Using R for Data Analysis in Social Sciences

a research project-oriented approach

QUAN LI

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`.

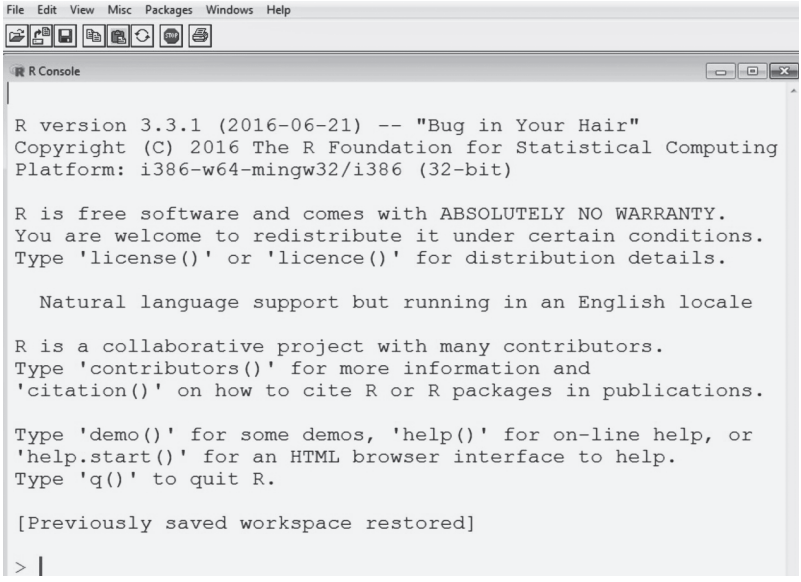
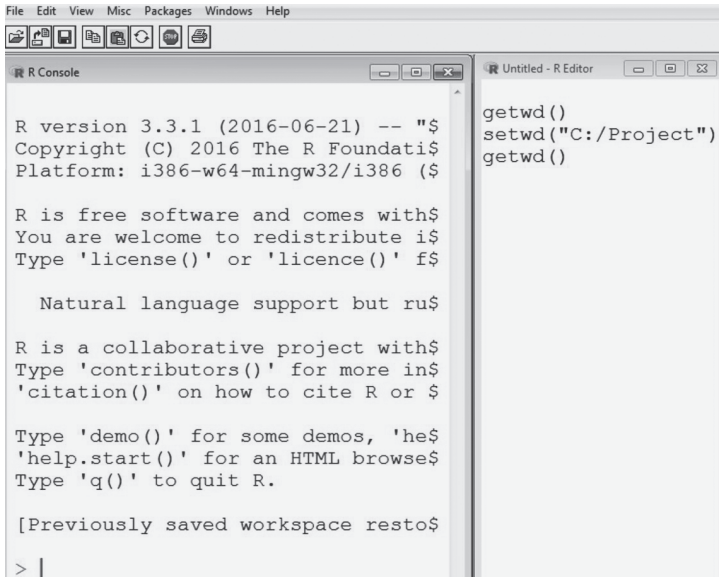
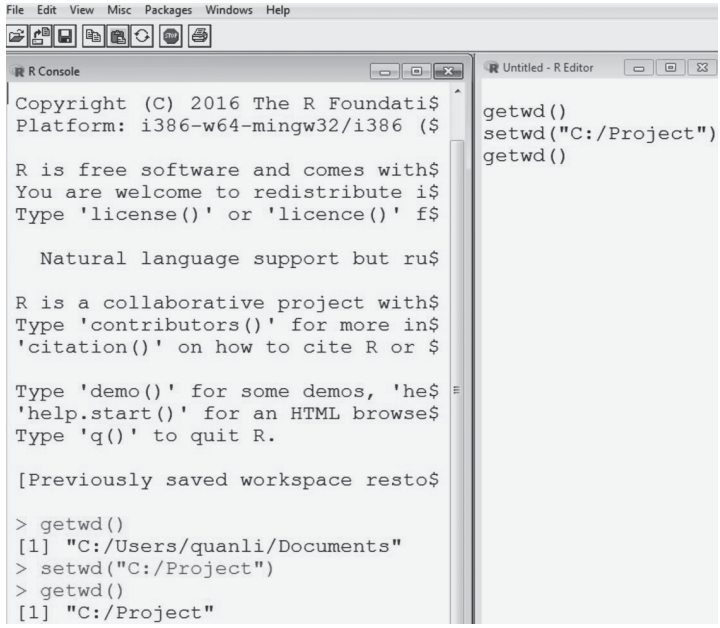


Figure 1.1 How to Write First Toy Program in R.



(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`.

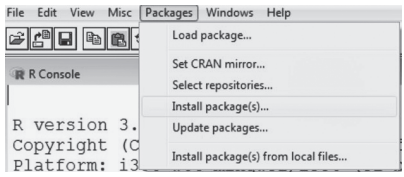
Format Output into Table

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.

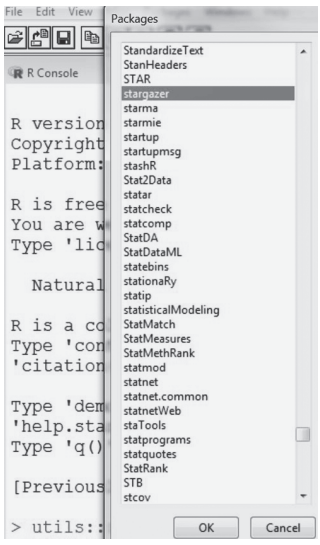
How to Install and Load an Add-on Package

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).

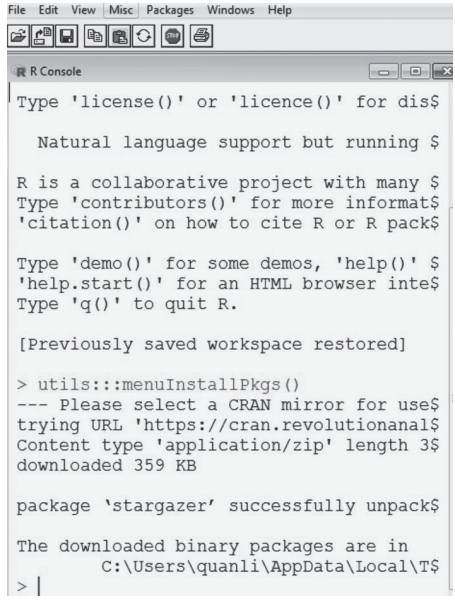


(a) picture 1

(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:

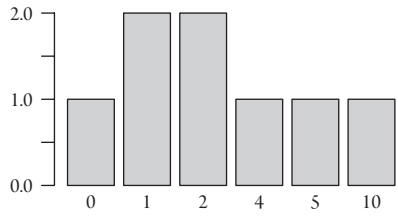


Figure 1.3 Distribution of Discrete Variable vd$v1: Bar Chart.

```
# 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,

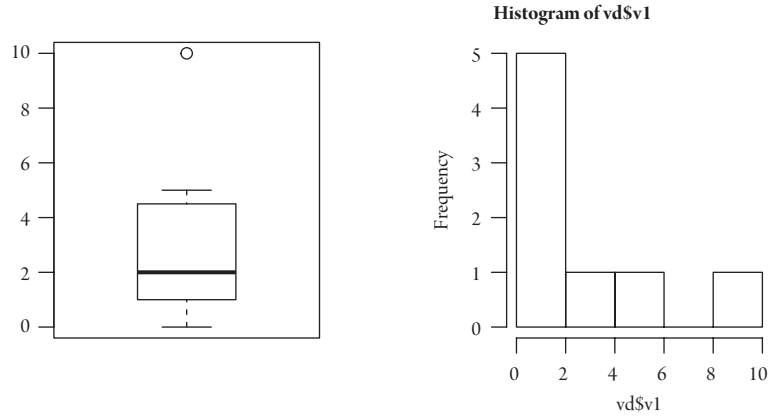


Figure 1.4 Distribution of Continuous Variable vd$v1: Boxplot and Histogram.

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))
```

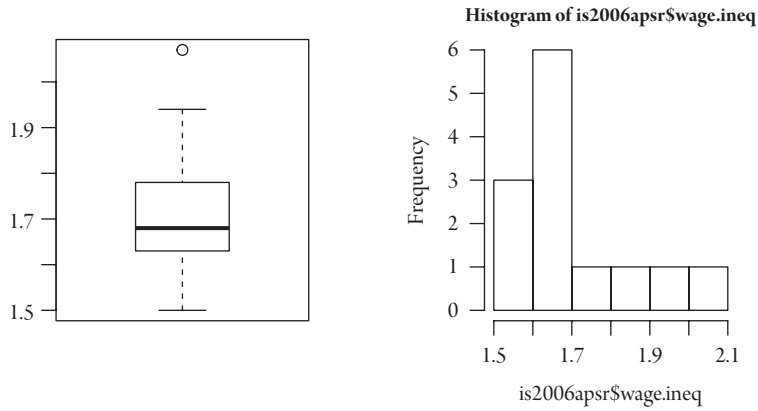


Figure 1.5 Distribution of Wage Inequality from Iversen and Soskice (2006).

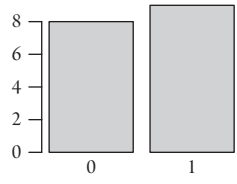


Figure 1.6 Distribution of PR and Majoritarian Systems from Iversen and Soskice (2006).

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
^	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()
```

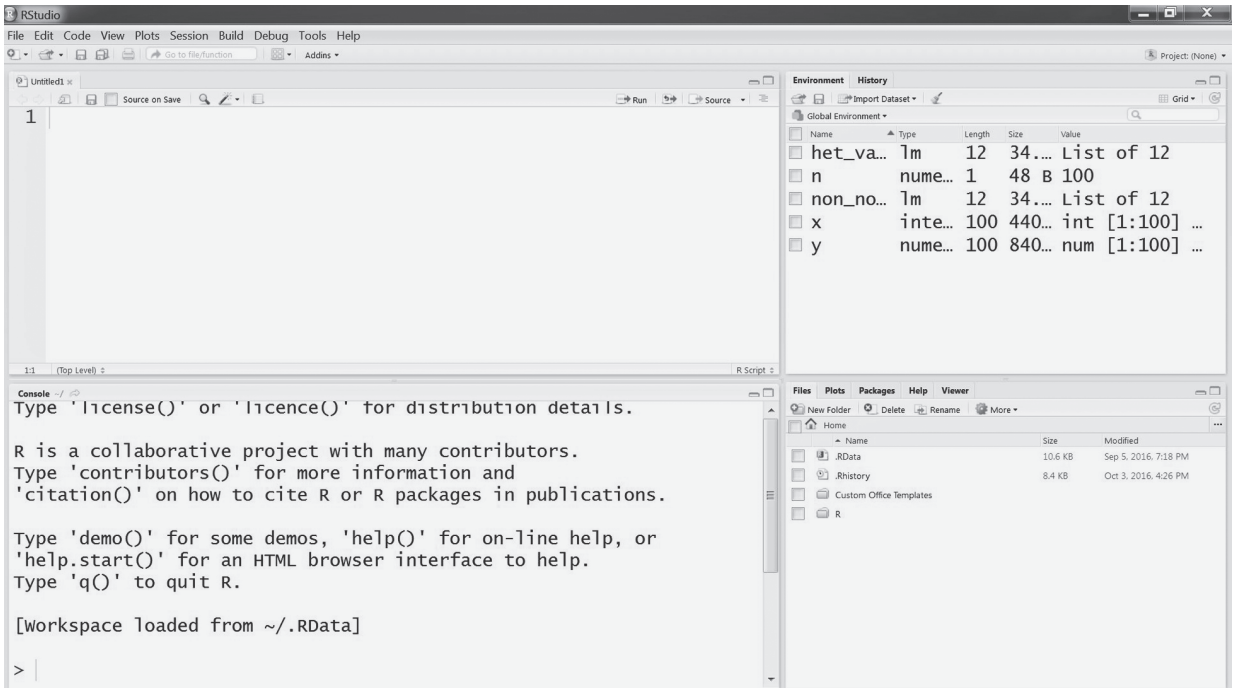


Figure 1.7 RStudio Screenshot.

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.