puts in a carriage return. You also can print any statistical output using the either `print` or `cat` commands. Remember, your output does not go to the log file unless you use one of the print commands. Another option is to simply copy and paste results from the R console window into Word or a text editor. To turn off the sink command, simply type:

```
sink()
```

1.4 Work Session Management

A key feature of R is that it is an *object-oriented* programming language. Variables, data frames, models, and outputs are all stored in memory as *objects*, or identified (and named) locations in memory with defined features. R stores in working memory any object you create using the name you define whenever you load data into memory or estimate a model. To list the objects you have created in a session use either of the following commands:

```
objects()
ls()
```

To remove all the objects in R type:

```
rm(list=ls(all=TRUE))
```

As a rule, it is a good idea to use the `rm` command at the start of any new program. If the previous user saved his or her workspace, then they may have used objects sharing the same name as yours, which can create confusion.

To quit R either close the console window or type:

```
q()
```

At this point, R will ask if you wish to save the workspace image. Generally, it is advisable not to do this, as starting with a clean slate in each session is more likely to prevent programming errors or confusion on the versions of objects loaded in memory.

Finally, in many R sessions, we will need to load *packages*, or batches of code and data offering additional functionality not written in R's base code. Throughout this book we will load several packages, particularly in Chap. 8, where

³The arrow (`<-`) is the traditional assignment operator, though a single equals sign (`=`) also can serve for assignments.

our focus will be on example packages written by prominent Political Scientists to implement cutting-edge methods. The necessary commands to load packages are `install.packages`, a command that automatically downloads and installs a package on a user's copy of R, and `library`, a command that loads the package in a given session. Suppose we wanted to install the package `MCMCpack`. This package provides tools for Bayesian modeling that we will use in Chap. 8. The form of the syntax for these commands is:

```
install.packages("MCMCpack")  
library(MCMCpack)
```

Package installation is case and spelling sensitive. R will likely prompt you at this point to choose one of the CRAN mirrors from which to download this package: For faster downloading, users typically choose the mirror that is most geographically proximate. The `install.packages` command only needs to be run once per R installation for a particular package to be available on a machine. The `library` command needs to be run for every session that a user wishes to use the package. Hence, in the next session that we want to use `MCMCpack`, we need only type: `library(MCMCpack)`.

1.5 Resources

Given the wide array of base functions that are available in R, much less the even wider array of functionality created by R packages, a book such as this cannot possibly address everything R is capable of doing. This book should serve as a resource introducing how a researcher can use R as a basic statistics program and offer some general pointers about the usage of packages and programming features. As questions emerge about topics not covered in this space, there are several other resources that may be of use:

- Within R, the *Help* pull down menu (also available by typing `help.start()` in the console) offers several manuals of use, including an “Introduction to R” and “Writing R Extensions.” This also opens an HTML-based search engine of the help files.
- UCLA’s Institute for Digital Research and Education offers several nice tutorials (<http://www.ats.ucla.edu/stat/r/>). The CRAN website also includes a variety of online manuals (<http://www.cran.r-project.org/other-docs.html>).
- Some nice interactive tutorials include `swirl`, which is a package you install in your own copy of R (more information: <http://www.swirlstats.com/>), and Try R, which is completed online (<http://tryr.codeschool.com/>).
- Within the R console, the commands `?`, `help()`, and `help.search()` all serve to find documentation. For instance, `?lm` would find the documentation for the linear model command. Alternatively, `help.search("linear model")` would search the documentation for a phrase.

- To search the internet for information, Rseek (<http://www.rseek.org/>, powered by Google) is a worthwhile search engine that searches only over websites focused on R.
- Finally, Twitter users reference R through the hashtag `#rstats`.

At this point, users should now have R installed on their machine, hold a basic sense of how commands are entered and output is generated, and recognize where to find the vast resources available for R users. In the next six chapters, we will see how R can be used to fill the role of a statistical analysis or econometrics software program.

1.6 Practice Problems

Each chapter will end with a few practice problems. If you have tested all of the code from the in-chapter examples, you should be able to complete these on your own. If you have not done so already, go ahead and install R on your machine for free and try the in-chapter code. Then try the following questions.

1. Compute the following in R:

- (a) -7×2^3
- (b) $\frac{8}{8^2+1}$
- (c) $\cos \pi$
- (d) $\sqrt{81}$
- (e) $\ln e^4$

- 2. What does the command `cor` do? Find documentation about it and describe what the function does.
- 3. What does the command `runif` do? Find documentation about it and describe what the function does.
- 4. Create a vector named `x` that consists of 1000 draws from a standard normal distribution, using code just like you see in Sect. 1.3. Create a second vector named `y` in the same way. Compute the correlation coefficient between the two vectors. What result do you get, and why do you get this result?
- 5. Get a feel for how to decide when add-on packages might be useful for you. Log in to <http://www.rseek.org> and look up what the `stringr` package does. What kinds of functionality does this package give you? When might you want to use it?

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:

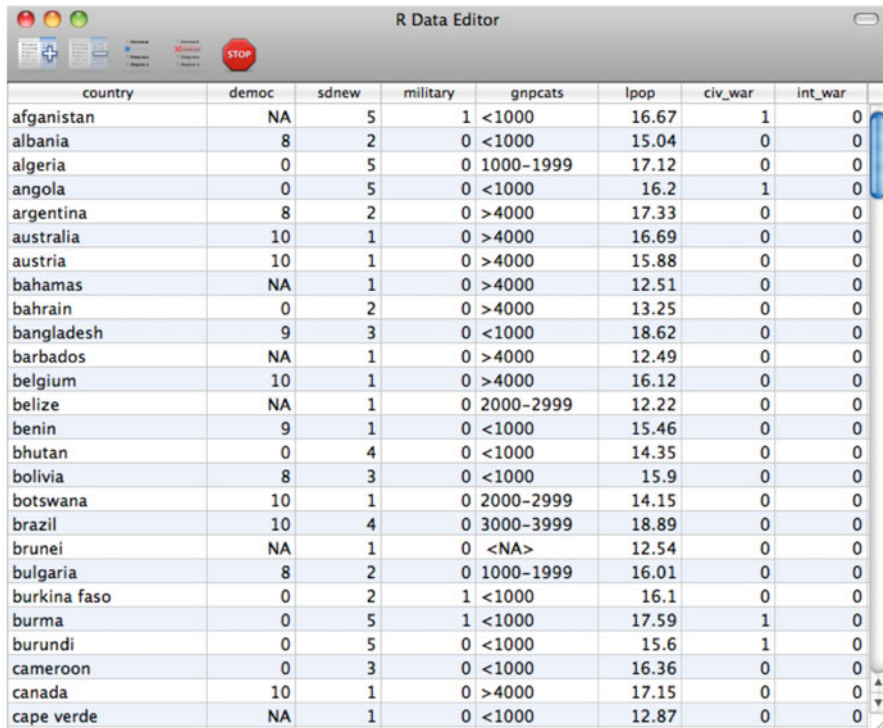
```
names(hmnrghts)
```

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?

Chapter 3

Visualizing Data

Visually presenting data and the results of models has become a centerpiece of modern political analysis. Many of Political Science's top journals, including the *American Journal of Political Science*, now ask for figures in lieu of tables whenever both can convey the same information. In fact, Kestellec and Leoni (2007) make the case that figures convey empirical results better than tables. Cleveland (1993) and Tufte (2001) wrote two of the leading volumes that describe the elements of good quantitative visualization, and Yau (2011) has produced a more recent take on graphing. Essentially these works serve as style manuals for graphics.¹ Beyond the suggestions these scholars offer for the sake of readers, viewing one's own data visually conveys substantial information about the data's univariate, bivariate, and multivariate features: Does a variable appear skewed? Do two variables substantively appear to correlate? What is the proper functional relationship between variables? How does a variable change over space or time? Answering these questions for oneself as an analyst and for the reader generally can raise the quality of analysis presented to the discipline.

On the edge of this graphical movement in quantitative analysis, R offers state-of-the-art data and model visualization. Many of the commercial statistical programs have tried for years to catch up to R's graphical capacities. This chapter showcases these capabilities, turning first to the `plot` function that is automatically available as part of the `base` package. Second, we discuss some of the other graphing commands offered in the `base` library. Finally, we turn to the `lattice` library, which allows the user to create Trellis Graphics—a framework for visualization

Electronic supplementary material: The online version of this chapter (doi: [10.1007/978-3-319-23446-5_3](https://doi.org/10.1007/978-3-319-23446-5_3)) contains supplementary material, which is available to authorized users.

¹Other particularly key historical figures in the development of graphical measures include Halley (1686), Playfair (1786/2005), and Tukey (1977). A more comprehensive history is presented by Beniger and Robyn (1978).

developed by Becker, Cleveland, and others to put Cleveland's (1993) suggestions into practice. Although space does not permit it here, users are also encouraged to look up the `ggplot2` packages, which offers additional graphing options. Chang (2013), in particular, offers several examples of graphing with `ggplot2`.

In this chapter, we work with two example datasets. The first is on health lobbying in the 50 American states, with a specific focus on the proportion of firms from the health finance industry that are registered to lobby (Lowery et al. 2008). A key predictor variable is the total number of health finance firms open for business, which includes organizations that provide health plans, business services, employer health coalitions, and insurance. The dataset also includes the lobby participation rate by state, or number of lobbyists as a proportion of the number of firms, not only in health finance but for all health-related firms and in six other subareas. These are cross-sectional data from the year 1997. The complete variable list is as follows:

stno: Numeric index from 1–50 that orders the states alphabetically.

raneyfolded97: Folded Ranney Index of state two-party competition in 1997.²

healthagenda97: Number of bills related to health considered by the state legislature in 1997.

supplybusiness: Number of health finance establishments.

businesssupplysq: Squared number of health finance establishments.

partratebusiness: Lobby participation rate for health finance—number of registrations as a percentage of the number of establishments.

predictbuspartrate: Prediction of health finance participation rate as a quadratic function of number of health finance establishments. (No control variables in the prediction.)

partratetotalhealth: Lobby participation rate for all of health care (including seven subareas).

partratedpc: Lobby participation rate for direct patient care.

partratepharmprod: Lobby participation rate for drugs and health products.

partrateprofessionals: Lobby participation rate for health professionals.

partrateadvo: Lobby participation rate for health advocacy.

partrategov: Lobby participation rate for local government.

rnmedschoolpartrate: Lobby participation rate for health education.

Second, we analyze Peake and Eshbaugh-Soha's (2008) data on the number of television news stories related to energy policy in a given month. In this data frame, the variables are:

Date: Character vector of the month and year observed.

Energy: Number of energy-related stories broadcast on nightly television news by month.

Unemploy: The unemployment rate by month.

Approval: Presidential approval by month.

oilc: Price of oil per barrel.

²Nebraska and North Carolina are each missing observations of the Ranney index.

freeze1: An indicator variable coded 1 during the months of August–November 1971, when wage and price freezes were imposed. Coded 0 otherwise.

freeze2: An indicator variable coded 1 during the months of June–July 1973, when price wage and price freezes were imposed. Coded 0 otherwise.

embargo: An indicator variable coded 1 during the months of October 1973–March 1974, during the Arab oil embargo. Coded 0 otherwise.

hostages: An indicator variable coded 1 during the months of November 1979–January 1981, during the Iran hostage crisis. Coded 0 otherwise.

Presidential speeches: Additional indicators are coded as 1 during the month a president delivered a major address on energy policy, and 0 otherwise. The indicators for the respective speeches are called: **rmn1173**, **rmn1173a**, **grf0175**, **grf575**, **grf575a**, **jec477**, **jec1177**, **jec479**, **grf0175s**, **jec479s**, and **jec477s**.

3.1 Univariate Graphs in the base Package

As a first look at our data, displaying a single variable graphically can convey a sense of the distribution of the data, including its mode, dispersion, skew, and kurtosis. The `lattice` library actually offers a few more commands for univariate visualization than `base` does, but we start with the major built-in univariate commands. Most graphing commands in the `base` package call the `plot` function, but `hist` and `boxplot` are noteworthy exceptions.

The `hist` command is useful to simply gain an idea of the relative frequency of several common values. We start by loading our data on energy policy television news coverage. Then we create a **hist**ogram of this time series of monthly story counts with the `hist` command. First, download Peake and Eshbaugh-Soha's data on energy policy coverage, the file named `PESenergy.csv`. The file is available from the Dataverse named on page vii or the chapter content link on page 33. You may need to use `setwd` to point `R` to the folder where you have saved the data. After this, run the following code:

```
pres.energy<-read.csv("PESenergy.csv")
hist(pres.energy$Energy,xlab="Television Stories",main="")
abline(h=0,col='gray60')
box()
```

The result this code produces is presented in Fig. 3.1. In this code, we begin by reading Peake and Eshbaugh-Soha's (2008) data. The data file itself is a comma-separated values file with a header row of variable names, so the defaults of `read.csv` suit our purposes. Once the data are loaded, we plot a histogram of our variable of interest using the `hist` command: `pres.energy$Energy` calls the variable of interest from its data frame. We use the `xlab` option, which allows us to define the label `R` prints on the horizontal axis. Since this axis shows us the values of the variable, we simply wish to see the phrase “Television Stories,” describing in brief what these numbers mean. The `main` option defines a title printed over the top of the figure. In this case, the only way to impose a blank title is to include quotes with no content between them. A neat feature of plotting in the `base` package is

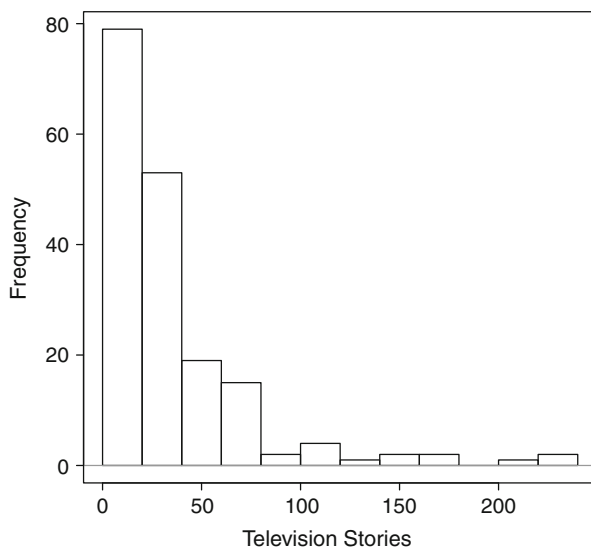


Fig. 3.1 Histogram of the monthly count of television news stories related to energy

that a few commands can add additional information to a plot that has already been drawn. The `abline` command is a flexible and useful tool. (The name **a-b line** refers to the linear formula $y = a + bx$. Hence, this command can draw lines with a slope and intercept, or it can draw a horizontal or vertical line.) In this case, `abline` adds a horizontal line along the 0 point on the vertical axis, hence $h=0$. This is added to clarify where the base of the bars in the figure is. Finally, the `box()` command encloses the whole figure in a box, often useful in printed articles for clarifying where graphing space ends and other white space begins. As the histogram shows, there is a strong concentration of observations at and just above 0, and a clear positive skew to the distribution. (In fact, these data are reanalyzed in Fogarty and Monogan (2014) precisely to address some of these data features and discuss useful means of analyzing time-dependent media counts.)

Another univariate graph is a box-and-whisker plot. R allows us to obtain this solely for the single variable, or for a subset of the variable based on some other available measure. First drawing this for a single variable:

```
boxplot(pres.energy$Energy, ylab="Television Stories")
```

The result of this is presented in panel (a) of Fig. 3.2. In this case, the values of the monthly counts are on the vertical axis; hence, we use the `ylab` option to label the vertical axis (or **y-axis label**) appropriately. In the figure, the bottom of the box represents the first quartile value (25th percentile), the large solid line inside the box represents the median value (second quartile, 50th percentile), and the top of the box represents the third quartile value (75th percentile). The whiskers, by default, extend to the lowest and highest values of the variable that are no more than 1.5 times the interquartile range (or difference between third and first quartiles) away

from the box. The purpose of the whiskers is to convey the range over which the bulk of the data fall. Data falling outside of this range are portrayed as dots at their respective values. This boxplot fits our conclusion from the histogram: small values including 0 are common, and the data have a positive skew.

Box-and-whisker plots also can serve to offer a sense of the conditional distribution of a variable. For our time series of energy policy coverage, the first major event we observe is Nixon's November 1973 speech on the subject. Hence, we might create a simple indicator where the first 58 months of the series (through October 1973) are coded 0, and the remaining 122 months of the series (November 1973 onward) are coded 1. Once we do this, the `boxplot` command allows us to condition on a variable:

```
pres.energy$post.nixon<-c(rep(0,58),rep(1,122))
boxplot(pres.energy$Energy~pres.energy$post.nixon,
        axes=F,ylab="Television Stories")
axis(1,at=c(1,2),labels=c("Before Nov. 1973",
                           "After Nov. 1973"))
axis(2)
box()
```

This output is presented in panel (b) of Fig. 3.2. The first line of code defines our pre v. post November 1973 variable. Notice here that we again define a vector with `c`. Within `c`, we use the `rep` command (for **repeat**). So `rep(0,58)` produces 58 zeroes, and `rep(1,122)` produces 122 ones. The second line draws our boxplots, but we add two important caveats relative to our last call to `boxplot`: First, we list `pres.energy$Energy~pres.energy$post.nixon` as our data argument. The argument before the tilde (`~`) is the variable for which we want the distribution, and the argument afterward is the conditioning variable. Second, we add the `axes=F` command. (We also could write `axes=FALSE`, but **R** accepts `F` as an abbreviation.) This gives us more control over how the horizontal and

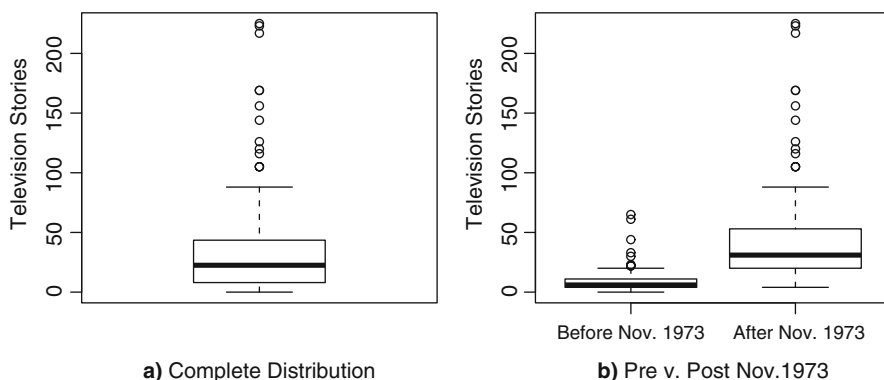


Fig. 3.2 Box-and-whisker plots of the distribution of the monthly count of television new stories related to energy. Panel (a) shows the complete distribution, and panel (b) shows the distributions for the subsets before and after November 1973

vertical axes are presented. In the subsequent command, we add axis 1 (the bottom horizontal axis), adding text labels at the tick marks of 1 and 2 to describe the values of the conditioning variable. Afterward, we add axis 2 (the left vertical axis), and a box around the whole figure. Panel (b) of Fig. 3.2 shows that the distribution before and after this date is fundamentally different. Much smaller values persist before Nixon's speech, while there is a larger mean and a greater spread in values afterward. Of course, this is only a first look and the effect of Nixon's speech is confounded with a variety of factors—such as the price of oil, presidential approval, and the unemployment rate—that contribute to this difference.

3.1.1 Bar Graphs

Bar graphs can be useful whenever we wish to illustrate the value some statistic takes for a variety of groups as well as for visualizing the relative proportions of nominal or ordinal measured data. For an example of barplots, we turn now to the other example data set from this chapter, on health lobbying in the 50 American states. Lowery et al. offer a bar graph of the means across all states of the lobbying participation rate—or number of lobbyists as a percentage of number of firms—for all health lobbyists and for seven subgroups of health lobbyists (2008, Fig. 3). We can recreate that figure in R by taking the means of these eight variables and then applying the `barplot` function to the set of means. First we must load the data. To do this, download Lowery et al.'s data on lobbying, the file named `constructionData.dta`. The file is available from the Dataverse named on page vii or the chapter content link on page 33. Again, you may need to use `setwd` to point R to the folder where you have saved the data. Since these data are in Stata format, we must use the `foreign` library and then the `read.dta` command:

```
library(foreign)
health.fin<-read.dta("constructionData.dta")
```

To create the actual figure itself, we can create a subset of our data that only includes the eight predictors of interest and then use the `apply` function to obtain the mean of each variable.

```
part.rates<-subset(health.fin,select=c(
  partratetotalhealth,partratedpc,
  partratepharmprod,partrateprofessionals,partrateadvo,
  partratebusiness,partrategov,rnmedschoolpartrate))
lobby.means<-apply(part.rates,2,mean)
names(lobby.means)<-c("Total Health Care",
  "Direct Patient Care","Drugs/Health Products",
  "Health Professionals","Health Advocacy","
  Health Finance","Local Government","Health Education")
```

In this case, `part.rates` is our subsetted data frame that only includes the eight lobby participation rates of interest. On the last line, the `apply` command allows us to take a matrix or data frame (`part.rates`) and apply a function of interest (`mean`) to either the rows or the columns of the data frame. We want the mean of

each variable, and the columns of our data set represent the variables. The 2 that is the second component of this command therefore tells `apply` that we want to apply mean to the *columns* of our data. (By contrast, an argument of 1 would apply to the *rows*. Row-based computations would be handy if we needed to compute some new quantity for each of the 50 states.) If we simply type `lobby.means` into the R console now, it will print the eight means of interest for us. To set up our figure in advance, we can attach an English-language name to each quantity that will be reported in our figure's margin. We do this with the `names` command, and then assign a vector with a name for each quantity.

To actually draw our bar graph, we use the following code:

```
par(mar=c(5.1, 10, 4.1, 2.1))
barplot(lobby.means,xlab="Percent Lobby Registration",
        xlim=c(0,26),horiz=T,cex.names=.8,las=1)
text(x=lobby.means,y=c(.75,1.75,3,4.25,5.5,6.75,8,9),
     labels=paste(round(lobby.means,2)),pos=4)
box()
```

The results are plotted in Fig. 3.3. The first line calls the `par` command, which allows the user to change a wide array of defaults in the graphing space. In our

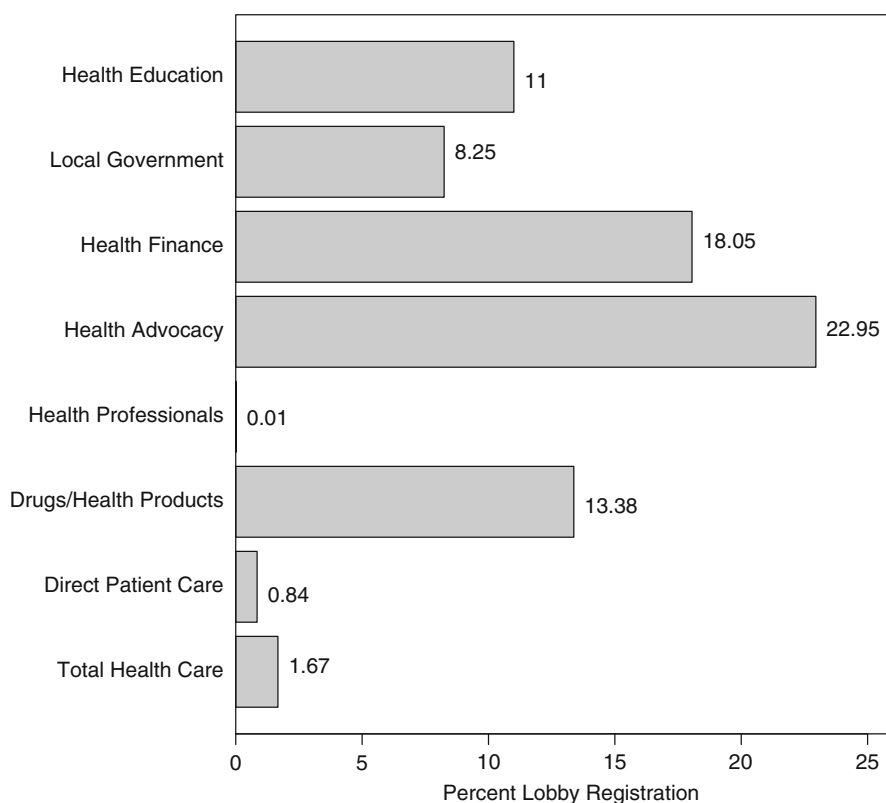


Fig. 3.3 Bar graph of the mean lobbying participation rate in health care and in seven sub-guilds across the 50 U.S. states, 1997

case, we need a bigger left margin, so we used the `mar` option to change this, setting the second value to the relatively large value of 10. (In general, the margins are listed as bottom, left, top, then right.) Anything adjusted with `par` is reset to the defaults after the plotting window (or device, if writing directly to a file) is closed. Next, we actually use the `barplot` command. The main argument is `lobby.means`, which is the vector of variable means. The default for `barplot` is to draw a graph with vertical lines. In this case, though, we set the option `horiz=T` to get horizontal bars. We also use the options `cex.names` (**c**haracter **e**xpansion for axis **n**ames) and `las=1` (**l**abel **a**xis **s**tyl**e**) to shrink our bar labels to 80 % of their default size and force them to print horizontally, respectively.³ The `xlab` command allows us to describe the variable for which we are showing the means, and the `xlim` (**x**-axis **l**imits) command allows us to set the space of our horizontal axis. Finally, we use the `text` command to print the mean for each lobby registration rate at the end of the bar. The `text` command is useful any time we wish to add text to a graph, be these numeric values or text labels. This command takes `x` coordinates for its position along the horizontal axis, `y` coordinates for its position along the vertical axis, and `labels` values for the text to print at each spot. The `pos=4` option specifies to print the text to the right of the given point (alternatively 1, 2, and 3 would specify below, left, and above, respectively), so that our text does not overlap with the bar.

3.2 The `plot` Function

We turn now to `plot`, the workhorse graphical function in the base package. The `plot` command lends itself naturally to bivariate plots. To see the total sum of arguments that one can call using `plot`, type `args(plot.default)`, which returns the following:

```
function (x, y=NULL, type="p", xlim=NULL, ylim=NULL,
  log="", main=NULL, sub=NULL, xlab=NULL, ylab=NULL,
  ann=par("ann"), axes=TRUE, frame.plot=axes,
  panel.first=NULL, panel.last=NULL, asp=NA, ...)
```

Obviously there is a lot going on underneath the generic `plot` function. For the purpose of getting started with figure creation in **R** we want to ask what is essential. The answer is straightforward: one variable `x` must be specified. Everything else has either a default value or is not essential. To start experimenting with `plot`, we continue to use the 1997 state health lobbying data loaded in Sect. 3.1.1.

With `plot`, we can plot the variables separately with the command `plot(varname)`, though this is definitively less informative than the kinds of

³The default `las` value is 0, which prints labels parallel to the axis. 1, our choice here, prints them horizontally. 2 prints perpendicular to the axis, and 3 prints them vertically.

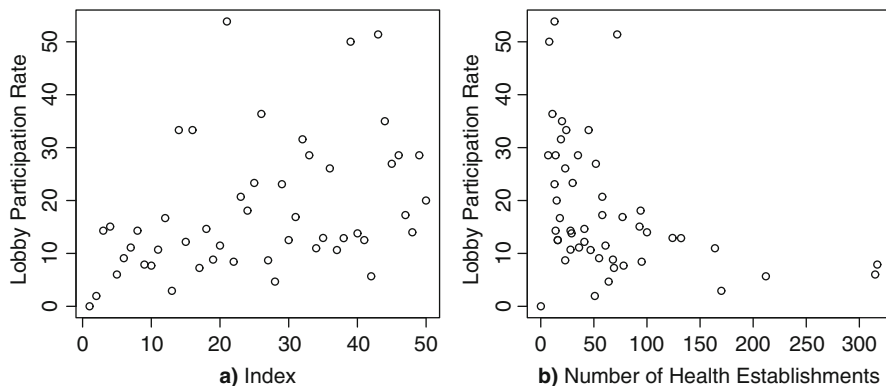


Fig. 3.4 Lobby participation rate of the health finance industry alone and against the number of health finance business establishments. (a) Index. (b) Number of Health Establishments

graphs just presented in Sect. 3.1. That said, if we simply wanted to see all of the observed values of the lobby participation rate by state of health finance firms (`partratebusiness`), we simply type:

```
plot(health.fin$partratebusiness,
     ylab="Lobby Participation Rate")
```

Figure 3.4a is returned in the R graphics interface. Note that this figure plots the lobby participation rate against the row number in the data frame: With cross-sectional data this index is essentially meaningless. By contrast, if we were studying time series data, and the data were sorted on time, then we could observe how the series evolves over time. Note that we use the `ylab` option because otherwise the default will label our vertical axis with the tacky-looking `health.fin$partratebusiness`. (Try it, and ask yourself what a journal editor would think of how the output looks.)

Of course, we are more often interested in bivariate relationships. We can explore these easily by incorporating a variable x on the horizontal axis (usually an *independent* variable) and a variable y on the vertical axis (usually a *dependent* variable) in the call to `plot`:

```
plot(y=health.fin$partratebusiness,x=health.fin$supplybusiness,
     ylab="Lobby Participation Rate",
     xlab="Number of Health Establishments")
```

This produces Fig. 3.4b, where our horizontal axis is defined by the number of health finance firms in a state, and the vertical axis is defined by the lobby participation rate of these firms in the respective state. This graph shows what appears to be a decrease in the participation rate as the number of firms rises, perhaps in a curvilinear relationship.

One useful tool is to plot the functional form of a bivariate model onto the scatterplot of the two variables. In the case of Fig. 3.4b, we may want to compare

how a linear function versus a quadratic, or squared, function of the number of firms fits the outcome of lobby participation rate. To do this, we can fit two linear regression models, one that includes a linear function of number of firms, and the other that includes a quadratic function. Additional details on regression models are discussed later on in Chap. 6. Our two models in this case are:

```
finance.linear<-lm(partratebusiness~supplybusiness,
  data=health.fin)
summary(finance.linear)
finance.quadratic<-lm(partratebusiness~supplybusiness+
  I(supplybusiness^2),data=health.fin)
summary(finance.quadratic)
```

The `lm` (**linear model**) command fits our models, and the `summary` command summarizes our results. Again, details of `lm` will be discussed in Chap. 6. With the model that is a linear function of number of firms, we can simply feed the name of our fitted model (`finance.linear`) into the command `abline` in order to add our fitted regression line to the plot:

```
plot(y=health.fin$partratebusiness,x=health.fin$supplybusiness,
  ylab="Lobby Participation Rate",
  xlab="Number of Health Establishments")
abline(finance.linear)
```

As mentioned before, the `abline` command is particularly flexible. A user can specify `a` as the intercept of a line and `b` as the slope. A user can specify `h` as the vertical-axis value where a horizontal line is drawn, or `v` as the horizontal-axis value where a vertical line is drawn. Or, in this case, a regression model with one predictor can be inserted to draw the best-fitting regression line. The results are presented in Fig. 3.5a.

Alternatively, we could redraw this plot with the quadratic relationship sketched on it. Unfortunately, despite `abline`'s flexibility, it cannot draw a quadratic

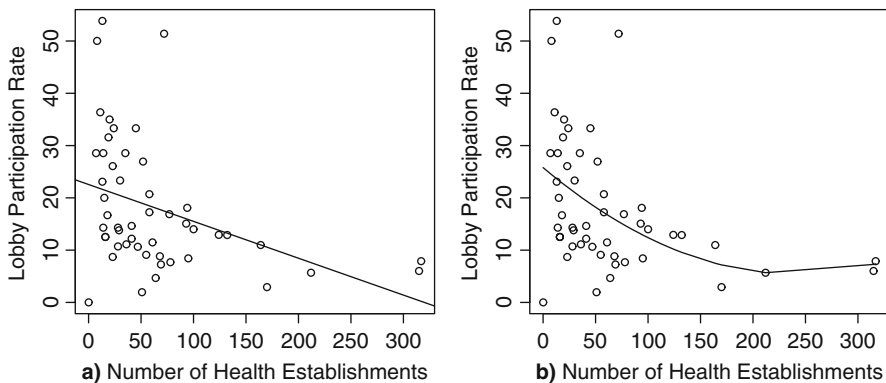


Fig. 3.5 Lobby participation rate of the health finance industry against the number of health establishments, linear and quadratic models. (a) Linear function. (b) Quadratic function

relationship by default. The easiest way to plot a complex functional form is to save the predicted values from the model, reorder the data based on the predictor of interest, and then use the `lines` function to add a connected line of all of the predictions. Be sure the data are properly ordered on the predictor, otherwise the line will appear as a jumbled mess. The code in this case is:

```
plot(y=health.fin$partratebusiness,x=health.fin$supplybusiness,
     ylab="Lobby Participation Rate",
     xlab="Number of Health Establishments")
finance.quadratic<-lm(partratebusiness~supplybusiness+
  I(supplybusiness^2), data=health.fin)
health.fin$quad.fit<-finance.quadratic$fitted.values
health.fin<-health.fin[order(health.fin$supplybusiness),]
lines(y=health.fin$quad.fit,x=health.fin$supplybusiness)
```

This outcome is presented in Fig. 3.5b. While we will not get into `lm`'s details yet, notice that `I(supplybusiness^2)` is used as a predictor. `I` means “as is”, so it allows us to compute a mathematical formula on the fly. After redrawing our original scatterplot, we estimate our quadratic model and save the fitted values to our data frame as the variable `quad.fit`. On the fourth line, we reorder our data frame `health.fin` according to the values of our input variable `supplybusiness`. This is done by using the `order` command, which lists vector indices in order of increasing value. Finally, the `lines` command takes our predicted values as the vertical coordinates (*y*) and our values of the number of firms as the horizontal coordinates (*x*). This adds the line to the plot showing our quadratic functional form.

3.2.1 Line Graphs with `plot`

So far, our analyses have relied on the `plot` default of drawing a scatterplot. In time series analysis, though, a line plot over time is often useful for observing the properties of the series and how it changes over time. (Further information on this is available in Chap. 9.) Returning to the data on television news coverage of energy policy first raised in Sect. 3.1, let us visualize the outcome of energy policy coverage and an input of oil price.

Starting with number of energy stories by month, we create this plot as follows:

```
plot(x=pres.energy$Energy,type="l",axes=F,
     xlab="Month", ylab="Television Stories on Energy")
axis(1,at=c(1,37,73,109,145),labels=c("Jan. 1969",
    "Jan. 1972","Jan. 1975","Jan. 1978","Jan. 1981"),
     cex.axis=.7)
axis(2)
abline(h=0,col="gray60")
box()
```

This produces Fig. 3.6a. In this case, our data are already sorted by month, so if we only specify *x* with no *y*, R will show all of the values in correct temporal

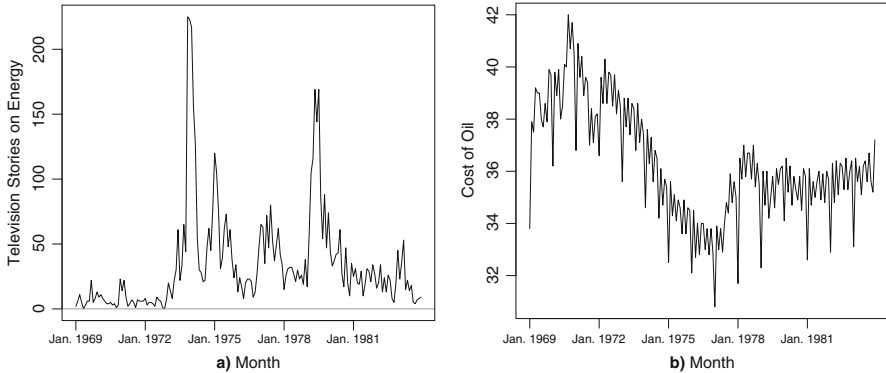


Fig. 3.6 Number of television news stories on energy policy and the price of oil per barrel, respectively, by month. (a) News coverage. (b) Oil price

order.⁴ To designate that we want a line plot instead of a scatterplot of points, we insert the letter `l` in the `type="l"` option. In this case, we have turned off the axes because the default tick marks for month are not particularly meaningful. Instead, we use the `axis` command to insert a label for the first month of the year every 3 years, offering a better sense of real time. Notice that in our first call to `axis`, we use the `cex.axis` option to shrink our labels to 70 % size. This allows all five labels to fit in the graph. (By trial and error, you will see that `R` drops axis labels that will not fit rather than overprint text.) Finally, we use `abline` to show the zero point on the vertical axis, since this is a meaningful number that reflects the complete absence of energy policy coverage in television news. As our earlier figures demonstrated, we see much more variability and a higher mean after the first 4 years. The figure of the price of oil per barrel can be created in a similar way:

```
plot(x=pres.energy$oilc,type="l",axes=F,xlab="Month",
     ylab="Cost of Oil")
axis(1,at=c(1,37,73,109,145),labels=c("Jan. 1969",
    "Jan. 1972","Jan. 1975","Jan. 1978","Jan. 1981"),
     cex.axis=.7)
axis(2)
box()
```

Again, the data are sorted, so only one variable is necessary. Figure 3.6b presents this graph.

⁴Alternatively, though, if a user had some time index in the data frame, a similar plot could be produced by typing something to the effect of: `pres.energy$Time<-1:180;`
`plot(y=pres.energy$Energy,x=pres.energy$Time,type="l").`

3.2.2 Figure Construction with plot: Additional Details

Having tried our hand with plots from the base package, we will now itemize in detail the basic functions and options that bring considerable flexibility to creating figures in R. Bear in mind that R actually offers the useful option of beginning with a blank slate and adding items to the graph bit-by-bit.

The Coordinate System: In Fig. 3.4, we were not worried about establishing the coordinate system because the data effectively did this for us. But often, you will want to establish the dimensions of the figure before plotting anything—especially if you are building up from the blank canvas. The most important point here is that your *x* and *y* must be of the same length. This is perhaps obvious, but missing data can create difficulties that will lead R to balk.

Plot Types: We now want to plot these series, but the `plot` function allows for different types of plots. The different types that one can include within the generic `plot` function include:

- `type="p"` This is the default and it plots the *x* and *y* coordinates as *points*.
- `type="l"` This plots the *x* and *y* coordinates as *lines*.
- `type="n"` This plots the *x* and *y* coordinates as *nothing* (it sets up the coordinate space only).
- `type="o"` This plots the *x* and *y* coordinates as *points and lines* overlaid (i.e., it “overplots”).
- `type="h"` This plots the *x* and *y* coordinates as *histogram-like vertical lines*. (Also called a *spike plot*.)
- `type="s"` This plots the *x* and *y* coordinates as *stair-step like lines*.

Axes: It is possible to turn off the axes, to adjust the coordinate space by using the `xlim` and `ylim` options, and to create your own labels for the axes.

`axes=` Allows you to control whether the axes appear in the figure or not. If you have strong preferences about how your axes are created, you may turn them off by selecting `axes=F` within `plot` and then create your own labels using the separate `axis` command:

- `axis(side=1, at=c(2, 4, 6, 8, 10, 12), labels=c("Feb", "Apr", "June", "Aug", "Oct", "Dec"))`

`xlim=, ylim=` For example, if we wanted to expand the space from the R default, we could enter:

- `plot(x=ind.var, y=dep.var, type="o", xlim=c(-5, 17), ylim=c(-5, 15))`

`xlab="", ylab=""` Creates labels for the *x*- and *y*-axis.

Style: There are a number of options to adjust the style in the figure, including changes in the line type, line weight, color, point style, and more. Some common commands include:

`asp=` Defines the **aspect** ratio of the plot. Setting `asp=1` is a powerful and useful option that allows the user to declare that the two axes are measured on the same scale. See Fig. 5.1 on page 76 and Fig. 8.4 on page 153 as two examples of this option.

`lty=` Selects the type of line (solid, dashed, short-long dash, etc.).

`lwd=` Selects the line width (fat or skinny lines).

`pch=` Selects the plotting symbol, can either be a numbered symbol (`pch=1`) or a letter (`pch="D"`).

`col=` Selects the color of the lines or points in the figure.

`cex=` **C**haracter **e**xpansion factor that adjusts the size of the text and symbols in the figure. Similarly, `cex.axis` adjusts axis annotation size, `cex.lab` adjusts font size for axis labels, `cex.main` adjusts the font size of the title, and `cex.sub` adjusts subtitle font size.

Graphing Parameters: The `par` function brings added functionality to plotting in R by giving the user control over the graphing **parameters**. One noteworthy feature of `par` is that it allows you to plot multiple calls to `plot` in a single graphic. This is accomplished by selecting `par(new=T)` while a plot window (or device) is still open and before the next call to `plot`. Be *careful*, though. Any time you use this strategy, include the `xlim` and `ylim` commands in each call to make sure the graphing space stays the same. Also be careful that graph margins are not changing from one call to the next.

3.2.3 Add-On Functions

There are also a number of add-on functions that one can use once the basic coordinate system has been created using `plot`. These include:

`arrows(x1, y1, x2, y2)` Create arrows within the plot (useful for labeling particular data points, series, etc.).

`text(x1, x2, "text")` Create text within the plot (modify size of text using the character expansion option `cex`).

`lines(x, y)` Create a plot that connects lines.

`points(x, y)` Create a plot of points.

`polygon()` Create a polygon of any shape (rectangles, triangles, etc.).

`legend(x, y, at = c("", ""), labels=c("", ""))` Create a legend to identify the components in the figure.

`axis(side)` Add an axis with default or customized labels to one of the sides of a plot. Set the side to 1 for bottom, 2 for left, 3 for top, and 4 for right.

`mtext(text, side)` Command to add **m**argin **t**ext. This lets you add an axis label to one of the sides with more control over how the label is presented. See the code that produces Fig. 7.1 on page 108 for an example of this.

3.3 Using lattice Graphics in R

As an alternative to the base graphics package, you may want to consider the `lattice` add-on package. These produce `trellis` graphics from the `S` language, which tend to make better displays of grouped data and numerous observations. A few nice features of the `lattice` package are that the graphs have viewer-friendly defaults, and the commands offer a `data=` option that does not require the user to list the data frame with every call to a variable.

To start, the first time we use the `lattice` library, we must install it. Then, on every reuse of the package, we must call it with the `library` command.

```
install.packages("lattice")
library(lattice)
```

To obtain a scatterplot similar to the one we drew with `plot`, this can be accomplished in `lattice` using the `xyplot` command:

```
xyplot(partratebusiness~supplybusiness,data=health.fin,
       col="black",ylab="Lobby Participation Rate",
       xlab="Number of Health Establishments")
```

Figure 3.7a displays this graph. The syntax differs from the `plot` function somewhat: In this case, we can specify an option, `data=health.fin`, that allows us to type the name of the relevant data frame once, rather than retype it for each variable. Also, both variables are listed together in a single argument using the form, `vertical.variable~horizontal.variable`. In this case, we also specify the option, `col="black"` for the sake of producing a black-and-white figure. By default `lattice` colors results cyan in order to allow readers to easily separate data information from other aspects of the display, such as axes and labels (Becker et al. 1996, p. 153). Also, by default, `xyplot` prints tick marks on the third and fourth axes to provide additional reference points for the viewer.

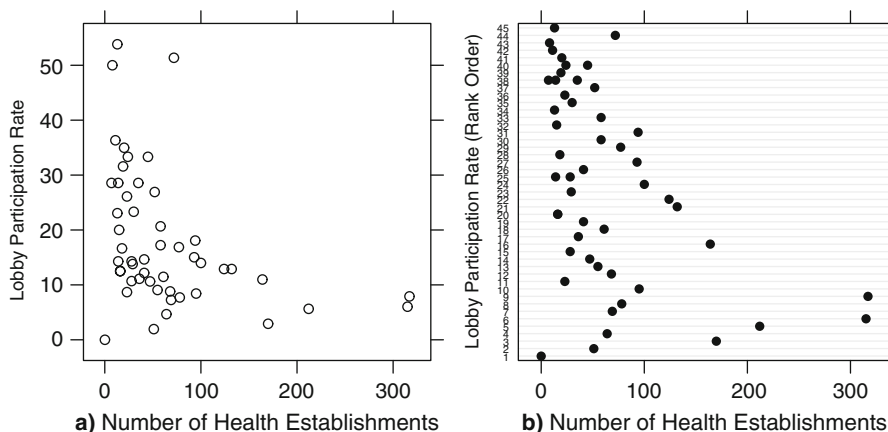


Fig. 3.7 Lobby participation rate of the health finance industry against the number of health establishments, (a) scatterplot and (b) dotplot

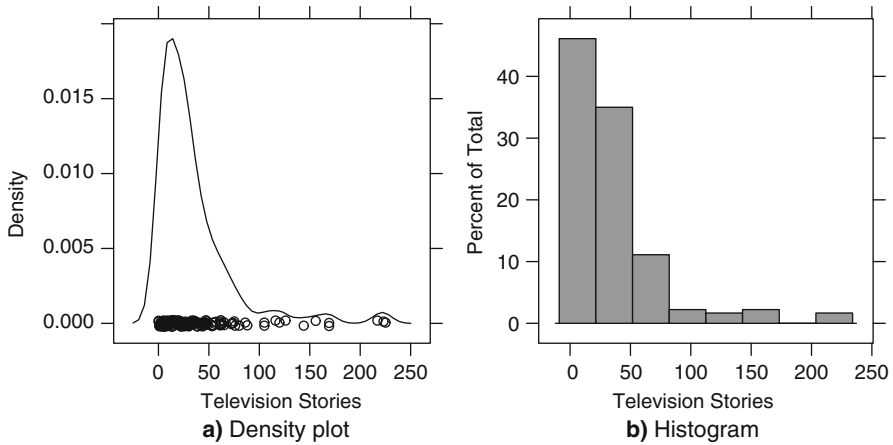


Fig. 3.8 (a) Density plot and (b) histogram showing the univariate distribution of the monthly count of television news stories related to energy

The `lattice` package also contains functions that draw graphs that are similar to a scatterplot, but instead use a rank-ordering of the vertical axis variable. This is how the `stripplot` and `dotplot` commands work, and they offer another view of a relationship and its robustness. The `dotplot` command may be somewhat more desirable as it also displays a line for each rank-ordered value, offering a sense that the scale is different. The `dotplot` syntax looks like this:

```
dotplot(partratebusiness~supplybusiness,
        data=health.fin,col="black",
        ylab="Lobby Participation Rate (Rank Order)",
        xlab="Number of Health Establishments")
```

Figure 3.7b displays this result. The `stripplot` function uses similar syntax.

Lastly, the `lattice` library again gives us an option to look at the distribution of a single variable by plotting either a histogram or a density plot. Returning to the presidential time series data we first loaded in Sect. 3.1, we can now draw a density plot using the following line of code:

```
densityplot(~Energy,data=pres.energy,
            xlab="Television Stories",col="black")
```

This is presented in Fig. 3.8a. This output shows points scattered along the base, each representing the value of an observation. The smoothed line across the graph represents the estimated relative density of the variable's values.

Alternatively, a histogram in `lattice` can be drawn with the `histogram` function:

```
histogram(~Energy, data=pres.energy,
          xlab="Television Stories", col="gray60")
```

This is printed in Fig. 3.8b. In this case, color is set to `col="gray60"`. The default again is for cyan-colored bars. For a good grayscale option in

this case, a medium gray still allows each bar to be clearly distinguished. A final interesting feature of `histogram` is left to the reader: The function will draw conditional histogram distributions. If you still have the `post.nixon` variable available that we created earlier, you might try typing `histogram(~Energy|post.nixon,data=pres.energy)`, where the vertical pipe (`|`) is followed by the conditioning variable.

3.4 Graphic Output

A final essential point is a word on how users can export their R graphs into a desired word processor or desktop publisher. The first option is to save the screen output of a figure. On Mac machines, user may select the figure output window and then use the dropdown menu *File*→*Save As...* to save the figure as a PDF file. On Windows machines, a user can simply right-click on the figure output window itself and then choose to save the figure as either a metafile (which can be used in programs such as Word) or as a postscript file (for use in L^AT_EX). Also by right-clicking in Windows, users may copy the image and paste it into Word, PowerPoint, or a graphics program.

A second option allows users more precision over the final product. Specifically, the user can write the graph to a graphics device, of which there are several options. For example, in writing this book, I exported Fig. 3.5a by typing:

```
postscript("lin.partrate.eps",horizontal=FALSE,width=3,
           height=3,onefile=FALSE,paper="special",pointsize=7)
plot(y=health.fin$partratebusiness,x=health.fin$supplybusiness,
     ylab="Lobby Participation Rate",
     xlab="Number of Health Establishments")
abline(finance.linear)
dev.off()
```

The first line calls the `postscript` command, which created a file called `lin.partrate.eps` that I saved the graph as. Among the key options in this command are `width` and `height`, each of which I set to three inches. The `pointsize` command shrank the text and symbols to neatly fit into the space I allocated. The `horizontal` command changes the orientation of the graphic from landscape to portrait orientation on the page. Change it to `TRUE` to have the graphic adopt a landscape orientation. Once `postscript` was called, all graphing commands wrote to the file *and not to the graphing window*. Hence, it is typically a good idea to perfect a graph before writing it to a graphics device. Thus, the `plot` and `abline` commands served to write all of the output to the file. Once I was finished writing to the file, the `dev.off()` command closed the file so that no other graphing commands would write to it.

Of course `postscript` graphics are most frequently used by writers who use the desktop publishing language of L^AT_EX. Writers who use more traditional word processors such as Word or Pages will want to use other graphics devices. The

available options include: `jpeg`, `pdf`, `png`, and `tiff`.⁵ To use any of these four graphics devices, substitute a call for the relevant function where `postscript` is in the previous code. Be sure to type `?png` to get a feel for the syntax of these alternative devices, though, as each of the five has a slightly different syntax.

As a special circumstance, graphs drawn from the `lattice` package use a different graphics device, called `trellis.device`. It is technically possible to use the other graphics devices to write to a file, but inadvisable because the device options (e.g., size of graph or font size) will not be passed to the graph. In the case of Fig. 3.7b, I generated the output using the following code:

```
trellis.device("postscript", file="dotplot.partrate.eps",
  theme=list(fontsize=list(text=7, points=7)),
  horizontal=FALSE, width=3, height=3,
  onefile=FALSE, paper="special")
dotplot(partratebusiness~supplybusiness,
  data=health.fin, col='black',
  ylab="Lobby Participation Rate (Rank Order)",
  xlab="Number of Health Establishments")
dev.off()
```

The first argument of the `trellis.device` command declares which driver the author wishes to use. Besides `postscript`, the author can use `jpeg`, `pdf`, or `png`. The second argument lists the file to write to. Font and character size must be set through the `theme` option, and the remaining arguments declare the other preferences about the output.

This chapter has covered univariate and bivariate graphing functions in R. Several commands from both the base and `lattice` packages have been addressed. This is far from an exhaustive list of R's graphing capabilities, and users are encouraged to learn more about the available options. This primer should, however, serve to introduce users to various means by which data can be visualized in R. With a good sense of how to get a feel for our data's attributes visually, the next chapter turns to numerical summaries of our data gathered through descriptive statistics.

3.5 Practice Problems

In addition to their analysis of energy policy coverage introduced in this chapter, Peake and Eshbaugh-Soha (2008) also study drug policy coverage. These data similarly count the number of nightly television news stories in a month focusing on drugs, from January 1977 to December 1992. Their data is saved in comma-separated format in the file named `drugCoverage.csv`. Download their data from the Dataverse named on page vii or the chapter content link on page 33. The variables in this data set are: a character-based time index showing month and year

⁵My personal experience indicates that `png` often looks pretty clear and is versatile.

(**Year**), news coverage of drugs (**drugsmedia**), an indicator for a speech on drugs that Ronald Reagan gave in September 1986 (**rwr86**), an indicator for a speech George H.W. Bush gave in September 1989 (**ghwb89**), the president's approval rating (**approval**), and the unemployment rate (**unemploy**).

1. Draw a histogram of the monthly count of drug-related stories. You may use either of the histogram commands described in the chapter.
2. Draw two boxplots: One of drug-related stories and another of presidential approval. How do these figures differ and what does that tell you about the contrast between the variables?
3. Draw two scatterplots:
 - (a) In the first, represent the number of drug-related stories on the vertical axis, and place the unemployment rate on the horizontal axis.
 - (b) In the second, represent the number of drug-related stories on the vertical axis, and place presidential approval on the horizontal axis.
 - (c) How do the graphs differ? What do they tell you about the data?
 - (d) Bonus: Add a linear regression line to each of the scatterplots.
4. Draw two line graphs:
 - (a) In the first, draw the number of drug-related stories by month over time.
 - (b) In the second, draw presidential approval by month over time.
 - (c) What can you learn from these graphs?
5. Load the `lattice` library and draw a density plot of the number of drug-related stories by month.
6. Bonus: Draw a bar graph of the frequency of observed unemployment rates. (*Hint*: Try using the `table` command to create the object you will graph.) Can you go one step further and draw a bar graph of the percentage of time each value is observed?