1.3 Introduction to R

This section provides a brief, self-contained introduction to R that is a prerequisite for the remainder of this book. R is an open-source statistical programming environment, which means that anyone can download it for free, examine source code, and make their own contributions. R is powerful and flexible, enabling us to handle a variety of data sets and create appealing graphics. For this reason, it is widely used in academia and industry. The *New York Times* described R as

> a popular programming language used by a growing number of data analysts inside corporations and academia. It is becoming their lingua franca... whether being used to set ad prices, find new drugs more quickly or fine-tune financial models. Companies as diverse as Google, Pfizer, Merck, Bank of America, the InterContinental Hotels Group and Shell use it.... "The great beauty of R is that you can modify it to do all sorts of things," said Hal Varian, chief economist at Google. "And you have a lot of prepackaged stuff that's already available, so you're standing on the shoulders of giants."[2]

To obtain R, visit `https://cran.r-project.org/` (The Comprehensive R Archive Network or CRAN), select the link that matches your operating system, and then follow the installation instructions.

While a powerful tool for data analysis, R's main cost from a practical viewpoint is that it must be learned as a programming language. This means that we must master various syntaxes and basic rules of computer programming. Learning computer programming is like becoming proficient in a foreign language. It requires a lot of practice and patience, and the learning process may be frustrating. Through numerous data analysis exercises, this book will teach you the basics of statistical programming, which then will allow you to conduct data analysis on your own. The core principle of the book is that we can learn data analysis only by analyzing data.

Unless you have prior programming experience (or have a preference for another text editor such as Emacs), we recommend that you use RStudio. RStudio is an open-source and free program that greatly facilitates the use of R. In one window, RStudio gives users a text editor to write programs, a graph viewer that displays the graphics we create, the R console where programs are executed, a help section, and many other features. It may look complicated at first, but RStudio can make learning how to use R much easier. To obtain RStudio, visit `http://www.rstudio.com/` and follow the download and installation instructions. Figure 1.1 shows a screenshot of RStudio.

In the remainder of this section, we cover three topics: (1) using R as a calculator, (2) creating and manipulating various objects in R, and (3) loading data sets into R.

1.3.1 ARITHMETIC OPERATIONS

We begin by using R as a calculator with standard arithmetic operators. In figure 1.1, the left-hand window of RStudio shows the R console where we can directly enter R

[2] Vance, Ashlee. 2009. "Data Analysts Captivated by R's Power." *New York Times*, January 6.

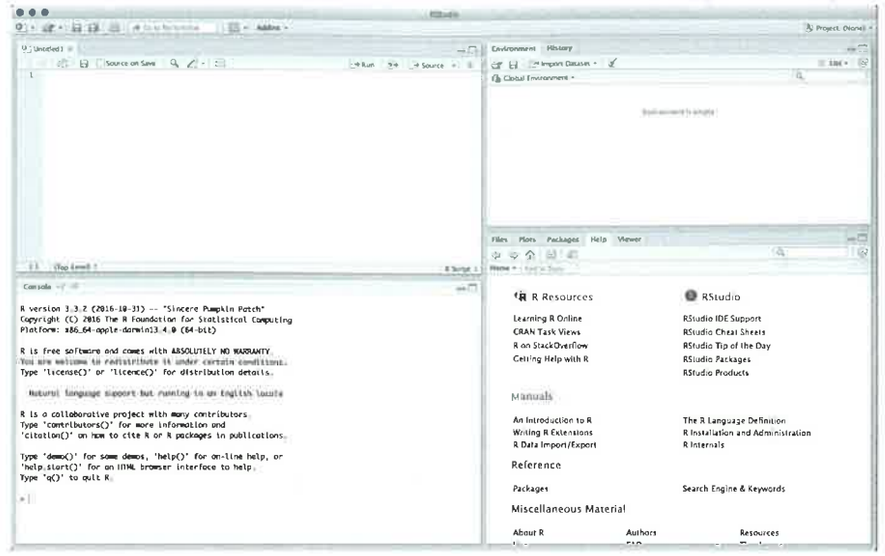

Figure 1.1. Screenshot of RStudio (version 1.0.44). The upper-left window displays a script that contains code. The lower-left window shows the console where R commands can be directly entered. The upper-right window lists R objects and a history of executed R commands. Finally, the lower-right window enables us to view plots, data sets, files and subdirectories in the working directory, R packages, and help pages.

commands. In this R console, we can type in, for example, `5 + 3`, then hit Enter on our keyboard.

```
5 + 3
## [1] 8
```

R ignores spaces, and so `5+3` will return the same result. However, we added a space before and after the operator + to make it easier to read. As this example illustrates, this book displays R commands followed by the outputs they would produce if entered in the R console. These outputs begin with `##` to distinguish them from the R commands that produced them, though this mark will not appear in the R console. Finally, in this example, `[1]` indicates that the output is the first element of a *vector* of length 1 (we will discuss vectors in section 1.3.3). It is important for readers to try these examples on their own. Remember that we can learn programming only by doing! Let's try other examples.

```
5 - 3
## [1] 2
5 / 3
## [1] 1.666667
5 ^ 3
```

```
## [1] 125
5 * (10 - 3)
## [1] 35
sqrt(4)
## [1] 2
```

The final expression is an example of a so-called *function*, which takes an input (or multiple inputs) and produces an output. Here, the function `sqrt()` takes a nonnegative number and returns its square root. As discussed in section 1.3.4, R has numerous other functions, and users can even make their own functions.

1.3.2 OBJECTS

R can store information as an *object* with a name of our choice. Once we have created an object, we just refer to it by name. That is, we are using objects as "shortcuts" to some piece of information or data. For this reason, it is important to use an intuitive and informative name. The name of our object must follow certain restrictions. For example, it cannot begin with a number (but it can contain numbers). Object names also should not contain spaces. We must avoid special characters such as `%` and `$`, which have specific meanings in R. In RStudio, in the upper-right window, called `Environment` (see figure 1.1), we will see the objects we created. We use the *assignment operator* `<-` to assign some value to an object.

For example, we can store the result of the above calculation as an object named `result`, and thereafter we can access the value by referring to the object's name. By default, R will print the value of the object to the console if we just enter the object name and hit Enter. Alternatively, we can explicitly print it by using the `print()` function.

```
result <- 5 + 3
result
## [1] 8
print(result)
## [1] 8
```

Note that if we assign a different value to the same object name, then the value of the object will be changed. As a result, we must be careful not to overwrite previously assigned information that we plan to use later.

```
result <- 5 - 3
result
## [1] 2
```


Another thing to be careful about is that object names are case sensitive. For example, `Hello` is not the same as either `hello` or `HELLO`. As a consequence, we receive an error in the R console when we type `Result` rather than `result`, which is defined above.

```
Result
## Error in eval(expr, envir, enclos): object 'Result' not found
```

Encountering programming errors or bugs is part of the learning process. The tricky part is figuring out how to fix them. Here, the error message tells us that the `Result` object does not exist. We can see the list of existing objects in the `Environment` tab in the upper-right window (see figure 1.1), where we will find that the correct object is `result`. It is also possible to obtain the same list by using the `ls()` function.

So far, we have assigned only numbers to an object. But R can represent various other types of values as objects. For example, we can store a string of characters by using quotation marks.

```
kosuke <- "instructor"
kosuke
## [1] "instructor"
```

In character strings, spacing is allowed.

```
kosuke <- "instructor and author"
kosuke
## [1] "instructor and author"
```

Notice that R treats numbers like characters when we tell it to do so.

```
Result <- "5"
Result
## [1] "5"
```

However, arithmetic operations like addition and subtraction cannot be used for character strings. For example, attempting to divide or take a square root of a character string will result in an error.

```
Result / 3
## Error in Result/3: non-numeric argument to binary operator
```

```
sqrt(Result)
## Error in sqrt(Result): non-numeric argument to mathematical function
```

R recognizes different types of objects by assigning each object to a *class*. Separating objects into classes allows R to perform appropriate operations depending on the objects' class. For example, a number is stored as a numeric object whereas a character string is recognized as a character object. In RStudio, the `Environment` window will show the class of an object as well as its name. The function (which by the way is another class) `class()` tells us to which class an object belongs.

```
result
## [1] 2
class(result)
## [1] "numeric"
Result
## [1] "5"
class(Result)
## [1] "character"

class(sqrt)
## [1] "function"
```

There are many other classes in R, some of which will be introduced throughout this book. In fact, it is even possible to create our own object classes.

1.3.3 VECTORS

We present the simplest (but most inefficient) way of entering data into R. Table 1.2 contains estimates of world population (in thousands) over the past several decades. We can enter these data into R as a numeric vector object. A *vector* or a one-dimensional array simply represents a collection of information stored in a specific order. We use the function `c()`, which stands for "concatenate," to enter a data vector containing multiple values with commas separating different elements of the vector we are creating. For example, we can enter the world population estimates as elements of a single vector.

```
world.pop <- c(2525779, 3026003, 3691173, 4449049, 5320817, 6127700,
   6916183)
world.pop
## [1] 2525779 3026003 3691173 4449049 5320817 6127700 6916183
```


Table 1.2. World Population Estimates.

Year	*World population* (thousands)
1950	2,525,779
1960	3,026,003
1970	3,691,173
1980	4,449,049
1990	5,320,817
2000	6,127,700
2010	6,916,183

Source: United Nations, Department of Economic and Social Affairs, Population Division (2013). *World Population Prospects: The 2012 Revision, DVD Edition.*

We also note that the `c()` function can be used to combine multiple vectors.

```
pop.first <- c(2525779, 3026003, 3691173)
pop.second <- c(4449049, 5320817, 6127700, 6916183)
pop.all <- c(pop.first, pop.second)
pop.all
## [1] 2525779 3026003 3691173 4449049 5320817 6127700 6916183
```

To access specific elements of a vector, we use square brackets `[ ]`. This is called *indexing*. Multiple elements can be extracted via a vector of indices within square brackets. Also within square brackets the dash, `-`, removes the corresponding element from a vector. Note that none of these operations change the original vector.

```
world.pop[2]
## [1] 3026003

world.pop[c(2, 4)]
## [1] 3026003 4449049

world.pop[c(4, 2)]
## [1] 4449049 3026003

world.pop[-3]
## [1] 2525779 3026003 4449049 5320817 6127700 6916183
```

Since each element of this vector is a numeric value, we can apply arithmetic operations to it. The operations will be repeated for each element of the vector. Let's

give the population estimates in millions instead of thousands by dividing each element of the vector by 1000.

```
pop.million <- world.pop / 1000
pop.million
## [1] 2525.779 3026.003 3691.173 4449.049 5320.817 6127.700
## [7] 6916.183
```

We can also express each population estimate as a proportion of the 1950 population estimate. Recall that the 1950 estimate is the first element of the vector `world.pop`.

```
pop.rate <- world.pop / world.pop[1]
pop.rate
## [1] 1.000000 1.198047 1.461400 1.761456 2.106604 2.426063
## [7] 2.738238
```

In addition, arithmetic operations can be done using multiple vectors. For example, we can calculate the percentage increase in population for each decade, defined as the increase over the decade divided by its beginning population. For example, suppose that the population was 100 thousand in one year and increased to 120 thousand in the following year. In this case, we say, "the population increased by 20%." To compute the percentage increase for each decade, we first create two vectors, one without the first decade and the other without the last decade. We then subtract the second vector from the first vector. Each element of the resulting vector equals the population increase. For example, the first element is the difference between the 1960 population estimate and the 1950 estimate.

```
pop.increase <- world.pop[-1] - world.pop[-7]
percent.increase <- (pop.increase / world.pop[-7]) * 100
percent.increase
## [1] 19.80474 21.98180 20.53212 19.59448 15.16464 12.86752
```

Finally, we can also replace the values associated with particular indices by using the usual assignment operator (`<-`). Below, we replace the first two elements of the `percent.increase` vector with their rounded values.

```
percent.increase[c(1, 2)] <- c(20, 22)
percent.increase
## [1] 20.00000 22.00000 20.53212 19.59448 15.16464 12.86752
```

1.3.4 FUNCTIONS

Functions are important objects in R and perform a wide range of tasks. A function often takes multiple input objects and returns an output object. We have already seen

several functions: `sqrt()`, `print()`, `class()`, and `c()`. In R, a function generally runs as `funcname(input)` where `funcname` is the function name and `input` is the input object. In programming (and in math), we call these inputs *arguments*. For example, in the syntax `sqrt(4)`, `sqrt` is the function name and 4 is the argument or the input object.

Some basic functions useful for summarizing data include `length()` for the length of a vector or equivalently the number of elements it has, `min()` for the *minimum* value, `max()` for the *maximum* value, `range()` for the *range* of data, `mean()` for the *mean*, and `sum()` for the *sum* of the data. Right now we are inputting only one object into these functions so we will not use argument names.

```
length(world.pop)
## [1] 7

min(world.pop)
## [1] 2525779

max(world.pop)
## [1] 6916183

range(world.pop)
## [1] 2525779 6916183

mean(world.pop)
## [1] 4579529

sum(world.pop) / length(world.pop)
## [1] 4579529
```

The last expression gives another way of calculating the mean as the sum of all the elements divided by the number of elements.

When multiple arguments are given, the syntax looks like `funcname(input1, input2)`. The order of inputs matters. That is, `funcname(input1, input2)` is different from `funcname(input2, input1)`. To avoid confusion and problems stemming from the order in which we list arguments, it is also a good idea to specify the name of the argument that each input corresponds to. This looks like `funcname(arg1 = input1, arg2 = input2)`.

For example, the `seq()` function can generate a vector composed of an increasing or decreasing sequence. The first argument `from` specifies the number to start from; the second argument `to` specifies the number at which to end the sequence; the last argument `by` indicates the interval to increase or decrease by. We can create an object for the `year` variable from table 1.2 using this function.

```
year <- seq(from = 1950, to = 2010, by = 10)
year
## [1] 1950 1960 1970 1980 1990 2000 2010
```

Notice how we can switch the order of the arguments without changing the output because we have named the input objects.

```
seq(to = 2010, by = 10, from = 1950)
## [1] 1950 1960 1970 1980 1990 2000 2010
```

Although not relevant in this particular example, we can also create a decreasing sequence using the `seq()` function. In addition, the colon operator `:` creates a simple sequence, beginning with the first number specified and increasing or decreasing by 1 to the last number specified.

```
seq(from = 2010, to = 1950, by = -10)
## [1] 2010 2000 1990 1980 1970 1960 1950

2008:2012
## [1] 2008 2009 2010 2011 2012

2012:2008
## [1] 2012 2011 2010 2009 2008
```

The `names()` function can access and assign names to elements of a vector. Element names are not part of the data themselves, but are helpful attributes of the R object. Below, we see that the object `world.pop` does not yet have the names attribute, with `names(world.pop)` returning the `NULL` value. However, once we assign the `year` as the labels for the object, each element of `world.pop` is printed with an informative label.

```
names(world.pop)
## NULL

names(world.pop) <- year
names(world.pop)
## [1] "1950" "1960" "1970" "1980" "1990" "2000" "2010"
```



```
world.pop
##    1950    1960    1970    1980    1990    2000    2010
## 2525779 3026003 3691173 4449049 5320817 6127700 6916183
```

In many situations, we want to create our own functions and use them repeatedly. This allows us to avoid duplicating identical (or nearly identical) sets of code chunks, making our code more efficient and easily interpretable. The `function()` function can create a new function. The syntax takes the following form.

```
myfunction <- function(input1, input2, ..., inputN) {

    DEFINE "output" USING INPUTS

    return(output)
}
```

In this example code, `myfunction` is the function name, `input1, input2, ..., inputN` are the input arguments, and the commands within the braces `{ }` define the actual function. Finally, the `return()` function returns the output of the function. We begin with a simple example, creating a function to compute a summary of a numeric vector.

```
my.summary <- function(x){ # function takes one input
  s.out <- sum(x)
  l.out <- length(x)
  m.out <- s.out / l.out
  out <- c(s.out, l.out, m.out) # define the output
  names(out) <- c("sum", "length", "mean") # add labels
  return(out) # end function by calling output
}
z <- 1:10
my.summary(z)
##    sum length   mean
##   55.0   10.0    5.5

my.summary(world.pop)
##       sum    length      mean
## 32056704         7   4579529
```

Note that objects (e.g., `x`, `s.out`, `l.out`, `m.out`, and `out` in the above example) can be defined within a function independently of the environment in which the function is being created. This means that we need not worry about using identical names for objects inside a function and those outside it.

1.3.5 DATA FILES

So far, the only data we have used has been manually entered into R. But, most of the time, we will load data from an external file. In this book, we will use the following two data file types:

- *CSV* or comma-separated values files represent tabular data. This is conceptually similar to a spreadsheet of data values like those generated by Microsoft Excel or Google Spreadsheet. Each observation is separated by line breaks and each field within the observation is separated by a comma, a tab, or some other character or string.
- *RData* files represent a collection of R objects including data sets. These can contain multiple R objects of different kinds. They are useful for saving intermediate results from our R code as well as data files.

Before interacting with data files, we must ensure they reside in the *working directory*, which R will by default load data from and save data to. There are different ways to change the working directory. In RStudio, the default working directory is shown in the bottom-right window under the `Files` tab (see figure 1.1). Oftentimes, however, the default directory is not the directory we want to use. To change the working directory, click on `More > Set As Working Directory` after choosing the folder we want to work from. Alternatively, we can use the RStudio pull-down menu `Session > Set Working Directory > Choose Directory...` and pick the folder we want to work from. Then, we will see our files and folders in the bottom-right window.

It is also possible to change the working directory using the `setwd()` function by specifying the full path to the folder of our choice as a character string. To display the current working directory, use the function `getwd()` without providing an input. For example, the following syntax sets the working directory to `qss/INTRO` and confirms the result (we suppress the output here).

```
setwd("qss/INTRO")
getwd()
```

Suppose that the United Nations population data in table 1.2 are saved as a CSV file `UNpop.csv`, which resembles that below:

```
year, world.pop
1950, 2525779
1960, 3026003
1970, 3691173
1980, 4449049
1990, 5320817
2000, 6127700
2010, 6916183
```

In RStudio, we can read in or load CSV files by going to the drop-down menu in the upper-right window (see figure 1.1) and clicking `Import Dataset > From Text File ...`. Alternatively, we can use the `read.csv()` function. The following syntax loads the data as a data frame object (more on this object below).


```
UNpop <- read.csv("UNpop.csv")
class(UNpop)

## [1] "data.frame"
```

On the other hand, if the same data set is saved as an object in an RData file named `UNpop.RData`, then we can use the `load()` function, which will load all the R objects saved in `UNpop.RData` into our R session. We do not need to use the assignment operator with the `load()` function when reading in an RData file because the R objects stored in the file already have object names.

```
load("UNpop.RData")
```

Note that R can access any file on our computer if the full location is specified. For example, we can use syntax such as `read.csv("Documents/qss/INTRO/UNpop.csv")` if the data file `UNpop.csv` is stored in the directory `Documents/qss/INTRO/`. However, setting the working directory as shown above allows us to avoid tedious typing.

A data frame object is a collection of vectors, but we can think of it like a spreadsheet. It is often useful to visually inspect data. We can view a spreadsheet-like representation of data frame objects in RStudio by double-clicking on the object name in the `Environment` tab in the upper-right window (see figure 1.1). This will open a new tab displaying the data. Alternatively, we can use the `View()` function, which as its main argument takes the name of a data frame to be examined. Useful functions for this object include `names()` to return a vector of variable names, `nrow()` to return the number of rows, `ncol()` to return the number of columns, `dim()` to combine the outputs of `ncol()` and `nrow()` into a vector, and `summary()` to produce a summary.

```
names(UNpop)

## [1] "year"      "world.pop"

nrow(UNpop)

## [1] 7

ncol(UNpop)

## [1] 2

dim(UNpop)

## [1] 7 2
```

```
summary(UNpop)
##       year         world.pop
##  Min.   :1950   Min.   :2525779
##  1st Qu.:1965   1st Qu.:3358588
##  Median :1980   Median :4449049
##  Mean   :1980   Mean   :4579529
##  3rd Qu.:1995   3rd Qu.:5724258
##  Max.   :2010   Max.   :6916183
```

Notice that the `summary()` function yields, for each variable in the data frame object, the minimum value, the first *quartile* (or 25th *percentile*), the *median* (or 50th percentile), the third quartile (or 75th percentile), and the maximum value. See section 2.6 for more discussion.

The `$` operator is one way to access an individual variable from within a data frame object. It returns a vector containing the specified variable.

```
UNpop$world.pop
## [1] 2525779 3026003 3691173 4449049 5320817 6127700 6916183
```

Another way of retrieving individual variables is to use indexing inside square brackets `[ ]`, as done for a vector. Since a data frame object is a two-dimensional array, we need two indexes, one for rows and the other for columns. Using brackets with a comma `[rows, columns]` allows users to call specific rows and columns by either row/column numbers or row/column names. If we use row/column numbers, sequencing functions covered above, i.e., `:` and `c()`, will be useful. If we do not specify a row (column) index, then the syntax will return all rows (columns). Below are some examples, demonstrating the syntax of indexing.

```
UNpop[, "world.pop"] # extract the column called "world.pop"
## [1] 2525779 3026003 3691173 4449049 5320817 6127700 6916183

UNpop[c(1, 2, 3),]   # extract the first three rows (and all columns)
##   year world.pop
## 1 1950   2525779
## 2 1960   3026003
## 3 1970   3691173

UNpop[1:3, "year"]   # extract the first three rows of the "year" column
## [1] 1950 1960 1970
```


When extracting specific observations from a variable in a data frame object, we provide only one index since the variable is a vector.

```
## take elements 1, 3, 5, ... of the "world.pop" variable
UNpop$world.pop[seq(from = 1, to = nrow(UNpop), by = 2)]
## [1] 2525779 3691173 5320817 6916183
```

In R, missing values are represented by `NA`. When applied to an object with missing values, functions may or may not automatically remove those values before performing operations. We will discuss the details of handling missing values in section 3.2. Here, we note that for many functions, like `mean()`, the argument `na.rm = TRUE` will remove missing data before operations occur. In the example below, the eighth element of the vector is missing, and one cannot calculate the mean until R has been instructed to remove the missing data.

```
world.pop <- c(UNpop$world.pop, NA)
world.pop
## [1] 2525779 3026003 3691173 4449049 5320817 6127700  6916183
## [8]      NA

mean(world.pop)
## [1] NA

mean(world.pop, na.rm = TRUE)
## [1] 4579529
```

1.3.6 SAVING OBJECTS

The objects we create in an R session will be temporarily saved in the *workspace*, which is the current working environment. As mentioned earlier, the `ls()` function displays the names of all objects currently stored in the workspace. In RStudio, all objects in the workspace appear in the `Environment` tab in the upper-right corner. However, these objects will be lost once we terminate the current session. This can be avoided if we save the workspace at the end of each session as an RData file.

When we quit R, we will be asked whether we would like to save the workspace. We should answer no to this so that we get into the habit of explicitly saving only what we need. If we answer yes, then R will save the entire workspace as `.RData` in the working directory without an explicit file name and automatically load it next time we launch R. This is not recommended practice, because the `.RData` file is invisible to users of many operating systems and R will not tell us what objects are loaded unless we explicitly issue the `ls()` function.

In RStudio, we can save the workspace by clicking the `Save` icon in the upper-right `Environment` window (see figure 1.1). Alternatively, from the navigation bar, click

on `Session > Save Workspace As...`, and then pick a location to save the file. Be sure to use the file extension `.RData`. To load the same workspace the next time we start RStudio, click the `Open File` icon in the upper-right `Environment` window, select `Session > Load Workspace...`, or use the `load()` function as before.

It is also possible to save the workspace using the `save.image()` function. The file extension `.RData` should always be used at the end of the file name. Unless the full path is specified, objects will be saved to the working directory. For example, the following syntax saves the workspace as `Chapter1.RData` in the `qss/INTRO` directory provided that this directory already exists.

```
save.image("qss/INTRO/Chapter1.RData")
```

Sometimes, we wish to save only a specific object (e.g., a data frame object) rather than the entire workspace. This can be done with the `save()` function as in `save(xxx, file = "yyy.RData")`, where `xxx` is the object name and `yyy.RData` is the file name. Multiple objects can be listed, and they will be stored as a single RData file. Here are some examples of syntax, in which we again assume the existence of the `qss/INTRO` directory.

```
save(UNpop, file = "Chapter1.RData")
save(world.pop, year, file = "qss/INTRO/Chapter1.RData")
```

In other cases, we may want to save a data frame object as a CSV file rather than an RData file. We can use the `write.csv()` function by specifying the object name and the file name, as the following example illustrates.

```
write.csv(UNpop, file = "UNpop.csv")
```

Finally, to access objects saved in the RData file, simply use the `load()` function as before.

```
load("Chapter1.RData")
```

1.3.7 PACKAGES

One of R's strengths is the existence of a large community of R users who contribute various functionalities as R packages. These packages are available through the Comprehensive R Archive Network (CRAN; `http://cran.r-project.org`). Throughout the book, we will employ various packages. For the purpose of illustration, suppose that we wish to load a data file produced by another statistical software package such as Stata or SPSS. The **foreign** package is useful when dealing with files from other statistical software.

To use the package, we must load it into the workspace using the `library()` function. In some cases, a package needs to be installed before being loaded. In RStudio, we can do this by clicking on `Packages > Install` in the bottom-right window (see figure 1.1), where all currently installed packages are listed, after choosing the desired packages to be installed. Alternatively, we can install from the R console using the `install.packages()` function (the output is suppressed below). Package installation needs only to occur once, though we can update the package later upon the release of a new version (by clicking `Update` or reinstalling it via the `install.packages()` function).

```
install.packages("foreign") # install package
library("foreign") # load package
```

Once the package is loaded, we can use the appropriate functions to load the data file. For example, the `read.dta()` and `read.spss()` functions can read Stata and SPSS data files, respectively (the following syntax assumes the existence of the `UNpop.dta` and `UNpop.sav` files in the working directory).

```
read.dta("UNpop.dta")
read.spss("UNpop.sav")
```

As before, it is also possible to save a data frame object as a data file that can be directly loaded into another statistical software package. For example, the `write.dta()` function will save a data frame object as a Stata data file.

```
write.dta(UNpop, file = "UNpop.dta")
```

1.3.8 PROGRAMMING AND LEARNING TIPS

We conclude this brief introduction to R by providing several practical tips for learning how to program in the R language. First, we should use a text editor like the one that comes with RStudio to write our program rather than directly typing it into the R console. If we just want to see what a command does, or quickly calculate some quantity, we can go ahead and enter it directly into the R console. However, for more involved programming, it is always better to use the text editor and save our code as a text file with the `.R` file extension. This way, we can keep a record of our program and run it again whenever necessary.

In RStudio, use the pull-down menu `File > New File > R Script` or click the `New File` icon (a white square with a green circle enclosing a white plus sign) and choose `R Script`. Either approach will open a blank document for text editing in the upper-left window where we can start writing our code (see figure 1.2). To run our code from the RStudio text editor, simply highlight the code and press the `Run` icon. Alternatively, in Windows, Ctrl+Enter works as a shortcut. The equivalent shortcut for Mac is Command+Enter. Finally, we can also run the entire code in the background

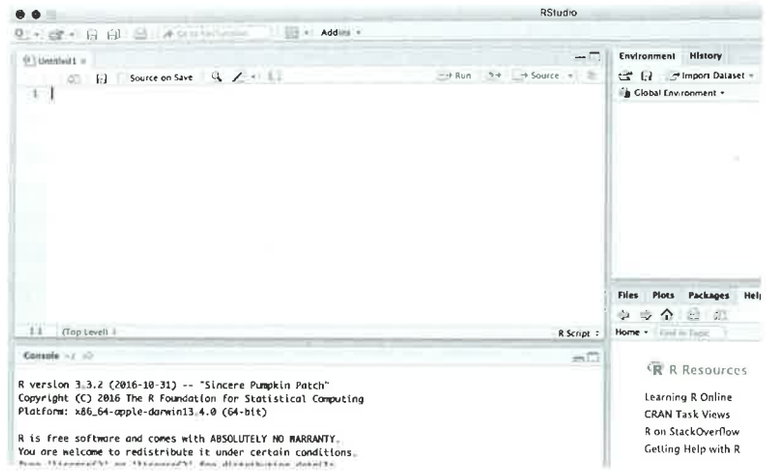

Figure 1.2. Screenshot of the RStudio Text Editor. Once we open an R script file in RStudio, the text editor will appear as one of the windows. It can then be used to write our code.

(so, the code will not appear in the console) by clicking the `Source` icon or using the `source()` function with the code file name (including a full path if it is not placed in the working directory) as the input.

```
source("UNpop.R")
```

Second, we can annotate our R code so that it can be easily understandable to ourselves and others. This is especially important as our code gets more complex. To do this, we use the comment character #, which tells R to ignore everything that follows it. It is customary to use a double comment character ## if a comment occupies an entire line and use a single comment character # if a comment is made within a line after an R command. An example is given here.

```
##
## File: UNpop.R
## Author: Kosuke Imai
## The code loads the UN population data and saves it as a Stata file
##

library(foreign)
UNpop <- read.csv("UNpop.csv")
UNpop$world.pop <- UNpop$world.pop / 1000  # population in millions
write.dta(UNpop, file = "UNpop.dta")
```


Third, for further clarity it is important to follow a certain set of coding rules. For example, we should use informative names for files, variables, and functions. Systematic spacing and indentation are essential too. In the above examples, we place spaces around all binary operators such as <-, =, +, and -, and always add a space after a comma. While comprehensive coverage of coding style is beyond the scope of this book, we encourage you to follow a useful R style guide published by Google at `https://google.github.io/styleguide/Rguide.xml`. In addition, it is possible to check our R code for potential errors and incorrect syntax. In computer science, this process is called *linting*. The `lint()` function in the **lintr** package enables the linting of R code. The following syntax implements the linting of the `UNpop.R` file shown above, where we replace the assignment operator <- in line 8 with the equality sign = for the sake of illustration.

```
library(lintr)
lint("UNpop.R")

## UNpop.R:7:7: style: Use <-, not =, for assignment.
## UNpop = read.csv("UNpop.csv")
##       ^
```

Finally, R Markdown via the **rmarkdown** package is useful for quickly writing documents using R. R Markdown enables us to easily embed R code and its output within a document using straightforward syntax in a plain-text format. The resulting documents can be produced in the form of HTML, PDF, or even Microsoft Word. Because R Markdown embeds R code as well as its output, the results of data analysis presented in documents are reproducible. R Markdown is also integrated into RStudio, making it possible to produce documents with a single click. For a quick start, see `http://rmarkdown.rstudio.com/`.

1.4 Summary

This chapter began with a discussion of the important role that quantitative social science research can play in today's data-rich society. To make contributions to this society through data-driven discovery, we must learn how to analyze data, interpret the results, and communicate our findings to others. To start our journey, we presented a brief introduction to R, which is a powerful programming language for data analysis. The remaining pages of this chapter are dedicated to exercises, designed to ensure that you have mastered the contents of this section. Start with the **swirl** review questions that are available via links from `http://press.princeton.edu/qss/`. If you answer these questions incorrectly, be sure to go back to the relevant sections and review the materials before moving on to the exercises.