1

Learn about R and Write First Toy Programs

Chapter Objectives

In this first chapter, we will aim to achieve the following objectives:

1. Understand when to use R in a research project.
2. Learn about the basic background of R, software installation, and getting help.
3. Learn to set up a project folder for R programs and data files.
4. Learn to write and execute simple toy programs.
5. Learn to find and set the working directory for a project in R.
6. Learn to create a data vector.
7. Learn to calculate descriptive statistics and handle missing values.
8. Learn to convert a data vector into a data frame.
9. Learn to refer to a variable within a data frame.
10. Learn to install an add-on package, "stargazer," load it into R, and use it to get a descriptive statistics table.
11. Learn to graph the distribution of a variable.
12. Apply all the lessons learned to a real-world data example.
13. Learn about common coding errors and how to get help.

Materials in this chapter need about an hour and a half for a class of about 10 students to cover in a lab, including brief lecturing and hands-on practice. Larger classes or self-study could take longer.

When to Use R in a Research Project

To complete an empirical research project involves several stages, often starting with the identification of a research problem and ending with the report of findings and implications:

1. Identify a research problem
2. Survey the literature (Find out what is known about the problem)
3. Formulate a theoretical argument and some testable hypothesis
4. Measure concepts
5. Collect data
6. Prepare data
7. Analyze data
8. Report findings and implications

The tasks of identifying a significant and interesting research problem, surveying the extant literature, formulating a coherent theoretical argument and some testable hypothesis that explain the research puzzle, measuring concepts in the theory empirically, and collecting data for the empirical indicators of the concepts—tasks (1) to (5)—are generally dealt with in substantive and research design courses in a field. Those topics are beyond the scope of this little R book. Yet tasks (6) to (8) may all involve R as a research instrument. Specifically, using R for actual research projects is to analyze particular research problems, such as evaluating the impact of a policy or testing the impact of a causal factor (or an independent variable) on an outcome (or a dependent variable) of interest, as postulated by pre-specified theoretical expectations. How to accomplish tasks (6) to (8) will be illustrated in the following chapters.

A research project of this type presents at least two challenges, for which R will be useful. First, in practice, such a project involves a range of tasks, such as importing data into software, merging different datasets together, verifying data, creating new variables, recoding and renaming variables, visualizing data, running statistical estimation procedures, carrying out diagnostic tests, and so on. Second, an analyst needs to be able to reproduce his or her own analysis, including dataset construction and estimation results, even years later. The first challenge concerns the efficiency of an analysis, whereas the second concerns the reproducibility and integrity of the analysis.

To achieve both efficiency and reproducibility, experienced analysts always choose to write down their computing code in one or more programs so that the code can be submitted, revised, and resubmitted to reproduce an analysis speedily and whenever necessary. Hence, in this book, we will focus on how to write and submit R programs for specific tasks in a program editor, rather than the interactive use or menu-driven interface of R. For all practical purposes,

the programming approach is much more efficient and consistent than the interactive or menu-driven approach.

Before we step into how to use R, we will need to clarify some related organizational and housekeeping issues. In this chapter, we will first offer a very brief introduction to R, then demonstrate how to install R, write and execute R programs, install and load add-on packages, and produce graphical and numerical output, and then turn to essential reference information about important symbols and common coding errors. Notably, each line of R code will likely appear three times: presented as a stand-alone command line preceded or followed by an explanation of its purpose and function, listed together with the output from its execution, and collated with all other program code in the chapter for the sake of convenient reference. We will end the chapter with a section about miscellaneous issues of interest to ambitious readers and a section on exercises.

Essentials about R

A One-Paragraph Introduction to R

R is a computer language and an environment for statistical computing and graphics with important advantages. Started by Robert Gentleman and Ross Ihaka of the University of Auckland in 1995, it is now maintained by the R core-development team of volunteer developers. R is referred to as a computer language because as a dialect of the S language developed in the late 1980s at AT&T's labs, R allows users to follow the algorithms, define and add new functions, and write new analytic methods, rather than merely supplying canned routines. R is also a coherent system which provides an environment with an integrated suite of software facilities for data storage, manipulation, analysis, and visualization. In addition, R is flexible. It runs on Windows, UNIX, and Mac OS X. It can be easily extended in terms of new functions and state-of-the-art statistical methods; the over 10,000 add-on packages by the end of January 2017 through the CRAN family of internet sites testify to this fact. Last but not least, R is free, as are its numerous add-on packages. Hence, R is popular among practitioners in many fields and scholars in many disciplines, including the social sciences.

Installation

As an open source software for statistical computing, R can be easily downloaded from the following site: http://www.r-project.org/. We may simply click on the highlighted `download R` link to reach a list of CRAN mirror sites. Clicking on any site we prefer directs us to the page for downloading the software for three different platforms: Linux, Windows, and Mac. R works slightly differently across

the three platforms. For the purpose of this book, we will focus on the language and functionality specific to the Windows platform. Mac users may consult the Miscellaneous Q&A Section later in the chapter for some brief explanation.

R is being constantly updated to new versions by developers. It is worth noting that some R programs and packages used in this book could require 3.3.2 or newer. If the version of R on a machine is not up to date, one may simply uninstall the old version and install the latest version following the procedures described previously, or refer to the subsection on how to update R in the Miscellaneous Q&A Section.

How to Start A Project Folder and Write Our First R Program

Learn to Set up A Project Folder for Programs and Data Files

The first step in a project is to set up a project folder to hold relevant datasets, programs, and output files. We can think of a project folder as our home mailing address, and all the relevant datasets, programs, and output files as the mail and packages to be delivered to us. Without the mailing address, the packages and mail will not be delivered to the right place. Hence, a project folder allows us to easily find all the relevant files and avoid having them mingled and conflated with those files for other projects or purposes.

In Windows, we can create a project folder via the following steps: Open `My Computer` or `File Explorer`; right click on the root directory, such as C: or D:; click on `New`; click on `New Folder` or `Folder`; and type in a meaningful name for the new folder, such as `Project`.

Learn to Find and Set A Working Directory for A Project

When we open R, the default interface is the R Console page, which is based on the interactive mode. To create an R program, we should go to the R Editor page. To do so, we can open R, click on `File` on the menu bar, and then click on `New script` to open the R program editor. Now, click on the `Save` button on the menu bar or the `Save` option under `File`, and we will be prompted to enter a filename for an R program file that ends with .R. For an experiment, name the file `session1.R` (remember to end it with `.R`), and then save the file in the `Project` folder.

Learn to Write and Execute the Simplest Toy Program

Now is the time for us to learn to write an extremely simple R program and run it. R has a default working directory or folder (think of it as the post office address for mail and packages). We are interested in telling R to change the current default working directory to the `Project` folder. It is like directing our mail to

be delivered to our own home address, rather than the post office address. The `Project` folder is where we keep our program and data files.

To do so, we first identify the working directory of the current R session, then change it to the `Project` folder, and finally verify that the change is successful.

In the program editor, first type in

```
getwd()
```

The `getwd` function lists the name of the current working directory.

Next, type in

```
setwd("C:/Project")
```

The `setwd` function changes the current working directory for the current R session. The argument of the function is inside the parentheses, between double quotation marks, and employs one forward slash; it specifies the path to the `Project` folder as the new current working directory for the current R session. This line of code makes it possible, during the rest of our R session, for us to refer to the files within the `Project` folder without specifying the path again. Finally, note that R is case sensitive. Hence, R will treat **Project** and **project** as two different folders. If there is a mismatch in spelling between the program code and the actual folder name, R will produce an error message. Also note that any mismatch in terms of quotation marks, colon, etc. will cause R to produce an error message.

In specifying the path, we may use one forward slash as above, or alternatively, two double back slashes as follows:

```
setwd("C:\\Project")
```

Please note the double back slashes. This is very important. If we copy the path from our computer `File Explorer`, the copied and pasted path will contain only one back slash. For R, we will need to add an extra back slash, or change it to one forward slash.

Finally, type in again

```
getwd()
```

This allows us to verify the task is done as instructed.

We save these three lines of code into a program file called `session1.R`. This three-line R program asks R to display the default working directory, then sets the `Project` folder as our new current working directory, and finally asks R to display the current working directory again.

Having the `.R` suffix in the program filename is a good practice for two reasons. It helps us see immediately that it is an R program file. When we open a program file in the R editor, all files with `.R` suffix will appear automatically in the list of files for us to choose to open. If the program file does not have the `.R` suffix and if we want to open it in the R program editor, it will not show up automatically in the list of files. We will have to choose `"All Files (*.*)"` from file type in the lower right corner in order to see all files in the folder.

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

To execute this little program in R, we may choose one of the following two ways:

1. If we want to execute the program line by line, put the cursor anywhere in that line of code, then we can execute it in one of three ways: (a) hit two keys `Ctrl+R` on the keyboard together; (b) right click the mouse and then click on `Run line or selection`; (c) click on the third little icon (right next to the second `save script` icon) on the upper left corner, representing `Run line or selection`.
2. If we want to execute the whole program in one run, highlight the whole program in R editor, and then either right click the mouse and click on `Run line or selection`, or hit two keys `Ctrl+R` on the keyboard together.

When we execute the program above, we will get the following output in R:

```
getwd()

[1] "C:/Users/Quan Li/Box Sync/R Book/Rnw_oup_formal"

setwd("C:/Project")
getwd()

[1] "C:/Project"
```

Note that the first line of code `getwd()` shows that the default current working directory was `"C:/Users/Quan Li/Box Sync/R Book/Rnw_oup_formal"`, in which I have kept my knitr .Rnw and LaTex files for writing this R book. Then, the second line of code asks R to set the current working directory to `"C:/Project"` instead. The third line of code shows that for the rest of this R session, files will be drawn from or saved to this new working directory unless otherwise specified via a different file path.

One essential point about programming is that one should document the purpose of a program as a whole and what each line of code does so that days, weeks, or months from now, we or any others who open up the program will be able to understand what the program does and how it does it. For this purpose, we insert comment lines that begin with the # sign into a program. The # sign tells R not to execute that line. Note that the first comment specifies the purpose, time, and software version used. After adding comment lines, the little toy R program above will now be complete and look like the following:

```
# First R toy program, today's date, R version 3.2.3
# show current working directory path
getwd()

# change the working directory for program to project folder
setwd("C:/Project")

# show current working directory path again to verify
getwd()
```

To demonstrate the general process graphically, Figure 1.1 presents four screenshots from R console and editor, which proceed from opening R (picture 1), to opening R editor (picture 2), to typing the three lines of code in R editor (picture 3), to running those three lines and producing output in R console (picture 4).

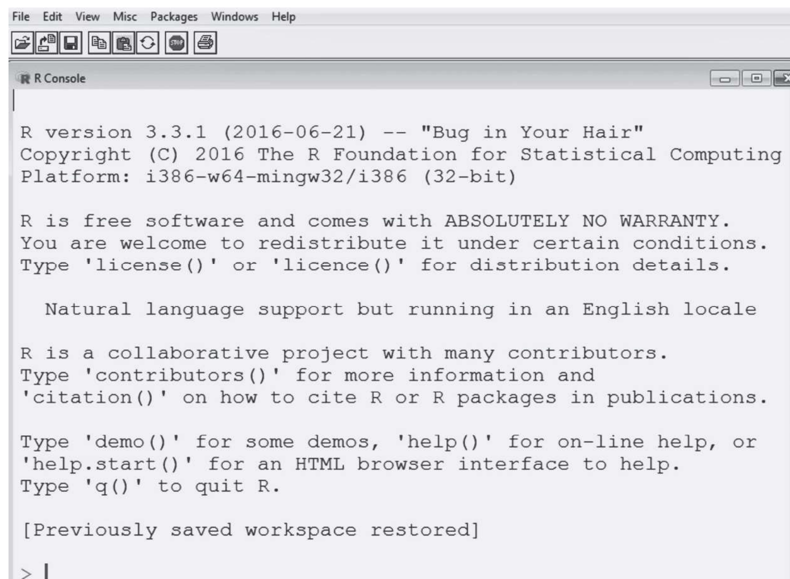
The biggest benefit of writing and saving a program is to reproduce the same output at any time so long as the functions of the software remain unchanged. For a reader unfamiliar with this approach of working with a software package, it might be useful to close R, open it again, and re-run the little program for replication and verification. Remember to save the program first before closing it; otherwise, we will lose all the changes since we last saved it. The ability to run the same program and produce the same result years after is one of the most important reasons why we prefer to program, rather than using the interactive mode via point and click.

Create, Describe, and Graph A Vector: A Simple Toy Example

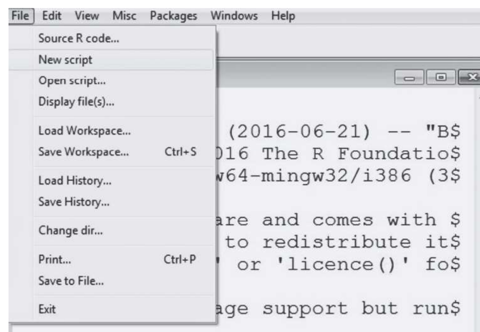
Since R is an object-oriented programming language, it is useful to know something about how R works with data. A simplified view is that R relies on a variety of functions, which take in data as input and then produce desired output

objects. In other words, R treats data as objects and operates on data objects through functions (which are themselves treated as objects as well), in order to manipulate data, create graphs, and conduct statistical analysis.

The most basic data object in R is a vector. A vector is a combination of elements, such as numbers, characters, or logical statements (like TRUE, FALSE). The elements within a single vector must be of the same type, i.e., numeric, character, or logical. In a simple example provided in the R manual, a vector named x consists of five numbers ordered as `10.4, 5.6, 3.1, 6.4, 21.7`.

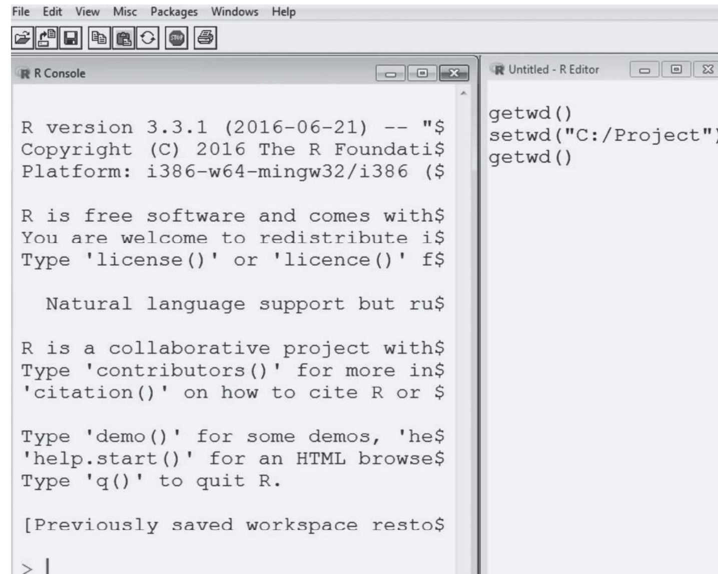


(a) picture 1

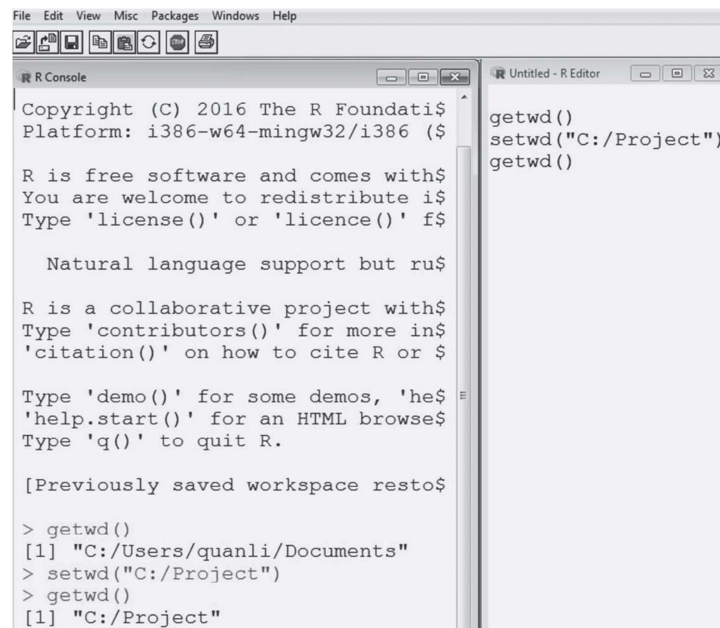


(b) picture 2

Figure 1.1 How to Write First Toy Program in R.



(c) picture 3



(d) picture 4

Figure 1.1 (Continued)

In this section, we will focus on expanding our first toy program around one artificial numeric data vector, carrying out various operations, turning it into a data frame, and then carrying out more operations. Our specific learning objectives in this section include the following:

1. Learn to use c() function to create a data vector
2. Learn to calculate descriptive statistics for this data vector
3. Learn to deal with missing values of a data vector
4. Learn to convert a data vector into a data frame
5. Learn to refer to a variable in a data frame
6. Learn to install and upload an add-on package
7. Learn to report the descriptive statistics of a variable in table format
8. Learn to graph the distribution of a variable in a data frame
9. Learn to combine multiple plots into one figure

Create A Vector Using c() Function

We will show a simple example of a numeric vector as a data object. To show how R operates on data, we will first introduce one function and one assignment operator for objects in R. The first function is c(). The c() function combines or concatenates the terms or arguments inside the parentheses together into a vector. For example, the R code below creates a vector that contains some arbitrary numbers, separated by commas, inside a pair of parentheses.

```
c(1, 2, 0, 2, 4, 5, 10, 1)
```

To save the vector for later use, we assign it to an R object with an arbitrary name. R uses the assignment symbol `<-` (a `less than` symbol followed by a `minus` sign, with no space in between) to link the name of the object and the c() function. Then, in a new line of code, we type the object name given to the vector, which simply displays what is in that object. The R code is as follows:

```
v1 <- c(1, 2, 0, 2, 4, 5, 10, 1)
v1
```

As noted, R is an object-oriented programming language. R can work with a variety of objects, such as "numeric," "logical," "character," "list," "matrix," "array," "factor," or "data.frame." Different objects have different attributes. Even though we will not go into details about all their differences in this introductory book, it

is always a good idea to know what type of an object we have created. To identify the object type of v1, we simply apply the class() function.

```
class(v1)
```

Collecting these four lines of code, mingled with explanatory comment lines R ignores in execution, we produce the following R program:

```
# use c() function to create a vector object
c(1, 2, 0, 2, 4, 5, 10, 1)

# assign the c function output to an object named v1
v1 <- c(1, 2, 0, 2, 4, 5, 10, 1)

# call object v1 to see what is in it
v1

# identify object type of v1
class(v1)
```

Running this program in R produces the following output:

```
# use c() function to create a vector object
c(1, 2, 0, 2, 4, 5, 10, 1)

[1]  1  2  0  2  4  5 10  1

# assign the c function output to an object named v1
v1 <- c(1, 2, 0, 2, 4, 5, 10, 1)

# call object v1 to see what is in it
v1

[1]  1  2  0  2  4  5 10  1

# identify object type of v1
class(v1)

[1] "numeric"
```

Calculate Descriptive Statistics for A Vector

Now that we have created our first data vector in R, we can work with it to practice some useful R functions. For example, we can ask R to tell us how many observations are in v1, its sample mean, and its sample variance, as well as

various other summary statistics regarding v1. Notice how we use comment lines to keep track of what we did for ourselves and to tell others who will see our code in the future what we did.

```
# find the number of observations in variable v1
length(v1)

# find v1's sample mean (two ways: mean function or formula)
mean(v1)
sum(v1)/length(v1)

# find v1's sample variance (variance function or formula)
var(v1)
sum((v1-mean(v1))^2)/(length(v1)-1)

# find v1's sample standard deviation (function
# or square root of sample variance)
sd(v1)
sqrt(var(v1))

# find v1's sample minimum and maximum using functions
max(v1)
min(v1)
```

Several issues on the code above are worth clarification. First, eight new functions are introduced, including length, mean, sum, var, sd, sqrt, min, and max. They identify the number of observations, mean, total sum, sample variance, sample standard deviation, minimum, and maximum of v1, respectively.

Second, the code employs some of the commonly used mathematical operators in R, including addition (+), subtraction (–), multiplication (*), division (/), and exponentiation (^). These are self-explanatory and should be memorized for future use.

Third, the code above shows alternative ways to compute sample mean, variance, and standard deviations. In other words, sample mean is computed as the sum of all observations of v1 divided by the number of observations, sample variance is calculated as the sum of squared deviations of v1 from its mean divided by (the number of observations minus one), and sample standard deviation is the square root of sample variance. More details about these will be discussed in Chapter 3.

Finally, parentheses are used to structure the order of mathematical operations. Always start with calculations inside parentheses. Also, perform exponentiation, multiplication, and division from left to right; then, perform addition and subtraction from left to right.

Executing the code above in R produces the following output:

```
# find total number of observations in variable v1
length(v1)

[1] 8

# find v1's sample mean (two ways: mean function or formula)
mean(v1)

[1] 3.125

sum(v1)/length(v1)

[1] 3.125

# find v1's sample variance (variance function or formula)
var(v1)

[1] 10.41071

sum((v1-mean(v1))^2)/(length(v1)-1)

[1] 10.41071

# find v1's sample standard deviation (function
# or square root of sample variance)
sd(v1)

[1] 3.226564

sqrt(var(v1))

[1] 3.226564

# find v1's sample minimum and maximum using functions
max(v1)

[1] 10

min(v1)

[1] 0
```

Handle Missing Values in Descriptive Statistics

The artificial variable v1 above does not have any missing value. However, missing values are common in real-world data. What happens when missing values are present? This question is relevant both when we compute descriptive statistics for a vector and when we want to know how many missing or non-missing values are in a vector.

Suppose we create a new variable called v2, which is identical to v1 except for that v2 has two missing values. In R, the default missing value is denoted by NA. Hence, we replace two observations in v1 with NA to generate v2.

```
# create v2 with missing values
v2 <- c(1, 2, 0, 2, NA, 5, 10, NA)
```

If we compute the mean of v2 with the mean() function without removing the missing values, we obtain the following output:

```
# compute mean of v2 without removing missing values
mean(v2)

[1] NA
```

Apparently, if there are missing values in a variable and R is not told to consider their presence when executing a function, then the output of that function will be NA. Hence, we need to tell R to ignore observations that are NA (missing values). We do so by inserting the na.rm=TRUE option inside the function, meaning that it is true to remove NA observations.

```
# compute mean of v2 after removing missing values
mean(v2, na.rm = TRUE)

[1] 3.333333
```

In actual data analysis, it is also important that we get correct information on the numbers of total, missing, and non-missing observations for a variable of interest. The length() function introduced earlier only identifies the total number of elements of a vector, including both missing and non-missing values. To identify the number of missing values in a variable, we need to use a new function, is.na(). The is.na() function produces a vector of logical values (TRUE or FALSE) for the vector listed in the function: TRUE if an element is a missing value; FALSE if it is not missing. Basically, we can think of the function as if it were asking whether each element in a vector is na or not. In contrast, to verify if an element of a vector is a non-missing value or not, we add the ! sign (meaning not) before the is.na() function. Applying is.na() and !is.na() to v2 gives us the

following TRUE or FALSE output with respect to each element of v2. The R code and output are listed below. Note how for each element of v2, is.na() and !is.na() give us exactly opposite logical values.

```
# display output from is.na() function
is.na(v2)

[1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE

# display output from !is.na() function
!is.na(v2)

[1]  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE FALSE
```

To find how many missing values are in v2, we use the sum() function to count the total number of TRUE values in the output of is.na(). For the number of non-missing values, we apply !is.na() instead to count the number of TRUE values. Note how in the output below, the total number of observations = missing values + non-missing values.

```
# find total number of observations in v2
length(v2)

[1] 8

# find the number of missing values in v2
sum(is.na(v2))

[1] 2

# find the number of non-missing values in v2
sum(!is.na(v2))

[1] 6
```

Convert A Vector into A Data Frame

So far, we have been working with vectors like v1 and v2. In actual data analysis, we typically work with a dataset that has observations as rows and variables as columns. In R, we also work with this type of data object, called a data frame. A data frame in R is often displayed in matrix form, which is a two-dimensional data object consisting of rows by columns. For our purposes, we can think of a data frame as a typical dataset like in Excel, with rows indicating observations and columns indicating variables. Each column or variable of a data frame must

be of the same data type (character, numeric, logical), but different columns or variables could be of different data types.

How do we convert a vector into a data frame in R? We simply apply the data.frame() function to v1, which will turn it into a data frame, and then assign the output from the function into a data frame arbitrarily named. The R code is as follows:

```
# convert data vector v1 into a data frame vd
vd <- data.frame(v1)
```

Recall that a data frame is a two-dimensional data object with rows by columns, just like a typical dataset with rows indicating observations and columns indicating variables. Hence, we can think of vd as a dataset containing one variable called v1 with eight observations.

We may also combine vectors v1 and v2 together into a data frame by applying the data.frame() function. By combining v1 and v2, the new data frame now consists of two variables, v1 and v2, and eight observations. The R code is as follows:

```
# combine vectors v1 and v2 into data frame vd of two
# variables v1 and v2
vd <- data.frame(v1, v2)
```

The R output for executing the code above and displaying the data frame content is as follows:

```
# convert data vector v1 into a data frame vd
vd <- data.frame(v1)

# display vd
vd

  v1
1  1
2  2
3  0
4  2
5  4
6  5
7 10
8  1

# combine vectors v1 and v2 into data frame vd of two
# variables v1 and v2
```

```
vd <- data.frame(v1, v2)

# display vd
vd

  v1 v2
1  1  1
2  2  2
3  0  0
4  2  2
5  4 NA
6  5  5
7 10 10
8  1 NA
```

Note that we first created a dataset or data frame `vd` with one variable and eight observations, and then we created another dataset `vd` that overwrote the previous dataset of the same name. We could have given the second one a different name to avoid overwriting the first one.

Now that we have created a dataset or data frame `vd` with two variables and eight observations, how do we refer to a variable in the data frame? R uses the $ sign to link a data frame and a variable in the data frame. Hence, `vd$v1` refers to variable `v1` in `vd`, and `vd$v2` refers to variable `v2` in `vd`.

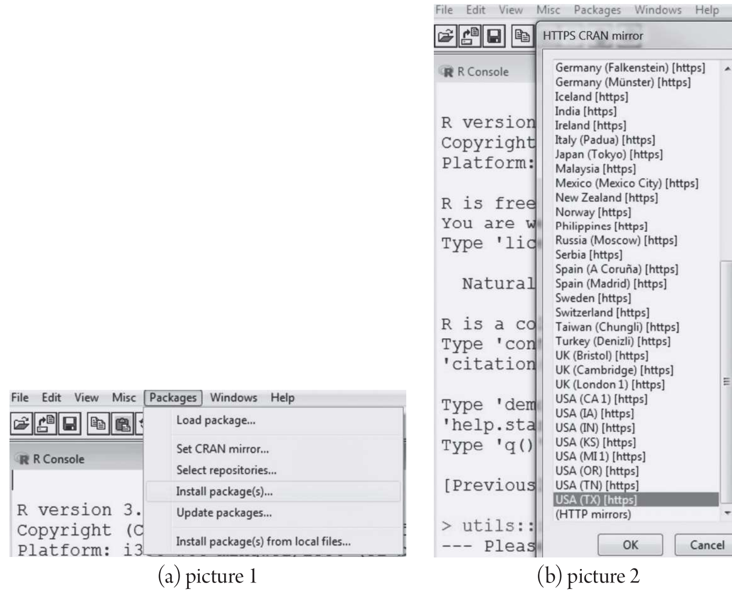
Format Output into Table

An important part of presenting research findings is to format output into a table. The base R package, however, does not produce nicely formatted output. As noted earlier, one of the most important strengths of R is the availability of add-on packages R users provide for free through the CRAN family of internet sites. R is reported to have nearly 12,000 free add-on packages as of 2017. One free package, `stargazer`, allows us to present the statistical results in a formatted table.

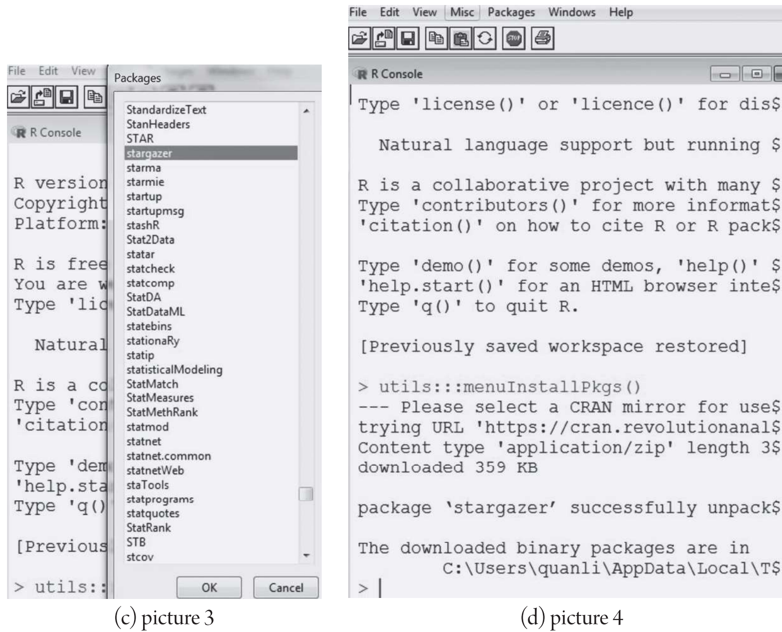
To apply the package to our data frame `vd`, we will learn how to complete the following tasks: install a user-written package, load the package into R, and apply the stargazer() function to produce a table.

How to Install and Load an Add-on Package

To install the stargazer package, follow the steps in Figure 1.2: Open R in Windows, click on `Packages` on the menu bar, click on `Install Packages` from the options (picture 1), select a CRAN mirror site nearby and click `OK` (picture 2), select the package to install and click `OK` (picture 3), and a successful installation will be shown in R console (picture 4).



(a) picture 1 (b) picture 2



(c) picture 3 (d) picture 4

Figure 1.2 How to Install Add-on Package.

Note that an add-on package only needs to be installed once for future use. Hence, we do not have to include and run the installation command in our R program. However, in every R session, if we plan to use an add-on package, we will have to load it into R via the library() function. After loading, a citation of each package will also show up for users. The R code is as follows:

```
# install.packages('stargazer')
library(stargazer)
```

Before we move on, it needs to be emphasized that all add-on packages must be first installed before they can be loaded into R using the library() function. Yet all add-on packages only need to be installed once. Thus, in the rest of the chapter, before we use the library() function, we will always keep a line of code on package installation as a reminder that the package must be installed first, and yet we will always comment out that line of code since it only needs to be installed once.

Produce Descriptive Statistics Table

Now that we have loaded the stargazer package, we may produce a formatted table of descriptive statistics for variables in `vd` via the stargazer() function. The R code is as follows:

```
# produce formatted descriptive statistics of variables in
# data frame vd
stargazer(vd, type = "text")
```

Two issues are worth clarification. First, to get descriptive statistics for `v1`, we have to apply the stargazer() function to the data frame `vd` containing `v1`; if we apply the function to `v1` alone, we will get a display of all its observations rather than its descriptive statistics (try it!). Second, the stargazer() function allows one to choose among three types of output formats: "latex" (default) for LaTeX code, "html" for HTML/CSS code, and "text" for ASCII text output. For users not familiar with the first two types, we specify `type="text"`. The R output is as follows:

```
# install package first and for once only
# install.packages('stargazer')

# load stargazer into R
library(stargazer)

# produce formatted descriptive statistics of variable(s)
# in data frame vd
```

```
stargazer(vd, type = "text")

==================================
Statistic N Mean  St. Dev. Min Max
----------------------------------
v1        8 3.125  3.227    0  10
v2        6 3.333  3.670    0  10
----------------------------------
```

We may modify the code above to generate a variety of formatted output. The following code and their comment lines illustrate several possible variations.

```
# display dataset in a table format
stargazer(vd, type = "text", summary = FALSE, rownames = FALSE)
```

```
# add additional statistics to be reported median,
# interquartile range (25th and 75th percentile)
stargazer(vd, type = "text", median = TRUE, iqr = TRUE)
```

```
# use c() function to choose statistics to be reported
stargazer(vd, type = "text", summary.stat = c("n", "mean",
    "median", "sd"))
```

For example, select descriptive statistics of variables in `vd` may be presented as follows:

```
# produce formatted select descriptive statistics of
# variables in vd
stargazer(vd, type = "text", summary.stat = c("n", "mean",
    "sd", "min", "p25", "median", "p75", "max"))

===========================================================
Statistic N Mean  St. Dev. Min Pctl(25) Median Pctl(75) Max
-----------------------------------------------------------
v1        8 3.125  3.227    0     1       2      4.2    10
v2        6 3.333  3.670    0    1.2      2      4.2    10
-----------------------------------------------------------
```

Graph the Distribution of A Variable

Visualizing the distribution of a variable of interest is an important part of data analysis. To illustrate, we will use several R functions to graph the distribution of variable `v1` in dataset `vd`. How we graph `vd$v1` depends on what type of variable

we think it is, i.e., whether it is discrete or continuous. A discrete variable is one that only takes on a finite number of values, e.g., gender, age, and the number of children in a household. In contrast, a continuous variable is one that could take on an infinite number of possible values, even within a range, like weight, distance, and income.

If we treat `v1` as a discrete variable, then we can use frequency table and bar chart to demonstrate its distribution. A frequency table displays data values of a variable and how often each value occurs. It is often first used to examine the distribution of a discrete variable. In R, the table() function computes the frequency count for each value of a variable. The R code and output are as follows:

```
# display the frequency count of v1
table(vd$v1)


 0  1  2  4  5 10
 1  2  2  1  1  1
```

The top row in output shows each value of `vd$v1`, and the second row the frequency count of each value.

A bar chart displays the distribution of a discrete variable in terms of the frequencies of its values or some other measures. In R, the barplot() function uses the frequency count output produced from the table() function to show a bar plot. Figure 1.3 is created using the code below:

```
# graph distribution of discrete variable vd$v1: bar chart
barplot(table(vd$v1))
```

If we treat `vd$v1` as a continuous variable, then we are more interested in the center, symmetry or skewness, and outliers in the distribution of the variable. We can use boxplot and histogram to demonstrate its distribution. In R, the hist() function produces histogram, and the boxplot() function creates box plot. The R code for plotting `vd$v1` is as follows:

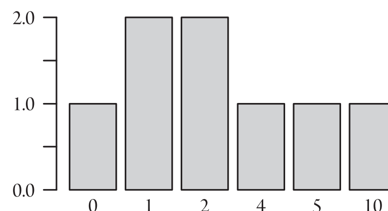


Figure 1.3 Distribution of Discrete Variable vd$v1: Bar Chart.

```
# graph distribution of continuous variable vd$v1: box
# plot and histogram
boxplot(vd$v1)
hist(vd$v1)
```

Sometimes, we are interested in placing the plots together into one figure for ease of comparison. To do so, we use the par() function to set graphical parameters, such as the number of rows and columns in a figure. If we want a figure of two plots side by side, that is, organized into one row and two columns, we specify `mfrow=c(1,2)` in the par() function, where the first value refers to the number of rows and the second the number of columns. Figure 1.4 is created using the code below:

```
# create a figure with two plots
# set graphical parameters for a figure of one row, two columns
par(mfrow = c(1, 2))

# graph distribution of continuous variable vd$v1: box
# plot and histogram
boxplot(vd$v1)
hist(vd$v1)
```

A box plot presents important attributes of a plotted variable: (1) the median, indicated by the dark horizontal line inside the box; (2) the first (25th) and third (75th) quartiles (i.e., interquartile range), represented by the lower and upper boundaries of the box, respectively; (3) the short horizontal lines outside the box, together with the dotted lines linking to the box, are called whiskers and represent the minimum and maximum values excluding outliers; (4) outliers (if any), denoted by dots that are 1.5 times larger than the upper quartile or 1.5 times less than the lower quartile. In Figure 1.4, the box plot shows that for `vd$v1`, the median value is 2, the 25th and 75th percentile values are 1 and 4.5, the minimum and maximum values are 0 and 5, and the outlier is 10. Overall, the box plot shows that the variable is centered within the interquartile range, yet it is not symmetrically distributed but skewed toward an outlier.

A histogram is also frequently used to show the distribution of a continuous variable in terms of its center, symmetry, and outliers. It puts values of the plotted variable into bins (i.e., bars or intervals). Each bin contains the number of times or frequency of data values contained within the bin. The area of a bin represents the frequency of occurrences within the bin. In Figure 1.4, the histogram shows that the 0–2 bin contains five observations, the bin from above 2 to 4 contains one observation, and the bin from above 4 to 6 contains one observation, and the bin from above 8 to 10 contains one observation. Overall,

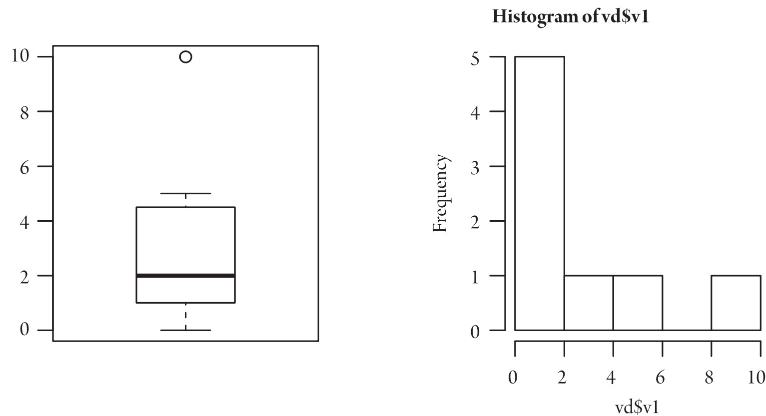


Figure 1.4 Distribution of Continuous Variable vd$v1: Boxplot and Histogram.

the histogram shows that the variable is single peaked but it is seriously skewed to the right.

R allows us to modify and polish these plots in many different ways. However, to make graphing accessible to beginners, we intentionally make the plots as simple as possible, without any additional details.

Simple Real-World Example: Data from Iversen and Soskice (2006)

In this section, we will apply the code from the simple toy example to a real-world data example, drawn from the following published article:

> Iversen, Torben, and David Soskice. 2006. "Electoral Institutions and the Politics of Coalitions: Why Some Democracies Redistribute More Than Others." *American Political Science Review* 100(2): 165–81.

In this article, Iversen and Soskice study the variations in government redistribution across democracies. They argue that proportional representation electoral systems enable center-left governments to dominate and produce more redistribution whereas majoritarian systems enable center-right governments to dominate and engage in less redistribution. Table A1 in their article provides country means for variables used in their statistical analysis, as shown in Table 1.1.

Next, we will demonstrate how to read data on several variables in that table into R, provide descriptive statistics, and show distribution plots for some key variables.

Table 1.1 **Country Means for Variables Used in Regression Analysis (from Iverson and Soskice, 2006)**

	Redistribution (reduction (in Gini)	Inequality (wages)	Partisanship (right)	Voter Turnout	Union-ization	Veto Points	Electoral System (PR)	Left Frag-mentation	Right over-Representaion	Per capita Income	Female Labor Force Participation	Unemploym
Australia	23.97	1.70	0.47	84	46	3	0	−0.39	0.10	10909	46	4.63
Austria	—	—	0.30	87	54	1	1	−0.18	0.04	8311	51	2.76
Belgium	35.56	1.64	0.36	88	48	1	1	−0.34	0.27	8949	43	7.89
Canada	21.26	1.82	0.36	68	30	2	0	0.18	−0.11	11670	48	6.91
Denmark	37.89	1.58	0.35	84	67	0	1	−0.40	0.07	9982	63	6.83
Finland	35.17	1.68	0.30	79	53	1	1	−0.18	0.09	8661	66	4.48
France	25.36	1.94	0.40	66	18	1	0	0.10	0.09	9485	51	4.57
Germany	18.70	1.70	0.39	81	34	4	1	−0.13	0.15	9729	51	4.86
Ireland	—	—	0.42	75	48	0	0	−0.33	0.70	5807	37	9.09
Italy	12.13	1.63	0.37	93	34	1	1	0.20	0.08	7777	38	8.12
Japan	—	—	0.78	71	31	1	0	0.22	0.28	7918	56	1.77
Netherlands	30.59	1.64	0.31	85	33	1	1	0.18	−0.36	9269	35	4.62
New Zealand	—	—	0.43	85	23	0	0	−0.40	0.98	—	47	—
Norway	27.52	1.50	0.15	80	54	0	1	−0.02	−0.32	9863	52	2.28
Sweden	37.89	1.58	0.17	84	67	0	1	−0.40	−0.03	9982	63	6.83
U.K.	22.67	1.78	0.52	76	42	0	0	0.08	0.07	9282	54	5.01
U.S.	17.60	2.07	0.40	56	23	5	0	0.00	−0.17	13651	53	5.74

Note: Time coverage is 1950–96 except for redistribution and inequality, which are restricted to the available LIS observations.

```
# create dataset from Iversen and Soskice

# assign c() function output to vector object country
country <- c("Australia","Austria","Belgium","Canada",
"Denmark","Finland","France","Germany","Ireland", "Italy",
"Japan","Netherlands","New Zealand","Norway","Sweden","U.K.",
"US")

# assign c() output to object gini.red for reduction in GINI
gini.red <- c(23.97,NA,35.56,21.26,37.89,35.17,25.36,18.7,NA,
              12.13,NA,30.59,NA,27.52,37.89,22.67,17.6)

# assign c() output to object wage.ineq for wage inequality
wage.ineq <- c(1.7,NA,1.64,1.82,1.58,1.68,1.94,1.7,NA,1.63,NA,
               1.64,NA,1.5,1.58,1.78,2.07)

# assign c() output to object pr for electoral system
pr <- c(0,1,1,0,1,1,0,1,0,1,0,1,0,1,1,0,0)

# use data.frame function to combine four vector objects
# assign data.frame function output to data frame is2006apsr
is2006apsr <- data.frame(country, gini.red, wage.ineq, pr)

# call data frame is2006apsr to display content
is2006apsr

       country gini.red wage.ineq pr
1    Australia    23.97      1.70  0
2      Austria       NA        NA  1
3      Belgium    35.56      1.64  1
4       Canada    21.26      1.82  0
5      Denmark    37.89      1.58  1
6      Finland    35.17      1.68  1
7       France    25.36      1.94  0
8      Germany    18.70      1.70  1
9      Ireland       NA        NA  0
10       Italy    12.13      1.63  1
11       Japan       NA        NA  0
12 Netherlands    30.59      1.64  1
13 New Zealand       NA        NA  0
14      Norway    27.52      1.50  1
15      Sweden    37.89      1.58  1
```

```
16        U.K.    22.67      1.78  0
17          US    17.60      2.07  0
```

Now that the dataset is ready, we can compute the descriptive statistics for `gini.red`, `wage.ineq`, and `pr`, and present them in a formatted table, using the stargazer package. We will load the package and then apply the stargazer() function as in the R code below.

```
# produce formatted table of descriptive statistics
# install package first and for once only
# install.packages("stargazer")

# load stargazer into R
library(stargazer)

# produce descriptive statistics table of select variables
stargazer(is2006apsr, type="text", title="Summary Statistics",
          median=TRUE, covariate.labels=c("GINI reduction",
          "wage inequality","PR system"))
```

Several added features in the stargazer code need clarification. First, we add a title to the table with the `title=""` option. Second, the default summary statistics reported include n, mean, standard deviation, minimum, and maximum. We now also ask for the median value of each variable with `median=TRUE`. We may even request for the 25th and 75th percentiles for each variable with a similar option like `iqr=TRUE`. Third, instead of using abbreviated variable names, we now give each variable a more meaningful variable label in the table. This is reflected in the `covariates.labels=` option.

Table 1.2 reports the summary statistics for the three numeric variables. Note that we intentionally did not input all variables from Table A1 in Iversen and Soskice (2006), expecting that readers will enter the rest of the data themselves as a take-home assignment.

Table 1.2 **Statistics of Imported Data from Iversen and Soskice (2006)**

```
Descriptive Statistics
=======================================================
Statistic       N   Mean  St. Dev.  Min   Median  Max
-------------------------------------------------------
GINI reduction  13 26.639  8.320   12.130 25.360 37.890
wage inequality 13 1.712   0.157   1.500  1.680  2.070
PR system       17 0.529   0.514     0      1      1
-------------------------------------------------------
```

Next, we will apply some graphing code to the `wage.ineq` variable in `is2006apsr`. Since it is a continuous variable, we produce its box plot and histogram. Figure 1.5 is created using the R code below:

```
# create a figure with two plots
# set graphic parameters for figure of one row, two columns
par(mfrow=c(1,2))

# graph distribution of wage inequality
boxplot(is2006apsr$wage.ineq)
hist(is2006apsr$wage.ineq)
```

Finally, we provide a bar plot for the discrete electoral system variable `pr`, with a value of 1 indicating a proportional representation system and 0 indicating a majoritarian system. Figure 1.6 is created using the R code below:

```
# graph distribution of electoral system variable pr
barplot(table(is2006apsr$pr))
```

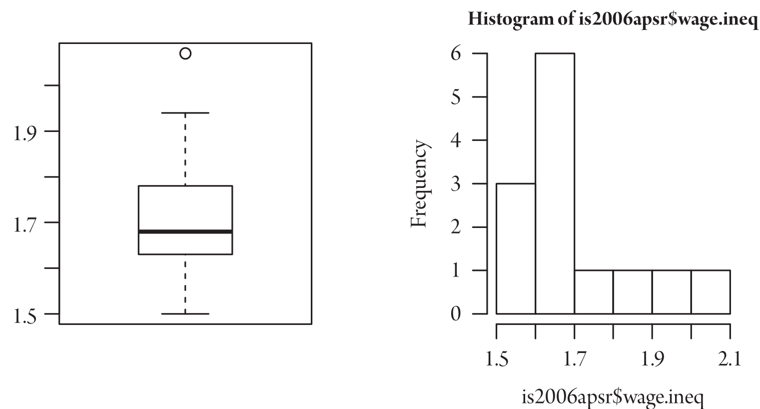


Figure 1.5 Distribution of Wage Inequality from Iversen and Soskice (2006).

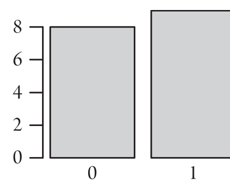


Figure 1.6 Distribution of PR and Majoritarian Systems from Iversen and Soskice (2006).

Chapter 1: R Program Code

By now, we have gone over individual lines of R code to accomplish different tasks. Could we compile them into one coherent R program? The answer is yes, and to do so is easy. Below we list all the R code used in this chapter, which one could copy and paste into R program editor and save them as one program file into our project folder. As noted earlier, the biggest advantage of keeping the R code in a permanent program file is that we can replicate at any time what we have produced. This benefit cannot be overemphasized.

First R Toy Program

```
# How to Start a Project Folder and Write First R Program
# show current working directory path
getwd()

# change working directory for program to this folder one
# could also use setwd('C:\\project')
setwd("C:/Project")

# show current working directory path again to verify
getwd()
```

Simple Toy Program: Create, Describe, and Graph a Variable

```
# use c() function to create a variable or vector object
c(1, 2, 0, 2, 4, 5, 10, 1)

# assign the c function output to an object named v1
v1 <- c(1, 2, 0, 2, 4, 5, 10, 1)

# call object v1 to see what is in it
v1

# identify object type of v1
class(v1)

# calculate descriptive statistics for v1:
# find total number of observations in variable v1
```

```
length(v1)

# find v1's sample mean (two ways: mean function or formula)
mean(v1)
sum(v1)/length(v1)

# find v1's sample variance (variance function or formula)
var(v1)
sum((v1-mean(v1))^2)/(length(v1)-1)

# find v1's sample standard deviation (function
# or square root of sample variance)
sd(v1)
sqrt(var(v1))

# find v1's sample minimum and maximum
max(v1)
min(v1)

# handle missing values in descriptive statistics:
# create variable v2 with missing values
v2 <- c(1, 2, 0, 2, NA, 5, 10, NA)

# compute mean of v2 without removing missing values
mean(v2)

# compute mean of v2 after removing missing values
mean(v2, na.rm=TRUE)

# display output from is.na() function
is.na(v2)

# display output from !is.na() function
!is.na(v2)

# find total number of observations in v2
length(v2)

# find the number of missing values in v2
sum(is.na(v2))
```

```
# find the number of non-missing values in v2
sum(!is.na(v2))

# convert vector v1 into a data frame vd with a variable v1
vd <- data.frame(v1)

# display vd
vd

# combine vectors v1 and v2 into data frame vd
# with two variables v1 and v2
vd <- data.frame(v1, v2)

# display vd
vd

# Format descriptive statistics output into table:
# install package first and for once only
# install.packages("stargazer")

# load stargazer into R
library(stargazer)

# produce descriptive statistics table for data frame vd
stargazer(vd, type="text")

# display dataset in a table format
stargazer(vd, type="text", summary=FALSE, rownames=FALSE)

# add additional statistics to be reported:
# median, interquartile range (25th and 75th percentile)
stargazer(vd, type="text", median=TRUE, iqr=TRUE)

# use c() function to choose statistics to be reported
stargazer(vd, type="text", summary.stat=c("n", "mean",
              "median", "sd"))

# produce select descriptive statistics table for vd
stargazer(vd, type="text", summary.stat=c("n", "mean", "sd",
              "min", "p25", "median", "p75", "max"))
```

```
# Graph the distribution of variable:
# display the frequency count of v1 in vd
table(vd$v1)

# graph distribution of discrete variable vd$v1: bar chart
barplot(table(vd$v1))

# graph distribution of continuous variable vd$v1:
# box plot and histogram
boxplot(vd$v1)
hist(vd$v1)

# create a figure with two plots
# set graphic parameters for figure of one row, two columns
par(mfrow=c(1,2))
# graph distribution of continuous variable vd$v1:
# box plot and histogram
boxplot(vd$v1)
hist(vd$v1)
```

Simple Real-World Example: Data from Iversen and Soskice (2006)

```
# create dataset from Iversen and Soskice

# assign c() function output to vector object country
country <- c("Australia","Austria","Belgium","Canada",
"Denmark","Finland","France","Germany","Ireland", "Italy",
"Japan","Netherlands","New Zealand","Norway","Sweden","U.K.",
"US")

# assign c() output to object gini.red for reduction in GINI
gini.red <- c(23.97,NA,35.56,21.26,37.89,35.17,25.36,18.7,NA,
              12.13,NA,30.59,NA,27.52,37.89,22.67,17.6)

# assign c() output to object wage.ineq for wage inequality
wage.ineq <- c(1.7,NA,1.64,1.82,1.58,1.68,1.94,1.7,NA,1.63,NA,
               1.64,NA,1.5,1.58,1.78,2.07)

# assign c() output to object pr for electoral system
pr <- c(0,1,1,0,1,1,0,1,0,1,0,1,0,1,1,0,0)
```

```
# use data.frame function to combine four vector objects
# assign data.frame function output to data frame is2006apsr
is2006apsr <- data.frame(country, gini.red, wage.ineq, pr)

# call data frame is2006apsr to display content
is2006apsr

# produce formatted table of descriptive statistics
# install package first and for once only
# install.packages("stargazer")

# load stargazer into R
library(stargazer)

# produce descriptive statistics table of select variables
stargazer(is2006apsr, type="text", title="Summary Statistics",
          median=TRUE, covariate.labels=c("GINI reduction",
          "wage inequality","PR system"))

# create a figure with two plots
# set graphic parameters for figure of one row, two columns
par(mfrow=c(1,2))

# graph distribution of wage inequality
boxplot(is2006apsr$wage.ineq)
hist(is2006apsr$wage.ineq)

# graph distribution of electoral system variable pr
barplot(table(is2006apsr$pr))
```

Troubleshoot and Get Help

One important principle to keep in mind in using R is that we will always have coding errors, serious or minor, such that troubleshooting and getting help is an indispensable part of using R. Knowing where to look for coding errors and where and how to get help is critical and can save one many hours. When a program fails to execute, it is common for a beginner to spend half an hour searching for a major error in their code, only to find that their error was due to a missing comma or parenthesis, a misspelled word, or a mix-up in the upper or lower case.

These occurrences are just too common to ignore. Here we provide information about common coding errors and useful resources for beginners in R.

Common Coding Errors for Beginners

When we create our first programs in R, we will definitely make errors. Learning to debug these errors is part of getting proficient with R. It is common to make programming errors in R, just like in any other software environment. The best suggestion is: Don't Panic!

It is useful to remember that there is more often than not a very simple reason for why we get an error message. R's error messages are not always clear or useful. We can always help ourselves by checking the following places to identify simple errors:

- Spelling: Go through the code to make sure spellings are correct.
- Case: R is sensitive to upper or lower case.
- Path: Is the file path correctly pointing to the right folder? Do we use double back slashes or one forward slash?
- Quotation: Should quotation marks be used? Are they symmetric (beginning and ending) ones?
- Parentheses: Are they matched? . . .

How to Get Help

Using R involves continuous learning. So knowing how to get help is important. If we have questions about a particular function, say, library(), we can type in at the prompt either *help*(*library*) or ?*library*, and R will direct us to the documentation page explaining this function.

```
> help(library)
>?library
```

More often, we need to seek help outside R itself. The last several years have witnessed an exponential increase in online communities and resources on the use of R. They are extremely valuable and useful, especially for beginner users. Here are a few possibilities.

- Rseek.org: a search engine that allows us to search for help on the official website, the CRAN, the archives of the mailing lists, and the documentation of R.
- http://www.r-bloggers.com/: R-bloggers is a blog aggregator that allows us to search for help through all articles by R-user bloggers.

- http://www.inside-r.org/: inside-R is a community site for R sponsored by Revolution Analytics that allows us to get help through searching blogs and asking questions.
- http://www.cookbook-r.com/: Cookbook for R, created by Winston Chang, provides useful information on various topics, including R basics, numbers, strings, formulas, data input and output, data manipulation, statistical analysis, graphs, scripts and functions, and tools for experiments.
- http://www.statmethods.net/: Quick-R, created by Rob Kabacoff and based on his popular book *R in Action*, provides useful information on the R code related to data input and management, basic and advanced statistics, and graphs.

Important Reference Information: Symbols, Operators, and Functions

In this section, we provide several tables to which we will refer later in the book. They concern important reference information, including symbols frequently used in R programs, common arithmetic and logical operators, and common mathematical and statistical functions. Table 1.3 provides a list of symbols frequently used in R. Table 1.4 provides a list of arithmetic operators. The order of operations follows the usual rules, with parentheses for grouping operations. Table 1.5 provides a list of logical operators. Table 1.6 provides a list of common statistical and mathematical functions. Note that these lists are by no means exhaustive.

Table 1.3 **Important Symbols in R**

Symbol	*Description*
<- or =	assignment symbol (assign function output from right to object on left)
#	comment character (indicating the code for own reference which R ignores)
$	symbol linking an R data object and a variable in the object
( )	parentheses for arguments of functions,
\\ or /	symbol for specifying path to folder or file (instead of default \ in Windows)
[ ]	square brackets for referencing entries in vectors, data frames, arrays, or lists

Table 1.4 **Arithmetic Operators**

Arithmetic	***Description***
+	addition
-	subtraction
*	multiplication
/	division
ˆ	exponentiation (raising to a power)
%/%	integer part of quotient or division
%%	remainder part (modulo)

Table 1.5 **Logical Operators**

Logical	***Description***
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	exactly equal to
!=	not equal to
!x	not x
x|y	x or y
x&y	x and y
x%in%y	binary logical operator for whether x matches y or not

Summary

This first chapter provides an overview of the steps for completing a research project, offers a one-paragraph introduction to R, shows how to install R and its add-on packages, mentions how to get help, presents an example of how to write and execute a simple R program as an ice-breaker, then demonstrates how to create, describe, and graph a variable in R with a simple numerical example; next it illustrates how to report descriptive statistics in a table, and finally it concludes with applying the R code to a real-world data example from a published article. This chapter is worth spending a lot of time on to mull over the code and make it work in R as smoothly as possible. Starting from the next chapter, we will work with larger raw datasets used in applied research. In Chapter 2, we will focus on getting a dataset ready for statistical analysis.

Table 1.6 **Common Statistical and Mathematical Functions**

Function	*Description*	*Example*
length(x)	number of values in x	length(c(3, 1, 6, 0, 6)) = 5
max(x)	maximum value of x	max(c(3, 1, 6, 0, 6)) = 6
min(x)	minimum value of x	min(c(3, 1, 6, 0, 6)) = 0
mean(x)	arithmetic mean of x	mean(c(3, 1, 6, 0, 6)) = 3.2
median(x)	median of x	median(c(3, 1, 6, 0, 6)) = 3
quantile(x, c(.25,.75))	quartile values of x	quantile(c(3, 1, 6, 0, 6), c(0.25, 0.75)) = 1, 6
range(x)	range	range(c(3, 1, 6, 0, 6)) = (0,6)
sd(x)	sample standard deviation	sd(c(3, 1, 6, 0, 6)) = 2.774887
sum(x)	sum of x	sum(c(3, 1, 6, 0, 6)) = 16
var(x)	sample variance	var(c(3, 1, 6, 0, 6)) = 7.7
abs(x)	absolute value of x	$abs(2-10) = 8$
factorial(x)	factorial of x	$factorial(10) = 3628800$
exp(x)	antilog, e raiased to a power x	$exp(2.302585) = 10$
log(x)	natural log (base e) of x	$log(10) = 2.302585$
log10(x)	log (base 10) of x	log10(100) = 2
rank(x)	find ranks to values in vector x	rank(c(3, 1, 6)) = (2 1 3)
round(x,n)	rounding x to nth digit	$round(log(10), 3) = 2.303$
rnorm(n)	n random numbers from standard normal distribution	rnorm(2): −0.4959407, −1.4102038
sqrt(x)	square root of x	$sqrt(10) = 3.162278$

Before heading in Chapter 2, it is useful for us to address some miscellaneous questions many beginning R users often come across.

Miscellaneous Q&As for Ambitious Readers

This section supplements the materials already covered or introduces some special topics beyond the scope of the main text. For supplementary materials, students do not need to learn immediately the additional materials to move forward to the next chapter, but more ambitious students will find these materials help improve their proficiency with R. For special topics, students may consult the discussion here or other online materials.

How to Update R to a Newer Version

R is a constantly evolving and improving software, which makes it necessary to update R to its newer version. The steps to update R on Windows are rather easy. First, install an add-on package called "installr"; then execute the following R code.

```
# load package
library(installr)

# update R
updateR()
```

How to Use R on Mac

To use R on a Mac machine, we have to first install R for Mac, as noted in the installation section. Once R is installed, running it on Mac requires only slight modifications to the R code for Windows. The most obvious difference is how to refer to the path to a file.

To illustrate the difference, we may first create a Project folder on the Mac desktop by clicking on the *NewFolder* option under *File* on the upper left corner of the home screen. Then to obtain the path for the Project folder, we first click on the folder, then click *File* on the upper left corner of the screen, next click the *GetInfo* option, and then copy the path right after *where* : in the *Projectinfo* page for our R program. Now take our first toy program for Windows as an example. All we need to modify in that program for Mac is to replace the Windows path with the Mac path, as follows:

```
# First R practice program, July 2016, R version 3.2.3
# show current working directory path
getwd()

# change working directory for program to project folder
# first copy the path from getinfo
# and then add the project folder name
setwd("/Users/quanli/Desktop/Project")

# show current working directory path again to verify
getwd()
```

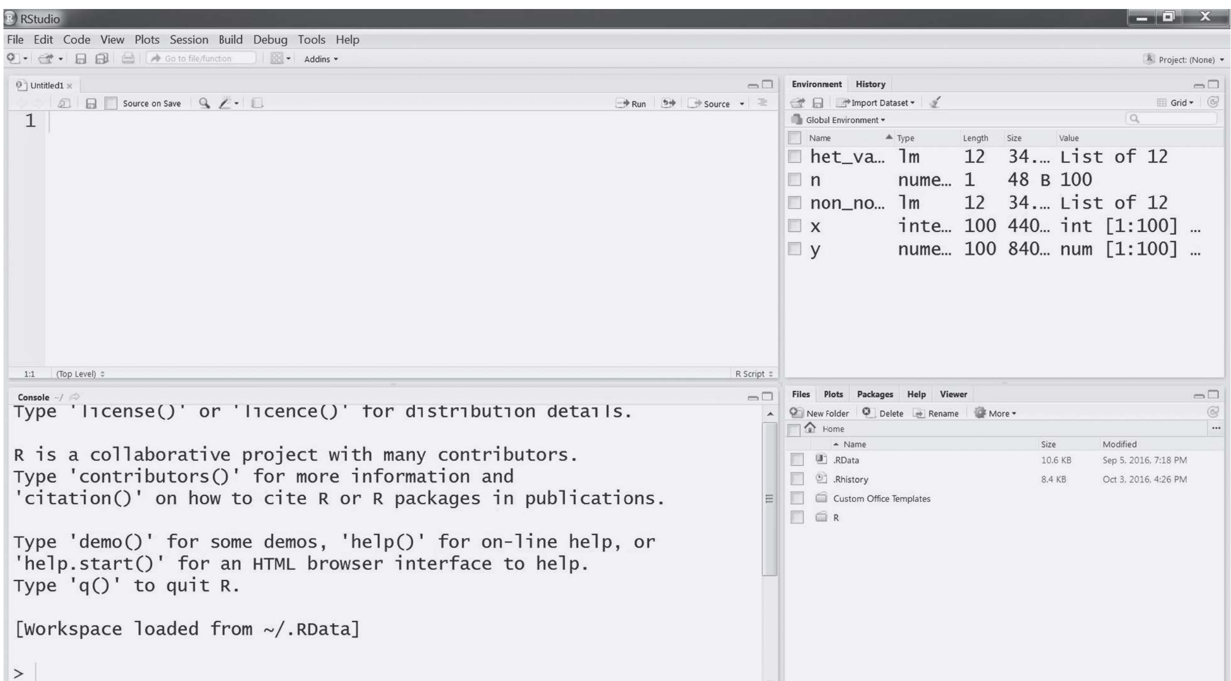


Figure 1.7 RStudio Screenshot.

We may write down this program in the R editor on Mac, which we can open by clicking on *NewDocument* option under *File*.

We can execute the R program on Mac in two ways. We can place the cursor in the line of code to be executed and then hold down two key strokes *Command* and *Return* at the same time. Alternatively, we can place the cursor in the line of code to be executed and then click *Edit* and *Execute* sequentially. To execute the entire program, we highlight the whole program and then apply either method above.

What Is RStudio, and How Do I Use It?

RStudio as an open source software provides an integrated development environment (IDE) for R. The software includes a console, syntax-highlighting editor, plotting, history, debugging, and workspace management. It certainly offers an analyst many more nice and convenient features than R does. For more information, one could visit https://www.rstudio.com/. The installation is straightforward and requires very little effort.

Figure 1.7 provides a screenshot of the RStudio interface, which is split into four panels. The upper left is the R script or program editor; the lower left is the R console for interactive use, with the version of R on top and the prompt below; the upper right is the workspace showing all active objects in the current session, as well as the history of commands used; the lower right is where tabs for files, plots, packages, help, and data viewer are located. Note that how these different panels are organized in the interface can be adjusted and re-organized. One can use RStudio just like R, but RStudio is easier to use (e.g., installing packages) and friendlier (e.g., seeing program and output at the same time).

How to Export R Output to a File

There are many ways to export R output. First, the simplest way to export all output from an R program into a text file is to employ the *sink*() function. The *sink*() function simply redirects output from the console to a file. We may insert the following two *sink*() function lines of code at the beginning and the end of the segment of an R program whose output we plan to export. We use v1 as an example below.

```
# change working directory for program to project folder
setwd("C:/Project")

# redirect and export console output to a file named output.txt
sink("output.txt")
```

```
# Simple Example: Create, Describe, and Graph a Variable
# use c() function to create a variable or vector object
c(1, 2, 0, 2, 4, 5, 10, 1)

# assign the c function output to an object named v1
v1 <- c(1, 2, 0, 2, 4, 5, 10, 1)

# call object v1 to see what is in it
v1

# restore output to the screen
sink()
```

Running this program and then opening output.txt, we will find the following output:

```
[1]  1  2  0  2  4  5 10  1
[1]  1  2  0  2  4  5 10  1
```

Two caveats are worth noting. The output between the two *sink*() functions is redirected to output.txt and thus, no longer shows up in the console. The file output.txt contains only the output from R functions but not the R code.

A second way to export R output is primarily to convert the select output into formatted statistical tables. As noted earlier, the stargazer package provides one way to export statistical output into a formatted table in the text or LaTeX format. Alternatively, we may send the computed statistics to a data object, use the write.table() function to export the data object to a tab- or comma-delimited file, and then format in Excel or Word. Furthermore, we use the *xtable*() function to create the LaTeX code for any statistical output object so that a nicely formatted table can be created in LaTeX.

A third way to export R output is to integrate the paper or report text, the R program code, and the R output into one file, via the *knitr* package plus *pandoc* for creating a Word file for the final paper or report, or via the *knitr* package plus *RStudio* for creating a .Rnw or .Rmd file that compiles into a pdf file of the final paper or report. This approach requires much greater investments in time and effort on the part of a user, but the final paper or report is typeset and professionally formatted. In fact, this R textbook was written as many chapter files in .Rnw format and then compiled into one book in pdf format.

How to Save a Graph into a File of pdf or Other Formats

To save a graph into a pdf file, we employ the *pdf*() function to first generate a graph called "graph1.pdf," and then shut down the graphing device with *dev.off*()

function so that we may open the graph file to view it in another application such as Acrobat Reader.

```
# pdf() function creates a pdf file for subsequent graph
# function output
pdf("graph1.pdf")

# hist() function creates a histogram for variable v1
hist(v1)

# dev.off() returns back to screen
dev.off()
```

Note that because we did not tell R where to save the graph file, R will then save it in the working directory specified in the *setwd*() function earlier. Alternatively, we may specify the path to another folder in the *pdf*() function.

Sometimes we need to save a graph into other formats. To do so, we simply replace the pdf line of code with any one of the following:

```
# create image files of alternative formats
bmp("graph1.bmp")
jpeg("graph1.jpeg")
png("graph1.png")
postscript("graph1.ps")
```

Alternatively, one may simply copy and paste an image from R graph output into a Microsoft Word or PowerPoint file.

Why Do I See = Also Used as an Assignment Symbol Rather Than < −?

The composite symbol of less than and minus, <-, works as the assignment symbol in R; that is, the function output on the right of the composite symbol is assigned to an R object on the left of the symbol. Starting with version 1.4.0 in 2001, however, R also allows the equal sign, =, to serve as an assignment operator, but in this book, to be consistent, we will use <- throughout.

Other Than Vector and Data Frame, What Are Some Other Data Objects Often Used in R?

Two other data objects often used are array and list. We will discuss them in detail when we come across them in future sessions.

An array is a three-dimensional object with rows by columns by height, such as several matrices stacked on top of each other. Like in a matrix, all elements in an array must be of the same data type.

A list in R is a set of objects, which could be vectors, matrices, arrays, data frames, and other lists. A list allows us to gather these objects under one name.

How to Create a Box Plot of a Variable With Respect to Each Value of Another Variable

Often we would like to compare the distribution of a continuous variable across different groups (i.e., with respect to another variable). The R code is as follows:

```
# compare distribution of x1 across different groups of x2
boxplot(x1 ~ x2, data = filename)
```

Note that since we explicitly specify the data frame name with the `data=` option, we no longer need to use the $ sign to link the dataset to each variable.

Exercises

1. Create a homework project folder and then an R program file that is pointing to and saved in that folder.
2. Read the Soskice and Iversen (2006) article, identify the definitions of the variables in Table A1, and include variable definitions as comment lines in the program file.
3. Create a dataset of all the variables in Table 1.1.
4. Produce a table of summary statistics for the dataset as in the format of Table 1.2.
5. Reproduce Figure 1.5, and then discuss the distribution of the wage inequality variable.
6. Graph one by-group boxplot figure for wage inequality and electoral system, that is, one boxplot for the PR system and the other for the majoritarian system. Discuss the differences in the distribution of wage inequality between the two systems.
7. Save the R program, exit R, and then rerun the program in R to make sure the results can be replicated.

2

Get Data Ready: Import, Inspect, and Prepare Data

Chapter Objectives

In this chapter, we will begin to work with an original dataset used in actual data analysis. Working with an original dataset helps us to observe the whole process from importing data to reporting statistical results. For illustration purposes, we will focus on the following research question: Do countries that are more open to international trade grow faster economically and have higher income? To address this question empirically, we will first need to find a dataset that contains both measures of trade openness (i.e., $\frac{exports+imports}{GDP}$) and economic growth and then get data ready for statistical analysis. Getting data ready involves importing, inspecting, and managing data. Therefore, in this chapter, we will aim to achieve the following objectives:

1. Learn to read an original dataset into R and create a corresponding data object.
2. Learn to inspect imported data by viewing raw data in a spreadsheet format, identifying dataset attributes, and graphing select variables.
3. Learn to prepare data for analysis by managing datasets, observations, and variables.
4. Get familiar with a variety of logical and mathematical operators.

Preparation

Before we discuss how to accomplish these objectives, we must make sure that we are ready to move forward from Chapter 1. We have to be prepared in two important aspects. First, we have made adequate logistic preparations such as setting up a project folder, knowing how to write and execute a program in R, and locating and downloading necessary data and codebook files. Second, we have developed a conceptual roadmap for the different possible tasks that have to be completed in Chapter 2 and their interconnections. We will discuss each of these in detail.

Logistic Preparations

Before heading into Chapter 2, we should have completed the following tasks:

1. Set up a project folder to hold data, program, and output files.
2. Create a well-documented R program that resets the working directory to the project folder, and then save the program in the project folder.
3. Execute that program in R.
4. Obtain a codebook or readme file for a dataset to be read into R.
5. Download the dataset to be read into R and place it in the project folder.

While tasks (1) to (3)—are covered in the previous chapter, tasks (4) and (5)—are new. As a matter of principle and good practice, we should always obtain the codebook or readme file for any dataset we use. The codebook or readme file should include important information such as the format of a dataset, the sample information (e.g., year and country coverage), the unit of analysis (for example, country year, individual respondent, etc.), the number of variables, the number of observations, variable names, variable definitions and measurements, variable types, value labels if any, data sources, and sometimes, descriptive statistics for the variables (mean, maximum, minimum, variance or standard deviation, number of observations).

A codebook is important for three reasons. First, we can make sure a dataset suits our purpose in terms of variable and sample coverage. Second, we can choose the right command or function to import a dataset into R and verify whether it is read into R correctly or not. Finally, we can refer to the codebook when we prepare data for analysis. R does not do a very good job in handling variable labels. So we need the codebook to know what variables we have and how to manage them.

In social and behavioral sciences, raw data is often organized in a tabular or rectangular form, where each row is an observation and each column a variable. In this chapter, we will discuss how to import tabular data files in formats that are common in social and behavioral sciences, including comma- or tab-delimited text files, fixed-width text files, Excel files, Stata and SPSS files. But our first focus is to learn to import a commonly used original dataset on economic variables in the comma-delimited format, the Penn World Table Version 7.0 database.

The Penn World Table Version 7.0 database can be found at the following link: http://www.rug.nl/ggdc/productivity/pwt/pwt-releases/pwt-7.0. In the event that the link does not work because of site changes, readers may find it by googling it on the internet. The Penn World Table dataset was initially constructed by Alan Heston, Robert Summers, and Bettina Aten at the Center for International Comparisons of Production, Income and Prices at the University of Pennsylvania, and it has now been taken over by the University of California,

Davis and the University of Groningen. The dataset provides a large number of variables related to purchasing power parity and national income accounts converted to international prices for 189 countries and territories for the period 1950–2009, with 2005 as the reference year. It has been widely used by many scholars in social science disciplines. For our purposes, the dataset contains variables that measure trade openness and national income of many countries in different years.

On the website listed in the preceding paragraph, we will choose to download PWT7.0 data and readme files in one zipped file by clicking on "Complete Data Download." After downloading the zipped file, place it into the project folder. The next step depends on whether we are working with R in Windows or on a Mac machine. If we are using the Windows operating system, we need to download and install the 7zip freeware from this website: http://www.7-zip.org/, and then unzip the PWT7.0 zip file. After unzipping the PWT7.0 zip file in the project folder, we should see two files. One is `pwt70_Vars_forWeb.xls`, a readme or codebook file that defines variable names and labels; the other one is `pwt70_w_country_names.csv`, a comma-delimited data file that places variable names in the header row, observations in rows, and variables in columns. We will import the data file into R later.

Alternatively, if we are using a Mac machine, we do not need to download any additional software to unzip the file, but we do need to place the two files individually into the project folder. If we don't do this, R will give an error message when reading the data file because R is not able to find the data file inside the zip file.

Conceptual Overview of Chapter 2

Chapter 2 will help us learn how to import a dataset into R, how to inspect the imported data, and how to manage variables, observations, and datasets. Because Chapter 2 involves a large quantity of information, it can be difficult for us to see the big picture. As a result, we should develop a conceptual overview of how the different tasks, functions, and methods are related to each other in the overall scheme of things. For this purpose, Table 2.1 lists various data preparation tasks, add-on packages that need to be installed first, and relevant R functions and methods. For the sake of presenting a wholistic view, Table 2.1 includes not only the functions and methods discussed in the main text of Chapter 2 but also those covered in the Miscellaneous Q&A section.

Import Penn World Table 7.0 Dataset

Importing datasets into R is the first biggest hurdle for many beginners. It is not a trivial issue. Yet when a dataset is finally successfully read into R, sometimes

Table 2.1 **List of Data Preparation Tasks and Related R Functions**

Tasks	*Functions and Methods*
Import data	Add-on packages: foreign, Hmisc, readstata13, RODBC, gdata, haven, XLConnect
comma-delimited file	read.csv, read.table
tab-delimited file	read.table
Stata file	stata.get, read.dta, read.dta13
SPSS file	spss.get
SAS file	read.xport, read_sas
Excel file	odbcConnectExcel, sqlFetch, odbcClose, read.xls, readWorksheet
R data file	load
Inspect data	Add-on packages: None
view imported data	View, head, tail
learn dataset attributes	dim, names, str
plot select variables	hist, boxplot, qqnorm, par
edit imported data	fix
Manage datasets	Add-on packages: reshape2, DataCombine
sort data	order
select rows and columns	indexing method, logical operators, c
create a subset dataset	indexing method, operators, assignment, subset
merge datasets	merge
reshape data from long to wide	dcast
reshape data from wide to long	melt
Manage observations	Add-on packages: None
remove select observations	indexing method, operators, assignment
find and remove duplicates	duplicated, !duplicated
Manage variables	Add-on packages: Hmisc, car, reshape, pwt
create new variables	mathematical and logical operators, assignment, factor
show variable type	class
create leading and lagged variables	slide
create group-specific statistics	by, aggregate
rename variables	indexing method, names, assignment, rename
recode variable values	indexing method, assignment, recode
create variable labels	label

after much struggle, it can provide a great sense of achievement for students. To learn the importation of datasets into R, we will start with the comma-delimited file `pwt70_w_country_names.csv`.

Before importing the dataset, we must first finish a few tasks: (1) Install only once the add-on packages that will be used in the chapter. Assuming the packages have all been installed, the code `install.packages()` is currently commented out; otherwise, remove the comment symbol, run this line of code once, and then put the comment symbol back in. (2) Clean R's workspace—its temporary work area. (3) Set the current working directory to the project folder. It is a good practice to always start with these tasks at the beginning of a new program.

```
# install following packages only once, then comment out code
# install.packages(c('reshape2', 'DataCombine',
# 'Hmisc', 'haven', 'foreign', 'gdata', 'XLConnect',
# 'pwt', 'reshape', 'doBy'), dependencies=TRUE)

# remove all objects from workspace
rm(list = ls(all = TRUE))

# change working directory to point to project folder
setwd("C:/Project")
```

The ls() function returns a vector of the names of the objects in the current R session, with `all=TRUE` referring to all objects created. The rm() function removes all objects listed in the output of the ls() function. Hence, the line of code above tells R to remove all the objects from the workspace.

The setwd(), discussed extensively in the previous chapter, resets the current working directory in R. In the rest of the program, R will seek to find files in that directory without us specifying the pathname each time. In this case, we first created a folder named `Project` in the C drive, where the data files are stored.

To read the Penn World Table dataset into R and create a data object called `pwt7`, we will employ the following code:

```
# import comma-delimited file, create data object pwt7
pwt7 <- read.csv("pwt70_w_country_names.csv", header = TRUE,
    strip.white = TRUE, stringsAsFactors = FALSE,
    na.strings = c("NA", ""))
```

Since this code can be confusing to understand, we will provide a detailed discussion of each element in the code below.

1. The read.csv() function reads a comma-delimited data file into R. The arguments of the function are listed inside the parentheses.
2. The argument `"pwt70_w_country_names.csv"` represents the raw data file to be read into R, where .csv indicates that it is a comma-delimited file (a common file format option, which we can easily see in Excel when choosing to save a data file in Excel). Note that if we did not first specify `setwd("C:/Project")`, the first argument would have to be `"C:/Project/pwt70_w_country_names.csv"` instead. Be sure that the file name is surrounded by double quotation marks!
3. The argument `header=TRUE` is a logical statement, meaning that it is true that the first row contains variable names. If the raw data file does not have variable names in the first row, change the option to `header=FALSE`. We may simplify the argument to `header=T` or `header=F`.
4. The argument `strip.white=TRUE` removes any white space at the start and end of character fields. Spaces are often introduced accidentally during data entry. R treats the following three as different values: **"USA"** and **" USA"** or **"USA "** because of the absence or presence of white space before or after USA. We further illustrate this concept in the Miscellaneous Q&A section.
5. The argument `stringsAsFactors = FALSE` tells R to keep character variables as they are, rather than converting them to factors. We will explain what factors are later on.
6. The argument `na.strings = c("NA","")` tells R to treat the following two types of values as missing: "NA" which is R's default recognized coding of a missing value; blank space or empty string, denoted by the double quotation marks and without any white space in between. By default, R treats a blank cell in a column of character data as a character string of zero length rather than as missing. Finally, note how the c() function allows us to specify multiple symbols, all of which are coded as missing and separated by commas. Missing values are a common occurrence in social science data, with different software packages treating missing values a little bit differently. For R, NA is the default missing value code. If we find a dataset uses some other value or symbol to represent a missing value, we must tell R so that the data will be read into R correctly. For example, if datasets use . or −999 to represent missing values, we can tell R simply by specifying `na.strings="."` or `na.strings="-999"`. Alternatively, we can use `na.strings = c("NA", "", ".", "-999")` instead.
7. The data output from the read.csv function is assigned via `<-` into a data object arbitrarily named as `pwt7`. The data object `pwt7` is a data frame, which is a matrix style data object in R. Alternatively, a data frame can be thought of in simpler terms as a table in Excel. Once the new data object `pwt7` is created, we can ask R to operate on it for analysis. If we only use the read.csv() function part of the code without assigning its output to a data object, R will merely display the dataset on the screen.

8. To read data into R smoothly, we need to follow R's rules for naming variables inside the raw dataset. A proper variable name can use letters, numbers, and dot or underline characters, but should never contain mathematical operators. Also, R does not like the presence of any blank space inside a variable name. Another factor worth considering is R's sensitivity to upper- and lower-case. For instance, `name` and `Name` are two different variables to R. Thus, it is common for an analyst to have to change some variable names in a raw dataset before reading it into R. In this type of situation, it is recommended that the analyst makes a copy of the original raw dataset and operates on the copied data file instead, leaving the original dataset intact.

It is important to note that the datasets we work with are often in various formats other than in a comma-delimited file. As one of its great strengths, R is extremely flexible and versatile in its ability to import datasets of various formats, as suggested by Table 2.1. As of right now, we will not engage the R code for importing datasets of different formats, but will learn that later in this chapter.

Inspect Imported Data

Eyeball Imported Dataset

After we import a dataset, it is both useful and important to take a direct look at the imported dataset itself. Examining the data provides an intuitive means for us to verify that the data has been imported correctly. Through quick glances, we can get a feel for the data, and spot check if observations have been imported correctly and whether variable names appear to be correct or not. There are many ways for us to look through and spot check an imported dataset.

The first method of eyeballing imported data is to open up a spreadsheet style viewer using the View() function. Note that the first letter of the function must be in uppercase.

```
# inspect dataset pwt7 in a spreadsheet style data viewer
View(pwt7)
```

Figure 2.1 shows that the dataset `pwt7` is organized with observations in rows and variables in columns, a general principle worth remembering. The header row contains the variable names.

Another way of eyeballing the data is to use two functions, head() and tail(), to look at the first and last couple of observations, respectively. The option `n=` allows us to specify how many observations we choose to look at.

	country	isocode	year	POP	XRAT	Currency_Unit	ppp	tcgdp
1	Afghanistan	AFG	1950	8150.368	NA	NA	NA	NA
2	Afghanistan	AFG	1951	8284.473	NA	NA	NA	NA
3	Afghanistan	AFG	1952	8425.333	NA	NA	NA	NA
4	Afghanistan	AFG	1953	8573.217	NA	NA	NA	NA
5	Afghanistan	AFG	1954	8728.408	NA	NA	NA	NA
6	Afghanistan	AFG	1955	8891.209	1.680000e-02	Afghani	NA	NA
7	Afghanistan	AFG	1956	9061.938	2.000000e-02	Afghani	NA	NA
8	Afghanistan	AFG	1957	9240.934	2.000000e-02	Afghani	NA	NA
9	Afghanistan	AFG	1958	9428.556	2.000000e-02	Afghani	NA	NA
10	Afghanistan	AFG	1959	9624.606	2.000000e-02	Afghani	NA	NA

Figure 2.1 Using View() Function to View Raw Data.

```
# list first one observation in dataset pwt7
head(pwt7, n = 1)

# list last one observation in dataset pwt7
tail(pwt7, n = 1)
```

As an illustration, the R output for listing the last observation of `pwt7` looks as follows:

```
# list last one observation in dataset pwt7
tail(pwt7, n = 1)

       country isocode year   POP    XRAT   Currency_Unit
11400 Zimbabwe     ZWE 2009 11383 1.4e+17 Zimbabwe Dollar
           ppp   tcgdp     cgdp    cgdp2     cda2       cc
11400 40289.96 1906.05 167.4471 174.4197 180.2302 81.78356
            cg       ci        p       p2       pc       pg
11400 7.759873 13.78792 2.87e-11 2.87e-11 2.91e-11 1.34e-11
            pi    openc     cgnp         y       y2
11400 3.55e-11 60.31122 94.11682 0.3670967 0.382383
         rgdpl   rgdpl2  rgdpch       kc       kg       ki
11400 142.5955 136.7233 142.564 86.89917 7.905525 14.74367
         openk rgdpeqa rgdpwok rgdpl2wok rgdpl2pe rgdpl2te
11400 83.74953 182.613      NA        NA       NA 314.1711
      rgdpl2th   rgdptt
11400       NA 151.4353
```

As shown in the output, the last observation in `pwt7` (i.e., observation 11400) is a country called Zimbabwe in year 2009, with isocode ZWE, a population (POP) of 11383 (thousands), an exchange rate to US dollar (XRAT) being 1.4e+17, etc.

If we would like to see more observations at the beginning or end of the dataset, we can simply change the value after `n=`.

Identify Dataset Attributes: Dimension, Variable Names, and Dataset Structure

After a dataset has been read into R and we have visually inspected the imported data, we need to further verify that R imported the dataset correctly. We can get summary information about the dataset and then verify it against the information in the codebook. To do so, we introduce three functions, i.e., dim(), names(), and str(). They help us to learn about the dimensions of the dataset (its rows and columns, i.e., number of observations and number of variables), variable names, and the structure of the dataset, respectively.

```
# dimensions of pwt7: number of observations and number of
# variables
dim(pwt7)

# variable names in dataset pwt7
names(pwt7)

# structure of dataset pwt7
str(pwt7)
```

The R output looks as follows:

```
# dimensions of pwt7: number of observations and number of
# variables
dim(pwt7)

[1] 11400    37
```

The output of `dim(pwt7)` shows that the dataset includes 11,400 observations and 37 variables.

```
# variable names in dataset pwt7
names(pwt7)

 [1] "country"       "isocode"       "year"
 [4] "POP"           "XRAT"          "Currency_Unit"
 [7] "ppp"           "tcgdp"         "cgdp"
[10] "cgdp2"         "cda2"          "cc"
[13] "cg"            "ci"            "p"
[16] "p2"            "pc"            "pg"
[19] "pi"            "openc"         "cgnp"
[22] "y"             "y2"            "rgdpl"
[25] "rgdpl2"        "rgdpch"        "kc"
[28] "kg"            "ki"            "openk"
[31] "rgdpeqa"       "rgdpwok"       "rgdpl2wok"
[34] "rgdpl2pe"      "rgdpl2te"      "rgdpl2th"
[37] "rgdptt"
```

The output of the `names(pwt7)` function lists the names of all 37 variables of `pwt7`, which should be identical to those in the codebook. The numerical values inside the brackets correspond to the indexed column positions of the respective variables next to the brackets. For example, `[1]` means the variable `country` is in column 1, thus the variable `year` at the end of that row is in the 3rd column.

The 37th column contains the last variable rgdptt. This is the first time we have come across the use of square brackets in R. Since we will rely heavily on the use of brackets in this book, it is important to remember that R uses the square brackets [] to reference entries in vectors and data frames, an issue we will discuss extensively below.

```
# structure of dataset pwt7
str(pwt7)

'data.frame': 11400 obs. of  37 variables:
 $ country      : chr  "Afghanistan" "Afghanistan" "Afghanistan"
                  "Afghanistan" ...
 $ isocode      : chr  "AFG" "AFG" "AFG" "AFG" ...
 $ year         : int  1950 1951 1952 1953 1954 1955 1956 1957 1958
                  1959 ...
 $ POP          : num  8150 8284 8425 8573 8728 ...
 $ XRAT         : num  NA NA NA NA NA 0.0168 0.02 0.02 0.02 0.02 ...
 $ Currency_Unit: chr  NA NA NA NA ...
 $ ppp          : num  NA NA NA NA NA NA NA NA NA NA ...
 $ tcgdp        : num  NA NA NA NA NA NA NA NA NA NA ...
 $ cgdp         : num  NA NA NA NA NA NA NA NA NA NA ...
 $ cgdp2        : num  NA NA NA NA NA NA NA NA NA NA ...
 $ cda2         : num  NA NA NA NA NA NA NA NA NA NA ...
 $ cc           : num  NA NA NA NA NA NA NA NA NA NA ...
 $ cg           : num  NA NA NA NA NA NA NA NA NA NA ...
 $ ci           : num  NA NA NA NA NA NA NA NA NA NA ...
 $ p            : num  NA NA NA NA NA NA NA NA NA NA ...
 $ p2           : num  NA NA NA NA NA NA NA NA NA NA ...
 $ pc           : num  NA NA NA NA NA NA NA NA NA NA ...
 $ pg           : num  NA NA NA NA NA NA NA NA NA NA ...
 $ pi           : num  NA NA NA NA NA NA NA NA NA NA ...
 $ openc        : num  NA NA NA NA NA NA NA NA NA NA ...
 $ cgnp         : num  NA NA NA NA NA NA NA NA NA NA ...
 $ y            : num  NA NA NA NA NA NA NA NA NA NA ...
 $ y2           : num  NA NA NA NA NA NA NA NA NA NA ...
 $ rgdpl        : num  NA NA NA NA NA NA NA NA NA NA ...
 $ rgdpl2       : num  NA NA NA NA NA NA NA NA NA NA ...
 $ rgdpch       : num  NA NA NA NA NA NA NA NA NA NA ...
 $ kc           : num  NA NA NA NA NA NA NA NA NA NA ...
 $ kg           : num  NA NA NA NA NA NA NA NA NA NA ...
 $ ki           : num  NA NA NA NA NA NA NA NA NA NA ...
 $ openk        : num  NA NA NA NA NA NA NA NA NA NA ...
 $ rgdpeqa      : num  NA NA NA NA NA NA NA NA NA NA ...
 $ rgdpwok      : num  NA NA NA NA NA NA NA NA NA NA ...
 $ rgdpl2wok    : num  NA NA NA NA NA NA NA NA NA NA ...
 $ rgdpl2pe     : num  NA NA NA NA NA NA NA NA NA NA ...
```

```
 $ rgdpl2te     : num  NA NA NA NA NA NA NA NA NA NA ...
 $ rgdpl2th     : num  NA NA NA NA NA NA NA NA NA NA ...
 $ rgdptt       : num  NA NA NA NA NA NA NA NA NA NA ...
```

The output of the `str(pwt7)` function provides a lot of information about the nature and structure of `pwt7`, showing its type as a data object (data.frame), its number of observations (11400 obs.), its number of variables (37), the name of each variable (right after each `$` sign), the type of each variable (right after each colon: `chr` means character, `int` integer, `num` numeric), and the first few observations of each variable (right after each variable type). Notice how R treats some variables as character such as `country`, `isocode`, `Currency_Unit` (denoted as `chr`), some others as integer such as `year` (denoted as `int`), and still others as numeric (denoted as `num`). Also of note is how missing values are denoted as `NA`.

For specific definitions of the variables, please refer to the readme file downloaded earlier. For the purpose of this chapter, we will pay particular attention to three variables: `country`, `year`, and `rgdpl`. The variable `country` is a character string, containing the names of countries. The variable `year` is an integer variable, showing years. The variable `rgdpl` is a numeric variable for the real GDP per capita in international dollars, which is widely used as a measure of income per person in a country. Both `country` and `year` are id variables that allow us to uniquely identify each observation. The variable `rgdpl` is the focus of our discussion in this section.

Graph a Variable of Interest for Inspection

We may also graph the distributions of select variables of interest to inspect whether they are imported into R correctly. For example, we can use some code from the previous chapter to plot the distribution of `rgdpl`, as shown in Figure 2.2. Note how we first use the par() function to specify graphical parameters for a figure with plots arranged into various rows and columns. Recall that `par(mfrow=c(2,2))` means to create a figure containing plots that are arranged into two rows and two columns.

```
# create a figure with three plots
# set graphic parameters for figure of two rows, two columns
par(mfrow=c(2,2))

# graph distribution of variable rgdpl in data frame pwt7
hist(pwt7$rgdpl)
boxplot(pwt7$rgdpl)
qqnorm(pwt7$rgdpl)
```

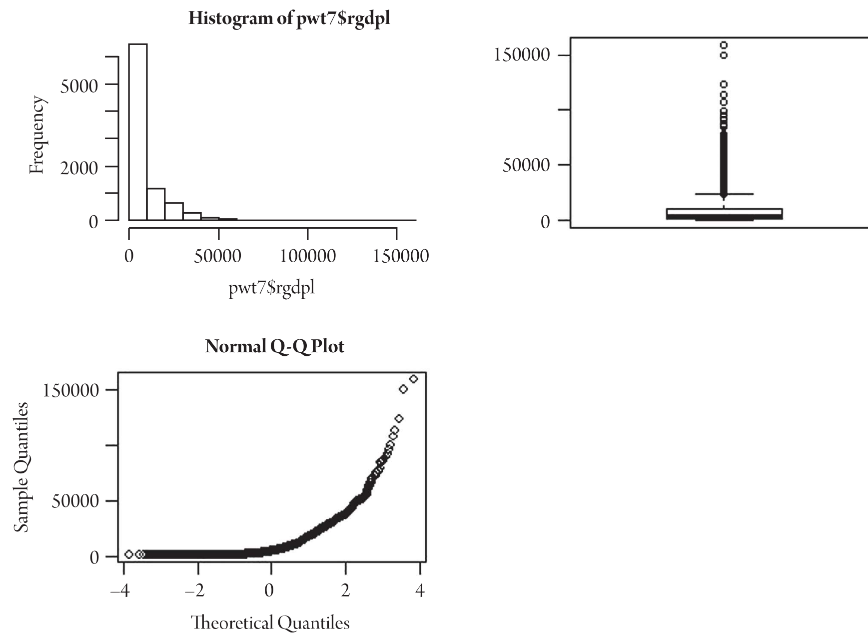


Figure 2.2 Distribution of Variable rgdpl.

Prepare Data I: Variable Types and Indexing

Preparing the imported data for analysis can be time-consuming and involves data manipulation and cleaning. Data management typically focuses on getting variables, observations, and datasets in order, with the end goal of getting a clean dataset ready for analysis. Before we go into the specifics of data management, we first need to learn some prerequisite information, primarily regarding variable types and the indexing method for referring to observations and variables.

Variable Types in R

R uses a variety of data objects, including scalar, vector, matrix, array, data frame, and list. For our purposes, we will focus on variables inside a data frame. Typically in statistics, variables are nominal, ordinal, or continuous. Nominal variables are categorical and their values are not ordered in any meaningful way. For example, in the `pwt7` dataset, the variable `country` is nominal because its values are just country names and not rank ordered in terms of magnitude. Ordinal variables could also be categorical, but their values are rank ordered in terms of magnitude; yet ordinal variables do not carry any information about the amount of difference between values, unlike continuous variables. For example,

we may create an ordinal variable for income group based on real GDP per capita in the `pwt7` dataset, with the purpose of classifying countries into different income categories. High-income countries, for example, are rank ordered as being wealthier than low-income ones, but their exact amount of difference is unknown. Finally, a continuous variable may take on any value in a certain range, with the difference between two values representing both rank order and size of magnitude. The variable `rgdpl` is a good example, with the difference between its two values indicating not only which one is larger but also how much larger.

Now, R has its own ways of defining variable types. For R, variables in a data frame can be classified as character, factor, or numeric. For example, the variable `country` is a character variable, as noted earlier, which is the same as a nominal variable. Its values are country names or strings. The variable `rgdpl` is a numeric variable, which is the same as a continuous variable. While these two variable types are rather conventional, the factor variable in R is somewhat distinct. A factor variable in R is a categorical variable and is a vector of integer values with corresponding character values. Both character and numeric variables can be converted into factor variables, but a factor variable's levels are always character values. For example, both the nominal variable `country` and the ordinal variable income group discussed previously can be turned into factor variables in R. The most important use of the factor variable is that it can be used in statistical modeling directly. This even applies to a variable like `country` after it is converted into a factor variable. We will discuss the issue of variable types further in the next section.

Indexing

Knowing how R refers to the observations in a variable or the observations and variables in a dataset is critical. In this section, we will learn about indexing, which is the most basic method of reference in R, and we will rely on this referencing or indexing method throughout the book. In the indexing method, we place the position values or identifying conditions inside a pair of square brackets `[]` to reference entries in variables or data frames.

Referencing Observation in Variable

We will start with how to reference an observation in a variable. For illustration, we use the population variable in dataset `pwt7`, denoted as `pwt7$POP`, as an example. As shown in Figure 2.1, the variable's fifth observation equals 8728.408. We may refer to the fifth observation of `pwt7$POP` in two alternative methods, via either its position value or other unique identifying conditions.

For the first method, we specify the variable name, followed by a pair of square brackets and with the position value of its fifth observation (i.e., 5) placed inside.

To ask R to display the value of that observation, the R code and output are as follows:

```
# reference fifth observation of pwt7$POP
pwt7$POP[5]

[1] 8728.408
```

We may also refer to the fifth observation of `pwt7$POP` by specifying unique identifying conditions. For example, the observation is uniquely identified by relevant information for the `country` and `year` variables, i.e., Afghanistan in 1954. Hence, we may refer to that observation by placing the unique identifying information for `country` and `year` inside the square brackets. The R code and output are as follows:

```
# reference specific observation of pwt7$POP
pwt7$POP[pwt7$country == "Afghanistan" & pwt7$year == 1954]

[1] 8728.408
```

As expected, both referencing methods produce the same output value: 8728.408. The first method requires the position value of an observation in a variable, and the second method requires the corresponding values of the unique identifying variables. For the latter, note how two logical operators (double equal sign and &) are used; also note how the value of the character variable `country` must be surrounded by quotation marks, but not that of the numeric variable `year`.

Referencing Rows and Columns in Dataset

To reference entries in a data frame is different because a data frame has both rows and columns. Recall from Figure 2.1, the data frame `pwt7` is a rectangle of data where rows represent observations and columns represent variables. Similar to the case with variable or vector, we also refer to a specific observation of a specific variable in a dataset in two alternative methods, via either relevant position values or unique identifying conditions.

Once again, take the fifth observation of the population variable in the `pwt7` dataset as an example. We can refer to that entry in the context of a dataset rather than a variable. In the first method, we specify the data frame name (instead of a variable name), followed by a pair of square brackets, with the position values of the relevant row and column placed inside and separated by a comma. As shown in Figure 2.1, the fifth observation of the population variable lies in the intersection of the fifth observation and the fourth variable. The R code

and output are as follows. Note how inside the square brackets, the two position values are observation first and variable second.

```
# reference fifth observation of fourth variable in pwt7
pwt7[5, 4]

[1] 8728.408
```

In the second method, we refer to a specific entry of a dataset by specifying unique row and column conditions separated by a comma. For example, the fifth observation and the fourth variable of `pwt7` is uniquely identified by the relevant row information based on `country` and `year` and the relevant column information (`POP`). Hence, we may refer to that entry by placing the relevant row and column conditions inside the square brackets and separated by a comma. The R code and output are as follows. Note how the row condition comes first and the column condition second, with the column variable name surrounded by quotation marks. Please remember the row and column conditions must be separated by a comma.

```
# reference certain observation of certain variable in pwt7
pwt7[pwt7$country=="Afghanistan" & pwt7$year==1954, "POP"]

[1] 8728.408
```

As expected, both referencing methods produce the same output value: 8728.408. It is worth noting that even though the four lines of code produce the same output value, the first two lines concern how to reference an element in a vector, and the latter two lines concern how to reference an entry in a dataframe.

To generalize, `dataframe[Select Rows, Select Columns]` allows us to refer to specific rows and columns from any data frame, with position values or identifying conditions placed inside square brackets and separated by a comma. The index value or condition before the comma refers to specific rows or observations, and the index value or condition after the comma refers to specific columns or variables.

As an extension, if we leave an index position empty, it refers to all rows or columns. For example, `pwt7[5,]` refers to the fifth observation of all variables in `pwt7`; the empty index position after the comma tells R to include all variables. In contrast, `pwt7[,4]` refers to all observations of the fourth variable in `pwt7`; the empty index position before the comma tells R to include all observations. We may use various conditional expressions in these index positions so that we can refer to observations and variables in a dataset in a variety of ways. We will learn more about this issue in the next section. Needless to say, data management relies heavily on the indexing method.

Prepare Data II: Manage Datasets

Sort Observations

Before any data manipulation, it is a good practice to sort data according to a dataset's key ID variables. The key ID variables refer to those by which each observation of a dataset is uniquely identified. Take the `pwt7` dataset as an example. Its key ID variables are `country` and `year`, which uniquely identify each row in the dataset. Thus we can sort `pwt7` first by `country` and then by `year` within each country, both in ascending order. We can use the order() function, applied to all rows and all columns of `pwt7`. Note that in order to denote all columns, we need to leave a blank entry right after the comma inside the brackets.

```
# sort data first by country and then by year
pwt7 <- pwt7[order(pwt7$country, pwt7$year), ]
```

Now if necessary, we could also sort the dataset in descending order by year. In the code below, we add a minus sign in front of `year` to indicate descending order.

```
# sort data by country (ascending) and by year (descending)
pwt7 <- pwt7[order(pwt7$country, -pwt7$year),]
```

Select Observations and Variables

When we work with an original dataset in research, we often need to choose observations and variables for several purposes. First, we want to find out the values of certain observations that are of special interest to us. For example, using the Penn World Table data, we may want to know the value of GDP per capita for a certain country in a certain year. Second, we often do not need all the observations and all the variables, particularly if the original dataset is too large. Keeping all observations and variables can be unwieldy and costs computing time. Thus, we often choose select observations and variables to form a new dataset for analysis.

There are many ways to choose observations and variables in R. Here we illustrate how to use the most basic indexing method introduced earlier with several examples. The first example below demonstrates how to select the 100th and 102nd observations of all variables in `pwt7`.

```
# show select rows; example: observations 100 and 102
pwt7[c(100, 102), ]
```

Recall from Chapter 1 that the c() function combines the arguments inside the parentheses together into a vector. Also, note how the second index position

after the comma is blank, which tells R to include all columns or variables for observations 100 and 102.

If we want to choose observations from 100 to 102, all we have to do is to replace the comma inside the c() function with a colon, which identifies continuous rows.

```
# show select rows; example: observations from 100 to 102
pwt7[c(100:102), ]
```

The second example concerns how to select all observations of certain variables in a data frame. The example below asks R to combine variables `country`, `year`, and `rgdpl` from `pwt7` and show all their observations. In the first index value position inside the brackets, we use a blank entry to reference all rows or observations; in the second index value position, we use the c() function to select variables. The quotation marks, brackets, and commas help to tell R what to do.

```
# show select variables
pwt7[, c("country", "year", "rgdpl")]
```

The third example shows how we can combine the two examples above to select observations from 100 to 102 for three variables `country`, `year`, and `rgdpl` only.

```
# combine two selections: certain rows and certain columns
pwt7[c(100:102), c("country", "year", "rgdpl")]
```

Now let us see some more selection examples. If we want to show the value of real GDP per capita in purchasing power parity for Afghanistan since 2006, we can use the following code:

```
# show certain observations and certain variables that meet
# conditions: Afghanistan since 2006 for three variables
pwt7[pwt7$year>=2006 & pwt7$country=="Afghanistan",
c("country", "year", "rgdpl")]
```

We use >=, &, and == to specify the conditions for observations in terms of years and countries. Specifically, we use `pwt7$year>=2006` to select years since 2006, `pwt7$country=="Afghanistan"` to select Afghanistan, and the & sign to specify that both conditions must be met for the observations. The R output looks like the following:

```
       country year    rgdpl
57 Afghanistan 2006  687.7274
58 Afghanistan 2007  736.4802
```

```
59 Afghanistan 2008 1009.8645
60 Afghanistan 2009 1170.9935
```

Now let us see a more complicated example. If we want to compare the values of GDP per capita of China and India for five different years, we can use the %in% as a logical operator to find those items from its left that match those to its right. In the code below, we choose years that equal to 1970, 1980, 1990, 2000, or 2009 and countries that are "India" or "China Version 1".

```
# select multiple countries and non-consecutive years
# for select variables
pwt7[pwt7$year %in% c(1970, 1980, 1990, 2000, 2009) &
    pwt7$country %in% c("India", "China Version 1"),
    c("country", "year", "rgdpl")]
```

The output looks like the following:

```
             country year     rgdpl
2121 China Version 1 1970  390.4169
2131 China Version 1 1980  640.2915
2141 China Version 1 1990 1262.7539
2151 China Version 1 2000 2888.3145
2160 China Version 1 2009 7008.1742
4641           India 1970  887.1750
4651           India 1980 1019.6258
4661           India 1990 1407.2194
4671           India 2000 1860.2439
4680           India 2009 3237.8371
```

Now if we are interested in all years between 1970 and 1990 instead, we just replace `c(1970, 1980, 1990, 2000, 2009)` with `c(1970:1990)`.

It is worth noting that we can use a variety of logical operators to pick and choose rows or observations and columns or variables from a data frame. Recall from Table 1.5 some of the most common logical operators: < (less than), <= (less than or equal to), > (greater than), >= (greater than or equal to), == (equal to), != (not equal to), | (or), & (and).

Create New Dataset with Subset of Data

Now that we have learned how to select certain observations and variables, it is time to learn how to take those selections and create a second dataset. That turns out to be really simple. The following line of code tells R to choose all observations

of three variables from pwt7 and assign them to a new dataset called pwt7new, which includes all observations but only three variables from pwt7.

```
# create a new dataset with three select variables
pwt7new <- pwt7[, c("country", "year", "rgdpl")]
```

If we use the same data frame name pwt7 for the new dataset, then the new dataset will overwrite the old dataset with the same name.

Merge Datasets

In applied data analysis, we often have to merge datasets because the variables we need come from different sources. Below we will provide an example of how to merge datasets in R through the use of the merge() function.

Suppose we use pwt7 to create two datasets in R, called pwt7.tmp1 and pwt7.tmp2, respectively, each with a different variable but two common merging ID variables (isocode and year).

```
# create two temporary pwt7 subsets for merging example
pwt7.tmp1 <- pwt7[, c("isocode", "year", "rgdpl")]
pwt7.tmp2 <- pwt7[, c("isocode", "year", "openk")]
```

To merge these two datasets, we can apply the merge() function and match merge observations from the two different datasets according to isocode and year.

```
# merge two datasets
pwt7.m <- merge(pwt7.tmp1, pwt7.tmp2, by = c("isocode",
    "year"), all = TRUE, sort = TRUE)
```

Inside the merge() function, we first specify the two datasets to be merged and then utilize the by= option and the c() function to specify the sorting ID variables according to which the two datasets are to be merged. Next, through the use of all=TRUE, we tell R that all extra unmatched observations from both datasets should be kept. Finally, the option sort=TRUE tells R that the merged dataset should be sorted by the sorting ID variables.

It is worth noting that this is a rather contrived example. A more practical example on merging datasets will be discussed later when we try to combine a dataset on group-specific statistics with the pwt7 dataset.

Two other functions—cbind() and rbind()—are sometimes used in combining datasets. Here we provide some generic examples. In the event that we want to combine horizontally two matrices with an equal number of rows but different columns, we can use the cbind() function without referring to any sorting ID.

In contrast, if we want to append vertically two datasets of an equal number of columns but different rows, we can use the rbind() function without using any sorting id. Note that in the R code below, data1 and data2 are just generic data frame names and do not refer to any specific data objects. Hence, executing these two lines of code without modification will produce an error message in R.

```
# combine datasets of equal observations, different variables
new.data <- cbind(data1, data2)

# append datasets of equal variables, different observations
new.data <- rbind(data1, data2)
```

Reshape Data Structure

We often need to reshape how a dataset is structured for analytical purposes. Suppose we are interested in comparing the values of GDP per capita, measured by `rgdpl`, between two countries over time. The most intuitive structure is to have their `rgdpl` values listed side by side for each year. This requires reorganzing the dataset structure. We will provide some examples by using some functions from the add-on package reshape2.

To illustrate how reshaping works, we first create a new dataset called `pwt7.ip`, which contains six years and three variables for two countries: India and Pakistan. It is organized by `country` and `year` and thus, is in a long form as displayed below. The R code and output are as follows:

```
# create a subset of pwt7 (India and Pakistan, three
# variables and six years)
pwt7.ip <- pwt7[pwt7$year %in% c(1950, 1960, 1970, 1980,
    1990, 2000) & pwt7$country %in% c("India", "Pakistan"),
    c("country", "year", "rgdpl")]

# display the long form dataset
pwt7.ip

      country year     rgdpl
4621    India 1950  594.1658
4631    India 1960  713.6806
4641    India 1970  887.1750
4651    India 1980 1019.6258
4661    India 1990 1407.2194
4671    India 2000 1860.2439
7681 Pakistan 1950  732.1559
```

```
7691 Pakistan 1960  732.4439
7701 Pakistan 1970 1148.8489
7711 Pakistan 1980 1453.3522
7721 Pakistan 1990 1933.9449
7731 Pakistan 2000 1858.5410
```

Next, we use the following R code to convert the newly created dataset long form `pwt7.ip` into a wide form. We first install and load the reshape2 package and then apply the dcast() function to reshape `pwt7.ip`. We expect that the `rgdpl` values of India and Pakistan will be arranged side by side for each year. We refer to this process as reshaping a dataset from a long form to a wide form.

```
# load reshape2 package
library(reshape2)

# reshape pwt7.ip from long to wide form
pwt7.ip2 <- dcast(pwt7.ip, year ~ country, value.var = "rgdpl")
```

The dcast() function turns a long form country-year dataset `pwt7.ip` into a wide form year dataset `pwt7.ip2`. Inside the dcast() function, we first specify the long dataset to be reshaped, then the variable to be retained as the sorting ID, followed by a tilde and the variable whose values will be used as the names of new columns (i.e., India and Pakistan), and end with the values of new variables (India and Pakistan) filled with the values of the variable specified in the `value.var=` option. The new dataset `pwt7.ip2`, as displayed below, is organized the way we expected.

```
# display the wide form dataset
pwt7.ip2

  year     India  Pakistan
1 1950  594.1658  732.1559
2 1960  713.6806  732.4439
3 1970  887.1750 1148.8489
4 1980 1019.6258 1453.3522
5 1990 1407.2194 1933.9449
6 2000 1860.2439 1858.5410
```

At this point, one may naturally ask how we can reshape a dataset from a wide form into a long form. To do so, we can apply the melt() function in reshape2. The R code and output below show how we can reshape the wide form `pwt7.ip2` back into a long form.

```
# reshape a dataset from wide to long
melt(pwt7.ip2, id.vars = "year", variable.name = "country",
    value.name = "rgdpl")

   year  country     rgdpl
1  1950    India  594.1658
2  1960    India  713.6806
3  1970    India  887.1750
4  1980    India 1019.6258
5  1990    India 1407.2194
6  2000    India 1860.2439
7  1950 Pakistan  732.1559
8  1960 Pakistan  732.4439
9  1970 Pakistan 1148.8489
10 1980 Pakistan 1453.3522
11 1990 Pakistan 1933.9449
12 2000 Pakistan 1858.5410
```

Inside the melt() function, the option `id.vars="year"` specifies `year` is an ID variable; the option `variable.name=` specifies the name of a new variable that converts variable names from the wide form dataset into variable values in the long form dataset (wide-to-long); and the option `value.name=` specifies the name of a new variable that stores the values of the wide-to-long variables in the wide form dataset.

Prepare Data III: Manage Observations

Remove Select Observations

Sometimes we want to remove select observations that satisfy certain conditions. The easiest solution is to apply logical operators and the indexing method to a dataset. Take the dataset `pwt7` as an example. In the dataset, there are two versions of GDP data for China, which are based on different price data, one official and the other non-official. Thus, for the same years for China, one set of observations is denoted by isocode "CHN", and another set by isocode "CH2". In data analysis, it is certainly inappropriate to include both sets of observations at the same time. The code below demonstrates a simple method to remove all the observations where `isocode` equals "CH2" and save the output into a new dataset, `pwt7.nc`.

```
# remove observations with isocode equal CH2
pwt7.nc <- pwt7[pwt7$isocode != "CH2", ]
```

To show the difference between the two datasets, pwt7 and pwt7.nc, we will apply the dim() function to both datasets and see that they have the same number of variables but that pwt7.nc has 60 fewer observations. We will further show that pwt7 has 60 years from 1950 to 2009 that meet the condition that isocode equals "CH2" and pwt7.nc has zero years that meet that condition.

```
# remove second set of China observations with isocode CH2
pwt7.nc <- pwt7[pwt7$isocode != "CH2", ]

# compare pwt7 and pwt7.nc
dim(pwt7)

[1] 11400    37

dim(pwt7.nc)

[1] 11340    37

# display the number of years under CH2
pwt7$year[pwt7$isocode == "CH2"]

 [1] 1950 1951 1952 1953 1954 1955 1956 1957 1958 1959 1960
[12] 1961 1962 1963 1964 1965 1966 1967 1968 1969 1970 1971
[23] 1972 1973 1974 1975 1976 1977 1978 1979 1980 1981 1982
[34] 1983 1984 1985 1986 1987 1988 1989 1990 1991 1992 1993
[45] 1994 1995 1996 1997 1998 1999 2000 2001 2002 2003 2004
[56] 2005 2006 2007 2008 2009

pwt7.nc$year[pwt7.nc$isocode == "CH2"]

integer(0)
```

Find and Remove Duplicate Observations

Datasets sometimes contain duplicate observations that must be removed. Below we will demonstrate how to remove duplicate observations based on some key sorting variables. For the sake of illustration, we will first create an artificial dataset with duplicate observations. Recall the dataset pwt7.ip listed in the previous section. Below we will employ the rbind() function to append all India observations from pwt7.ip to the dataset pwt7.ip itself, thus producing a dataset with one duplicate India observation for each year. The display of pwt7.dup shows the creation of duplicate observations is as expected.

```
# create dataset with duplicate India observations
pwt7.dup <- rbind(pwt7.ip, pwt7.ip[pwt7.ip$country=="India",])
```

```
# display data with duplicate observations
pwt7.dup
```

```
       country year    rgdpl
4621     India 1950  594.1658
4631     India 1960  713.6806
4641     India 1970  887.1750
4651     India 1980 1019.6258
4661     India 1990 1407.2194
4671     India 2000 1860.2439
7681  Pakistan 1950  732.1559
7691  Pakistan 1960  732.4439
7701  Pakistan 1970 1148.8489
7711  Pakistan 1980 1453.3522
7721  Pakistan 1990 1933.9449
7731  Pakistan 2000 1858.5410
46211    India 1950  594.1658
46311    India 1960  713.6806
46411    India 1970  887.1750
46511    India 1980 1019.6258
46611    India 1990 1407.2194
46711    India 2000 1860.2439
```

Now we turn to apply the !duplicated() function to remove duplicate observations in `pwt7.dup`. The R code is as follows:

```
# remove duplicate observations
pwt7.dup[!duplicated(pwt7.dup[, c("country", "year")]), ]
```

The way the duplicated() function works is to assign a logical value TRUE or FALSE to each observation based on duplicate values for the key sorting variables: `country` and `year`. Since we have two key ID variables, we use the c() function to refer to them. We remove duplicate observations by asking R to select only observations that do *not* have duplicate values for `country` and `year`. In contrast to the duplicated() function, the !duplicated() function assigns TRUE to non-duplicated observations according to `country` and `year`. The R output is as follows:

```
      country year    rgdpl
4621    India 1950  594.1658
4631    India 1960  713.6806
4641    India 1970  887.1750
```

```
4651    India 1980 1019.6258
4661    India 1990 1407.2194
4671    India 2000 1860.2439
7681 Pakistan 1950  732.1559
7691 Pakistan 1960  732.4439
7701 Pakistan 1970 1148.8489
7711 Pakistan 1980 1453.3522
7721 Pakistan 1990 1933.9449
7731 Pakistan 2000 1858.5410
```

If we are interested in saving the dataset without duplicate observations, we simply assign the output to a new data object.

Prepare Data IV: Manage Variables

Managing variables or columns of a data frame often involves the creation of new variables, renaming variables in terms of variable names, recoding variables in terms of variable values, and the creation of variable labels. This section relies heavily on the earlier discussion on variable types.

Create New Variables

In order to conduct data analysis to answer a research question, we often have to create new variables. Here we provide some examples on how to create numeric, character, and factor variables, how to construct leading, lagging, and growth rate variables, and how to compute a new variable representing group mean.

Numeric Variables: Real Investment Per Capita and Total Real Investment

We begin with some simple examples of numeric variables. Suppose we want to use `pwt7` to create two new variables on investment: real investment per capita and total real investment in a country. For this task, the relevant variables include `ki`, `rgdpl`, and `POP`, which are defined in the readme file as follows:

- The variable `ki` is “Investment Share of PPP Converted GDP Per Capita at 2005 constant prices [rgdpl], in percent”;
- The variable `rgdpl` is "PPP Converted GDP Per Capita (Laspeyres), derived from growth rates of c, g, i, at 2005 constant prices (2005 International dollar per person);
- The variable `POP` is "Population (in thousands)."

Therefore, real investment per capita (in 2005 international dollars) ought to be computed as $rgdpl * ki/100$, and total real investment (in 2005 international dollar) ought to be computed as $rgdpl * POP * 1000 * ki/100 = rgdpl * POP * ki * 10$.

The R code for creating these two variables is as follows:

```
# create real per capita investment in 2005 international $
pwt7$investpc <- pwt7$rgdpl*pwt7$ki/100

# create total real investment in 2005 international $
pwt7$invest <- pwt7$rgdpl*pwt7$POP*pwt7$ki*10
```

Character and Factor Variables: Income Group and Decade

We will demonstrate how to create character and factor variables. As noted earlier, the advantage of the factor variable is that it can be used directly in statistical modeling. Below we will demonstrate how to construct an income group variable, first as a character type and then converted into a factor one.

Suppose we want to create an income group variable that classifies countries into one of four categories: low-income, lower-middle-income, upper-middle-income, and high-income. The World Bank employs the following criteria based on 2012 Gross National Income: low-income is $1,035 or less; lower-middle-income is $1,036–$4,085; upper-middle-income is $4,086–$12,615; and high-income is $12,616 or more. For the sake of illustration, we will employ a similar set of cutoffs.

We will first show how to create a character variable using 1,000, 4,000, and 12,000 as the thresholds. We use the indexing method and various operators to specify conditions for each category of income group. For example, the new variable `income.group` is assigned a value of "low-income" if `rgdpl` is less than 1,000 (i.e., `[pwt7$rgdpl<1000]`). It is important to remember that the character value such as "low-income" needs to be placed inside double quotation marks.

```
# create a character variable income.group
pwt7$income.group <- NA
pwt7$income.group[pwt7$rgdpl<1000] <- "low-income"
pwt7$income.group[pwt7$rgdpl>1000&pwt7$rgdpl<4000]<-"low-middle"
pwt7$income.group[pwt7$rgdpl>4000&pwt7$rgdpl<12000]<-"up-middle"
pwt7$income.group[pwt7$rgdpl>12000] <- "high-income"
```

We can illustrate the variable type of `income.group` using the class() function. Recall that the class() function identifies the class of an R object, which could be "numeric," "logical," "character," "list," "matrix," "array," "factor," or "data.frame."

```
# show variable type using class() function
class(pwt7$income.group)

[1] "character"
```

We can convert this character variable into a factor type using the factor() function.

```
# convert character variable into factor type
pwt7$income.group <- factor(pwt7$income.group,
    levels = c("low-income", "low-middle", "up-middle",
        "high-income"), ordered = TRUE)
```

The `levels=` and `ordered=TRUE` options inside the factor() function enable the following assignment of integer values: 1=low-income, 2=lower-middle, 3=upper-middle, and 4=high-income, for the variable `income.group`.

Any character variable can be converted into a factor one using the factor() function. As noted earlier, a character variable can not be used directly in a lot of statistical modeling such as regression analysis, but a factor variable can be.

Again, we can confirm the variable type using the class() function. We can also use the table() function to show the frequency count of observations in each income category.

```
# show variable type again
class(pwt7$income.group)

[1] "ordered" "factor"

# show frequency count in each category
table(pwt7$income.group)


 low-income  low-middle   up-middle high-income
       1613        2750        2421        1941
```

Next, we will show how to create a numeric variable to indicate income group categories, check its variable type, and then convert it into a factor variable.

```
# create a numeric variable income.group2
pwt7$income.group2 <- NA
pwt7$income.group2[pwt7$rgdpl<1000] <- 1
pwt7$income.group2[pwt7$rgdpl>1000 & pwt7$rgdpl<4000] <- 2
pwt7$income.group2[pwt7$rgdpl>4000 & pwt7$rgdpl<12000] <- 3
pwt7$income.group2[pwt7$rgdpl>12000] <- 4
```

```
# show variable type using class() function
class(pwt7$income.group2)

[1] "numeric"
```

Note that when converting a numeric variable into a factor one, we use the `labels=` option in the factor() function, which will assign labels to the different levels of the numeric variable.

```
# convert numeric variable into factor variable
pwt7$income.group2 <- factor(pwt7$income.group2, labels =
    c("low-income", "low-middle", "up-middle", "high-income"))
```

Finally, we will illustrate how to create a decade factor variable to represent whether an observation is from the 1950s, 1960s, 1970s, 1980s, 1990s, or 2000s.

```
# create a character variable for decade
pwt7$decade <- NA
pwt7$decade[pwt7$year >= 1950 & pwt7$year <= 1959] <- "1950s"
pwt7$decade[pwt7$year >= 1960 & pwt7$year <= 1969] <- "1960s"
pwt7$decade[pwt7$year >= 1970 & pwt7$year <= 1979] <- "1970s"
pwt7$decade[pwt7$year >= 1980 & pwt7$year <= 1989] <- "1980s"
pwt7$decade[pwt7$year >= 1990 & pwt7$year <= 1999] <- "1990s"
pwt7$decade[pwt7$year >= 2000] <- "2000s"

# convert character variable into factor variable
pwt7$decade <- factor(pwt7$decade, levels = c("1950s", "1960s",
    "1970s", "1980s", "1990s", "2000s"), ordered = TRUE)

# show frequency count in each decade
table(pwt7$decade)


1950s 1960s 1970s 1980s 1990s 2000s
 1900  1900  1900  1900  1900  1900
```

Leading, Lagged, and Growth Rate Variables

We often need to create variables that lead or lag behind the current year or variables that represent annual growth rates. Take the annual economic growth rate of a country as an example. We often measure economic growth by dividing the annual change in GDP per capita with the level of GDP per capita in the

previous year. We may denote economic growth in year `t` for country `i` as follows:

$$Growth_{t,i} = \frac{rgdpl_{t,i} - rgdpl_{t\text{-}1,i}}{rgdpl_{t\text{-}1,i}}$$

For example, country `i`'s annual growth rate in 1990 is:

$$Growth_{1990,i} = \frac{rgdpl_{1990,i} - rgdpl_{1989,i}}{rgdpl_{1989,i}}.$$

In order to compute growth, we need to have the current year `rgdpl` and the one-year lagged `rgdpl`. The former is directly available as a variable, but the latter is not. To generate the one-year lagged `rgdpl` for each country, we apply the slide() function from the DataCombine package. The specific steps include: install the DataCombine package, load it into R, sort data in ascending order first by country and then by year, use the slide() function to create a new leading or lagged variable, assign the slide() function output to `pwt7`, and then compute the annual growth rate for `rgdpl`. The R code is as follows:

```
# create leading and lagging variables in a panel data
# load DataCombine package in order to use slide function
library(DataCombine)

# sort data first by country and then by year
pwt7 <- pwt7[order(pwt7$country, pwt7$year),]

# create a one-year leading variable for rgdpl
pwt7 <- slide(pwt7,Var="rgdpl", NewVar="rgdplead",
              GroupVar="country", slideBy=1)

# create a one-year lagged variable for rgdpl
pwt7 <- slide(pwt7, Var="rgdpl", NewVar="rgdplag",
              GroupVar="country", slideBy=-1)

# create annual growth rate for rgdpl
pwt7$growth <- (pwt7$rgdpl-pwt7$rgdplag)/pwt7$rgdplag
```

The arguments in the slide() function can be difficult to understand and thus, require clarification.

1. Specify the dataset used.
2. The option `Var=` identifies the variable whose leading or lagged version is created.

3. The option NewVar= specifies the name of the newly created leading or lagged variable.
4. The option GroupVar= identifies the cross-sectional group for which the leading or lagged variable is created.
5. The option slideBy= specifies by how many rows (time units like year) the new variable is to be lagged (-) or leading (+). For example, 1 indicates one-year leading whereas -1 means one year lagged.

Create a Group-Specific Variable: World Average Annual Growth

We often need to create by-group variables. World average economic growth rate is a good example of a by-group variable. It is an average of the growth rates of all countries by each year, which allows us to compare a country's own growth rate with that of the world economy. Below we will demonstrate how to create a by-group variable in R using the by() function.

```
#  create world average annual growth using by() function
pwt7$growth.w <- by(pwt7$growth, pwt7$year, FUN=mean,
                    na.rm=TRUE)
```

The new variable indicating world average annual growth rate is called growth.w, assigned through the output of the by() function. The by() function first specifies the original variable growth that is used for computing group mean, then the group or index variable year for which the group mean is computed, followed by the FUN= option which specifies the statistic computed for each group. Since missing values are present in the growth variable, na.rm=TRUE must be specified to remove missing values in computation.

The by() function applies a function across subsets of data defined by the group variable. Hence, a variety of by-group statistics can be computed. For a list of such statistics, refer to Table 1.6.

Create and Merge More Group-Specific Statistics

We may also be interested in finding out the average economic growth rate for each income group during each decade. For this purpose we can use the aggregate() function as follows:

```
# compute by-group economic growth rates
aggregate(growth ~ decade + income.group, data=pwt7, FUN=mean)
```

Inside the aggregate() function, we first specify the variable whose values are to be aggregated, and then following a tilde, we must specify the group variables by which group statistics are to be computed. After that, we specify which dataset

is to be used with `data=`, and which statistic is to be computed with `FUN=`. We may choose to compute different group-specific statistics from the list in Table 1.6. The R output is as follows:

```
   decade income.group       growth
1   1950s   low-income  0.016622330
2   1960s   low-income  0.019106637
3   1970s   low-income  0.007661474
4   1980s   low-income  0.002613433
5   1990s   low-income -0.004449034
6   2000s   low-income  0.015688352
7   1950s   low-middle  0.023500379
8   1960s   low-middle  0.026975806
9   1970s   low-middle  0.031768607
10  1980s   low-middle  0.008659334
11  1990s   low-middle  0.020114909
12  2000s   low-middle  0.035340001
13  1950s    up-middle  0.033518688
14  1960s    up-middle  0.048562439
15  1970s    up-middle  0.039638180
16  1980s    up-middle  0.010777898
17  1990s    up-middle  0.015921743
18  2000s    up-middle  0.038937086
19  1950s  high-income  0.019925973
20  1960s  high-income  0.036524799
21  1970s  high-income  0.035745528
22  1980s  high-income  0.021335852
23  1990s  high-income  0.019735372
24  2000s  high-income  0.026003399
```

The by() and aggregate() functions both produce summarizing information for groups. Yet they are different in several ways. First, the former works with any function such as mean or range, but the latter only works with a function that returns a single value such as mean. Second, the former may produce multiple by-group statistics at the same time, but the latter only one at a time. Finally, the output of the former is a list that requires `as.data.frame((as.table(output)))` to turn into a data frame, and the output of the latter is by default a data frame.

We may want to save the aggregate() output into a dataset by simply assigning it to a new data frame object called `pwt7.ag`. The R code is as follows:

```
# save by-group economic growth rates to a dataset
pwt7.ag <- aggregate(growth ~ decade + income.group,
                     data=pwt7, FUN=mean)
```

Often, though, we are interested in comparing a country's GDP per capita in a year with its income group's decade average. But the former lies in the dataset `pwt7` whereas the latter is in `pwt7.ag`. We may merge the latter into the former by using the code discussed earlier. One complication in the process is that the variable `growth` in `pwt7.ag` is a group-specific statistic, whereas the variable `growth` in `pwt7` is the value of GDP per capita for a country in a year. Hence, they measure different things even though they have the same variable name, as shown below.

```
# show variable names in pwt7.ag
names(pwt7.ag)

[1] "decade"       "income.group" "growth"
```

Therefore, we must first rename the `growth` variable in `pwt7.ag`, by using the code below, and then we merge the two datasets `pwt7` and `pwt7.ag` according to two sorting ID variables `income.group` and `decade`. The R code is as follows:

```
# rename the growth variable in pwt7.ag
names(pwt7.ag)[names(pwt7.ag) == "growth"] <- "growth.di"

# merge pwt7.ag into pwt7
pwt7 <- merge(pwt7, pwt7.ag, by = c("decade", "income.group"),
    all = TRUE, sort = TRUE)
```

Often times, we need to create multiple group-specific statistics for multiple variables. The aggregate() function is limited for that purpose. The summaryBy() function in the doBy package can help produce multiple statistics for multiple variables over multiple groups at the same time. For example, if we are interested in producing both the mean and standard deviation for three variables—`growth`, `openk`, and `POP`—for each country in each decade, we may use the following R code. Inside summaryBy(), we first specify multiple variables whose group statistics need to be computed, and then after the tilde, specify multiple group variables; the statistics are specified via `FUN=`. For illustration, we save the output into a new dataset `pwt7.cs` and then display its last observation. Note how variable names in `pwt7.cs` combine original variable names and statistics computed.

```
# load package
library(doBy)

# generate multiple group-statistics for multiple
# variables
```

```
pwt7.cs <- summaryBy(growth + openk + POP ~ isocode + decade,
    FUN = c(mean, sd), data = pwt7, na.rm = TRUE)

# display last observation
tail(pwt7.cs, n = 1)
```

```
     isocode decade growth.mean openk.mean POP.mean
1140     ZWE  2000s -0.05472721   81.33989 11646.44
      growth.sd openk.sd   POP.sd
1140 0.05515508 4.734968 203.5446
```

Rename Variables

It often happens that the name of a variable from an original dataset is not what we would prefer to call it. Suppose we would like to refer to the `POP` variable in `pwt7` as `population` instead. We may use the names() function, the square brackets, and the assignment symbol to do so.

```
# rename variable by changing column name of dataset with
# names() function and indexing brackets
names(pwt7)[names(pwt7) == "POP"] <- "population"
```

Running this line of code and `names(pwt7)` shows that the variable name `POP` is now changed to `population`.

```
# rename variable by changing column name of dataset with
# names() function and indexing brackets
names(pwt7)[names(pwt7) == "POP"] <- "population"

# confirm rename from output of names() function
names(pwt7)
```

```
 [1] "country"       "isocode"       "year"
 [4] "population"    "XRAT"          "Currency_Unit"
 [7] "ppp"           "tcgdp"         "cgdp"
[10] "cgdp2"         "cda2"          "cc"
[13] "cg"            "ci"            "p"
[16] "p2"            "pc"            "pg"
[19] "pi"            "openc"         "cgnp"
[22] "y"             "y2"            "rgdpl"
[25] "rgdpl2"        "rgdpch"        "kc"
[28] "kg"            "ki"            "openk"
```

```
[31] "rgdpeqa"       "rgdpwok"       "rgdpl2wok"
[34] "rgdpl2pe"      "rgdpl2te"      "rgdpl2th"
[37] "rgdptt"        "income.group"  "income.group2"
[40] "decade"        "rgdplead"      "rgdplag"
[43] "growth"        "growth.w"
```

Recode Variable Values

Often we have to change how variable values are coded. Our first example will be how to recode certain values of two character variables in `pwt7`.

```
# recode variable value
pwt7$isocode[pwt7$isocode == "CH2"] <- "CHN"
pwt7$country[pwt7$country == "China Version 1"] <- "China"
```

The logic of this recoding approach is to assign the new values to select observations of a certain variable. The first line of code above selects the variable pwt7$isocode, and then, through the use of the indexing method, those observations where `isocode` equals `CH2` are selected and assigned the value `CHN`. The double quotation marks must be used around the values in the case of a character variable. If we are recoding numeric variables, quotation marks are not necessary.

Now, recoding is particularly important for datasets in which missing values are assigned special numerical values such as `-99` or `-999`. If we do not recode those special missing values into R's default missing value `NA`, they will be treated incorrectly as normal numerical values. Thus, we must recode them into R's default missing value `NA`. Hypothetically, if `rgdpl` used `-999` as its missing value code, then we could use the following line of code to recode it.

```
# recode hypothetical missing value -999 in rgdpl
pwt7$rgdpl[pwt7$rgdpl == -999] <- NA
```

Create Variable Labels

We often use variable labels to keep track of variable definitions. R does not do as good a job with variable labels as other software packages like STATA or SAS. A remedy is to use the label() function from the Hmisc package that allows us to create meaningful labels or definitions for variables. Below we will show how to assign labels to certain variables in `pwt7`. We will first install the package Hmisc if we have not done so, or skip the step if we have, then load the package, and apply the label function to each variable to which we want to assign a meaningful label. Next, we assign a long label surrounded by double quotation marks to the output

of the label function for each variable. Finally, we apply the label function to the whole dataset pwt7 to display the variable labels assigned. The R code, without output, is as follows:

```
# load Hmisc package
library(Hmisc)

# use label function to assign variable labels
label (pwt7$isocode) <- "Penn World Table country code"
label (pwt7$rgdpl) <- "PPP Converted GDP Per Capita (Laspeyres)
derived from growth rates of c, g, i, at 2005 constant prices"
label (pwt7$openk) <- "Openness at 2005 constant prices in
                       percent"
label (pwt7$pop) <- "Population (in thousands)"
label (pwt7$growth) <- "annual economic growth rate, based on
                        RGDPL"

# display labels of all variables
label(pwt7)
```

An important weakness of the label() function is that the assigned labels are only useful for functions from the Hmisc package. Still, it offers a good way to keep track of variable labels inside an R program.

Chapter 2 Program Code

Set Working Directory; Import and Inspect Data

```
# install packages only once, then comment out code
# install.packages(c("reshape2", "DataCombine", "Hmisc",
# "haven", "foreign", "gdata", "XLConnect", "pwt",
# "reshape", "doBy"), dependencies=TRUE)

# remove all objects from workspace
rm(list=ls(all=TRUE))

# change working directory to point to project folder
setwd("C:/Project")

# import comma-delimited file, create data object pwt7
pwt7 <- read.csv("pwt70_w_country_names.csv", header=TRUE,
                 strip.white=TRUE, stringsAsFactors = FALSE,
```

```
                 na.strings=c("NA",""))

# Inspect Imported Data

# inspect dataset pwt7 in a spreadsheet style data viewer
View(pwt7)

# list first one observation in dataset pwt7
head(pwt7, n=1)

# list last one observation in dataset pwt7
tail(pwt7, n=1)

# dimensions of pwt7: number of observations and number of
# variables
dim(pwt7)

# variable names in dataset pwt7
names(pwt7)

# structure of dataset pwt7
str(pwt7)

# create a figure with three plots
# set graphic parameters for figure of two rows, two columns
par(mfrow=c(2,2))

# graph distribution of variable rgdpl in data frame pwt7
hist(pwt7$rgdpl)
boxplot(pwt7$rgdpl)
qqnorm(pwt7$rgdpl)
```

Manage Datasets, Observations, and Variables

```
# reference fifth observation of pwt7$POP
pwt7$POP[5]

# reference specific observation of pwt7$POP
pwt7$POP[pwt7$country=="Afghanistan" & pwt7$year==1954]

# reference fifth observation of fourth variable in pwt7
```

```
pwt7[5, 4]

# reference certain observation of certain variable in pwt7
pwt7[pwt7$country=="Afghanistan" & pwt7$year==1954, "POP"]

# sort data in ascending order by country and by year
pwt7 <- pwt7[order(pwt7$country, pwt7$year),]

# sort data by country (ascending) and by year (descending)
pwt7 <- pwt7[order(pwt7$country, -pwt7$year),]

# show select rows; example: observations 100 and 102
pwt7[c(100, 102),]

# show select rows; example: observations from 100 to 102
pwt7[c(100:102),]

# show select variables
pwt7[, c("country", "year", "rgdpl")]

# combine two selections: certain rows and certain columns
pwt7[c(100:102), c("country", "year", "rgdpl")]

# show certain observations and certain variables that meet
# conditions: Afghanistan since 2006 for three variables
pwt7[pwt7$year>=2006 & pwt7$country=="Afghanistan",
     c("country", "year", "rgdpl")]

# select multiple countries and non-consecutive years
# for select variables
pwt7[pwt7$year %in% c(1970, 1980, 1990, 2000, 2009) &
pwt7$country %in% c("India","China Version 1"),
c("country","year","rgdpl")]

# create a new dataset with three select variables
pwt7new <- pwt7[, c("country", "year", "rgdpl")]

# create two temporary pwt7 subsets for merging example
pwt7.tmp1 <- pwt7[,c("isocode", "year", "rgdpl")]
pwt7.tmp2 <- pwt7[,c("isocode", "year", "openk")]
```

```
# merge two datasets
pwt7.m<-merge(pwt7.tmp1, pwt7.tmp2, by=c("isocode","year"),
              all=TRUE, sort=TRUE)

# combine datasets of equal observations, different variables
new.data <- cbind(data1, data2)

# append datasets of equal variables, different observations
new.data <- rbind(data1, data2)

# create a subset of pwt7
# India and Pakistan, three variables and six years
pwt7.ip <- pwt7[pwt7$year %in% c(1950,1960,1970,1980,1990,2000)
        & pwt7$country %in% c("India","Pakistan"),
        c("country","year","rgdpl")]

# display the long form dataset
pwt7.ip

# load reshape2 package
library(reshape2)

# reshape pwt7.ip from long to wide form
pwt7.ip2 <- dcast(pwt7.ip, year~country, value.var="rgdpl")

# display the wide form dataset
pwt7.ip2

# reshape a dataset from wide to long
melt(pwt7.ip2, id.vars="year", variable.name="country",
     value.name="rgdpl")

# remove second set of China observations with isocode CH2
pwt7.nc <- pwt7[pwt7$isocode!="CH2",]

# compare pwt7 and pwt7.nc
dim(pwt7)
dim(pwt7.nc)

# display the number of years under CH2
pwt7$year[pwt7$isocode=="CH2"]
```

```
pwt7.nc$year[pwt7.nc$isocode=="CH2"]

# create dataset with duplicate India observations
pwt7.dup <- rbind(pwt7.ip, pwt7.ip[pwt7.ip$country=="India",])

# display data with duplicate observations
pwt7.dup

# remove duplicate observations
pwt7.dup[!duplicated(pwt7.dup[,c("country", "year")]),]

# create real per capita investment in 2005 international $
pwt7$investpc <- pwt7$rgdpl*pwt7$ki/100

# create total real investment in 2005 international $
pwt7$invest <- pwt7$rgdpl*pwt7$POP*pwt7$ki*10

# create a character variable income.group
pwt7$income.group <- NA
pwt7$income.group[pwt7$rgdpl<1000] <- "low-income"
pwt7$income.group[pwt7$rgdpl>1000&pwt7$rgdpl<4000]<-"low-middle"
pwt7$income.group[pwt7$rgdpl>4000&pwt7$rgdpl<12000]<-"up-middle"
pwt7$income.group[pwt7$rgdpl>12000] <- "high-income"

# show variable type using class() function
class(pwt7$income.group)

# convert character variable income.group into factor type
pwt7$income.group <- factor(pwt7$income.group,
            levels=c("low-income", "low-middle", "up-middle",
            "high-income"), ordered=TRUE)

# show variable type again
class(pwt7$income.group)

# show frequency count in each category
table(pwt7$income.group)

# create a numeric variable income.group2
pwt7$income.group2 <- NA
pwt7$income.group2[pwt7$rgdpl <1000] <- 1
```

```
pwt7$income.group2[pwt7$rgdpl>1000 & pwt7$rgdpl<4000] <- 2
pwt7$income.group2[pwt7$rgdpl>4000 & pwt7$rgdpl<12000] <- 3
pwt7$income.group2[pwt7$rgdpl > 12000] <- 4

# show variable type using class() function
class(pwt7$income.group2)

# convert numeric variable into factor variable
pwt7$income.group2 <- factor(pwt7$income.group2, labels=
    c("low-income", "low-middle", "up-middle", "high-income"))

# show variable type again
class(pwt7$income.group2)

# show frequency count within each category
table(pwt7$income.group2)

# create a character variable for decade
pwt7$decade <- NA
pwt7$decade[pwt7$year>=1950 & pwt7$year<=1959] <- "1950s"
pwt7$decade[pwt7$year>=1960 & pwt7$year<=1969] <- "1960s"
pwt7$decade[pwt7$year>=1970 & pwt7$year<=1979] <- "1970s"
pwt7$decade[pwt7$year>=1980 & pwt7$year<=1989] <- "1980s"
pwt7$decade[pwt7$year>=1990 & pwt7$year<=1999] <- "1990s"
pwt7$decade[pwt7$year>=2000] <- "2000s"

# convert character variable into factor variable
pwt7$decade <- factor(pwt7$decade, levels=c("1950s", "1960s",
           "1970s", "1980s", "1990s", "2000s"), ordered=TRUE)

# show frequency count in each decade
table(pwt7$decade)

# create leading and lagging variables in a panel dataset
# load DataCombine package in order to use slide function
library(DataCombine)

# sort data first by country and then by year
pwt7 <- pwt7[order(pwt7$country, pwt7$year),]

# create a one-year leading variable for rgdpl
```

```
pwt7 <- slide(pwt7,Var="rgdpl", NewVar="rgdplead",
              GroupVar="country", slideBy=1)

# create a one-year lagged variable for rgdpl
pwt7 <- slide(pwt7, Var="rgdpl", NewVar="rgdplag",
              GroupVar="country", slideBy=-1)


# create annual growth rate for rgdpl
pwt7$growth <- (pwt7$rgdpl-pwt7$rgdplag)/pwt7$rgdplag

#  create world average annual growth using by() function
pwt7$growth.w <- by(pwt7$growth, pwt7$year, FUN=mean,
                    na.rm=TRUE)

# compute by-group economic growth rates
aggregate(growth ~ decade + income.group, data=pwt7, FUN=mean)

# save by-group economic growth rates to a dataset
pwt7.ag <- aggregate(growth~decade+income.group, data=pwt7,
                     FUN=mean)

# rename the growth variable in pwt7.ag
names(pwt7.ag)[names(pwt7.ag)=="growth"] <- "growth.di"

# merge pwt7.ag into pwt7
pwt7 <- merge(pwt7, pwt7.ag, by=c("decade","income.group"),
              all=TRUE, sort=TRUE)

# load package
library(doBy)

# generate multiple group-statistics for multiple variables
pwt7.cs <- summaryBy(growth + openk + POP ~ isocode + decade,
                     FUN=c(mean, sd), data=pwt7, na.rm=TRUE)

# display last observation
tail(pwt7.cs, n=1)

# rename variable by changing column name of dataset
# with names() function and indexing brackets
```

```
names(pwt7)[names(pwt7)=="POP"] <- "population"

# confirm rename from output of names() function
names(pwt7)

# recode variable value
pwt7$isocode[pwt7$isocode=="CH2"] <- "CHN"
pwt7$country[pwt7$country=="China Version 1"] <- "China"

# recode hypothetical missing value -999 in rgdpl
pwt7$rgdpl[pwt7$rgdpl==-999] <- NA

# load Hmisc package
library(Hmisc)

# use label function to assign variable labels
label (pwt7$isocode) <- "Penn World Table country code"
label (pwt7$rgdpl) <- "PPP Converted GDP Per Capita (Laspeyres)
derived from growth rates of c, g, i, at 2005 constant prices"
label (pwt7$openk) <- "Openness at 2005 constant prices in
                       percent"
label (pwt7$pop) <- "Population (in thousands)"
label (pwt7$growth) <- "annual economic growth rate, based on
                        RGDPL"

# display labels of all variables
label(pwt7)
```

Summary

This chapter shows how to read an original raw dataset in the comma-delimited format into R, how to create a corresponding data object, how to inspect the imported data visually, how to obtain information on dataset attributes (dimensions, variable names, etc.), how to graph select variables, and how to manage variables, observations, and datasets in order to get data ready for analysis. Chapter 2 covers a large amount of materials that are necessary for using R to get ready for practical data analysis, even at the beginner's level. The large quantity of information, however, can make it difficult for us to see the big picture. Therefore, at this moment, it is useful to revisit Table 2.1 which provides a conceptual roadmap of the information in Chapter 2.

Now that we have learned how to prepare a dataset for analysis, it is time for us to learn how to use R to obtain descriptive statistics and make basic statistical inferences about an outcome variable of interest. These are the topics of Chapter 3. Yet before heading into Chapter 3, it is useful to address some miscellaneous questions that beginning R users often come up with.

Miscellaneous Q&As for Ambitious Readers

How do we Import Datasets of Other Formats into R?

The Penn World Table 7.0 dataset example illustrates how R imports a comma-delimited file. In practice, we will always come across datasets in many other formats. R is powerful in terms of data importation. Below we will present some sample code for importing datasets of several common formats. All the code below will assume we have executed setwd() function and specified where the datasets are stored.

(1) Tab-delimited file. The R code for a tab-delimited file is very similar to those for the comma-delimited file, except for how to denote that a file is tab-delimited.

```
dataframe.name <- read.table("filename.txt", header=TRUE,
                  sep="\t", na.strings=".", strip.white=TRUE,
                  stringsAsFactors=False)
```

The read.table() function imports a tab-delimited file called filename.txt, in which the header row contains variable names, the columns are separated by tabs, and the observation values denoted by "." are treated as missing. The output from the read.table function is assigned to a data object arbitrarily called dataframe.name.

We use the option `sep="\t"` to indicate the file is tab-delimited. The read.table function can also import the comma delimited file by specifying the option `sep=","` instead.

(2) Stata file. To import stata files into R, we may use one of the following several ways.

```
# install.packages("foreign")
# install.packages("Hmisc")
library(foreign)
library(Hmisc)
dataframe.name <- stata.get("filename.dta")
```

In the example above, we first use the library function to load into R two add-on packages, foreign and Hmisc. Note that the code assumes that the

packages have been installed; if they have not been, we will get an error message when trying to load the packages. Then we use the stata.get function to read the stata file and create a data object called dataframe.name.

In a second way to import a stata file, we first use the library function to load into R the add-on package, foreign. Then we use the read.dta function to read the stata file, and create a data object dataframe.name.

```
library(foreign)
dataframe.name <- read.dta("filename.dta")
```

There are three important caveats to remember when we import stata files into R. First, many variable names in stata files contain underscores like in `variable_name`, which R does not like. We could use the option `convert.underscore = TRUE` to replace an underscore with a period. Second, many variables in stata files are often read into R as factor variables, which could be hard to work with for beginners. We could use the option `convert.factors = FALSE` to tell R to import numerical variables as numerical ones only. With these options added, the R code may appear as follows:

```
dataframe.name <- stata.get("filename.dta",
               convert.underscore=TRUE, convert.factors=FALSE)
dataframe.name <- read.dta("filename.dta",
               convert.underscore=TRUE, convert.factors=FALSE)
```

Finally, both stata.get and read.dta functions can not import files in the STATA13 format. A new package for importing Stata 13 data files is now available. The following code will allow us to do that.

```
# install.packages('readstata13')
library(readstata13)
dataframe.name <- read.dta13("filename.dta")
```

We first install and load the readstata13 package, then use the read.dta13 function to import the stata file, and assign it to a data object in R.

Stata files often contain very informative variable labels. Here is one way to preserve those variable labels in a separate file, which we refer to as "codebook" in the R code below.

```
dataframe.name <- read.dta("filename.dta")
var.labels <- attr(dataframe.name,"var.labels")
codebook <- data.frame(var.name=names(dataframe.name),
                       var.labels)
```

(3) SPSS file. Similar to stata files, we can use two alternative functions for importing SPSS files: read.spss in the foreign package, and spss.get in the Hmisc package. The latter is preferable because it automatically takes care of many options, making data importation smoother. The R code for the latter is as follows:

```
library(foreign)
library(Hmisc)
dataframe.name <- spss.get("filename.sav")
```

In many SPSS files, variables often come with value labels. If we want to keep the variables with those same levels defined by value labels, we can add an option to the argument of the function and use the following code instead.

```
dataframe.name <- spss.get("filename.sav",
                           use.value.labels=TRUE)
```

(4) SAS file. SAS files can be read into R using the foreign and haven packages among others. We can use the read.xport() function in the foreign package and the read_sas() function in the haven package to read two different SAS data formats, respectively.

```
library(foreign)
dataframe.name <- read.xport("filename.xport")
```

```
library(haven)
dataframe.name <- read_sas("filename.sas7bdat")
```

(5) Excel file. Most frequently students work with Excel spreadsheet files. The best data importing method, recommended in many books on R, is to export those files from within Excel to comma-delimited or tab-delimited files and then import them into R using the code described earlier. We encourage R beginners to follow this advice.

However, R does have the ability to directly communicate with Excel spreadsheets for data importation. The Excel file needs some preparation before we read it into R. For example, keep the variable names in the first row of the spreadsheet, and give the sheet to be read into R a worksheet name so that we can refer to it in R. Here is the R code:

```
# install.packages('RODBC')
library(RODBC)
channel <- odbcConnectExcel("filename.xls")
```

```
dataframe1 <- sqlFetch(channel, "worsheet1")
odbcClose(channel)
```

In the code, we first install and load the RODBC package into R; ignore the installation step if the package has been installed. The odbcConnectExcel() function reads an excel file called filename.xls, and returns an RODBC connection object called channel. Then the sqlFetch() function uses the channel object, imports the Excel worksheet labeled worksheet1, and assigns the output to an R data object called dataframe1. Finally, the odbcClose() function removes the RODBC connection object called channel.

Sometimes, we would like to import multiple worksheets from an Excel file. Suppose we have three worksheets, labeled worksheet1, worksheet2, and worksheet3. The code below shows how we can import the three worksheets into three R data objects called dataframe1, dataframe2, and dataframe3, respectively.

```
library(RODBC)
channel <- odbcConnectExcel("filename.xls")
dataframe1 <- sqlFetch(channel, "worsheet1")
dataframe2 <- sqlFetch(channel, "worsheet2")
dataframe3 <- sqlFetch(channel, "worsheet3")
odbcClose(channel)
```

A couple of other packages also can help import Excel files into R, including gdata and XLConnect. To use these packages, one may try the following code:

```
library(gdata)
dataframe <- read.xls("datafile.xls", sheet = 1)
```

```
library(XLConnet)
workbook <- loadWorkbook("datafile.xls")
dataframe <- readWorksheet(workbook, sheet = "Sheet1")
```

(6) R data format. If a data object is saved in R data format, it can be directly loaded into R using the load() function. The R code below shows how to save a data object in the workspace, and then load it into R.

```
save(data.object, file = "datafile.RData")
load("datafile.RData")
```

Can we Edit an Observation's Values in the Spreadsheet Viewer?

Recall that we used the `View(pwt7)` to open and eyeball the pwt7 data frame in a spreadsheet interface. In addition to `View(pwt7)`, one can also use `fix(pwt7)` to view data. Yet while both can be used to view data, they are extremely different functions. View() just allows one to view data, but fix() allows one to not only view but also to edit and change values of a dataset within the spreadsheet viewer.

Could we use the Indexing Method to Indicate Variables?

We can use the indexing numbers to refer to variables in a dataset. To do so, we first use the names() function to find the indexing numbers for relevant variables, and then we can refer to them in other functions. The following is an example of how to choose `country`, `year`, and `rgdpl` from the `pwt7` dataset. Note that based on the output for names() function, the number inside the brackets is the indexing number for the variable right next to it. In this case below, `country` is in the first column, `year` in the third column, and `rgdpl` in the 24th column. So the two lines of code produce the same output.

```
names(pwt7)

 [1] "country"       "isocode"       "year"
 [4] "population"    "XRAT"          "Currency_Unit"
 [7] "ppp"           "tcgdp"         "cgdp"
[10] "cgdp2"         "cda2"          "cc"
[13] "cg"            "ci"            "p"
[16] "p2"            "pc"            "pg"
[19] "pi"            "openc"         "cgnp"
[22] "y"             "y2"            "rgdpl"
[25] "rgdpl2"        "rgdpch"        "kc"
[28] "kg"            "ki"            "openk"
[31] "rgdpeqa"       "rgdpwok"       "rgdpl2wok"
[34] "rgdpl2pe"      "rgdpl2te"      "rgdpl2th"
[37] "rgdptt"        "income.group"  "income.group2"
[40] "decade"        "rgdplead"      "rgdplag"
[43] "growth"        "growth.w"
```

```
pwt7[, c(1, 3, 24)]
pwt7[, c("country", "year", "rgdpl")]
```

Does White Space Really Matter for Character Fields?

We use the following example to show whether R considers "USA" (without any white space) to be the same as " USA" (with one white space in front of USA).

```
# show that white space matters for character fields
# isTRUE tests if it is TRUE or FALSE that 'USA' is the
# same as ' USA'
isTRUE("USA" == " USA")
```

```
isTRUE("USA" == " USA")

[1] FALSE
```

The output shows that the statement that "USA" is the same as " USA" is FALSE.

Is there Another Way to get a Subset of a Dataset?

Earlier we showed how to use indexing to rows and columns as a way to select observations and variables in order to obtain a subset of `pwt7`. Another way to do this is to use the subset() function. Below is an example of using both indexing and subset to produce the same subset of `pwt7`.

```
# create a new dataset with five select variables
# excluding CH2 observations
pwt7.v1 <- pwt7[pwt7$isocode != "CH2", c("country", "isocode",
           "year","rgdpl","openk")]

# get the same result by using subset function
pwt7.v2 <- subset(pwt7, isocode != "CH2", c(country, isocode,
           year, rgdpl, openk))
```

Note that within the subset() function, we first specify the data frame name, then specify rows meeting conditions, then select the columns or variables using the c() function. Note that quotation marks are not needed inside the c() function within the subset() function.

We can also see a fuller expression of the same code below.

```
# a fuller expression of the same code
pwt7.v3 <- subset(pwt7, subset = (isocode != "CH2"), select =
    c("country", "isocode", "year", "rgdpl", "openk", "POP") )
```

Is there Another Way to Recode Variable Values?

An alternative way to recode variable values is to use the recode function from the car package. Since the package is an add-on, we have to install the car package first and then load it into R.

```
# install.packages('car')
library(car)
pwt7$isocode <- recode(pwt7$isocode, "'CH2'='CHN'")
```

Notice how we use single quotation marks around the strings, though single quotation marks are not necessary for numeric values.

How to Save Duplicate Observations Into a Different Dataset

Earlier we learned how to remove duplicate observations. Often it is necessary for us to know why duplicate observations exist and whether the observations that are duplicates according to the sorting variables also have duplicate values for other variables in the dataset. Hence, we often would like to send the duplicate observations to a separate dataset for examination. The following R code shows how to do that in two different ways.

```
# create a dataset of duplicated observations
pwt7.d <- pwt7[duplicated(pwt7[, c("isocode", "year")]), ]
```

An alternative way for inspecting duplicate observations is to assign a logical value TRUE or FALSE to each observation in the original dataset with TRUE indicating an observation has duplicated values for sorting variables, and assign the output to a new dataset. Then we can apply the View() function to directly view which observations are duplicates, and apply the table() function to get a frequency count of the number of duplicate observations in the dataset. The R code is listed below.

```
# assign a logical value TRUE or FALSE to each observation
# based on duplicate values for sorting variables, and
# assign the output to a data object
idx <- duplicated(pwt7[, c("isocode", "year")])

# directly view which observations are duplicates
View(idx)

# obtain a frequency count of the number of duplicate
# observations
table(idx)
```

Is there Another Way to Rename Variables?

The add-on package reshape provides a rename() function for us to easily rename variables.

```
# load the reshape package
library(reshape)

# apply the rename function to rename three variables and
# send output to pwt7
pwt7 <- rename(pwt7, c(POP = "population", openk = "trade",
    rgdpl = "rgdppc"))
```

Alternative Source of Penn World Table Data

We showed how to download the pwt7.0 dataset in a comma-delimited format earlier from the following link: www.rug.nl/ggdc/productivity/pwt/pwt-releases/pwt-7.0. Some scholars have put together an R package pwt, which contains six different earlier versions of Penn World Table data (pwt5.6, 6.1, 6.2, 7.0, and 7.1). So an easy way to access the pwt 7.0 dataset is to install the pwt package, upload the package in R, and then directly read the pwt7.0 dataset. The R code below shows how to upload the package and then read pwt7.0 in R.

```
library(pwt)
data(pwt7.0)
```

Exercises

For beginners who apply R to data analysis, the first biggest hurdle is reading data into R. The following set of exercises will focus on importing data sets to be used in this book.

1. Create a homework project folder and then an R program file that is pointing to and saved in that folder.
2. In Excel, save the Penn World Table 7 dataset as a tab-delimited file, then read the dataset into R, and produce a table of summary statistics.
3. Follow the instructions in Chapter 6, read the dataset used in Braithwaite (2006) into R, and produce a table of summary statistics.
4. Follow the instructions in Chapter 6, read the dataset used in Bénabou et al. (2015) into R, and produce a table of summary statistics.
5. Following the instructions in Chapter 8, read the dataset of the World Value Survey (WVS) Wave 6 into R, and produce a table of summary statistics.