5

Introduction to R

R is an extraordinarily powerful open source software program built for working with data. It is one of the most popular data science tools because of its ability to efficiently perform statistical analysis, implement machine learning algorithms, and create data visualizations. R is the primary programming language used throughout this book, and understanding its foundational operations is key to being able to perform more complex tasks.

5.1 Programming with R

R is a **statistical programming language** that allows you to write code to work with data. It is an **open source** programming language, which means that it is free and continually improved upon by the R community. The R language has a number of capabilities that allow you to read, analyze, and visualize data sets.

> **Fun Fact:** R is called "R" in part because it was inspired by the language "S," a language for Statistics developed by AT&T, and because it was developed by Ross Ihaka and Robert Gentleman.

In previous chapters, you leveraged formal language to give instructions to your computer, such as by writing syntactically precise instructions at the command line. Programming in R works in a similar manner: you write instructions using R's special language and syntax, which the computer **interprets** as instructions for how to work with data.

However, as projects grow in complexity, it becomes useful if you can write down all the instructions in a single place, and then order the computer to *execute* all of those instructions at once. This list of instructions is called a **script**. Executing or "running" a script will cause each instruction (line of code) to be run *in order, one after the other*, just as if you had typed them in one by one. Writing scripts allows you to save, share, and reuse your work. By saving instructions in a file (or set of files), you can easily check, change, and re-execute the list of instructions as you figure out how to use data to answer questions. And, because R is an *interpreted* language, rather than a *compiled* language like C or Java, R programming environments give you the ability to separately execute each individual line of code in your script if you desire.

As you begin working with data in R, you will be writing multiple instructions (lines of code) and saving them in files with the **.R** extension, representing R scripts. You can write this R code in any text editor (such as Atom), but we recommend you usually use **RStudio**, a program that is specialized for writing and running R scripts.

5.2 Running R Code

There are a few different ways in which you can have your computer execute code that you write in the R language. The most user-friendly approach is to use RStudio.

5.2.1 Using RStudio

RStudio is an open source **integrated development environment (IDE)** that provides an informative user interface for interacting with the R interpreter. Generally speaking, IDEs provide a platform for writing *and* executing code, including viewing the results of the code you have run. This is distinct from a code *editor* (like Atom), which is used just to *write* code.

When you open the RStudio program, you will see an interface similar to that in Figure 5.1. An RStudio session usually involves four sections ("panes"), though you can customize this layout if you wish:

- **Script**: The top-left pane is a simple text editor for writing your R code as different script files. While it is not as robust as a text editing program like Atom, it will colorize code, auto-complete text, and allow you to easily execute your code. Note that this pane is hidden if there are no open scripts; select `File > New File > R Script` from the menu to create a new script file.

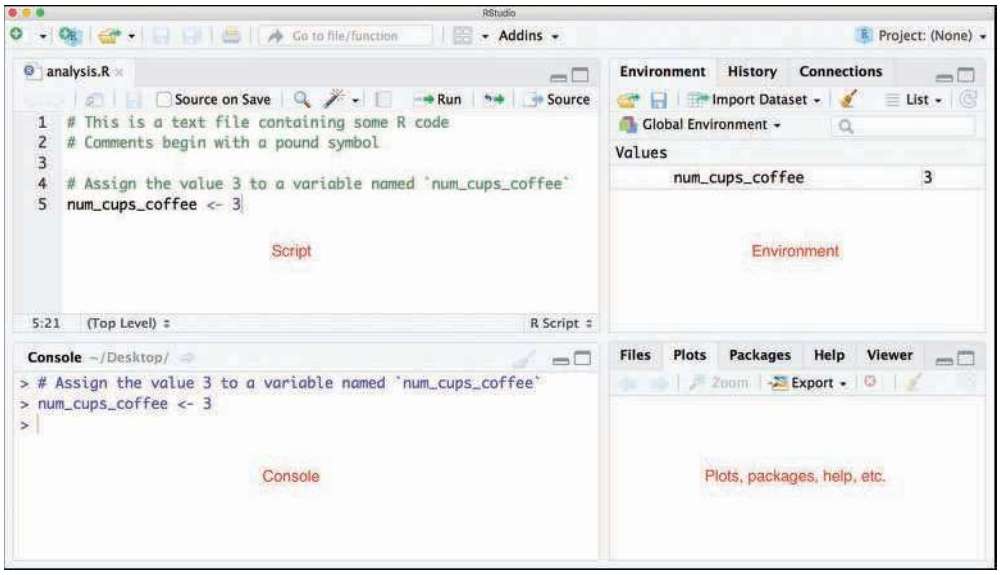

Figure 5.1 RStudio's user interface, showing a script file. Red notes are added.

To execute (run) the code you write, you have two options:

1. You can execute a section of your script by selecting (highlighting) the desired code and clicking the "Run" button (or use the keyboard shortcut[1]: `cmd+enter` on Mac, or `ctrl+enter` on Windows). If no lines are selected, this will run the line currently containing the cursor. This is the most common way to execute code in RStudio.

   > **Tip**: Use `cmd+a` (Mac) or `ctrl+a` (Windows) to select the entire script!

2. You can execute an entire script by clicking the "Source" button (at the top right of the Script pane, or via `shift+cmd+enter`) to execute all lines of code in the script file, one at a time, from top to bottom. This command will treat the current script file as the "source" of code to run. If you check the "Source on Save" option, your entire script will be executed every time you save the file (which may or may not be appropriate, depending on the complexity of your script and its output). You can also hover your mouse over this or any other button to see keyboard shortcuts.

   > **Fun Fact**: The Source button actually calls an `R` function called `source()`, described in Chapter 14.

- **Console**: The bottom-left pane is a console for entering `R` commands. This is identical to an interactive session you would run on the command line, in which you can type and execute one line of code at a time. The console will also show the printed results of executing the code from the Script pane. If you want to perform a task *once*, but don't want to save that task in your script, simply type it in the console and press `enter`.

  > **Tip**: Just as with the command line, you can use the *up arrow* to easily access previously executed lines of code.

- **Environment**: The top-right pane displays information about the current `R` environment—specifically, information that you have stored inside of *variables*. In Figure 5.1 the value 3 is stored in a variable called `num_cups_coffee`. You will often create dozens of variables within a script, and the Environment pane helps you keep track of which values you have stored in which variables. *This is incredibly useful for "debugging" (identifying and fixing errors)!*

- **Plots, packages, help, etc.**: The bottom-right pane contains multiple tabs for accessing a variety of information about your program. When you create visualizations, those plots will be rendered in this section. You can also see which packages you have loaded or look up information about files. Most importantly, you can access the official documentation for the `R` language in this pane. If you ever have a question about how something in `R` works, this is a good place to start!

[1] **RStudio Keyboard Shortcuts**: https://support.rstudio.com/hc/en-us/articles/200711853-Keyboard-Shortcuts

Note that you can use the small spaces between the quadrants to adjust the size of each area to your liking. You can also use menu options to reorganize the panes.

> **Tip**: RStudio provides a built-in link to a "Cheatsheet" for the IDE—as well as for other packages described in this text—through the `Help > Cheatsheets` menu.

5.2.2 Running R from the Command Line

While RStudio is the interface that we suggest for running R code, you may find that in certain situations you need to execute some code without the IDE. It is possible to issue R instructions (run lines of code) one by one at the command line by starting an **interactive R session** within your command shell. This will allow you to type R code directly into the terminal, and your computer will interpret and execute each line of code (if you just typed R syntax directly into the terminal, your computer wouldn't understand it).

With the R software installed, you can start an interactive R session on a Mac by typing R (or lowercase r) into the Terminal to run the R program. This will start the session and provide you with some information about the R language, as shown in Figure 5.2.

Notice that this description also includes *instructions on what to do next*—most importantly, `"Type 'q()' to quit R."`

> **Remember**: Always read the output carefully when working on the command line!

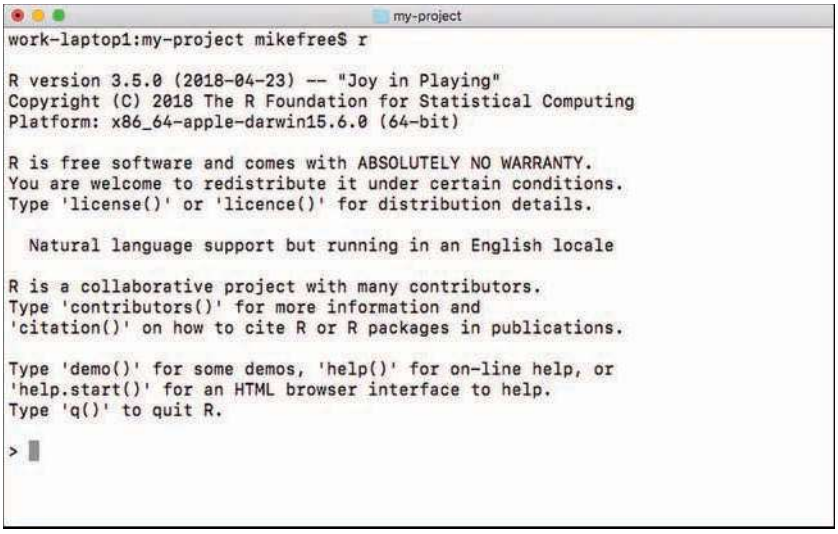

Figure 5.2 An interactive R session running in a command shell.

Once you've started running an interactive R session, you can begin entering one line of code at a time at the prompt (>). This is a nice way to experiment with the R language or to quickly run some code. For example, you can try doing some math at the command prompt (e.g., enter `1 + 1` and see the output).

It is also possible to run entire scripts from the command line by using the `RScript` program, specifying the `.R` file you wish to execute, as shown in Figure 5.3. Entering the command shown in Figure 5.3 in the terminal would execute each line of R code written in the `analysis.R` file, performing all of the instructions that you had saved there. This is helpful if your data has changed, and you want to recalculate the results of your analysis using the same instructions.

On Windows (and some other operating systems), you may need to tell the computer where to find the R and `RScript` programs to execute—that is, the **path** to these programs. You can do this by specifying the *absolute path* to the `R.exe` program when you execute it, as in Figure 5.3.

> **Going Further**: If you use Windows and plan to run R from the command line regularly (which is *not* required or even suggested in this book), a better solution is to add the folder containing these programs to your computer's **PATH variable**. This *system-level* variable contains a list of folders that the computer searches when finding programs to execute. The reason the computer knows where to find the `git.exe` program when you type `git` in the command line is because that program is "on the PATH."
>
> In Windows, you can add the `R.exe` and `RScript.exe` programs to your computer's PATH by editing your machine's **environment variables** through the *Control Panel*.[a] Overall, using R from the command line can be tricky; we recommend you just use RStudio instead as you're starting out.
>
> [a] https://helpdeskgeek.com/windows-10/add-windows-path-environment-variable/

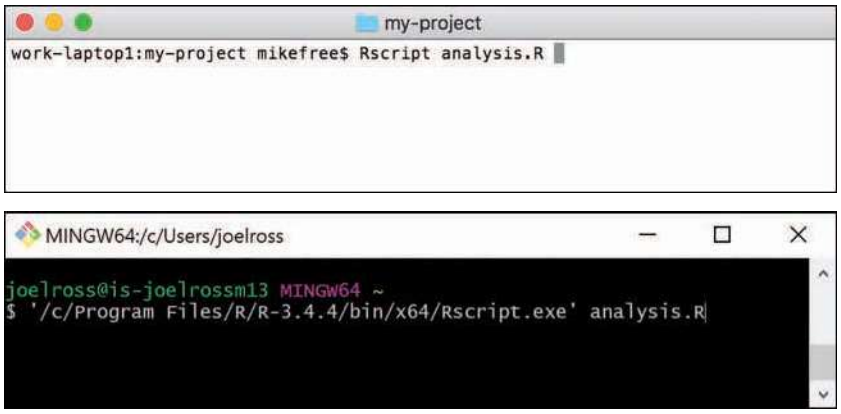

Figure 5.3 Using the `RScript` command to run an R script from a command shell: Mac (top) and Windows (bottom).

Caution: On Windows, the R interpreter download also installs an "RGui" application (e.g., "R x64 3.4.4"), which will likely be the default program for opening `.R` scripts. Make sure to use the RStudio IDE for working in R!

5.3 Including Comments

Before discussing how to write programs with R, it's important to understand the syntax that lets you add comments your code. Since computer code can be opaque and difficult to understand, developers use comments to help write down the meaning and purpose of their code. This is particularly important when someone else will be looking at your work—whether that person is a collaborator or simply a future version of you (e.g., when you need to come back and fix something and so need to remember what you were trying to do).

Comments should be clear, concise, and helpful. They should provide information that is not otherwise present or "obvious" in the code itself.

In R, you mark text as a comment by putting it after the pound symbol (**#**). Everything from the # until the end of the line is a comment. You put descriptive comments *immediately above* the code they describe, but you can also put very short notes at the end of the line of code, as in the following example (note that the R code syntax used is described in the following section):

```
# Calculate the number of minutes in a year
minutes_in_a_year <- 365 * 24 * 60 # 525,600 minutes!
```

(You may recognize this # syntax and commenting behavior from command line examples in previous chapters—because the same syntax is used in a Bash shell!)

5.4 Defining Variables

Since computer programs involve working with lots of information, you need a way to store and refer to that information. You do this with **variables**. Variables are labels for information; in R, you can think of them as "boxes" or "name tags" for data. After putting data in a variable box, you can then refer to that data by the label on the box.

In the R language, variable names can contain any combination of letters, numbers, periods (`.`), or underscores (_), though they must begin with a letter. Like almost everything in programming, variable names are **case sensitive**. It is best practice to make variable names descriptive and informative about what data they contain. For example, `x` is not a good variable name, whereas `num_cups_coffee` is a good variable name. Throughout this book, we use the formatting suggested in *the tidyverse style guide*.[2] As such, variable names should be all lowercase letters, separated by underscores (_). This is also known as `snake_case`.

[2] **Tidyverse style guide**: http://style.tidyverse.org

> **Remember**: There is an important distinction between *syntax* and *style*. The syntax of a language describes the rules for writing the code so that a computer can interpret it. Certain operations are permitted, and others are not. Conversely, styles are optional conventions that make it easier for other humans to interpret your code. The use of a *style guide* allows you to describe the conventions you will follow in your code to help keep things like variable names consistent.

Storing information in a variable is referred to as **assigning** a **value** to the variable. You assign a value to a variable using the *assignment operator* **`<-`**. For example:

```
# Assign the value 3 to a variable named `num_cups_coffee`
num_cups_coffee <- 3
```

Notice that the variable name goes on the left, and the value goes on the right.

You can see which value (data) is "inside" a variable by either executing that variable name as a line of code or by using R's built-in **`print()`** function (functions are detailed in Chapter 6):

```
# Print the value assigned to the variable `num_cups_coffee`
print(num_cups_coffee)
# [1] 3
```

The `print()` function prints out the value (3) stored in the variable (`num_cups_coffee`). The `[1]` in that output indicates that the first element stored in the variable is the number 3—this is discussed in detail in Chapter 7.

You can also use **mathematical operators** (e.g., `+`, `-`, `/`, `*`) when assigning values to variables. For example, you could create a variable that is the sum of two numbers as follows:

```
# Use the plus (+) operator to add numbers, assigning the result to a variable
too_much_coffee <- 3 + 4
```

Once a value (like a number) is in a variable, you can use that variable in place of any other value. So all of the following statements are valid:

```
# Calculate the money spent on coffee using values stored in variables
num_cups_coffee <- 3 # store 3 in `num_cups_coffee`
coffee_price <- 3.5 # store 3.5 in `coffee_price`
money_spent_on_coffee <- num_cups_coffee * coffee_price # total spent on coffee
print(money_spent_on_coffee)
# [1] 10.5

# Alternatively, you can use a mixture of numeric values and variables
# Calculate the money spent on 4 cups of coffee
money_spent_on_four_cups <- coffee_price * 4 # total spent on 4 cups of coffee
print(money_spent_on_four_cups)
# [1] 14
```

In many ways, script files are just note pads where you've jotted down the R code you wish to run. Lines of code can be (and often are) executed out of order, particularly when you want to change or fix a previous statement. When you do change a previous line of code, you will need to ***re-execute*** that line of code to have it take effect, as well as re-execute any subsequent lines if you want them to use the updated value.

As an example, if you had the following code in your script file:

```
# Calculate the amount of caffeine consumed using values stored in variables
num_cups_coffee <- 3 # line 1
cups_of_tea <- 2 # line 2
caffeine_level <- num_cups_coffee + cups_of_tea # line 3
print(caffeine_level) # line 4
# [1] 5
```

Executing all of the lines of code one after another would assign the variables and print a value `5`. If you edited line 1 to say `num_cups_coffee <- 4`, the computer wouldn't do anything different until you re-executed the line (by selecting it and pressing `cmd+enter`). And re-executing line 1 wouldn't cause another new value to be printed, since that command occurs at line 4! If you then re-executed line 4 (by selecting that line and pressing `cmd+enter`), it would still print out `5`—because you haven't told R to recalculate the value of `caffeine_level`! You would need to re-execute *all* of the lines of code (e.g., select them *all* and pressing `cmd+enter`) to have your script print out the desired (new) value of `6`. This kind of behavior is common for computer programming languages (though different from environments like Excel, where values are automatically updated when you change other referenced cells).

5.4.1 Basic Data Types

The preceding examples show the storage of numeric values in variables. R is a **dynamically typed language**, which means that you do not need to explicitly state which type of information will be stored in each variable you create. R is intelligent enough to understand that if you have code `num_cups_coffee <- 3`, then `num_cups_coffee` will contain a numeric value (and thus you can do math with it).

> **Going Further**: In **statically typed languages**, you need to declare the *type* of variable you want to create. For example, in the Java programming language (which is not used in this text), you have to indicate the type of variable you want to create: if you want the *integer* `10` to be stored in the variable `my_num`, you would have to write `int my_num = 10` (where `int` indicates that `my_num` will be an integer).

There are a few "basic types" (or *modes*) for data in R:

- **Numeric**: The default computational data type in R is numeric data, which consists of the set of real numbers (including decimals). You can use **mathematical operators** on numeric data (such as `+`, `-`, `*`, `-`, etc.). There are also numerous functions that work on numeric data (such as for calculating sums or averages).

Note that you can use multiple operators in a single **expression**. As in algebra, parentheses can be used to enforce order of operations:

```
# Calculate the number of minutes in a year
minutes_in_a_year <- 365 * 24 * 60

# Enforcing order of operations with parentheses
# Calculate the number of minutes in a leap year
minutes_in_a_leap_year <- (365 + 1) * 24 * 60
```

- **Character**: Character data stores *strings* of characters (e.g., letters, special characters, numbers) in a variable. You specify that information is character data by surrounding it with either single quotes (`'`) or double quotes (`"`); the tidyverse style guide suggests always using double quotes.

```
# Create character variable `famous_writer` with the value "Octavia Butler"
famous_writer <- "Octavia Butler"
```

Note that character data is still data, so it can be assigned to a variable just like numeric data.

There are no special operators for character data, though there are a many built-in functions for working with strings.

> **Caution**: If you see a plus sign (`+`) in the terminal as opposed to the typical greater than symbol (`>`)—as in Figure 5.4—you have probably forgotten to close a quotation mark. If you find yourself in this situation, you can press the `esc` key to cancel the line of code and start over. This will also work if you forget to close a set of parentheses (`( )`) or brackets (`[ ]`).

- **Logical**: Logical (boolean) data types store "yes-or-no" data. A logical value can be one of two values: `TRUE` or `FALSE`. Importantly, these *are not* the strings `"TRUE"` or `"FALSE"`; logical values are a different type! If you prefer, you can use the shorthand `T` or `F` in lieu of `TRUE` and `FALSE` in variable assignment.

> **Fun Fact**: Logical values are called "booleans" after mathematician and logician George Boole.

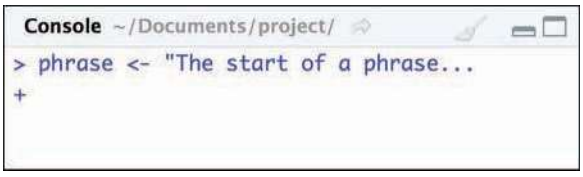

Figure 5.4 An unclosed statement in the RStudio console: press the `esc` key to cancel the statement and return to the command prompt.

Logical values are most commonly produced by applying a **relational operator** (also called a **comparison operator**) to some other data. Comparison operators are used to compare values and include < (less than), > (greater than), <= (less than or equal), >= (greater than or equal), == (equal), and != (not equal). Here are a few examples:

```
# Store values in variables (number of strings on an instrument)
num_guitar_strings <- 6
num_mandolin_strings <- 8

# Compare the number of strings on each instrument
num_guitar_strings > num_mandolin_strings # returns logical value FALSE
num_guitar_strings != num_mandolin_strings # returns logical value TRUE

# Equivalently, you can compare values that are not stored in variables
6 == 8 # returns logical value FALSE

# Use relational operators to compare two strings
"mandolin" > "guitar" # returns TRUE (m comes after g alphabetically)
```

If you want to write a more complex logical expression (i.e., for when something is true **and** something else is false), you can do so using **logical operators** (also called **boolean operators**). These include & (and), | (or), and ! (not).

```
# Store the number of instrument players in a hypothetical band
num_guitar_players <- 3
num_mandolin_players <- 2

# Calculate the number of band members
total_band_members <- num_guitar_players + num_mandolin_players # 5

# Calculate the total number of strings in the band
# Shown on two lines for readability, which is still valid R code
total_strings <- num_guitar_players * num_guitar_strings +
  num_mandolin_strings * num_mandolin_players # 34

# Are there fewer than 30 total strings AND fewer than 6 band members?
total_strings < 30 & total_band_members < 6 # FALSE

# Are there fewer than 30 total strings OR fewer than 6 band members?
total_strings < 30 | total_band_members < 6 # TRUE

# Are there 3 guitar players AND NOT 3 mandolin players?
# Each expression is wrapped in parentheses for increased clarity
(num_guitar_players == 3) & !(num_mandolin_players == 3) # TRUE
```

It's easy to write complex—even overly complex—expressions with logical operators. If you find yourself getting lost in your logic, we recommend rethinking your question to see if there is a simpler way to express it!

- **Integer**: Integer (whole-number) values are technically a different data type than numeric values because of how they are stored and manipulated by the `R` interpreter. This is something that you will rarely encounter, but it's good to know that you can specify that a number is of the integer type rather than the general numeric type by placing a capital `L` (for "long integer") after a value in variable assignment (`my_integer <- 10L`). You will rarely do this intentionally, but this is helpful for answering the question, *Why is there an L after my number...?*
- **Complex**: Complex (imaginary) numbers have their own data storage type in `R`, and are created by placing an `i` after the number: `complex_variable <- 2i`. We will not be using complex numbers in this book, as they rarely are important for data science.

5.5 Getting Help

As with any programming language, you will inevitably run into problems, confusing situations, or just general questions when working in `R`. Here are a few ways to start getting help.

1. **Read the error messages**: If there is an issue with the way you have written or executed your code, `R` will often print out an error message in your console (in red in RStudio). Do your best to decipher the message—read it carefully, and think about what is meant by each word in the message—or you can put that message directly into Google to search for more information. You will soon get the hang of interpreting these messages if you put the time into trying to understand them. For example, Figure 5.5 shows the result of accidentally mistyping a variable name. In that error message, `R` indicated that the object `cty` was not found. This makes sense, because the code never defined a variable `cty` (the variable was called `city`).
2. **Google**: When you're trying to figure out how to do something, it should come as no surprise that search engines such as Google are often the best resource. Try searching for queries like `"how to DO_THING in R"`. More frequently than not, your question will lead you to a Q&A forum called StackOverflow (discussed next), which is a great place to find potential answers.

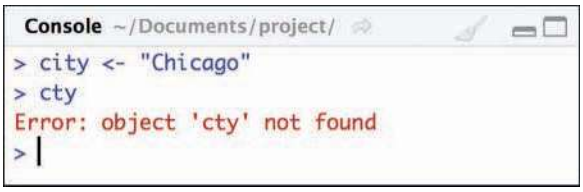

Figure 5.5 RStudio showing an error message due to a typo (there is no variable cty).

3. **StackOverflow**: StackOverflow is an amazing Q&A forum for asking/answering programming questions. Indeed, most basic questions have already been asked and answered there. However, don't hesitate to post your own questions to StackOverflow. Be sure to hone in on the specific question you're trying to answer, and provide error messages and sample code. You will often find that by the time you can articulate the question clearly enough to post it, you will have figured out your problem anyway.

   > **Tip**: There is a classical method of fixing errors called *rubber duck debugging*, which involves trying to explain your code/problem to an inanimate object (talking to pets works too). You will usually be able to fix the problem if you just step back and think about how you would explain it to someone else!

4. **Built-in documentation**: R's documentation is actually pretty good. Functions and behaviors are all described in the same format, and often contain helpful examples. To search the documentation within R (or in RStudio), type a question mark (`?`) followed by the function name you're using (e.g, `?sum`). You can perform a broader search of available documentation by typing two questions marks (`??`) followed by your search term (e.g., `??sum`).

   You can also look up help by using the `help()` function (e.g., `help(print)` will look up information on the `print()` function, just as `?print` does). There is also an `example()` function you can call to see examples of a function in action (e.g., `example(print)`). This will be more applicable starting in Chapter 6.

   In addition, *RDocumentation.org*[3] has a lovely searchable and readable interface to the R documentation.

5. **RStudio Community**: RStudio recently launched an online community[4] for R users. The intention is to build a more positive online community for getting programming help with R and engaging with the open source community using the software.

5.5.1 Learning to Learn R

This chapter has demonstrated the basics of the R programming language, and further features are detailed through the rest of the book. However, it's not possible to cover *all* features of a particular programming language—not to mention its surrounding ecosystem, such as the other frameworks used in data science—especially in a way that is accessible to those who are just getting started. While we will cover all of the material that you need to get started and ask questions of data using code, you will most certainly encounter problems in the future that aren't discussed in this text. *Doing* data science will require continuously learning new skills and techniques that are more advanced, more specific to your problem, or simply hadn't been invented when this book was written!

[3] **RDocumentation.org**: https://www.rdocumentation.org
[4] **RStudio Community**: https://community.rstudio.com

Luckily, you're not alone in this process! There is a huge number of resources that you can use to help you learn R or any other topic in programming or data science. This section provides an overview and examples of the types of resources you might use.

- **Books:** Many excellent text resources are available both in print and for free online. Books can provide a comprehensive overview of a topic, usually with a large number of examples and links to even more resources. We typically recommend them for beginners, as they help to cover all of the myriad steps involved in programming and their extensive examples help inform good programming habits. Free online books are easily accessible (and allow you to copy-and-paste code examples), but physical print can provide a useful point of reference (and typing out examples is a great way to practice).

  For learning R in particular, *R for Data Science*[5] is one of the best free online textbooks, covering the programming language through the lens of the `tidyverse` collection of packages (which are used in this book as well). Excellent print books include *R for Everyone*[6] and *The Art of R Programming*.[7]

- **Tutorials and videos:** The internet is also host to a large number of more informal explanations of programming concepts. These range from mini-books (such as the opinionated but clear introduction *aRrgh: a newcomer's (angry) guide to R*[8]), to tutorial series (such as those provided by *R Tutor*[9] or *Quick-R*[10]), to focused articles and guides (e.g., posts on *R-bloggers*[11]), to particularly informative StackOverflow responses. These smaller guides are particularly useful when you're trying to answer a specific question or clarify a single concept—when you want to know how to do one thing, not necessarily understand the entire language. In addition, many people have created and shared online video tutorials (such as Pearson's *LiveLessons*[12]), often in support of a course or textbook. Video code blogging is even more common in other programming languages such as JavaScript. Video demonstrations are great at showing you how to actually use a programming concept in practice—you can see all the steps that go into a program (though there is no substitute for doing it yourself).

  Because such guides can be created and hosted by anyone, the quality and accuracy may vary. It's always a good idea to confirm your understanding of a concept with multiple sources (do multiple tutorials agree?), with your own experience (does the solution actually work for your code?), and your own intuition (does that seem like a sensible explanation?). In general, we encourage you to start with more popular or official guides, as they are more likely to encourage best practices.

- **Interactive tutorials and courses:** The best way to learn any skill is by doing it, and there are multiple interactive websites that will let you learn and practice programming right in your web browser. These are great for seeing topics in action or for experimenting with different

[5] Wickham, H., & Grolemund, G. (2016). *R for Data Science*. O'Reilly Media, Inc. http://r4ds.had.co.nz
[6] Lander, J. P. (2017). *R for Everyone: Advanced Analytics and Graphics* (2nd ed.). Boston, MA: Addison-Wesley.
[7] Matloff, N. (2011). *The Art of R Programming: A Tour of Statistical Software Design*. San Francisco, CA: No Starch Press.
[8] **aRrgh: a newcomer's (angry) guide to R**: http://arrgh.tim-smith.us
[9] **R Tutor**: http://www.r-tutor.com/; start with the introduction at http://www.r-tutor.com/r-introduction
[10] **Quick-R**: https://www.statmethods.net/index.html; be sure and follow the hyperlinks.
[11] **R-Bloggers**: https://www.r-bloggers.com
[12] **LiveLessons** video tutorials: https://www.youtube.com/user/livelessons

options (though it is simple enough to experiment inside of RStudio—an approach taken by the *swirl*[13] package).

The most popular set of interactive tutorials for R programming are provided by *DataCamp*[14] and are presented as online courses (a sequence of explanations and exercises that you can learn to use a skill) on different topics. DataCamp tutorials provide videos and interactive tutorials for a wide range of different data science topics. While most of the introductory courses (e.g., *Introduction to R*[15]) are free, more advanced courses require you to sign up and pay for the service. Nevertheless, even at the free level, this is an effective set of resources for picking up new skills.

In addition to these informal interactive courses, it is possible to find more formal online courses in R and data science through massive open online course (MOOC) services such as *Coursera*[16] or *Udacity*.[17] For example, the *Data Science at Scale*[18] course from the University of Washington offers a deep introduction to data science (though it assumes some programming experience, so it may be more appropriate for *after* you've finished this book!). Note that these online courses almost always require a paid fee, though you can sometimes earn university credit or certifications from them.

- **Documentation:** One of the best places to start out when learning a programming concept is the official documentation. In addition to the base R documentation described in the previous section, many system creators will produce useful "getting started" guides and references—called "vignettes" in the R community—that you can use (to encourage adoption of their tool). For example, the `dplyr` package (described in great detail in Chapter 11) has an official "getting started" summary on its homepage[19] as well as a complete reference.[20] Further detail on a package may also often be found linked from that package's homepage on GitHub (where the documentation can be kept under version control); checking the GitHub page for a package or library is often an effective way to gain more information about it. Additionally, many R packages host their documentation in `.pdf` format on CRAN's website; to learn to use a package, you will need to read its explanation carefully and try out its examples!
- **Community resources**: As R is an open source language, many of the R resources described here are created by the community of programmers—and this community can be one of the best resources for learning to program. In addition to community-generated tutorials and answers to questions, in-person meet-ups can be an excellent source for getting help (particularly in larger urban areas). Check whether your city or town has a local "useR" group that may host events or training sessions.

[13] **swirl** interactive tutorial: http://swirlstats.com
[14] **DataCamp**: https://www.datacamp.com/home
[15] **DataCamp**: **Introduction to R**: https://www.datacamp.com/courses/free-introduction-to-r
[16] **Coursera**: https://www.coursera.org
[17] **Udacity**: https://www.udacity.com
[18] **Data Science at Scale**: online course from the University of Washington: https://www.coursera.org/specializations/data-science
[19] **`dplyr`** homepage: https://dplyr.tidyverse.org
[20] **`dplyr`** reference: https://dplyr.tidyverse.org/reference/index.html

This section lists only a few of the many, many resources for learning R. You can find many more online resources on similar topics by searching for "TOPIC tutorial" or "how to DO_SOMETHING in R." You may also find other compilations of resources. For example, RStudio has put together a list[21] of its recommended tutorials and resources.

In the end, remember that the best way to learn about anything—whether about programming or from a set of data—is to *ask questions*. For practice writing code in R and familiarizing yourself with RStudio, see the set of accompanying book exercises.[22]

[21] **RStudio: Online Learning** resource collection: https://www.rstudio.com/online-learning/
[22] **Introductory R exercises**: https://github.com/programming-for-data-science/chapter-05-exercises

This page intentionally left blank

6

Functions

As you begin to take on data science projects, you will find that the tasks you perform will involve multiple different instructions (lines of code). Moreover, you will often want to be able to repeat these tasks (both within and across projects). For example, there are many steps involved in computing summary statistics for some data, and you may want to repeat this analysis for different variables in a data set or perform the same type of analysis across two different data sets. Planning out and writing your code will be notably easier if can you group together the lines of code associated with each overarching task into a single step.

Functions represent a way for you to add a label to a group of instructions. Thinking about the tasks you need to perform (rather than the individual lines of code you need to write) provides a useful *abstraction* in the way you think about your programming. It will help you hide the details and generalize your work, allowing you to better reason about it. Instead of thinking about the many lines of code involved in each task, you can think about the task itself (e.g., `compute_summary_stats()`). In addition to helping you better reason about your code, labeling groups of instructions will allow you to save time by reusing your code in different contexts—repeating the task without rewriting the individual instructions.

This chapter explores how to use functions in R to perform advanced capabilities and create code that is flexible for analyzing multiple data sets. After considering a function in a general sense, it discusses using built-in R functions, accessing additional functions by loading R packages, and writing your own functions.

6.1 What Is a Function?

In a broad sense, a **function** is a named sequence of instructions (lines of code) that you may want to perform one or more times throughout a program. Functions provide a way of *encapsulating* multiple instructions into a single "unit" that can be used in a variety of contexts. So, rather than needing to repeatedly write down all the individual instructions for drawing a chart for every one of your variables, you can define a `make_chart()` function once and then just **call** (execute) that function when you want to perform those steps.

In addition to grouping instructions, functions in programming languages like R tend to follow the mathematical definition of functions, which is a set of operations (instructions!) that are performed on some **inputs** and lead to some **outputs**. Function inputs are called **arguments** (also

referred to as **parameters**); specifying an argument for a function is called **passing** the argument into the function (like passing a football). A function then **returns** an output to use. For example, imagine a function that can determine the largest number in a set of numbers—that function's input would be the set of numbers, and the output would be the largest number in the set.

Grouping instructions into reusable functions is helpful throughout the data science process, including areas such as the following:

- *Data management*: You can group instructions for loading and organizing data so they can be applied to multiple data sets.
- *Data analysis*: You can store the steps for calculating a metric of interest so that you can repeat your analysis for multiple variables.
- *Data visualization*: You can define a process for creating graphics with a particular structure and style so that you can generate consistent reports.

6.1.1 R Function Syntax

R functions are referred to by name (technically, they are values like any other variable). As in many programming languages, you call a function by writing the name of the function followed immediately (no space) by parentheses `()`. Inside the parentheses, you put the arguments (inputs) to the function separated by commas (`,`). Thus, computer functions look just like multi-variable mathematical functions, but with names longer than `f()`. Here are a few examples of using functions that are included in the R language:

```
# Call the print() function, passing it "Hello world" as an argument
print("Hello world")
# [1] "Hello world"

# Call the sqrt() function, passing it 25 as an argument
sqrt(25) # returns 5 (square root of 25)

# Call the min() function, passing it 1, 6/8, and 4/3 as arguments
# This is an example of a function that takes multiple arguments
min(1, 6 / 8, 4 / 3) # returns 0.75 (6/8 is the smallest value)
```

> **Remember:** In this text, we always include empty parentheses `()` when referring to a function by name to help distinguish between variables that hold functions and variables that hold values (e.g., `add_values()` versus `my_value`). This does not mean that the function takes no arguments; instead, it is just a useful shorthand for indicating that a variable holds a function (*not* a value).

If you call any of these functions interactively, R will display the returned value (the output) in the console. However, the computer is not able to "read" what is written in the console—that's for humans to view! If you want the computer to be able to *use* a returned value, you will need to give

that value a name so that the computer can refer to it. That is, you need to store the returned value in a variable:

```
# Store the minimum value of a vector in the variable `smallest_number`
smallest_number <- min(1, 6 / 8, 4 / 3)

# You can then use the variable as usual, such as for a comparison
min_is_greater_than_one <- smallest_number > 1 # returns FALSE

# You can also use functions inline with other operations
phi <- .5 + sqrt(5) / 2 # returns 1.618034

# You can pass the result of a function as an argument to another function
# Watch out for where the parentheses close!
print(min(1.5, sqrt(3)))
# [1] 1.5
```

In the last example, the resulting value of the "inner" function function—`sqrt()`—is immediately used as an argument. Because that value is used immediately, you don't have to assign it a separate variable name. Consequently, it is known as an **anonymous variable**.

6.2 Built-in R Functions

As you have likely noticed, R comes with a variety of functions that are built into the language (also referred to as "*base*" R functions). The preceding example used the `print()` function to print a value to the console, the `min()` function to find the smallest number among the arguments, and the `sqrt()` function to take the square root of a number. Table 6.1 provides a *very* limited list of functions you might experiment with (or see a few more from *Quick-R*[1]).

To learn more about any individual function, you can look it up in the R documentation by using `?FUNCTION_NAME` as described in Chapter 5.

> **Tip**: Part of learning any programming language is identifying which functions are available in that language and understanding how to use them. Thus, you should look around and become familiar with these functions—but do not feel that you need to memorize them! It's enough to be aware that they exist, and then be able to look up the name and arguments for that function. As you can imagine, Google also comes in handy here (i.e., *"how to DO_TASK in R"*).

This is just a tiny taste of the many different functions available in R. More functions will be introduced throughout the text, and you can also see a nice list of options in the *R Reference Card*[2] cheatsheet.

[1] **Quick-R: Built-in Functions**: http://www.statmethods.net/management/functions.html
[2] **R Reference Card:** cheatsheet summarizing built-in R functions: https://cran.r-project.org/doc/contrib/Short-refcard.pdf

Table 6.1 **Examples and descriptions of frequently used R functions**

Function Name	Description	Example
`sum(a, b, ...)`	Calculates the sum of all input values	`sum(1, 5)` # *returns* 6
`round(x, digits)`	Rounds the first argument to the given number of digits	`round(3.1415, 3)` # *returns* `3.142`
`toupper(str)`	Returns the characters in uppercase	`toupper("hi mom")` # *returns* `"HI MOM"`
`paste(a, b, ...)`	*Concatenates* (combines) characters into one value	`paste("hi", "mom")` # *returns* `"hi mom"`
`nchar(str)`	Counts the number of characters in a string (including spaces and punctuation)	`nchar("hi mom")` # *returns* 6
`c(a, b, ...)`	*Concatenates* (combines) multiple items into a *vector* (see Chapter 7)	`c(1, 2)` # *returns* `1, 2`
`seq(a, b)`	Returns a sequence of numbers from a to b	`seq(1, 5)` # *returns* `1, 2, 3, 4, 5`

6.2.1 Named Arguments

Many functions have both *required arguments* (values that you must provide) and *optional arguments* (arguments that have a "default" value, unless you specify otherwise). Optional arguments are usually specified using **named arguments**, in which you specify that an argument value has a particular name. As a result, you don't need to remember the order of optional arguments, but can instead simply reference them by name.

Named arguments are written by putting the name of the argument (which is like a variable name), followed by the equals symbol (`=`), followed by the value to pass to that argument. For example:

```
# Use the `sep` named argument to specify the separator is '+++'
paste("Hi", "Mom", sep = "+++") # returns "Hi+++Mom"
```

Named arguments are almost always optional (since they have default values), and can be included in any order. Indeed, many functions allow you to specify arguments either as **positional arguments** (called such because they are determined by their position in the argument list) or with a name. For example, the second positional argument to the `round()` function can also be specified as the named argument `digits`:

```
# These function calls are all equivalent, though the 2nd is most clear/common
round(3.1415, 3) # 3.142
round(3.1415, digits = 3) # 3.142
round(digits = 3, 3.1415) # 3.142
```

To see a list of arguments—required or optional, positional or named—available to a function, look it up in the documentation (e.g., using `?FUNCTION_NAME`). For example, if you look up the `paste()`

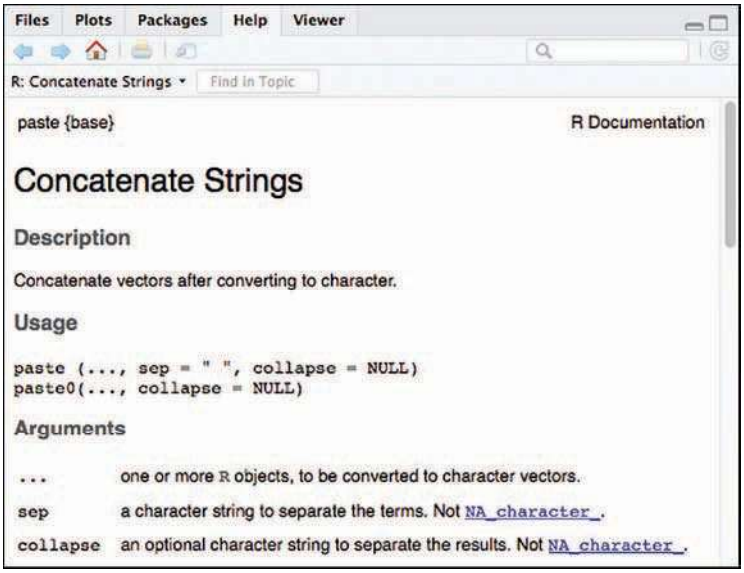

Figure 6.1 Documentation for the `paste()` function, as shown in RStudio.

function (using `?paste` in RStudio), you will see the documentation shown in Figure 6.1. The *usage* displayed —`paste (..., sep = " ", collapse = NULL)`— specifies that the function takes any number of positional arguments (represented by the `...`), as well as two additional named arguments: `sep` (whose default value is `" "`, making pasted words default to having a space between them) and `collapse` (used when pasting *vectors*, described in Chapter 7).

Tip: In R's documentation, functions that require a limited number of unnamed arguments will often refer to them as `x`. For example, the documentation for `round()` is listed as follows: `round(x, digits = 0)`. The `x` just means "the data value to run this function on."

Fun Fact: The mathematical operators (e.g., `+`) are actually functions in R that take two arguments (the operands). The familiar mathematical notation is just a shortcut.

```
# These two lines of code are the same:
x <- 2 + 3 # add 2 and 3
x <- '+'(2, 3) # add 2 and 3
```

6.3 Loading Functions

Although R comes with lots of built-in functions, you can always use more functions! **Packages** (also broadly, if inaccurately, referred to as *libraries*) are additional sets of R functions that are written and published by the R community. Because many R users encounter the same data management and analysis challenges, programmers are able to use these packages and thereby

benefit from the work of others. (This is the amazing thing about the open source community—people solve problems and then make those solutions available to others.) Popular R packages exist for manipulating data (`dplyr`), making beautiful graphics (`ggplot2`), and implementing machine learning algorithms (`randomForest`).

R packages do not ship with the R software by default, but rather need to be downloaded (once) and then loaded into your interpreter's environment (each time you wish to use them). While this may seem cumbersome, the R software would be huge and slow if you had to install and load *all* available packages to do anything with it.

Luckily, it is possible to install and load R packages from within R. The base R software provides **`install.packages()`** function for installing packages, and the **`library()`** function for loading them. The following example illustrates installing and loading the `stringr` package (which contains handy functions for working with character strings):

```
# Install the `stringr` package. Only needs to be done once per computer
install.packages("stringr")

# Load the package (make `stringr` functions available in this `R` session)
library("stringr") # quotes optional here, but best to include them
```

> **Caution**: When you install a package, you may receive a warning message about the package being built under a previous version of R. In all likelihood, this shouldn't cause a problem, but you should pay attention to the details of the messages and keep them in mind (especially if you start getting unexpected errors).
>
> Errors installing packages are some of the trickiest to solve, since they depend on machine-specific configuration details. Read any error messages carefully to determine what the problem may be.

The `install.packages()` function downloads the necessary set of R code for a given package (which explains why you need to do it only once per machine), while the `library()` function loads those scripts into your current R session (you connect to the "library" where the package has been installed). If you're curious *where* the library of packages is located on your computer, you can run the R function `.libPaths()` to see where the files are stored.

> **Caution**: Loading a package sometimes overrides a function of the same name that is already in your environment. This may cause a warning to appear in your R terminal, but it does not necessarily mean you made a mistake. Make sure to read warning messages carefully and attempt to decipher their meaning. If the warning doesn't refer to something that seems to be a problem (such as overriding existing functions you weren't going to use), you can ignore it and move on.

After loading a package with the `library()` function, you have access to functions that were written as part of that package. For example, `stringr` provides a function **`str_count()`** that

returns how many times a "substring" appears in a word (see the `stringr` documentation[3] for a complete list of functions included in that package):

```
# How many i's are in Mississippi?
str_count("Mississippi", "i") # 4
```

Because there are so many packages, many of them will provide functions with the same names. You thus might need to distinguish between the `str_count()` function from `stringr` and the `str_count()` function from somewhere else. You can do this by using the full package name of the function (called **namespacing** the function)—written as the package name, followed by a double colon (`::`), followed by the name of the function:

```
# Explicitly call the namespaced `str_count` function. Not very common.
stringr::str_count("Mississippi", "i") # 4

# Equivalently, call the function without namespacing
str_count("Mississippi", "i") # 4
```

Much of the work involved in programming for data science involves finding, understanding, and using these external packages (no need to reinvent the wheel!). A number of such packages will be discussed and introduced in this text, but you must also be willing to extrapolate what you learn (and research further examples) to new situations.

Tip: There are packages available to help you improve the style of your R code. The `lintr`[a] package detects code that violates the tidyverse style guide, and the `styler`[b] package applies suggested formatting to your code. After loading those packages, you can run `lint("MY_FILENAME.R")` and `style_file("MY_FILENAME.R")` (using the appropriate filename) to help ensure you have used good code style.

[a] https://github.com/jimhester/lintr
[b] http://styler.r-lib.org

6.4 Writing Functions

Even more exciting than loading other people's functions is writing your own. Anytime that you have a task that you may repeat throughout a script—or if you just want to organize your thinking—it's good practice to write a function to perform that task. This will limit repetition and reduce the likelihood of errors, as well as make things easier to read and understand (and identify flaws in your analysis).

[3] https://cran.r-project.org/web/packages/stringr/stringr.pdf

The best way to understand the syntax for defining a function is to look at an example:

```
# A function named `make_full_name` that takes two arguments
# and returns the "full name" made from them
make_full_name <- function(first_name, last_name) {
  # Function body: perform tasks in here
  full_name <- paste(first_name, last_name)

  # Functions will *return* the value of the last line
  full_name
}

# Call the `make_full_name()` function with the values "Alice" and "Kim"
my_name <- make_full_name("Alice", "Kim") # returns "Alice Kim" into `my_name`
```

Functions are in many ways like variables: they have a **name** to which you *assign* a value (using the same assignment operator: `<-`). One difference is that they are written using the **`function`** keyword to indicate that you are creating a function and not simply storing a value. Per the tidyverse style guide,[4] functions should be written in `snake_case` and named using **verbs**—after all, they define something that the code will *do*. A function's name should clearly suggest what it does (without becoming too long).

> **Remember**: Although tidyverse functions are written in `snake_case`, many built-in R functions use a dot `.` to separate words—for example, `install.packages()` and `is.numeric()` (which determines whether a value is a number and not, for example, a character string).

A function includes several different parts:

- **Arguments**: The value assigned to the function name uses the syntax `function(...)` to indicate that you are creating a function (as opposed to a number or character string). The words put between the parentheses are names for variables that will contain the values passed in as arguments. For example, when you call `make_full_name("Alice", "Kim")`, the value of the first argument (`"Alice"`) will be assigned to the first variable (`first_name`), and the value of the second argument (`"Kim"`) will be assigned to the second variable (`last_name`).

  Importantly, you can make the argument names anything you want (`name_first`, `given_name`, and so on), just as long as you then use that variable name to refer to the argument inside the function body. Moreover, these argument variables are available only while inside the function. You can think of them as being "nicknames" for the values. The variables `first_name`, `last_name`, and `full_name` exist only within this particular function; that is, they are accessible within the **scope** of the function.

- **Body**: The body of the function is a *block of code* that falls between curly braces **`{}`** (a "block" is represented by curly braces surrounding code statements). The cleanest style is to put the opening { immediately after the arguments list, and the closing } on its own line.

[4] **tidyverse style Guide**: http://style.tidyverse.org/functions.html

The function body specifies all the instructions (lines of code) that your function will perform. A function can contain as many lines of code as you want. You will usually want more than 1 line to make the effort of creating the function worthwhile, but if you have more than 20 lines, you might want to break it up into separate functions. You can use the argument variables in here, create new variables, call other functions, and so on. Basically, any code that you would write outside of a function can be written inside of one as well!

- **Return value**: A function will return (output) whatever value is evaluated in the last statement (line) of that function. In the preceding example, the final `full_name` statement will be returned.

  It is also possible to explicitly state what value to return by using the **`return()`** function, passing it the value that you wish your function to return:

```
# A function to calculate the area of a rectangle
calculate_rect_area <- function(width, height){
    return(width * height) # return a specific result
}
```

  However, it is considered good style to use the `return()` statement only when you wish to return a value before the final statement is executed (see Section 6.5). As such, you can place the value you wish to return as the last line of the function, and it will be returned:

```
# A function to calculate the area of a rectangle
calculate_rect_area <- function(width, height){
    # Store a value in a variable, then return that value
    area <- width * height # calculate area
    area # return this value from the function
}

# A function to calculate the area of a rectangle
calculate_rect_area <- function(width, height){
    # Equivalently, return a value anonymously (without first storing it)
    width * height # return this value from the function
}
```

You can call (execute) a function you defined the same way you call built-in functions. When you do so, R will take the arguments you pass in (e.g., `"Alice"` and `"Kim"`) and assign them to the argument variables. It then executes each line of code in the function body one at a time. When it gets to the last line (or the `return()` call), it will end the function and return the last expression, which could be assigned to a different variable outside of the function.

Overall, writing functions is an effective way to group lines of code together, creating an abstraction for those statements. Instead of needing to think about doing four or five steps at once, you can just think about a single step: calling the function! This makes it easier to understand your code and the analysis you need to perform.

6.4.1 Debugging Functions

A central part of writing functions is fixing the (inevitable) errors that you introduce in the process. Identifying errors within the functions you write is more complex than resolving an issue with a single line of code because you will need to search across the entire function to find the source of the error! The best technique for honing in on and identifying the line of code with the error is to run each line of code *one at a time*. While it is possible to execute each line individually in RStudio (using `cmd+enter`), this process requires further work when functions require *arguments*.

For example, consider a function that calculates a person's body mass index (BMI):

```
# Calculate body mass index (kg/m^2) given the input in pounds (lbs) and
# inches (inches)
calculate_bmi <- function(lbs, inches) {
  height_in_meters <- inches * 0.0254
  weight_in_kg <- lbs * 0.453592
  bmi <- weight_in_kg / height_in_meters ^ 2
  bmi
}

# Calculate the BMI of a person who is 180 pounds and 70 inches tall
calculate_bmi(180, 70)
```

Recall that when you execute a function, R evaluates each line of code, replacing the arguments of that function with the values you supply. When you execute the function (e.g., by calling `calculate_bmi(180, 70)`), you are essentially *replacing* the variable `lbs` with the value `180`, and replacing the variable `inches` with the value `70` throughout the function.

But if you try to run each statement in the function one at a time, then the variables `lbs` and `inches` won't have values (because you never actually called the function)! Thus a strategy for debugging functions is to assign sample values to your arguments, and then run through the function line by line. For example, you could do the following (either within the function, in another part of the script, or just in the console):

```
# Set sample values for the `lbs` and `inches` variables
lbs <- 180
inches <- 70
```

With those variables assigned, you can run each statement inside the function one at a time, checking the intermediate results to see where your code makes a mistake—and then you can fix that line and retest the function! Be sure to delete the temporary variables when you're done.

Note that while this will identify *syntax errors*, it will not help you identify *logical* errors. For example, this strategy will not help if you use the incorrect conversion between inches and meters, or pass the arguments to your function in the incorrect order. For example, `calculate_bmi(70, 180)` won't return an error, but it will return a *very* different BMI than `calculate_bmi(180, 70)`.

> **Remember:** When you pass arguments to functions, *order matters!* Be sure that you are passing in values in the order expected by the function.

6.5 Using Conditional Statements

Functions are a way to organize and control the flow of execution of your code (e.g., which lines of code get run in which order). In R, as in other languages, you can also control program flow by specifying different instructions that can be run based on a different set of conditions. **Conditional statements** allow you to specify different blocks of code to run when given different contexts, which is often valuable within functions.

In an abstract sense, a conditional statement is saying:

```
IF something is true
  do some lines of code
OTHERWISE
  do some other lines of code
```

In R, you write these conditional statements using the keywords **`if`** and **`else`** and the following syntax:

```
# A generic conditional statement
if (condition) {
  # lines of code to run if `condition` is TRUE
} else {
  # lines of code to run if `condition` is FALSE
}
```

Note that the `else` needs to be on the same line as the closing curly brace (}) of the `if` block. It is also possible to omit the `else` and its block, in case you don't want to do anything when the condition isn't met.

The `condition` can be any variable or expression that resolves to a logical value (`TRUE` or `FALSE`). Thus both of the following conditional statements are valid:

```
# Evaluate conditional statements based on the temperature of porridge

# Set an initial temperature value for the porridge
porridge_temp <- 125 # in degrees F

# If the porridge temperature exceeds a given threshold, enter the code block
if (porridge_temp > 120) { # expression is true
  print("This porridge is too hot!") # will be executed
}

# Alternatively, you can store a condition (as a TRUE/FALSE value)
# in a variable
too_cold <- porridge_temp < 70 # a logical value

# If the condition `too_cold` is TRUE, enter the code block
if (too_cold) { # expression is false
  print("This porridge is too cold!") # will not be executed
}
```

You can further extend the set of conditions evaluated using an `else if` statement (e.g., an `if` immediately after an `else`). For example:

```
# Function to determine if you should eat porridge
test_food_temp <- function(temp) {
  if (temp > 120) {
    status <- "This porridge is too hot!"
  } else if (temp < 70) {
    status <- "This porridge is too cold!"
  } else {
    status <- "This porridge is just right!"
  }
  status # return the status
}

# Use the function on different temperatures
test_food_temp(150) # "This porridge is too hot!"
test_food_temp(60)  # "This porridge is too cold!"
test_food_temp(119) # "This porridge is just right!"
```

Note that a set of conditional statements causes the code to *branch*—that is, only one block of the code will be executed. As such, you may want to have one block return a specific value from a function, while the other block might keep going (or return something else). This is when you would want to use the `return()` function:

```
# Function to add a title to someone's name
add_title <- function(full_name, title) {
  # If the name begins with the title, just return the name
  if (startsWith(full_name, title)) {
    return(full_name) # no need to prepend the title
  }

  name_with_title <- paste(title, full_name) # prepend the title
  name_with_title # last argument gets returned
}
```

Note that this example didn't use an explicit `else` clause, but rather just let the function "keep going" when the `if` condition wasn't met. While both approaches would be valid (achieve the same desired result), it's better code design to avoid `` `else` `` statements when possible and to instead view the `if` conditional as just handling a "special case."

Overall, conditionals and functions are ways to *organize* the flow of code in your program: to explicitly tell the R interpreter in which order lines of code should be executed. These structures become particularly useful as programs get large, or when you need to combine code from multiple script files. For practice using and writing functions, see the set of accompanying book exercises.[5]

[5] **Function exercises:** https://github.com/programming-for-data-science/chapter-06-exercises