Contents

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. Such research projects range from the examination of racial discrimination in the labor market to the impact evaluation of new curricula on students' educational achievements, and from the prediction of election outcomes to the analysis of social media usage. A similar data-driven approach has 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 of 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*, a spike in the amount and diversity of available data, through information sharing, making 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 essentially anyone can conduct data analysis using their personal computer and favorite data analysis software, without needing to access expensive computational facilities.

As a direct consequence of these technological changes, the sheer volume of data available to quantitative social scientists has rapidly grown. In the past, most researchers 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 groups of researchers (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 now 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 used for academic research.

New data sets have emerged across diverse areas. For example, detailed data about consumer transactions are available through electronic purchasing records. International trade data are now collected at the product level between many pairs of countries over several decades. Militaries have also contributed to the data revolution. During the recent war in Afghanistan, the United States and international forces gathered data on the geolocation, 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 now analyzing digitized texts as data, including legislative bills, newspaper articles, and speeches. 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, enabling, for example, researchers to analyze the legislative redistricting process and its outcomes. Others have used satellite imagery data to measure the level of electrification in rural areas of developing countries. Images, sounds, and even videos can now be analyzed using quantitative methods to answer 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 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, in the current day and age this traditional approach cannot meet the 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 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 the writing of this book. The book introduces 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 and 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, these 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 apply **R** to quantitative social science research.

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 problems of society 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, which are designed to have readers practice elementary **R** functionalities using data sets from published social science research. All code and data sets used in this book are freely available for download at https://github.com/kosukeimai/qss. This website also provides other useful materials, such as the review exercises for each chapter.

The original version of this textbook focused on the most basic syntax of **R**, which is often referred to as "base **R**." This book instead relies heavily on what is called the "**tidyverse**" (specifically version 1.3.1 of the **tidyverse**).[1] We will cover more on what that means as we introduce **R**. However, it is worth noting now that there are often many different ways to write code in **R** that produces the same results. You can think of the **tidyverse** as being a dialect or

[1] For more detail, see Hadley Wickham et al. (2019) "Welcome to the tidyverse." *Journal of Open Source Software*, vol. 4, no. 43, p. 1686, https://doi.org/10.21105/joss.01686. We are deeply indebted to an earlier contribution by Jeffrey Arnold (with additions from Nora Webb Williams and Calvin Garner), which is available at https://github.com/jrnold/qss-tidy.

specific syntax of **R** programming. It is another (sometimes more efficient or elegant) way of asking for the same thing from **R** as base **R**. This book is not a complete introduction to the **tidyverse**. We recommend *R for Data Science* by Hadley Wickham and Garrett Grolemond for additional reference.

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 or not 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 resumes of fictitious job applicants to potential employers after randomly choosing the applicants' names to sound either African American or White. 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.

In chapter 2, readers will also learn about causal inference in observational studies where researchers do not have control over treatment assignment. The main application is a classic study of the relationship between minimum wage and employment. The goal of the study was to figure out the impact of increasing 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, instead subject to many factors, like economic growth, that are themselves associated with employment. Since these factors influence which companies select themselves into 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 selection bias in observational studies. Despite the risk that we will inaccurately estimate treatment effects in observational studies, their results are often easier to generalize than those obtained from randomized controlled trials. Other examples in this chapter include field experiments concerning social pressure in get-out-the-vote mobilization. In terms of **R** programming, chapter 2 covers logical statements and subsetting. The exercises include a randomized experiment that investigates the impact of small class size in early education and an observational study about the assassination of political leaders.

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 attempt to measure the degree of support among Afghan citizens for international forces and the Taliban insurgency during the war in Afghanistan. We explain the power of randomization in survey sampling. Specifically, random sampling of respondents from a population allows us to obtain a representative sample of this population. As a result, we can infer the opinion of an entire population by analyzing one small representative sample. 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 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 some automated methods and visualization tools to identify consistent patterns in the data. First, we analyze texts as data. Our primary application here is the 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, which record information about the relationships among units. Within marriage networks in Renaissance Florence, we quantify the key role played by the Medici family. Various measures of centrality are introduced and applied to a Twitter network, an example of social media data.

Finally, chapter 5 introduces geospatial 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 change 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 focus on 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 frequentist and Bayesian perspectives. We then provide mathematical definitions of probability and conditional probability, and we introduce several fundamental rules of probability. One such rule is Bayes' rule. We show how we can use Bayes' rule to 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 the election predictions we produced using preelection polls in chapter 4. Another exercise adds uncertainty to the forecasts of election outcomes based on betting market data. Finally, chapter 6 introduces 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 as sample size increases and over repeated sampling. The last set of exercises in this chapter are about 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 off 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 hypothesis testing. In other words, the chapter is concerned with statistical inference. Our examples are drawn 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 the linear regression model. 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. The book is based on the following principle:

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

The book is not just for reading. Therefore, 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 at https://github.com/kosukeimai/qss/.

The book is cumulative. Materials that appear later in the book assume that readers are already familiar with most of the materials covered in earlier parts of the book. Hence, in general, it is not a good idea 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. In the authors' experience, however, 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, a 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 exercise 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.[2] 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. The reading assignment introduces a particular example and a general discussion of new concepts to students. The classroom discussion 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 mastered the

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

materials. To facilitate this, for each chapter, instructors can obtain, upon request, access to additional exercises and their solutions at a private repository.

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

- Week 1: Introduction (chapter 1)
- Weeks 2–3: Causality (chapter 2)
- Weeks 4–5: Measurement (chapter 3)
- Weeks 6–7: Prediction (chapter 4)
- Weeks 8–9: Discovery (chapter 5)
- Weeks 10–12: Probability (chapter 6)
- Weeks 13–15: Uncertainty (chapter 7)

For a shorter course, there are at least two ways to reduce the materials. One option is to focus on the aspects of "data analysis" 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.

1.3 Introduction to R and the tidyverse

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[3]

> 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."

To obtain **R**, go to 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. We used version 4.0.2 of **R** when writing this textbook.

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 syntax and rules of computer programming. Learning computer programming is like becoming

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

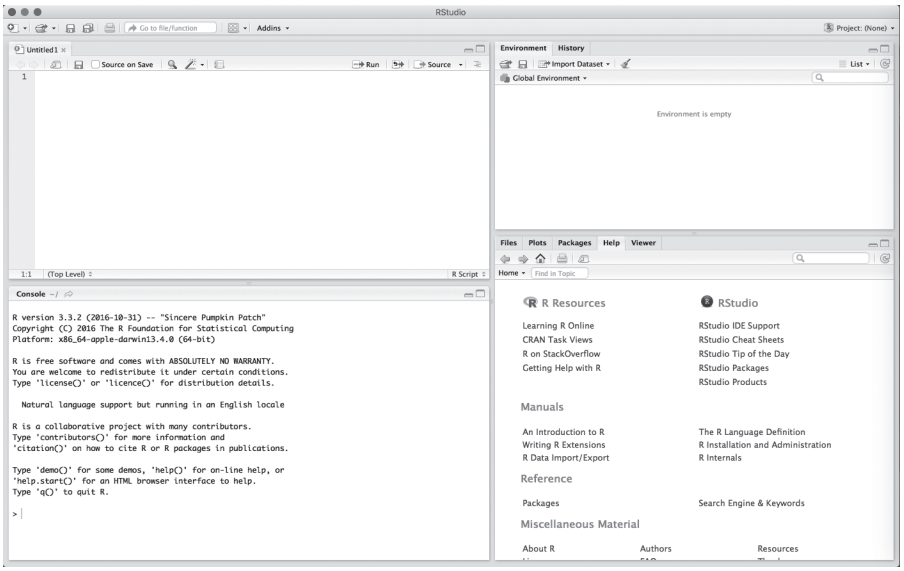

Figure 1.1. Screenshot of **RStudio**. The left window shows the **R** console where **R** commands can be entered. The upper-right window lists **R** objects once they are created and a history of executed **R** commands. Finally, the lower-right window enables us to view plots, data sets, files and subdirectories in the working directory, **R** packages, and help pages.

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**. For example, we wrote this entire book in **RStudio** using **bookdown**.[4] **RStudio** gives users a text editor where we write programs, a graph viewer which displays graphics we make, the **R** console where we execute our programs, a help section, and many other features. It may look complicated at first, but **RStudio** should make learning how to use **R** much easier. To obtain **RStudio**, go to 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 five main topics: (1) using **R** as a calculator, (2) writing **R** scripts, (3) loading packages, (4) creating and manipulating various objects in **R**, and (5) loading, subsetting, adding variables to, and summarizing data sets in **R**.

1.3.1 ARITHMETIC OPERATIONS: R AS A CALCULATOR

We begin by using **R** as a calculator with standard arithmetic operators. In the **R** console (the left panel in **RStudio**, with a > and a cursor; see figure 1.1), we can type in, for example, `5 + 3`, then hit `Enter` on our keyboard.

[4] Yihui Xie (2016) *bookdown: Authoring Books and Technical Documents with R Markdown*. Chapman and Hall/CRC.

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

R ignores spaces, and so typing `5+3` will return the identical result to `5 + 3`. However, we added a space before and after the operator `+` to make it easier to read. As this example illustrates, in this book, **R** commands will be displayed, followed by the outputs they would produce if entered in the **R** console. These outputs are marked by `##` to distinguish them from the **R** commands that produced them. You will not see the `##` in your **R** console, you will just see the output (`8`, in this case). 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 1.3.5). It is important for readers to try these examples on their own. Remember that we can only learn programming by doing! Let's try some additional 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 1.3.6, **R** has numerous other functions, and users can even make their own functions.

1.3.2 **R** SCRIPTS

You have now run your first commands in **R**! If you scroll up in the console, you will see a record of the commands you entered and the output. However, if you close **RStudio** and restart it, the console will start again clean. To save a record of our commands, we can use an **R** *script*. An **R** script is a text file with the file extension (ending) of `.R`. In **RStudio**, you can start a new script using the dropdown menu `File > New File > R Script`. There are at least two other ways to start a new **R** script in **RStudio**—see if you can find them.

Try out your new **R** script by typing in the commands from the section above on arithmetic operations into the script. Notice that if you hit `Enter`, you get a line break in the script but no output. In order to have **R** actually evaluate the command you have written,

highlight it and press the `Run` button in **RStudio**. You should see the command and the output appear in the console window. In **RStudio**, there are many ways to ask **R** to run **R** code from a script. For example, you can use keyboard shortcuts, which is much faster than clicking the `Run` button. The shortcut `Ctrl+Enter` (on a Windows machine) or `Cmd+Return` (on a Mac) will run the line of code your cursor is on. You can see all of the many **RStudio** keyboard shortcuts by searching online or clicking on `Tools > Keyboard Shortcuts Help`.

R scripts are a useful way to save **R** code. Go ahead and save the new script you have just created. The next time you open **RStudio**, you can now open that script and rerun the same **R** commands. **R** scripts can get very messy very quickly and there are many guides and suggestions for best practice. We offer two suggestions for now. First, use your **R** script to save finalized commands. Type in the console to practice and debug, then use the script to save what worked. Second, **R** scripts allow you to save comments, or annotations, to your code. Anything following the `#` symbol in a script will not be treated like an **R** command—essentially, **R** will skip over it. You can use comments to explain what your commands do. This is helpful for transparency when sharing code with others and for reminding yourself what you were doing when you next return to **R**. 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. The code below provides an example of commenting.

```
## this is the start of an R script
## the heading provides some information about the file

## File name: testing_arithm.R
## Author: Kosuke Imai and Nora Webb Williams
## Purpose: Practicing basic math commands and commenting in R
##

5 - 3 # what is 5 minus 3?
5 / 3
5 ^ 3
5 * (10 - 3) # a bit more complex
sqrt(4) # this function will take the square root of a number
```

1.3.3 LOADING PACKAGES

One of the benefits of using **R** is that many researchers have developed open-source *packages* that are easily installed and used for your own projects. Packages might contain data, functions, or other tools. In this book, we rely heavily on a family of packages collectively referred to as the **tidyverse**. The **tidyverse** packages include **ggplot2**, **dplyr**, **tidyr**, **readr**, **purrr**, **tibble**, and a few others. As mentioned previously, you can think of the **tidyverse** as a dialect of **R** programming. The **tidyverse** is in continuing development, meaning that some of the functions are subject to change (though many of the core functions are fairly stable). If you encounter difficulties getting our example code to run, it may be because the **tidyverse**

has changed. If this is the case, you can look up documentation about the package to learn how to update your code.

When you open an **R** script, you can usually tell fairly quickly whether the author is using base **R** or the **tidyverse** (or both—we can mix and match to tell **R** what we want it to do). Scripts in base **R** tend to have lots of dollar signs (`$`) and square brackets (`[]`) while scripts using the **tidyverse** tend to have lots of what we call the *pipe* operator (`%>%`).

The `install.packages()` command will download the required package materials to your computer. You only need to install a package once per computer, though we can update packages later upon the release of a new version (by clicking `Update` or reinstalling it via the `install.packages()` function). In **RStudio**, you can also install packages by using the `Tools > Install Packages...` dropdown menu. After you have successfully installed a package, you load it for your current **R** session with the `library()` command. Although you do not need to install packages at the start of every **R** session (or script), you do need to load your necessary packages every time. If you receive an error that a certain function is not found, it may mean that you forgot to run a `library()` command for the package that contains that function.

You can run the following commands to install the package **devtools**, which contains a function that will then allow you to install the **qss** package. This package has all the data you will need for the examples and exercises in this book. The double colons (`::`) in `devtools::install_github()` tell **R** to look in the **devtools** package for the `install_github()` function. This function is used to install a package from GitHub, which is a website to build and share software. Note that **R** is particular about quotation marks when installing and loading packages. Make sure to use quotes around the package name with the `install()` commands. You do not have to use quotes around the package name with `library()` (but you can).

```
install.packages("devtools") # install the package
library(devtools) # load the package
## install a package from GitHub
devtools::install_github("kosukeimai/qss-package",
                         build_vignettes = TRUE)
## you may need to allow R to update/install additional packages
## load the qss package
library("qss")
```

It is best practice to load the packages you will need at the start of your **R** session (or at the start of your **R** script). That way you can troubleshoot package installation at the start. Sometimes, installing a package will require you to install additional packages, often called *dependencies* (the other packages that the one you are installing *depends* on to run properly). You may also encounter difficulties depending on which version of **R** you have and which version of a package you are installing. The internet is a great resource for troubleshooting these issues. Your instructor or computing/statistics helpdesks may also be able to provide assistance. For now, practice by installing and loading the **tidyverse** family of packages.

```
library(tidyverse)
## if this command does not work, remember to install the package
```

1.3.4 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**, we will see the objects we created in the upper-right window called `Environment` (see figure 1.1). 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 once this is done 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. And so, 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, which will list all the current objects that exist in **R**.

So far, we have only assigned 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 by surrounding a number with quotation marks.

```
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 on an object depending on its class. For example, a number is stored as a *numeric* object whereas a character string is recognized

as a *character* object (sometimes also referred to as a *string*). 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 of objects in **R**, some of which will be introduced throughout this book. In fact, it is even possible to create our own object classes.

1.3.5 VECTORS

We present the simplest (but most inefficient) way of entering data into **R**. Table 1.1 contains estimates of the 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 with multiple values

Table 1.1. 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.

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
```

The `c()` function can be used to combine multiple vectors into one.

```
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, while the minus sign, `-`, removes the corresponding element from a vector. Note that none of these operations changes 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. Suppose that the population was 100 thousand in one year and increased to 120 thousand in the following year. In this case, we say that 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 from the decade before. For example, the first element is the difference between the 1960 population estimate and the 1950 estimate. We then divide each element in the vector of population increases by the population at the end of the decade to get the percentage increase.

```
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 (`<-`).

```
percent.increase[c(1, 2)] <- c(20, 22)
percent.increase

## [1] 20.00000 22.00000 20.53212 19.59448 15.16464 12.86752
```

1.3.6 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 length of a vector or equivalently the number of elements in the vector, `min()` for *minimum* value, `max()` for *maximum* value, `range()` for the *range* of the data (the highest and lowest values), `mean()` for the *mean* or *average*, and `sum()` for the *sum* of the data (adding all the values). Right now we are only inputting 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 above gives another way of calculating the mean: as the sum of all elements in a vector divided by the number of elements in that vector.

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 generates a vector composed of an increasing or decreasing sequence of numbers. 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 1.1 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 `seq()`. 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` vector 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 brackets `{ }` 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. It takes a vector as an input and returns the sum of all the numbers in the vector, the number of objects in the vector (the length), and the mean of the vector.

```
my.summary <- function(x){ # function takes one input, x
  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 # z is a vector from 1 to 10
my.summary(z) # run my.summary function on z

##    sum length   mean
##   55.0   10.0    5.5

my.summary(world.pop) # run my.summary function on 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.

1.3.7 DATA FILES: LOADING AND SUBSETTING

So far, we have only used data that we 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 Microsoft Excel or Google Sheets. 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 is the location on your computer where **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 picking the folder we want to work from. Alternatively, we can use the **RStudio** pull-down menu `Session > Set Working Directory > Choose Directory...` and then 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 will set the working directory to `qss/INTRO` and confirms the result (we suppress the output here).

```
getwd() # check what your current working directory is
setwd("qss/INTRO") # set your working directory with a path
getwd() # check that you changed your working directory
```

Suppose that the United Nations population data in table 1.1 are saved as a CSV file called `UNpop.csv` on our computer. The data resemble the example below. In tabular data like this, we often think of (and refer to) the columns as *variables* and the rows as *observations*. You will notice that we use the terms *column* and *variable* interchangeably, just as we use *row* and *observation* interchangeably.

```
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 dropdown menu in the upper-right `Environment` window (see figure 1.1) and clicking `Import Dataset > From Text File...`. Alternatively, we can use the `read_csv()` function from the `readr` package. The following syntax loads the data as a `tibble` object (more on this object below).

```
## if your working directory is where the .csv file is stored
UNpop <- read_csv("UNpop.csv")
class(UNpop) # what type of object is UNpop?
```

On the other hand, if the same data set is saved on our computer 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 with an RData file because the **R** objects stored in the file already have object names. Note that this code will overwrite the data set we loaded with `read_csv()` above because the objects have the same name. Although that is what we want in this case, there is a general lesson here that it is important to keep track of object names so that we do not accidentally overwrite objects.

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

Note that **R** can access any file on our computer if the full location is specified. The location of a file on your computer is called the *path*. 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 a lot of tedious typing (and typos).

Increasingly, researchers use **R** *projects* instead of changing the working directory at the start of each script. An **R** project tells **RStudio** which directory is home base. Then, instead of pointing **R** to look for files on your computer using *absolute* paths, such as `read_csv("Documents/qss/INTRO/UNpop.csv")`, you can instead use *relative* paths. For example, say your **R** project is in a directory called `QSS` on your computer, and you have a subdirectory in that folder called `INTRO`, where you are saving all the `.csv` files for this chapter. You could then use the following relative path in the `read_csv()` function to find `UNpop.csv`. To create an **R** project, you can use the `File > New Project` dropdown menu in **RStudio** or click on the project icon in the upper right.

```
## specifying a relative path to find and read in UNpop.csv
## will overwrite previously loaded UNpop object
UNpop <- read_csv("INTRO/UNpop.csv")
class(UNpop) # what type of object is UNpop?
## [1] "spec_tbl_df" "tbl_df"      "tbl"         "data.frame"
```

Having discussed multiple ways to read in data saved on your computer as a `.csv` or `.Rdata`, we can now make things even easier for the data used in this textbook. If you have successfully installed the `qss` package (see section 1.3.3), you can load in any of the referenced data with the following syntax and the `data()` command.

```
## load the package
library(qss)
## load the UN pop data
## will overwrite previously loaded UNpop object
data(UNpop, package = "qss")
```

Now that we've loaded `UNpop`, let's see what we have. We can think of a `data.frame` or `tibble` object as a spreadsheet. We can view a table-like representation of `data.frame` or `tibble` objects in **RStudio** by clicking on the object name in the `Environment` tab in the upper-right window (see figure 1.1). Alternatively, we can use the `View()` function with the object name as the input argument. This will open a new tab displaying the data. Useful functions for this object include the `names()` function to return a vector of variable names, the `nrow()` function to return the number of rows, the `ncol()` function to return the number of columns, and the `dim()` function to combine the outputs of `ncol()` and `nrow()` into a vector (also known as the *dimensions* of the data).

```
names(UNpop)
## [1] "year"      "world.pop"
nrow(UNpop)
## [1] 7
ncol(UNpop)
## [1] 2
dim(UNpop)
## [1] 7 2
```

There are several ways to access an individual variable. We begin with the base **R** syntax. First, one could use the `$` operator to extract a variable from a `data.frame` object, which returns a vector containing the specified variable.

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

Second, we could also use indexing `[]`, 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). Here are some syntax examples, which show how this indexing works.

```
## subset all rows for column called "world.pop" from UNpop data
UNpop[, "world.pop"]

## [1] 2525779 3026003 3691173 4449049 5320817 6127700 6916183

## subset the first three rows (and all columns)
UNpop[c(1, 2, 3),]

##   year world.pop
## 1 1950   2525779
## 2 1960   3026003
## 3 1970   3691173

## subset the first three rows of the "year" column
UNpop[1:3, "year"]

## [1] 1950 1960 1970
```

In the **tidyverse** syntax, extracting subsets of data looks a bit different. Instead of using lots of brackets, we will begin by using two functions: `slice()` and `select()`. The `slice()` function returns rows by number (or other criteria), while `select()` returns columns by name, number, or other criteria (more on this in a bit). The first input argument in `slice()` or `select()` is the data. The second input provides information about how to subset the data. Notice that the base **R** syntax (see code above) returns a vector, while the **tidyverse** syntax returns a `tibble`.

```
## subset the first three rows of UNpop with tidyverse
slice(UNpop, n = 1:3)

##   year world.pop
## 1 1950   2525779
## 2 1960   3026003
## 3 1970   3691173

## extract/subset the world.pop variable (column)
select(UNpop, world.pop)

##   world.pop
## 1   2525779
## 2   3026003
## 3   3691173
## 4   4449049
## 5   5320817
## 6   6127700
## 7   6916183
```

Let's say we wanted to subset the first three rows just for the variable `year`. We could do that in any of the following ways—notice that we can nest a `slice()` command

inside a `select()` command. **R** will select the `year` column and then slice the first three rows.

```
## base R subset the first three rows of the year variable
UNpop[1:3, "year"]

## [1] 1950 1960 1970

## or in tidyverse, combining slice() and select()
select(slice(UNpop, 1:3), year)

##   year
## 1 1950
## 2 1960
## 3 1970
```

Instead of nesting the `slice()` into `select()`, we could use the pipe operator `%>%` to link commands together. This tells **R** to do something and then do something else to the output of the first something. Chaining functions together like this will become very useful as our tasks become more complicated.

```
UNpop %>% # take the UNpop data we have loaded, and then...
  slice(1:3) %>% # subset the first three rows, and then...
  select(year) # subset the year column

##   year
## 1 1950
## 2 1960
## 3 1970
```

As another subsetting example, imagine that we want to extract every other row of the `world.pop` column from `UNpop` (i.e., we want rows 1, 3, 5, etc. for the `world.pop` variable). We could use an additional helper function, `n()`, which returns the number of rows in the `data.frame` or `tibble`.

```
UNpop %>%
  slice(seq(1, n(), by = 2)) %>% # using a sequence from 1 to n()
  select(world.pop)

##   world.pop
## 1   2525779
## 2   3691173
## 3   5320817
## 4   6916183
```

A final example of how to subset these specific rows and column uses the `filter()` function. `filter()` is to rows what `select()` is to columns—it subsets rows by name, order, or other criteria. These are two very commonly used functions in the **tidyverse**. To keep them straight, note that `select()` has a **c** in it (and therefore is for **c**olumns), while `filter()` has an **r** in it (and therefore is used for **r**ows). In the example below, `filter()` says to subset rows if their row number divided by 2 gives a remainder of 1. The `%%` operator returns the modulus, i.e., division remainder. The function `row_number()` returns the row number of an observation.

```
UNpop %>%
  filter(row_number() %% 2 == 1) %>%
  select(world.pop)
```

```
##   world.pop
## 1   2525779
## 2   3691173
## 3   5320817
## 4   6916183
```

The `filter(row_number() %% 2 == 1)` in the above code makes use of what is called a *conditional* or *logical* statement. We will discuss these more in depth in chapter 2. For now, think of these as "if" conditions, telling **R** to do something if a condition is met. The condition might be that something is equal to something else, such as the modulus being equal to 1 as in the example. The "equal to" condition is indicated in **R** with == (note that this is not the same as a single =). In the example, we are telling **R** to return the subset of rows where the modulus of dividing the row number by 2 is equal to 1 (in other words, returning the odd number observations 1, 3, 5, etc.).

For conditional statements, we can also use the "less than" (<), "less than or equal to" (<=), "greater than" (>), and "greater than or equal to" (>=) syntax. An exclamation point in a conditional indicates negation, so `!=` means "is not equal to."

The following code uses `filter()`, `select()`, and a conditional statement with the function `pull()` to extract a specific value from our data as a vector instead of a `tibble`. Let's say, for example, that we wanted to know what the world population was in 1970. We could use the following commands.

```
pop.1970 <- UNpop %>% # take the UNpop data and then...
  filter(year == 1970) %>% # subset rows where year is equal to 1970
  select(world.pop) %>% # subset just the world.pop column
  pull() # return a vector, not a tibble

## print the vector to the console to see it
print(pop.1970)
```

```
## [1] 3691173
```

1.3.8 ADDING VARIABLES

Suppose we wanted to take the population data and add an additional column based on a current column. For example, perhaps we want to have the world population in millions, instead of the raw figure in the original data. We can use the `mutate()` function to create that variable, which we call `world.pop.mill`, and add it to the `tibble`. We can then drop the original `world.pop` variable using the `select()` function with a – and the column name. In the example below, we also use `<-` to save a new version of the data that contains the new column. Note that if we put the same object name on both sides of the `<-`, that would overwrite the existing data. If you run the code below, you should have both `UNpop` and `UNpop.mill` in your `Environment`. You may want to look at the new data to confirm that the new variable is as you expect.

```
UNpop.mill <- UNpop %>% # create a new tibble from UNpop
  ## create new variable world.pop.mill
  mutate(world.pop.mill = world.pop / 1000) %>%
  select(-world.pop) # drop the original world.pop column
```

The `mutate()` function is a very useful command. We used it above to do an arithmetic operation on a column. We can also use it to combine columns based on our specifications, as in the example below. Let's say we wanted a variable that took the world population and divided it by the year (why we would want to do this is unclear, but let's go with it for now). The following code shows how we could do that by using the column names.

```
## adding a nonsense variable to the UNpop.mill data
UNpop.mill <- UNpop.mill %>%
  mutate(nonsense.var = world.pop.mill / year)
```

We can combine the `mutate()` function with conditional statements in useful ways by using the function `if_else()`. The function tells **R** to do something **if** a conditional statement is met and to do something **else** if the statement is not met. Say we wanted a new variable that indicates whether or not a row contains data from after 1980. We'll call this new variable `after.1980`. We want this variable to have two possible values: `1` if the row is from after 1980 and `0` if it's not. In the example below, we use `mutate()` to tell **R** that we want a new variable called `after.1980`. The value that each row will have for this new column is set by the `if_else()` function. The conditional statement is the first argument. It tells **R** to check the `year` column for a given row. If `year` is greater than or equal to 1980, then the value for the variable `after.1980` should be `1`. If `year` is not greater than or equal to 1980, then the value for the variable `after.1980` should be `0`. It is usually a good idea to check that your new variable looks the way you expect.

```
## adding a variable with if_else
UNpop.mill <- UNpop.mill %>%
  mutate(after.1980 = if_else(year >= 1980, 1, 0))
```

A final example with `mutate()` and `if_else()` uses a very helpful conditional symbol: `%in%`. We can follow `%in%` with a vector of values. **R** will then check whether a specific value matches something in that vector. For example, let's say that we also wanted to add a variable noting whether a row was from the following specific set of years (imagine that these years are of particular interest to us): 1950, 1980, and 2000. In the following code, we first create a vector of those years, then we reference it within `if_else()` to create a new variable, `years.of.interest`.

```
## creating a vector of the years of interest
specific.years <- c(1950, 1980, 2000)

## adding a variable with if_else and %in%
UNpop.mill <- UNpop.mill %>%
  mutate(year.of.interest = if_else(year %in% specific.years, 1, 0))
```

1.3.9 DATA FRAMES: SUMMARIZING

Having loaded our data and created some new variables, we now turn to some ways to summarize the data. The `summary()` function is useful for this. The `summary()` function yields, for each variable in the input `data.frame` or `tibble` 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. We can also use functions like `mean()` to compute summary values for specific variables in the data. See section 3.3 for more discussion.

```
summary(UNpop.mill)

##       year      world.pop.mill  nonsense.var
##  Min.   :1950   Min.   :2526   Min.   :1.295
##  1st Qu.:1965   1st Qu.:3359   1st Qu.:1.709
##  Median :1980   Median :4449   Median :2.247
##  Mean   :1980   Mean   :4580   Mean   :2.305
##  3rd Qu.:1995   3rd Qu.:5724   3rd Qu.:2.869
##  Max.   :2010   Max.   :6916   Max.   :3.441
##    after.1980     year.of.interest
##  Min.   :0.0000   Min.   :0.0000
##  1st Qu.:0.0000   1st Qu.:0.0000
##  Median :1.0000   Median :0.0000
##  Mean   :0.5714   Mean   :0.4286
##  3rd Qu.:1.0000   3rd Qu.:1.0000
##  Max.   :1.0000   Max.   :1.0000

mean(UNpop.mill$world.pop.mill)

## [1] 4579.529
```

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. Below we use the `add_row()` function to add a row with `NA` values to demonstrate the issue that arises with `mean()` and how to fix it.

```
## add a row where value for each column is NA
UNpop.mill.wNAs <- UNpop.mill %>%
  add_row(year = NA, world.pop.mill = NA,
          nonsense.var = NA, after.1980 = NA,
          year.of.interest = NA)
## take the mean of world.pop.mill (returns NA)
mean(UNpop.mill.wNAs$world.pop.mill)

## [1] NA

## take the mean of world.pop.mill (ignores the NA)
mean(UNpop.mill.wNAs$world.pop.mill, na.rm = TRUE)

## [1] 4579.529
```

The **tidyverse** offers a useful way to generate summaries with the `summarize()` function (or `summarise()`, both spellings are accepted). With this function, you can specify multiple functions to apply to variables within a data set, returning the results as new columns in a `tibble`. The example below returns both the median and the mean of `world.pop.mill`.

```
UNpop.mill %>%
  summarize(mean.pop = mean(world.pop.mill),
            median.pop = median(world.pop.mill))

##   mean.pop median.pop
## 1 4579.529   4449.049
```

What if we wanted to know the average (mean) world population but not for the full time period? For example, we might want to know what the average population was before 1980 and after 1980. To do this, we can combine the `summarize()` function with the `group_by()` function, which tells **R** to treat subsets of the data separately. We need to tell **R** which variable to use to group the rows. We can use the variable we created earlier, `after.1980`, for this purpose.

```
UNpop.mill %>%
  ## create subset group for each value of after.1980
  group_by(after.1980) %>%
  ## calculate mean for each group
  summarize(mean.pop = mean(world.pop.mill))
```

```
## # A tibble: 2 x 2
##   after.1980 mean.pop
##        <dbl>    <dbl>
## 1          0    3081.
## 2          1    5703.
```

1.3.10 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 in 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 explicitly 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 or select `Session > Load Workspace...`.

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 to save the workspace. 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")
```

Generally speaking, however, it is not recommended to save the workspace. Best practice is to turn off the **RStudio** prompt to save your workspace by going to `Tools > Global Options > General` and setting "Save workspace to .RData on exit" to "Never" (while you have the `Global Options` open, note that this is also where you can change the appearance of **RStudio**—changing to a different color scheme with a nonwhite background can be easier on the eyes.)

It is better to save your **R** source code (e.g., scripts), which you can rerun to reproduce your results. Sometimes we wish to save a specific object (e.g., a `data.frame` or `tibble` 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`

(continued...)

General Index

R Index

D

E

F

G

H

I