Chapter 1

Introduction

In God we trust; all others must bring data.
—William Edwards Deming

Quantitative social science is an interdisciplinary field encompassing a large number of disciplines, including economics, education, political science, public policy, psychology, and sociology. In quantitative social science research, scholars analyze data to understand and solve problems about society and human behavior. For example, researchers examine racial discrimination in the labor market, evaluate the impact of new curricula on students' educational achievements, predict election outcomes, and analyze social media usage. Similar data-driven approaches have been taken up in other neighboring fields such as health, law, journalism, linguistics, and even literature. Because social scientists directly investigate a wide range of real-world issues, the results of their research have enormous potential to directly influence individual members of society, government policies, and business practices.

Over the last couple of decades, quantitative social science has flourished in a variety of areas at an astonishing speed. The number of academic journal articles that present empirical evidence from data analysis has soared. Outside academia, many organizations—including corporations, political campaigns, news media, and government agencies—increasingly rely on data analysis in their decision-making processes. Two transformative technological changes have driven this rapid growth of quantitative social science. First, the Internet has greatly facilitated the *data revolution*, leading to a spike in the amount and diversity of available data. Information sharing makes it possible for researchers and organizations to disseminate numerous data sets in digital form. Second, the *computational revolution*, in terms of both software and hardware, means that anyone can conduct data analysis using their personal computer and favorite data analysis software.

As a direct consequence of these technological changes, the sheer volume of data available to quantitative social scientists has rapidly grown. In the past, researchers largely relied upon data published by governmental agencies (e.g., censuses, election outcomes, and economic indicators) as well as a small number of data sets collected by research groups (e.g., survey data from national election studies and hand-coded data sets about war occurrence and democratic institutions). These data sets still

play an important role in empirical analysis. However, the wide variety of new data has significantly expanded the horizon of quantitative social science research. Researchers are designing and conducting randomized experiments and surveys on their own. Under pressure to increase transparency and accountability, government agencies are making more data publicly available online. For example, in the United States, anyone can download detailed data on campaign contributions and lobbying activities to their personal computers. In Nordic countries like Sweden, a wide range of registers, including income, tax, education, health, and workplace, are available for academic research.

New data sets have emerged across diverse areas. Detailed data about consumer transactions are available through electronic purchasing records. International trade data are collected at the product level between many pairs of countries over several decades. Militaries have also contributed to the data revolution. During the Afghanistan war in the 2000s, the United States and international forces gathered data on the geo-location, timing, and types of insurgent attacks and conducted data analysis to guide counterinsurgency strategy. Similarly, governmental agencies and nongovernmental organizations collected data on civilian casualties from the war. Political campaigns use data analysis to devise voter mobilization strategies by targeting certain types of voters with carefully selected messages.

These data sets also come in varying forms. Quantitative social scientists are analyzing digitized texts as data, including legislative bills, newspaper articles, and the speeches of politicians. The availability of social media data through websites, blogs, tweets, SMS messaging, and Facebook has enabled social scientists to explore how people interact with one another in the online sphere. Geographical information system (GIS) data sets are also widespread. They enable researchers to analyze the legislative redistricting process or civil conflict with attention paid to spatial location. Others have used satellite imagery data to measure the level of electrification in rural areas of developing countries. While still rare, images, sounds, and even videos can be analyzed using quantitative methods for answering social science questions.

Together with the revolution of information technology, the availability of such abundant and diverse data means that anyone, from academics to practitioners, from business analysts to policy makers, and from students to faculty, can make data-driven discoveries. In the past, only statisticians and other specialized professionals conducted data analysis. Now, everyone can turn on their personal computer, download data from the Internet, and analyze them using their favorite software. This has led to increased demands for accountability to demonstrate policy effectiveness. In order to secure funding and increase legitimacy, for example, nongovernmental organizations and governmental agencies must now demonstrate the efficacy of their policies and programs through rigorous evaluation.

This shift towards greater transparency and data-driven discovery requires that students in the social sciences learn how to analyze data, interpret the results, and effectively communicate their empirical findings. Traditionally, introductory statistics courses have focused on teaching students basic statistical concepts by having them conduct straightforward calculations with paper and pencil or, at best, a scientific calculator. Although these concepts are still important and covered in this book, this

traditional approach cannot meet the current demands of society. It is simply not sufficient to achieve "statistical literacy" by learning about common statistical concepts and methods. Instead, all students in the social sciences should acquire basic data analysis skills so that they can exploit the ample opportunities to learn from data and make contributions to society through data-driven discovery.

The belief that everyone should be able to analyze data is the main motivation for writing this book. The book introduces the three elements of data analysis required for quantitative social science research: research contexts, programming techniques, and statistical methods. Any of these elements in isolation is insufficient. Without research contexts, we cannot assess the credibility of assumptions required for data analysis and will not be able to understand what the empirical findings imply. Without programming techniques, we will not be able to analyze data and answer research questions. Without the guidance of statistical principles, we cannot distinguish systematic patterns, known as signals, from idiosyncratic ones, known as noise, possibly leading to invalid inference. (Here, inference refers to drawing conclusions about unknown quantities based on observed data.) This book demonstrates the power of data analysis by combining these three elements.

1.1 Overview of the Book

This book is written for anyone who wishes to learn data analysis and statistics for the first time. The target audience includes researchers, undergraduate and graduate students in social science and other fields, as well as practitioners and even ambitious high-school students. The book has no prerequisite other than some elementary algebra. In particular, readers do not have to possess knowledge of calculus or probability. No programming experience is necessary, though it can certainly be helpful. The book is also appropriate for those who have taken a traditional "paper-and-pencil" introductory statistics course where little data analysis is taught. Through this book, students will discover the excitement that data analysis brings. Those who want to learn R programming might also find this book useful, although here the emphasis is on how to use R to answer quantitative social science questions.

As mentioned above, the unique feature of this book is the presentation of programming techniques and statistical concepts simultaneously through analysis of data sets taken directly from published quantitative social science research. The goal is to demonstrate how social scientists use data analysis to answer important questions about societal problems and human behavior. At the same time, users of the book will learn fundamental statistical concepts and basic programming skills. Most importantly, readers will gain experience with data analysis by examining approximately forty data sets.

The book consists of eight chapters. The current introductory chapter explains how to best utilize the book and presents a brief introduction to R, a popular open-source statistical programming environment. R is freely available for download and runs on Macintosh, Windows, and Linux computers. Readers are strongly encouraged to use RStudio, another freely available software package that has numerous features to make data analysis easier. This chapter ends with two exercises that are

designed to help readers practice elementary R functionalities using data sets from published social science research. All data sets used in this book are freely available for download via links from `http://press.princeton.edu/qss/`. Links to other useful materials, such as the review exercises for each chapter, can also be found on the website. With the exception of chapter 5, the book focuses on the most basic syntax of R and does not introduce the wide range of additional packages that are available. However, upon completion of this book, readers will have acquired enough R programming skills to be able to utilize these packages.

Chapter 2 introduces *causality*, which plays an essential role in social science research whenever we wish to find out whether a particular policy or program changes an outcome of interest. Causality is notoriously difficult to study because we must infer counterfactual outcomes that are not observable. For example, in order to understand the existence of racial discrimination in the labor market, we need to know whether an African-American candidate who did not receive a job offer would have done so if they were white. We will analyze the data from a well-known experimental study in which researchers sent the résumés of fictitious job applicants to potential employers after randomly choosing applicants' names to sound either African-American or Caucasian. Using this study as an application, the chapter will explain how the randomization of treatment assignment enables researchers to identify the average causal effect of the treatment.

Additionally, readers will learn about causal inference in observational studies where researchers do not have control over treatment assignment. The main application is a classic study whose goal was to figure out the impact of increasing the minimum wage on employment. Many economists argue that a minimum-wage increase can reduce employment because employers must pay higher wages to their workers and are therefore made to hire fewer workers. Unfortunately, the decision to increase the minimum wage is not random, but instead is subject to many factors, like economic growth, that are themselves associated with employment. Since these factors influence which companies find themselves in the treatment group, a simple comparison between those who received treatment and those who did not can lead to biased inference.

We introduce several strategies that attempt to reduce this type of selection bias in observational studies. Despite the risk that we will inaccurately estimate treatment effects in observational studies, the results of such studies are often easier to generalize than those obtained from randomized controlled trials. Other examples in chapter 2 include a field experiment concerning social pressure in get-out-the-vote mobilization. Exercises then include a randomized experiment that investigates the causal effect of small class size in early education as well as a natural experiment about political leader assassination and its effects. In terms of R programming, chapter 2 covers logical statements and subsetting.

Chapter 3 introduces the fundamental concept of *measurement*. Accurate measurement is important for any data-driven discovery because bias in measurement can lead to incorrect conclusions and misguided decisions. We begin by considering how to measure public opinion through sample surveys. We analyze the data from a study in which researchers attempted to measure the degree of support among Afghan citizens for international forces and the Taliban insurgency during the Afghanistan war. The chapter explains the power of randomization in survey sampling. Specifically,

random sampling of respondents from a population allows us to obtain a representative sample. As a result, we can infer the opinion of an entire population by analyzing one small representative group. We also discuss the potential biases of survey sampling. Nonresponses can compromise the representativeness of a sample. Misreporting poses a serious threat to inference, especially when respondents are asked sensitive questions, such as whether they support the Taliban insurgency.

The second half of chapter 3 focuses on the measurement of latent or unobservable concepts that play a key role in quantitative social science. Prominent examples of such concepts include ability and ideology. In the chapter, we study political ideology. We first describe a model frequently used to infer the ideological positions of legislators from roll call votes, and examine how the US Congress has polarized over time. We then introduce a basic clustering algorithm, *k*-means, that makes it possible for us to find groups of similar observations. Applying this algorithm to the data, we find that in recent years, the ideological division within Congress has been mainly characterized by the party line. In contrast, we find some divisions within each party in earlier years. This chapter also introduces various measures of the spread of data, including quantiles, standard deviation, and the Gini coefficient. In terms of R programming, the chapter introduces various ways to visualize univariate and bivariate data. The exercises include the reanalysis of a controversial same-sex marriage experiment, which raises issues of academic integrity while illustrating methods covered in the chapter.

Chapter 4 considers *prediction*. Predicting the occurrence of certain events is an essential component of policy and decision-making processes. For example, the forecasting of economic performance is critical for fiscal planning, and early warnings of civil unrest allow foreign policy makers to act proactively. The main application of this chapter is the prediction of US presidential elections using preelection polls. We show that we can make a remarkably accurate prediction by combining multiple polls in a straightforward manner. In addition, we analyze the data from a psychological experiment in which subjects are shown the facial pictures of unknown political candidates and asked to rate their competence. The analysis yields the surprising result that a quick facial impression can predict election outcomes. Through this example, we introduce linear regression models, which are useful tools to predict the values of one variable based on another variable. We describe the relationship between linear regression and correlation, and examine the phenomenon called "regression towards the mean," which is the origin of the term "regression."

Chapter 4 also discusses when regression models can be used to estimate causal effects rather than simply make predictions. Causal inference differs from standard prediction in requiring the prediction of counterfactual, rather than observed, outcomes using the treatment variable as the predictor. We analyze the data from a randomized natural experiment in India where randomly selected villages reserved some of the seats in their village councils for women. Exploiting this randomization, we investigate whether or not having female politicians affects policy outcomes, especially concerning the policy issues female voters care about. The chapter also introduces the regression discontinuity design for making causal inference in observational studies. We investigate how much of British politicians' accumulated wealth is due to holding political office. We answer this question by comparing those who barely won an election with those who narrowly lost it. The chapter introduces powerful but

challenging R programming concepts: loops and conditional statements. The exercises at the end of the chapter include an analysis of whether betting markets can precisely forecast election outcomes.

Chapter 5 is about the *discovery* of patterns from data of various types. When analyzing "big data," we need automated methods and visualization tools to identify consistent patterns in the data. First, we analyze texts as data. Our primary application here is authorship prediction of *The Federalist Papers*, which formed the basis of the US Constitution. Some of the papers have known authors while others do not. We show that by analyzing the frequencies of certain words in the papers with known authorship, we can predict whether Alexander Hamilton or James Madison authored each of the papers with unknown authorship. Second, we show how to analyze network data, focusing on explaining the relationships among units. Within marriage networks in Renaissance Florence, we quantify the key role played by the Medici family. As a more contemporary example, various measures of centrality are introduced and applied to social media data generated by US senators on Twitter.

Finally in chapter 5, we introduce geo-spatial data. We begin by discussing the classic spatial data analysis conducted by John Snow to examine the cause of the 1854 cholera outbreak in London. We then demonstrate how to visualize spatial data through the creation of maps, using US election data as an example. For spatial–temporal data, we create a series of maps as an animation in order to visually characterize changes in spatial patterns over time. Thus, the chapter applies various data visualization techniques using several specialized R packages.

Chapter 6 shifts the focus from data analysis to *probability*, a unified mathematical model of uncertainty. While earlier chapters examine how to estimate parameters and make predictions, they do not discuss the level of uncertainty in empirical findings, a topic that chapter 7 introduces. Probability is important because it lays a foundation for statistical inference, the goal of which is to quantify inferential uncertainty. We begin by discussing the question of how to interpret probability from two dominant perspectives, frequentist and Bayesian. We then provide mathematical definitions of probability and conditional probability, and introduce several fundamental rules of probability. One such rule is called Bayes' rule. We show how to use Bayes' rule and accurately predict individual ethnicity using surname and residence location when no survey data are available.

This chapter also introduces the important concepts of random variables and probability distributions. We use these tools to add a measure of uncertainty to election predictions that we produced in chapter 4 using preelection polls. Another exercise adds uncertainty to the forecasts of election outcomes based on betting market data. The chapter concludes by introducing two fundamental theorems of probability: the law of large numbers and the central limit theorem. These two theorems are widely applicable and help characterize how our estimates behave over repeated sampling as sample size increases. The final set of exercises then addresses two problems: the German cryptography machine from World War II (Enigma), and the detection of election fraud in Russia.

Chapter 7 discusses how to quantify the *uncertainty* of our estimates and predictions. In earlier chapters, we introduced various data analysis methods to find

patterns in data. Building on the groundwork laid in chapter 6, chapter 7 thoroughly explains how certain we should be about such patterns. This chapter shows how to distinguish signals from noise through the computation of standard errors and confidence intervals as well as the use of hypothesis testing. In other words, the chapter concerns statistical inference. Our examples come from earlier chapters, and we focus on measuring the uncertainty of these previously computed estimates. They include the analysis of preelection polls, randomized experiments concerning the effects of class size in early education on students' performance, and an observational study assessing the effects of a minimum-wage increase on employment. When discussing statistical hypothesis tests, we also draw attention to the dangers of multiple testing and publication bias. Finally, we discuss how to quantify the level of uncertainty about the estimates derived from a linear regression model. To do this, we revisit the randomized natural experiment of female politicians in India and the regression discontinuity design for estimating the amount of wealth British politicians are able to accumulate by holding political office.

The final chapter concludes by briefly describing the next steps readers might take upon completion of this book. The chapter also discusses the role of data analysis in quantitative social science research.

1.2 How to Use this Book

In this section, we explain how to use this book, which is based on the following principle:

One can learn data analysis only by doing, not by reading.

This book is not just for reading. The emphasis must be placed on gaining experience in analyzing data. This is best accomplished by trying out the code in the book on one's own, playing with it, and working on various exercises that appear at the end of each chapter. All code and data sets used in the book are freely available for download via links from `http://press.princeton.edu/qss/`.

The book is cumulative. Later chapters assume that readers are already familiar with most of the materials covered in earlier parts. Hence, in general, it is not advisable to skip chapters. The exception is chapter 5, "Discovery," the contents of which are not used in subsequent chapters. Nevertheless, this chapter contains some of the most interesting data analysis examples of the book and readers are encouraged to study it.

The book can be used for course instruction in a variety of ways. In a traditional introductory statistics course, one can assign the book, or parts of it, as supplementary reading that provides data analysis exercises. The book is best utilized in a data analysis course where an instructor spends less time on lecturing to students and instead works interactively with students on data analysis exercises in the classroom. In such a course, the relevant portion of the book is assigned prior to each class. In the classroom, the instructor reviews new methodological and programming concepts and then applies them to one of the exercises from the book or any other similar application of their choice. Throughout this process, the instructor can discuss the exercises interactively

with students, perhaps using the Socratic method, until the class collectively arrives at a solution. After such a classroom discussion, it would be ideal to follow up with a computer lab session, in which a small number of students, together with an instructor, work on another exercise.

This teaching format is consistent with the "particular general particular" principle.[1] This principle states that an instructor should first introduce a particular example to illustrate a new concept, then provide a general treatment of it, and finally apply it to another particular example. Reading assignments introduce a particular example and a general discussion of new concepts to students. Classroom discussion then allows the instructor to provide another general treatment of these concepts and then, together with students, apply them to another example. This is an effective teaching strategy that engages students with active learning and builds their ability to conduct data analysis in social science research. Finally, the instructor can assign another application as a problem set to assess whether students have mastered the materials. To facilitate this, for each chapter instructors can obtain, upon request, access to a private repository that contains additional exercises and their solutions.

In terms of the materials to cover, an example of the course outline for a 15-week-long semester is given below. We assume that there are approximately two hours of lectures and one hour of computer lab sessions each week. Having hands-on computer lab sessions with a small number of students, in which they learn how to analyze data, is essential.

Chapter title	*Chapter number*	*Weeks*
Introduction	1	1
Causality	2	2–3
Measurement	3	4–5
Prediction	4	6–7
Discovery	5	8–9
Probability	6	10–12
Uncertainty	7	13–15

For a shorter course, there are at least two ways to reduce the material. One option is to focus on aspects of data science and omit statistical inference. Specifically, from the above outline, we can remove chapter 6, "Probability," and chapter 7, "Uncertainty." An alternative approach is to skip chapter 5, "Discovery," which covers the analysis of textual, network, and spatial data, and include the chapters on probability and uncertainty.

Finally, to ensure mastery of the basic methodological and programming concepts introduced in each chapter, we recommend that users first read a chapter, practice all of the code it contains, and upon completion of each chapter, try the online review questions before attempting to solve the associated exercises. These review questions

[1] Frederick Mosteller (1980) "Classroom and platform performance." *American Statistician*, vol. 34, no. 1 (February), pp. 11–17.

Table 1.1. The **swirl** Review Exercises.

Chapter	***swirl** lesson*	*Sections covered*
1: Introduction	INTRO1	1.3
	INTRO2	1.3
2: Causality	CAUSALITY1	2.1–2.4
	CAUSALITY2	2.5–2.6
3: Measurement	MEASUREMENT1	3.1–3.4
	MEASUREMENT2	3.5–3.7
4: Prediction	PREDICTION1	4.1
	PREDICTION2	4.2
	PREDICTION3	4.3
5: Discovery	DISCOVERY1	5.1
	DISCOVERY2	5.2
	DISCOVERY3	5.3
6: Probability	PROBABILITY1	6.1–6.3
	PROBABILITY2	6.4–6.5
7: Uncertainty	UNCERTAINTY1	7.1
	UNCERTAINTY2	7.2
	UNCERTAINTY3	7.3

Note: The table shows the correspondence between the chapters and sections of the book and each set of **swirl** review exercises.

are available as **swirl** lessons via links from `http://press.princeton.edu/qss/`, and can be answered within R. Instructors are strongly encouraged to assign these **swirl** exercises prior to each class so that students learn the basics before moving on to more complicated data analysis exercises. To start the online review questions, users must first install the **swirl** package (see section 1.3.7) and then the lessons for this book using the following three lines of commands within R. Note that this installation needs to be done only once.

```
install.packages("swirl") # install the package
library(swirl) # load the package
install_course_github("kosukeimai", "qss-swirl") # install the course
```

Table 1.1 lists the available set of **swirl** review exercises along with their corresponding chapters and sections. To start a **swirl** lesson for review questions, we can use the following command.

```
library(swirl)
swirl()
```

More information about **swirl** is available at `http://swirlstats.com/`.

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.

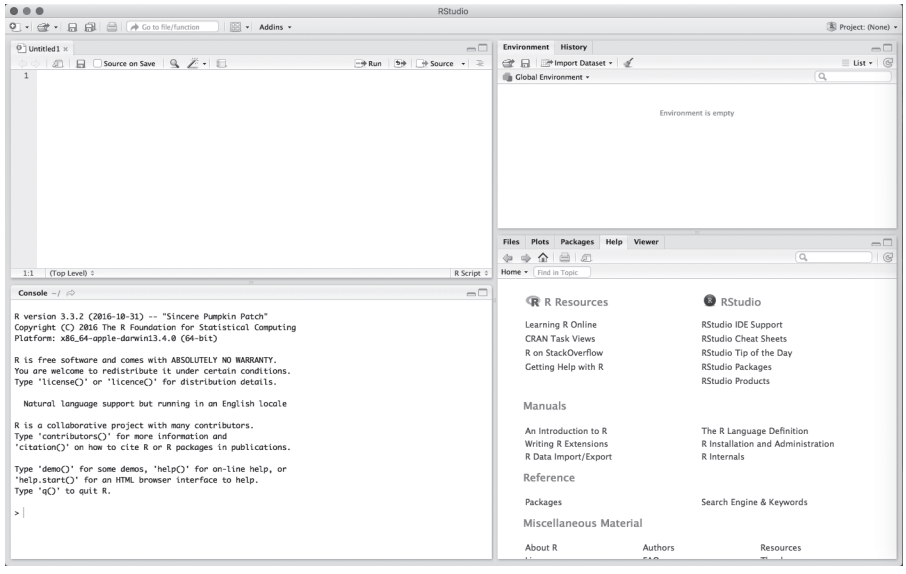

Figure 1.1. Screenshot of RStudio (version 1.0.44). The upper-left window displays a script that contains code. The lower-left window shows the console where R commands can be directly entered. The upper-right window lists R objects and a history of executed R commands. Finally, the lower-right window 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

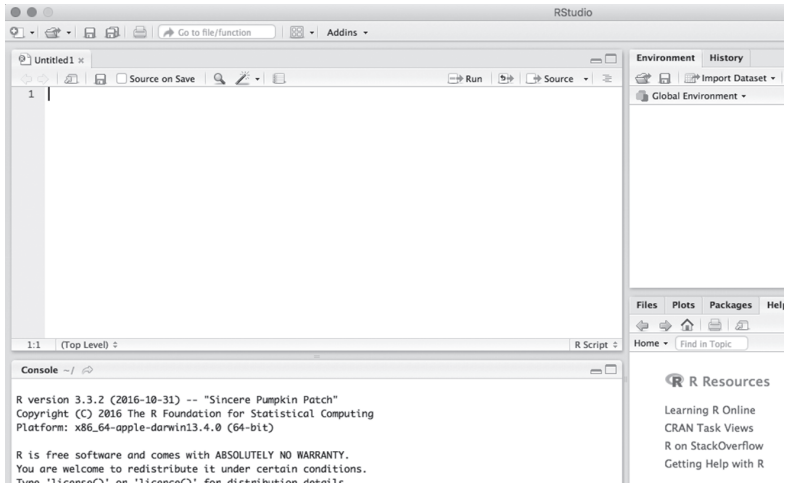

Figure 1.2. Screenshot of the RStudio Text Editor. Once we open an R script file in RStudio, the text editor will appear as one of the windows. It can then be used to write our code.

(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 `lintr()` 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:8:7: 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.

Table 1.3. US Election Turnout Data.

Variable	*Description*
`year`	election year
`ANES`	ANES estimated turnout rate
`VEP`	voting eligible population (in thousands)
`VAP`	voting age population (in thousands)
`total`	total ballots cast for highest office (in thousands)
`felons`	total ineligible felons (in thousands)
`noncitizens`	total noncitizens (in thousands)
`overseas`	total eligible overseas voters (in thousands)
`osvoters`	total ballots counted by overseas voters (in thousands)

1.5 Exercises

1.5.1 BIAS IN SELF-REPORTED TURNOUT

Surveys are frequently used to measure political behavior such as voter turnout, but some researchers are concerned about the accuracy of self-reports. In particular, they worry about possible *social desirability bias* where, in postelection surveys, respondents who did not vote in an election lie about not having voted because they may feel that they should have voted. Is such a bias present in the American National Election Studies (ANES)? ANES is a nationwide survey that has been conducted for every election since 1948. ANES is based on face-to-face interviews with a nationally representative sample of adults. Table 1.3 displays the names and descriptions of variables in the `turnout.csv` data file.

1. Load the data into `R` and check the dimensions of the data. Also, obtain a summary of the data. How many observations are there? What is the range of years covered in this data set?

2. Calculate the turnout rate based on the voting age population or VAP. Note that for this data set, we must add the total number of eligible overseas voters since the `VAP` variable does not include these individuals in the count. Next, calculate the turnout rate using the voting eligible population or VEP. What difference do you observe?

3. Compute the differences between the VAP and ANES estimates of turnout rate. How big is the difference on average? What is the range of the differences? Conduct the same comparison for the VEP and ANES estimates of voter turnout. Briefly comment on the results.

4. Compare the VEP turnout rate with the ANES turnout rate separately for presidential elections and midterm elections. Note that the data set excludes the year 2006. Does the bias of the ANES estimates vary across election types?

Table 1.4. Fertility and Mortality Estimate Data.

Variable	*Description*
`country`	abbreviated country name
`period`	period during which data are collected
`age`	age group
`births`	number of births (in thousands), i.e., the number of children born to women of the age group
`deaths`	number of deaths (in thousands)
`py.men`	person-years for men (in thousands)
`py.women`	person-years for women (in thousands)

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

5. Divide the data into half by election years such that you subset the data into two periods. Calculate the difference between the VEP turnout rate and the ANES turnout rate separately for each year within each period. Has the bias of ANES increased over time?

6. ANES does not interview prisoners and overseas voters. Calculate an adjustment to the 2008 VAP turnout rate. Begin by subtracting the total number of ineligible felons and noncitizens from the VAP to calculate an adjusted VAP. Next, calculate an adjusted VAP turnout rate, taking care to subtract the number of overseas ballots counted from the total ballots in 2008. Compare the adjusted VAP turnout with the unadjusted VAP, VEP, and the ANES turnout rate. Briefly discuss the results.

1.5.2 UNDERSTANDING WORLD POPULATION DYNAMICS

Understanding population dynamics is important for many areas of social science. We will calculate some basic demographic quantities of births and deaths for the world's population from two time periods: 1950 to 1955 and 2005 to 2010. We will analyze the following CSV data files: `Kenya.csv`, `Sweden.csv`, and `World.csv`. The files contain population data for Kenya, Sweden, and the world, respectively. Table 1.4 presents the names and descriptions of the variables in each data set. The data are collected for a period of 5 years where *person-year* is a measure of the time contribution of each person during the period. For example, a person who lives through the entire 5-year period contributes 5 person-years, whereas someone who lives only through the first half of the period contributes 2.5 person-years. Before you begin this exercise, it would be a good idea to directly inspect each data set. In R, this can be done with the `View()` function, which takes as its argument the name of the data frame to be examined. Alternatively, in RStudio, double-clicking a data frame in the `Environment` tab will enable you to view the data in a spreadsheet-like form.

1. We begin by computing *crude birth rate* (CBR) for a given period. The CBR is defined as

$$\text{CBR} = \frac{\text{number of births}}{\text{number of person-years lived}}.$$

Compute the CBR for each period, separately for Kenya, Sweden, and the world. Start by computing the total person-years, recorded as a new variable within each existing data frame via the `$` operator, by summing the person-years for men and women. Then, store the results as a vector of length 2 (CBRs for two periods) for each region with appropriate labels. You may wish to create your own function for the purpose of efficient programming. Briefly describe patterns you observe in the resulting CBRs.

2. The CBR is easy to understand but contains both men and women of all ages in the denominator. We next calculate the *total fertility rate* (TFR). Unlike the CBR, the TFR adjusts for age compositions in the female population. To do this, we need to first calculate the *age-specific fertility rate* (ASFR), which represents the fertility rate for women of the reproductive age range $[15, 50)$. The ASFR for the age range $[x, x+\delta)$, where x is the starting age and δ is the width of the age range (measured in years), is defined as

$$\text{ASFR}_{[x,\ x+\delta)} = \frac{\text{number of births to women of age } [x,\ x+\delta)}{\text{number of person-years lived by women of age } [x,\ x+\delta)}.$$

Note that square brackets, [and], include the limit whereas parentheses, (and), exclude it. For example, $[20, 25)$ represents the age range that is greater than or equal to 20 years old and less than 25 years old. In typical demographic data, the age range δ is set to 5 years. Compute the ASFR for Sweden and Kenya as well as the entire world for each of the two periods. Store the resulting ASFRs separately for each region. What does the pattern of these ASFRs say about reproduction among women in Sweden and Kenya?

3. Using the ASFR, we can define the TFR as the average number of children that women give birth to if they live through their entire reproductive age:

$$\text{TFR} = \text{ASFR}_{[15,\ 20)} \times 5 + \text{ASFR}_{[20,\ 25)} \times 5 + \cdots + \text{ASFR}_{[45,\ 50)} \times 5.$$

We multiply each age-specific fertility rate by 5 because the age range is 5 years. Compute the TFR for Sweden and Kenya as well as the entire world for each of the two periods. As in the previous question, continue to assume that the reproductive age range of women is $[15, 50)$. Store the resulting two TFRs for each country or the world as vectors of length 2. In general, how has the number of women changed in the world from 1950 to 2000? What about the total number of births in the world?

4. Next, we will examine another important demographic process: death. Compute the *crude death rate* (CDR), which is a concept analogous to the CBR, for each

period and separately for each region. Store the resulting CDRs for each country and the world as vectors of length 2. The CDR is defined as

$$\text{CDR} = \frac{\text{number of deaths}}{\text{number of person-years lived}}.$$

Briefly describe the patterns you observe in the resulting CDRs.

5. One puzzling finding from the previous question is that the CDR for Kenya during the period 2005–2010 is about the same level as that for Sweden. We would expect people in developed countries like Sweden to have a lower death rate than those in developing countries like Kenya. While it is simple and easy to understand, the CDR does not take into account the age composition of a population. We therefore compute the *age-specific death rate* (ASDR). The ASDR for age range $[x, x + \delta)$ is defined as

$$\text{ASDR}_{[x,\ x+\delta)} = \frac{\text{number of deaths for people of age } [x,\ x+\delta)}{\text{number of person-years of people of age } [x,\ x+\delta)}.$$

Calculate the ASDR for each age group, separately for Kenya and Sweden, during the period 2005–2010. Briefly describe the pattern you observe.

6. One way to understand the difference in the CDR between Kenya and Sweden is to compute the counterfactual CDR for Kenya using Sweden's population distribution (or vice versa). This can be done by applying the following alternative formula for the CDR:

$$\text{CDR} = \text{ASDR}_{[0,5)} \times P_{[0,5)} + \text{ASDR}_{[5,10)} \times P_{[5,10)} + \cdots,$$

where $P_{[x,x+\delta)}$ is the proportion of the population in the age range $[x, x + \delta)$. We compute this as the ratio of person-years in that age range relative to the total person-years across all age ranges. To conduct this counterfactual analysis, we use $\text{ASDR}_{[x,x+\delta)}$ from Kenya and $P_{[x,x+\delta)}$ from Sweden during the period 2005–2010. That is, first calculate the age-specific population proportions for Sweden and then use them to compute the counterfactual CDR for Kenya. How does this counterfactual CDR compare with the original CDR of Kenya? Briefly interpret the result.