CHESTER ISMAY

GETTING USED TO R, RSTUDIO, AND R MARKDOWN

Contents

1

Introduction

In the HTML version of this book, you can also download the PDF version of the book by clicking on PDF button in the top toolbar of the page. HTML is the preferred format but the PDF format may be preferred for some readers. Links to the different YouTube videos directly found in the HTML version are provided in the PDF version. You can also download the video files directly here. Note that no audio is attached to these screencast videos.

This resource is designed to provide new users to R, RStudio, and R Markdown with the introductory steps needed to begin their own reproducible research. Many screenshots and screencasts (with no audio) will be included, but if further clarification is needed on these or any other aspect of the book, please create a GitHub issue here or email me with a reference to the error/area where more guidance is necessary. Pull requests on GitHub for typos or improvements are preferred and you can easily do so by clicking on the Edit button near Search at the top of the HTML version of the book.

This book will evolve and be updated as needed based on feedback. You can always check the date below to see when the book was last updated.

It is recommended that you have R version 3.3.0 or later, RStudio Desktop version 1.0 or higher, and `rmarkdown` R package version 1.0 or higher. This is to make sure that the screenshots and GIF recordings match up with the behavior on your machine/set-up. Additionally, you may find that GIFs don't load sometimes. I haven't had any problems using Google Chrome and recommend that as your browser to view this book if you have troubles with others.

This book was last updated by Chester on Sunday, January 07, 2018 17:11:17 PST.

2

Why R?

If you are brand new to R and programming, you may be scared. You aren't used to having to type commands to tell the computer what to do. You may be more used to using drop-down menus and other graphical user interfaces that allow you to pick what you'd like to do. So why are so many companies, colleges/universities, and individuals of all disciplinary backgrounds shifting towards using R?

There are lots of answers to this question, but some of the most important for us now are:

1. R is free. RStudio is free.

   One of the biggest perks about working with R and RStudio is that they are both provided free of charge to use. R is an open-source programming language that has grown tremendously in recent years with developers adding more functionality and packages on a daily basis. Where other more proprietary packages are sometimes stuck in the dark ages (the 1990s, for example) of development and can be incredibly expensive to purchase, R continues to be a free alternative that allows users of all levels to contribute.

   RStudio is a graphical user interface that allows one to write R code and view the results of that code in an easy way. It is also free to download and work with.

2. Analyses done using R are reproducible.

   As many scientific fields push towards more reproducible analyses, the point-and-click proprietary systems actually serve as a hindrance to this process. If you need to re-run your analysis using these systems, you'll need to carefully copy-and-paste your analysis and plots into your text editors from potentially beginning to end. Anyone that has done this sort of copy-and-pasting knows that it is prone to errors and incredibly tedious.

   If you use the workflows described in this book, your analyses will be reproducible so you don't need to worry about these copy-and-pasting issues. As you might have guessed by now, it would be much better to be able to update your code/data inputs and re-run all of your analysis than to have to worry about manually moving your results from one program to another. Reproducibility also helps you as a programmer since your greatest collaborator

will probably be yourself a few months or years down the road. Instead of having to carefully write down all the steps you took to find the right drop-down menu option, your entire code is stored.

3. Using R makes collaboration easier.

   This also helps you with collaboration since, as you will see later, you can share an R Markdown file containing all of your analysis, documentation, commentary, and the code to others. This reduces the time to needed to work with others and reduces the likelihood of errors being made in following along with point-and-click analyses. The mantra here will be to **Say No to Copy-And-Paste!** both for your sanity and for the sake of science.

4. Finding answers to questions is much simpler.

   If you have ever had an issue with software, you know how difficult it is to find answers to your questions. "How can I describe the process to someone else? Do I need to take screenshots? Do I really need to call IT and wait for hours for someone to respond?" R is a programming language and so it is much easier (after a bit of practice) to use Google or Stack Overflow to find answers to your questions. You'll be amazed at how many other users have encountered the same sorts of errors you will see when you begin.

   I frequently (almost on a daily basis) Google things like "How do I make a side-by-side boxplot in R coloring by a third variable?". You'll become better at working with R by reaching out to others for help and by answering questions that others have. In addition, Chapter 6 describes many common errors and how you can fix them.

5. Struggling through programming helps you learn.

   We all know that learning isn't easy. Do you have trouble remembering how to follow a list of more than 10 steps or so? Do you find yourself going back over and over again because you can't remember what step comes next in the process? This is extremely common especially if you haven't done the procedure in awhile. Learning via following a procedure is easy in the short-term, but can be extremely frustrating to remember in the long-term. Programming (if done well) promotes long-term thinking to short-term fixes.

   One unfortunate thing that we frequently take for granted is that our brain tricks us into picking the easy route. If you truly want to learn how to do something (like programming with R), you'll need to feel frustrated at times. Any time you learn something you've been frustrated. (We tend to forget all the frustration and only think about where we currently are.) R still frustrates me from time to time, but I grow through practice and I look forward to the challenges. Hadley Wickham encapsulated this phenomenon nicely in the Prologue of the book "Hands-On Programming with R" (Grolemund, 2014):

   > As you learn to program, you are going to get frustrated. You are learning a new language, and it will take time to become fluent. But frustration is not just natural, it's actually a positive sign that you should watch for. Frustration is your brain's way of being lazy; it's trying to get you to quit and go do something easy or fun. If you want to get physically fitter, you need to push your body even though it complains. If you want to get better at programming, you'll need to push your brain. Recognize when you get frustrated and see it

as a good thing: you're now stretching yourself. Push yourself a little further every day, and you'll soon be a confident programmer.

3

R and RStudio Basics

3.1 What is R?

In Chapter 2, I discussed many of the reasons why you should be doing your analyses (especially those of the data type) using R. If you skipped over that chapter in the hopes of just hopping into learning about R, I request that you to go back to it and carefully read it over. As you begin working with R, it is especially important to review that introductory chapter from time to time.

3.1.1 R beginnings

R was created by a group of statisticians who wanted an open-source alternative to the costly proprietary options. Being created by statisticians (instead of computer scientists) means that R has some quirky aspects to it that take a little bit of time to get used to. We'll see that many packages have been developed to help with this and that you don't need to have advanced degrees in Statistics to be able to work with R now.

Getting back to the development of R... R was created by **R**oss Ihaka and **R**obert Gentleman in New Zealand at the University of Auckland. It is a spin-off of the S programming language and is named partly after the first names of its developers (as you can see in the emphasis above). The beginning ideas for creating R came in 1992 and the first version of R was released in 1994. You can find much more about the background of R and its features as well as its connections to the S language on its Wikipedia page.

3.1.2 R packages

I first learned to use R while a graduate student at Northern Arizona University from Dr. Philip Turk in 2007. At the time, I never thought that R could have exploded in users as we have seen since 2011. I never would have thought that students taking an introductory statistics course would be encouraged to learn to use R.

In 2007, it was still largely esoteric and tricky language used by statisticians to do analyses. Getting used to the syntax for producing plots and working with data was especially tricky for those with little to no programming experience. So what has changed since 2007 about learning R?

I believe one of the biggest developments has been the creation of packages to make R easier to work with for newbies. Packages are created by users of R to increase the functionality of the base R installation. Packages created by Hadley Wickham and others recently have greatly expanded the capabilities of R, while also working to make beginning with R simpler. From the Wikipedia page referenced earlier, as of January 2016, there were around 7800 additional R packages available on common R repositories.[1]

Another great development is the graphical user interface called RStudio and the package developed by the those that work for RStudio, Inc. called `rmarkdown`. We will discuss `rmarkdown` (also referred to as R Markdown) in Chapter 4, and will now focus on discussing RStudio.

[1] You'll see how to download these packages via `install.packages("dplyr")` and load them into your current R working environment via `library("dplyr")`, for example, in Chapter 5.

3.2   What is RStudio?

RStudio is a powerful, free, open-source integrated development environment for R. It began development in 2010 and its first beta release came in February 2011. It is available in two editions: RStudio Desktop and RStudio Server. This book will focus mostly on using the RStudio Server, but both versions are nearly identical to work with.

You can find instructions linked below for downloading R and RStudio on Windows and Mac machines. If you are using RStudio Server, your professor and members of your organization's IT department have done these steps for you. On the RStudio Server you log on using a web browser to an account sitting on the cloud. There are many advantages to using the RStudio Server for the beginning user including sharing of R projects to help with feedback and error resolution. Installation of software can also cause its own headaches and this is eliminated by using the RStudio Server.

Note for advanced users: You can also install your very own RStudio Server for around $5 per month on Digital Ocean. Instructions to do so can be found from Dean Attali here and on the Digital Ocean site here.

After you complete a few months of work with the RStudio Server, it is recommended that you download RStudio Desktop to your computer. The instructions to do so are below.

3.2.1   Installing R and RStudio Desktop

It is worth noting that you can't just install RStudio Desktop without installing R as RStudio needs to have R installed in order to run. A step-by-step guide to installing R and RStudio Desktop with screenshots can be found

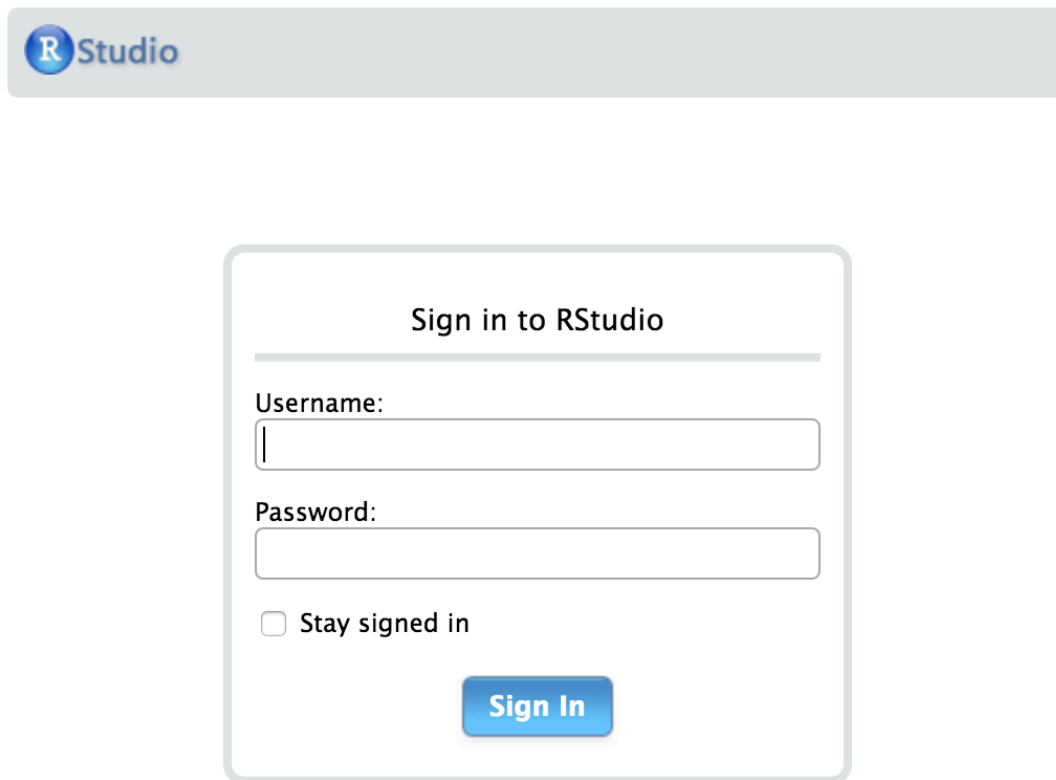- here for the Mac and
- here for a PC.

Unless you plan to create PDF documents (which requires a multiple gigabyte download of

LaTeX) you can skip some of the later steps of the installation. It is recommended that you select **HTML** as the Default Output Format for R Markdown. You'll see more about this in Chapter 4.

3.3   Working in RStudio Server

3.3.1   Logging in and initial screen

The RStudio Server provides a web-based way to run analyses in R. This means that you will only need an internet connection and a web browser to run your analyses. Your professor or administrator will provide you with a link to the web location of your RStudio Server. After entering the link, you'll see a page that looks something like:

Figure 3.1: Login page for RStudio Server

After logging in with your username and password, you should see a layout similar to what follows.

For your own reference, a screenshot of RStudio Desktop looks like:

As you can see they are almost identical. This makes working between the two different RStudio set-ups painless. A discussion of what each of the three different RStudio panes (will soon be four panes) and their corresponding tabs will occur in Chapter 4. You'll find that a lot of
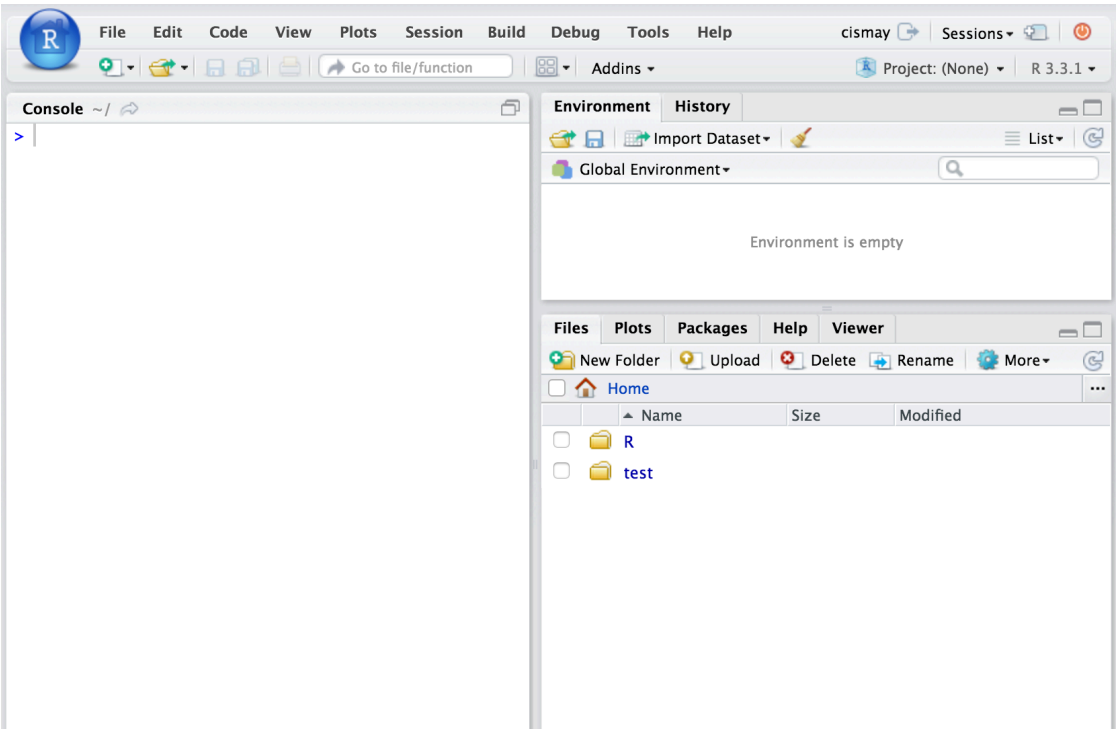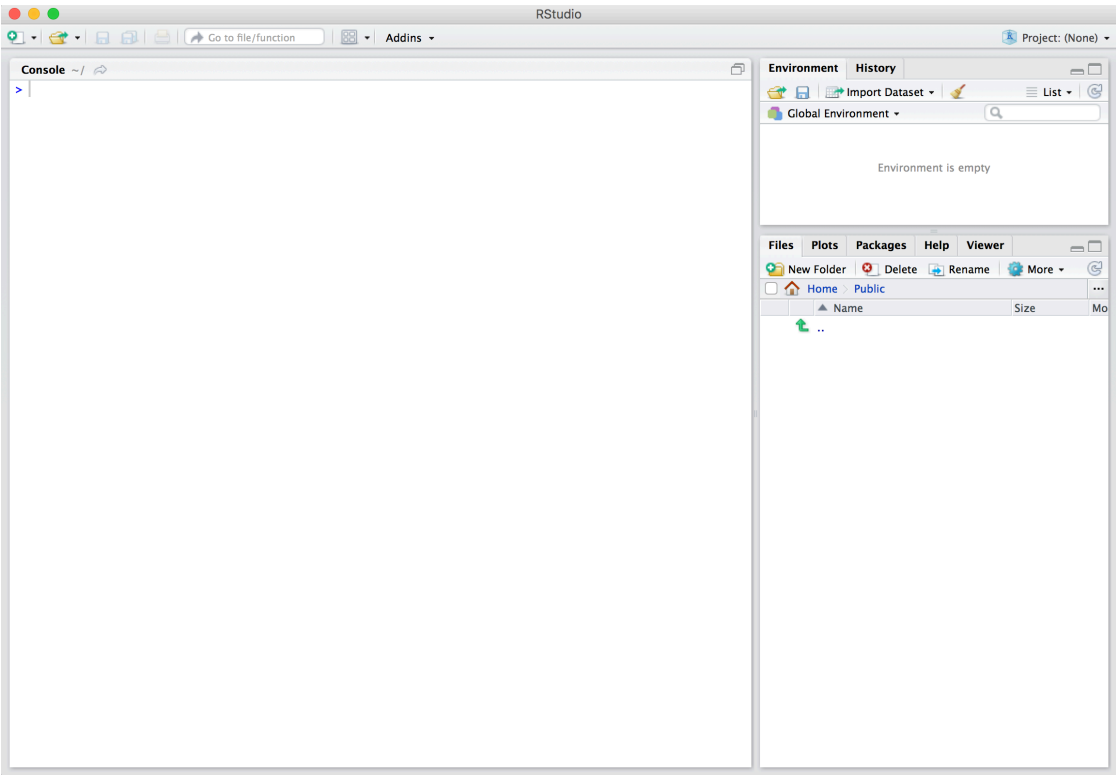
Figure 3.2: Initial page for RStudio Server

Figure 3.3: Initial page for RStudio Desktop

what follows also applies to RStudio Desktop (except for the Shared Projects feature), but it always recommended to create an RStudio project regardless of whether you are on the cloud or working locally.

3.3.2   Basic Workflow with RStudio

A good habit to get into whenever you start a new project with R code is to create a new RStudio project to go along with it. RStudio project files have the extension `.Rproj` and store metadata that goes along with the documents you've saved and information about the R environment you are working in. More information about RStudio projects is available from RStudio, Inc. here.

If you are sharing homework or lab assignments with your instructor using RStudio Server, for example, it might make sense to create an RStudio project, share it with your instructor, and then create new folders for each lab. We will follow this example below.

The video below shows you how to create a new RStudio project called `initial` and also your first R Markdown file. Note that you also may see a description about what version of R is running on your initial login like shown in the screencast below in the Console pane.

We have our **first_rmarkdown.Rmd** file set up.

3.3.3   Sharing Projects on RStudio Server Pro

You will now see an example of how to share this project with another user. This will enable you and collaborators (other students, your instructor, etc.) to work on the **Rmd** file at the same time. This is similar to working on a Google Doc at the same time as someone else.

RStudio Server comes in a couple different formats and you'll need to make sure you (or your IT administrator) have installed RStudio Server Pro to use the Shared Projects feature. You can find more information from RStudio, Inc. on this here. Below is an example video of sharing this **initial.Rproj** project file with another user of the RStudio Server.

Both myself and `bottk` can now work together on this project. We can type our commentary and code into **first_rmarkdown.Rmd** or other files and save files to the common folder where **initial.Rproj** resides.

In Chapter 4, you'll see why it is recommended you work in R Markdown files and you'll also begin to see some examples of how R works with R Markdown.

3.4   RStudio Layout

You may be initially a little overwhelmed by all the different panes and tabs that are available in RStudio. You'll soon learn to love this layout but, as with all things, it will take a little bit to get used to it. We will begin with the top left pane, proceed to the top right pane, then to the bottom left pane, and lastly to the bottom right pane. These panes can be customized but it is recommended for beginning users to keep things in this standard layout.

3.4.1   Code Editor / View Window

Likely the pane where you will spend a majority of your time is in the top left. When you first login this pane isn't there, but it was created when we made the **first_rmarkdown.Rmd**. This pane serves as a place to view the contents of files and objects in R. In the screencast below, you can see that we can change the text in this file and then save the file.

Note that the tab that gives the filename will change its color from black to red and an asterisk will appear after the filename. This is to remind you that the file is not saved. You'll need to get into the habit of saving files frequently. You can have multiple tabs open and view different files in this pane. You'll also see that you can `View` datasets in this pane in Chapter 4.

It may not be clear what these additions are really doing just yet. You'll be pressing the **Knit HTML** button near the top of the pane to put all of your text, code, and its output together. We aren't quite there just yet though!

3.4.2   Environment / History

The next pane includes by default an **Environment** tab and a **History** tab. To get a sense of what these tabs provide we'll need to also use the bottom left pane and the **Console** tab. I'll show you how to create multiple objects in R using the **Console**. Initially you will see that the **Environment** tab tells us that the "Environment is empty." If you click on the **History** tab, you should also see a blank screen with a few icons. We won't go over using all of these buttons here, but you are encouraged to hover over them and click on them to get a sense for what they do. As I enter code into the **Console** watch to see how the **Environment** and **History** tabs change.

You can think of the **Console** as a place to play around. It is your R sandbox. You can test your code to make sure it is working and then copy that text into your **Rmd** file into a chunk after you are satisfied with it. We'll see more examples of this in the chapters to come.

Note that, by default, when you enter the name of an object into the console like I did with `sum_1_2` it displays the result. You've also been shown what is called the assignment operator denoted by `<-`. You can read this as putting the contents of the right-hand-side into an object named whatever appears on the left-hand-side. For example, `num1` is the name of an object that stores the value 7.

A powerful feature of the R language has been introduced in the `sum_1_2 <- sum(num1, num2)` line. `sum` is a function. Functions are denoted by their name and then a parenthesis, their arguments separated by commas, and then a closing parenthesis. You'll see many examples of these going forward. Of course, we'll be using R for more than just a basic calculator shown here but this should give you a good idea of what the **Environment** and **History** tabs store.

3.4.3   Console

You'll frequently use the **Console** as a way to check your work or thoughts on how to solve a problem using R. Before RStudio came around, most users of R just had a window like the

Console provides where they entered their commands and then looked at the results in different windows. We will see that the **Console** and the **Code Editor/View Window** will allow us to store all of our code in a file and then "run" that code through the **Console** to check that it works. The example video below shows how this may be done.

Rules for naming objects

It is good practice to get in the habit of naming variables corresponding to what they actually represent. If you are dividing two different sums of numbers, you might want to choose a name like `ratio_of_sums` to refer to that object. R has a couple restrictions on what can be included in the name of R objects:

1. Object names cannot begin with a number.
2. Object names cannot contain symbols used for mathematics or to denote other operations native to R. These symbols include `$`, `@`, `!`, `^`, `+`, `-`, `/`, and `*`.

Another important property of R is that is case-sensitive. You'll see what this means in the video below that also includes examples of invalid names of objects. Note that we will continue to work inside the R chunk as we did in the last video. You'll see that these code chunks provide a nice way to keep track of our important analyses.

You may have noticed that R will do some checks and alert you to potential errors by placing a red X to the left of the line of code with an error. It won't catch all errors, but this can be helpful. Not also that `Name`, `name`, and `nAme` refer to three different values.

Another thing to notice is that you can store numbers into an object with a given name like we did earlier with `num1`. If we'd like to store a character string such as "Chester", we can provide the name of the object on the left-hand-side of `<-` and the string in quotation marks on the right-hand-side of `<-`. There are more complex object types that you will see in Chapter 5, but it is always important to think about the difference between a number and a character in R.

It's also a good idea to not call objects the names of functions that are built into R. You may want to call the addition of two numbers `sum` and R will allow this, but it is **HIGHLY** recommended that you create more descriptive names and not choose to name objects the same as common R functions. Something like `sum_densities` is better and less likely to be the name of a function.

3.4.4   The `help` function (`?`)

Of course, you won't know what all of the names of built-in functions are until you practice, but it is something to think about. If you are ever wondering if a function is built-in or is in a package you have included, you can use the `?` function in the **Console** to check. Some examples are shown below as a screencast.

3.4.5   The bottom right pane

The bottom right pane in RStudio contains the most tabs by default and is a useful place to view a variety of miscellaneous information about your RStudio project and its files.

Files

The leftmost tab here shows the file and folder structure. This shows you where the files are stored, what they are called, and any folders that may exist in your project folder. This can be thought of as similar to going to **My Computer** on a PC or opening **Finder** on a PC. Similarly, this gives the file and directory structure either on the cloud for RStudio Server or on your local machine for RStudio Desktop.

Plots / Viewer

You'll see clearer examples of what the **Plots** and **Viewer** tabs provide in Chapter 4. As you likely guessed **Plots** will show you the resulting graphs/figures that your R code has generated. The **Viewer** tab can show you the resulting HTML file created from an R Markdown **Knit**.

Packages

You can get a sense for what packages have been downloaded to your computer/your cloud server by clicking on the **Packages** tab. You can also see which packages are loaded into the current working environment by looking to see if a check-mark exists next to the package name. Note that you may not have all of the packages loaded onto your machine that I do below in the video. That's OK. This is just an example of what you may expect.

You'll also notice a **Description** of the package as well as the **Version** number here. Packages are frequently updated and improved so this is a way to check to see if you have the most up-to-date version of a package. Remember that this is likely taken care of for you if you are on an RStudio Server installation. If you are on RStudio Desktop, you might find the **Install** and **Update** buttons useful for downloading new packages or updating currently installed ones.

Help

We also saw an example of using the **Help** tab when we invoked the `?` function. This will show you documentation on R functions, datasets, and packages. If you see code that you aren't really sure about, it is often a good go-to to enter a question mark followed by the name and see if the built-in documentation can help you out.

4

R Markdown

R Markdown provides an easy way to produce a rich, fully-documented reproducible analysis. It allows its user to share a single file that contains all of the commentary, R code, and metadata needed to reproduce the analysis from beginning to end. R Markdown allows for chunks of R code to be included along with Markdown text to produce a nicely formatted HTML, PDF, or Word file without having to know any HTML or LaTeX code or have to fuss with getting the formatting just right in a Microsoft Word DOCX file.

One R Markdown file can generate a variety of different formats and all of this is done in a single text file with a few bits of formatting. I think you'll be pleasantly surprised at how easy it is to write an R Markdown document after your first few encounters.

4.1 Fixing Errors in an R Markdown file

We now shift back to the R Markdown file called **first_rmarkdown.Rmd** created in Chapter 3. We know that we left some errors in the creation of variables there, and while it might seem strange to show you errors, it is good exposure for someone new to this to see what errors one might see initially. We are going to see what happens when we click the **Knit HTML** button with these errors. Then we will clean up the code and see what the resulting file looks like from the **Knit**.

When you initially created an R Markdown file, a basic template with some code and text was inputted for you. This is to give you a sense of how to create your own R Markdown file with your own R code and your own commentary. We modified some of that code here. I decided to remove all of the lines in the `cars` named chunk of code even though the errors did not occur in the declaration of the objects that had names stored in them. We see that an HTML file is produced in the **Viewer** pane because **View in Pane** was selected.

As you look over the **Including Plots** text you may be surprised to see that there is no plot provided in the R Markdown file, but in the HTML file there is a scatter-plot showing temperature and pressure varying. This is something alluded to earlier. R Markdown runs the code stored in R chunks and then places that output into the HTML (or PDF or DOCX, etc.) format.

You can also see that the text appears as commentary before and after the R code. You'll

understand in a bit why the text "Including Plots" is so much larger than the other text.

Important note: Remember that all of the R code you want to run needs to be stored in a chunk (in the right order) for your analysis to be reproducible AND for you not to receive errors when you **Knit**. It is easy to do a lot of work in the R **Console** and then forget to add that work into a chunk in your **Rmd** file. This is probably the number one error you will see when you first begin working in RStudio. An example of this error is below in a video.

The `object not found` errors are the most frequently encountered errors and along with misspellings and not completing R code segments provide the vast majority of issues with R. You'll see a further breakdown of this in Chapter 6.

4.2   The Components of an R Markdown File

4.2.1   YAML

The top part of the file is called the YAML header. YAML stands for "YAML Ain't Markup Language" and its website at `http://yaml.org` makes the following statement as to what it is:

> YAML is a human friendly data serialization standard for all programming languages.

Essentially, the YAML stores the metadata needed for the document. You can see an example of a YAML header from our **first_rmarkdown.Rmd** file:

```
---
title: "First RMarkdown"
author: "Chester Ismay"
output: html_document
---
```

There are many other fields that can be customized in the YAML header. The important thing to notice here are the three hyphens that begin and end the YAML header. Indentation also plays a role with YAML so be careful with your alignment of text.

4.2.2   Headers

As you can see above, you can create a lot of different sized headers by simply adding one or more `#` in front of the text you'd like to denote the header.

4.2.3   Emphasis

Whenever you see a hash-tag in the text of your R Markdown document, you now know that this will correspond to bolded larger text[1] that denotes the start of a section of your document.

This is one of the nice features of R Markdown. You can simply look at the plain text and know what it will produce in the knitted document. We can also add different styles of emphasis to words, phrases, or sentences by surrounding them in matching symbols. Below are some examples.

[1] Unless you want to have a fourth, fifth, or sixth level header, but these are not common.

You are beginning to see how easy it is to customize your output. We'll next discuss ways to add links to URLs, create ordered and unordered lists, and other frequently used Markdown features.

4.2.4 Links

To add a link to a URL, you simply enclose the text you'd like displayed in the resulting HTML file inside [ ] and then the link itself inside ( ) right next to each other with no space in between.

4.2.5 Lists

The screencast below shows the process of creating both ordered and unordered lists.

Note that only numbers are needed as we saw by numbering "Warm up food" with a "1." We can also combine unordered and ordered lists by indenting the text two spaces.

In many of the examples that follow, you will see the actual text you'd type into your R Markdown document highlighted with a gray background and also the results of that text immediately below it.

```
1. Wake up
   - Get out of bed
1. Warm up food
   - Open kitchen door
   - Get plate out of cupboard
2. Make coffee
   i. Warm up water
   ii. Grind beans
3. Make breakfast

   We can have a paragraph (or two) here describing how we could go about making
   breakfast.  If we indent the paragraph a few spaces and create a newline, it
   will indent below the item.
```

1. Wake up

   - Get out of bed

2. Warm up food

   - Open kitchen door
   - Get plate out of cupboard

3. Make coffee

   i. Warm up water
   ii. Grind beans

4. Make breakfast

   We can have a paragraph (or two) here describing how we could go about making breakfast.

If we indent the paragraph a few spaces it will indent below the item.

4.2.6   Miscellaneous Markdown

Line breaks / white spacing

Line breaks in combination with white space are incredibly important pieces in Markdown. They frequently denote the start of a new paragraph.

```
Here is an example of text with only a line break.
You may expect this line to appear in a new paragraph but it doesn't.
```

Here is an example of text with only a line break. You may expect this line to appear in a new paragraph but it doesn't.

In order to start a new paragraph, you'll need to be sure that white space exists between the two paragraphs:

```
Here is an example of text with a line break and white space.


You may expect this line to appear in a new paragraph and it does.
```

Here is an example of text with a line break and white space.

You may expect this line to appear in a new paragraph and it does.

Horizontal rules

Another useful way to divide up different parts of your analysis is by including horizontal lines. These can be easily adding by placing three asterisks next to each other (or three hyphens):

```
***
```

```
---
```

Blockquotes

If you'd like to quote someone or produce an indented text block, you can easily do so by adding a > before the passage:

```
> Reproducible research is the idea that data analyses, and more generally,
scientific claims, are published with their data and software code so that
others may verify the findings and build upon them. - Roger Peng
```

> Reproducible research is the idea that data analyses, and more generally, scientific claims, are published with their data and software code so that others may verify the findings and build upon them. - Roger Peng

Commenting Text

There are times where you might want to comment out text inside an R Markdown document. Say you wrote something that you don't really want in the resulting knitted document, but you aren't quite sure if you should delete it completely. To do so you'll just need to enclose the

block of text in `<!-- -->` as seen below:

```
<!--
I'd like to save this text for later and don't want to delete it yet.
-->
```

You won't see the commented out results here in the book.

Equations

If you'd like nice mathematical formulas in your document, you can add them between two dollar signs:

```
$y = mx + b$
```

$y = mx + b$

4.2.7   R Chunks

It took us a bit to get to what I believe is the best part about R Markdown: the addition of R code into the document and a compilation of that code in the resulting knitted document. You've seen some R chunks in the R Markdown file already. There are some properties to get used to about them:

- They always begin and end with three backticks `` ``` ``.
- After the initial three backticks, you'll see R chunks begin with `{r`, potentially some names and other chunk options, and then close that first line with a `}`.
- The lines wrapped inside the beginning and closing three backticks is R code that you could run in the R Console.

Note that spaces in front of these backticks will produce

In our **first_rmarkdown.Rmd** file, let's explore an example of recognizing and creating our own R chunks:

This example introduces you to two different ways to create a vector of values. You'll see further discussion of this in Chapter 5. You can see that the code is automatically run when we pressed the **Knit HTML** button with its output in the knitted file.

Important note: Any other potential R chunks after this chunk will have access to the three variables created here: `count20`, `count100_by_5`, and `prod`. Any chunks before the `mult_vectors` named chunk *WILL NOT* have access to these variables yet. You can read the document like a book, so it is important to add objects and work with them in appropriate order. You'll receive errors from R if you don't.

4.2.8   Inline R code

We've seen that we can add R code and have that run in an R chunk of code enclosed by three backticks. We may want to do a simple calculation on the fly inside of the text of our document though.

Another crafty approach is to have the text produced in our document to automatically update based on the results of R code. To see an example of this, we will select a number at random from our `count20` vector. If it is greater than or equal to 10, we will say so. If it isn't, we will say so.

You also see that an error was made in that I didn't include the third argument to `ifelse` in quotation marks. When I fixed the code and pressed **Knit HTML** again the error went away.

4.2.9   Code Highlighting

As you see in the previous example, it is a good habit to highlight the names of your R objects to distinguish them from the usual text. This can be done by enclosing the word in a single backtick such as what we did with `one_value`.

4.3   R Markdown Chunk Options

The most common R chunk options you'll likely work with are `echo`, `eval`, and `include`. By default, all three of these options are set to `TRUE`.

- `echo` dictates whether the code that produces the result should be printed before the corresponding R output

- `eval` specifies whether the code should be evaluated or just displayed without its output

- `include` specifies whether the code AND its output should be included in the resulting knitted document. If it is set to `FALSE` the code is run but no remnant of the code or its output will be in the resulting document.

Since we specified that `eval=FALSE` and that was where we declared the `one_value` variable we now obtain an error. You can include multiple chunk options by separating with a comma.

4.4   General Guidelines for Writing R Markdown Files

White space is your friend. You should always include a blank white space between R chunks and your Markdown text. It makes your document much more readable and can reduce some potential errors. Also, leave a line of white space between header text and your paragraphs.

Commentary is always good. Explain yourself and your ideas whenever you can. Remember that your greatest collaborator is likely yourself a few months down the road. Be nice to yourself and explain what you are doing so that you can remember!

Remember that the Console and R Markdown environments (when Knitting) don't interact with each other. This forces you to include only the code in your R chunks that produces exactly the results you want to share with others. Don't inflate your document with extra output. You need to be concise and clear in exactly what you are doing.

The chunk options can really beautify your documents and customize them exactly to what you'd like the reader of your documents to see. You can find more information on all of the available R chunk options here.

4.5   Help -> Cheatsheets

RStudio includes really nice cheatsheets that can act as great references to many of the common tasks you will do inside of RStudio. You can get nice PDF versions of the files by going to **Help -> Cheatsheets** inside RStudio.

```
https://raw.githubusercontent.com/ismayc/rbasics-book/gh-pages/screenshots/cheatsheets.png
```

4.6   Spell-check

Near the top of your R Markdown editor window sits one of the more useful tools for writing documents: the spell-check button. It is the green check-mark with "ABC" above it:

Before you submit a document or share it with someone else, please run a spell-check of your document. You'll need to add some R commands to the dictionary or ignore them since those may not be words, but it is easy to misspell words as we type and this feature can really help.

5

Intro to R using R Markdown

In this chapter, you'll see many of the ways that R stores objects and more details on how you can use functions to solve problems in R. You'll be working with a common dataset derived from something that you likely have encountered before: the periodic table from chemistry.

5.1   A beginning directory/file workflow

"File organization and naming are powerful weapons against chaos." - Jenny Bryan

Something that is not frequently discussed when working with files and programming languages like R is the importance of naming your files something relevantly and having organization of your files in folders. You may be tempted to call a file `analysis.Rmd` but what happens when you need to change your analysis months from now and you've named many files for many different projects `analysis.Rmd` in many different folders. It's much better to give your future self a break and commit to concise naming strategies.

You can choose a variety of ways to name files. These guidelines are what I try to follow:

1. Group similar style files into the same folder whenever possible.

   Try not to have one folder that contains all of the different types of files you are working with. It is much easier to find what you are looking for if you put all of your `data` files in a `data` folder, all of your `figure` files in a `figure` folder, and so on.

   This rule can be broken if you only have one or two of each type. It can be the case that too many folders can waste your time when a useful search for the appropriate file may be able to find your file faster than digging through a complex hierarchy of directories.

2. Name your files consistently so that what the file contains can be easily identified from the name of your file, but be concise.

   Seeing `test1.Rmd` and `test2.Rmd` doesn't tell us much about what is actually in the files. It's OK to create a temporary file or two if you don't think you will be using it going forward, but you should be in the habit of reviewing your work at the end of your session and naming files that we needed appropriately.
   Something like `model_fit_sodium.Rmd` is so much better in the long-run.
   Remember to think about your future self whenever you can, especially with programming.

Be nice to yourself so that future self really appreciates past self.

3. Use an underscore to separate elements of the file name.

   The underscore _ produces a visually appealing way for us to realize there is a separation between content. You may be tempted to just use a space, but spaces cause problems in R and in other programming languages. Some folks prefer to just change the case of the first letter of the word to bring attention. Here are a few examples of file names. You can be the judge as to what is most appealing to you:

   `barplot_weight_height.Rmd` vs `barplotWeightHeight.Rmd` vs `barplotweightheight.Rmd`

   Whatever you choose for style, be consistent and think about other users as you name your files. If you were passed a smorgasbord of files that is a mess to deal with and hard to understand, you wouldn't like it, right? Don't be that person to someone else (or yourself)!

5.2   Using R with periodic table dataset

We now hop into the basics of working with a dataset in R. We will also explore the ways R stores data in **objects**, how to access specific elements in those objects, and also how to use **functions**, which are one of the most useful pieces of R to help with organization and clean code.

It is worthy to note that many of the functions here such as `table` and concepts like subsetting, indexing, and creating/modifying new variables can also be done using the great packages that Hadley Wickham has developed and in particular the `dplyr` package. It is still important to get a sense for how R stores objects and how to interact with objects in the "old-school" way. You'll still find some times where doing it this old way actually works just as nicely as the newer modern ways…but they are becoming fewer and fewer by the day. Anyways, you'll hopefully be well-ready to hop into some statistical analyses or data tasks after you follow along with this introduction to R.

5.2.1   Loading data from a file

One of the most common ways you'll want to work with data is by importing it from a file. A common file format that works nicely with R is the CSV (comma-separated values) file. The following R commands first download the CSV file from the internet on my webpage into the `periodic-table-data.csv` file on your computer, then reads in the CSV stored, and then gives the name `periodic_table` to the data frame that stores these values in R:

```
download.file(url = "http://ismayc.github.io/periodic-table-data.csv",
   destfile = "periodic-table-data.csv")
```

```
periodic_table <- read.csv("periodic-table-data.csv",
                           stringsAsFactors = FALSE)
```

We will be discussing both **strings** and **factors** in the data structures section (Section 5.3) and why this additional parameter `stringsAsFactors` set to `FALSE` is recommended.

It's good practice to check out the data after you have loaded it in:

```
View(periodic_table)
```

The video below walks through downloading the CSV file and loading it into the data frame object named `periodic_table`. In addition, it shows another way to view data frames that is built into RStudio without having to run the `View` function. Note that a new R Markdown file is created here as well called `chemistry_example.Rmd`. We are writing the commands directly into R chunks here, but you may find it nicer to play around in your R Console sandbox first.

I encourage you to take the R chunks that follow in this chapter, type/copy them into an R Markdown file that has loaded the `periodic_table` data set, and watch that your resulting output will match the output that I have presented for you here in this book. This book was written in R Markdown after all!

5.3 Data structures

5.3.1 Data frames

Data frames are by far the most common type of object you will work with in R. This makes sense since R is a statistical computing language at its core, so handling spreadsheet-like data is something it should be good at. The `periodic_table` data set is stored as a data frame. As you can see there are many different types of variables in this data set. We can get a glimpse as to the types and some of the values of the variables by using the `str` function in the R Console:

```
str(periodic_table)
```

Each of the names of the variables/columns in the data frame are listed immediately after the `$`. Then on each row of the `str` function call after the `:` we see what type of variable it is. For this `periodic_table` data set, we have four types: `int`, `chr`, `num`, and `logi`:

- `int` corresponds to integer values
- `chr` corresponds to character string values
- `num` corresponds to numeric (not necessarily integer) values
- `logi` corresponds to logical values (`TRUE` or `FALSE`)

5.3.2 Vectors

Data frames are most commonly just many vectors put together into a single object. Our `periodic_table` data frame has each row correspond to a chemical element and each column correspond to a different measurement or characteristic of that element. There are many different ways to create a vector that stands on its own outside of a data frame.

Using the c function

If you would like to list out many entries and put them into a vector object, you can do so via

the `c` function. If you enter `?c` in the R Console, you can gain information about it. The "c" stands for combine or concatenate.

Suppose we wanted to create a way to store four names:

```r
friend_names <- c("Bertha", "Herbert", "Alice", "Nathaniel")
friend_names
```

```
## [1] "Bertha"    "Herbert"    "Alice"    "Nathaniel"
```

You can see when `friend_names` is outputted that there are four entries to it. This is vector is known as a **strings** vector since it contains character strings. You can check to see what type an object is by using the `class` function:

```r
class(friend_names)
```

```
## [1] "character"
```

Next suppose we wanted to put the ages of our friends in another vector. We can again use the `c` function:

```r
friend_ages <- c(25L, 37L, 22L, 30L)
friend_ages
```

```
## [1] 25 37 22 30
```

```r
class(friend_ages)
```

```
## [1] "integer"
```

Note the use of the `L` value here. This tells R that the numbers entered have no decimal components. If we didn't designate the `L` we can see that the values are read in as `"numeric"` by default:

```r
ages_numeric <- c(25, 37, 22, 30)
class(ages_numeric)
```

```
## [1] "numeric"
```

There is not a huge difference in how these values are stored from a user's perspective, but it is a good habit to specify what class your variables are whenever possible to help with collaboration and documentation.

Using the seq function

The most likely way you will enter character values into a vector is via the `c` function. Numeric values can be entered in a couple different ways. You saw the first way using the `c` function above. Since numbers have an ordering to them, we can also specify a sequence of numbers with some starting value, ending value, and how much we'd like to increment each step in the sequence:

```r
sequence_by_2 <- seq(from = 0L, to = 100L, by = 2L)
sequence_by_2
```

```
## [1]    0   2   4   6   8  10  12  14  16  18  20  22  24  26  28  30  32
## [18]  34  36  38  40  42  44  46  48  50  52  54  56  58  60  62  64  66
## [35]  68  70  72  74  76  78  80  82  84  86  88  90  92  94  96  98 100
```

```
class(sequence_by_2)
```

```
## [1] "integer"
```

You now might have a better sense to what the numbers in the [ ] before the output refer to. This is done to help you keep track of where you are in the printing of the output. So the first element denoted by [1] is 0, the 18th entry ([18]) is 34, and the 35th entry ([35]) is 68. This will serve as a nice introduction into indexing and subsetting in Section 5.5.

We can also set the sequence to go by a negative number or a decimal value. We will do both in the next example.

```
dec_frac_seq <- seq(from = 10, to = 3, by = -0.2)
dec_frac_seq
```

```
##  [1] 10.0  9.8  9.6  9.4  9.2  9.0  8.8  8.6  8.4  8.2  8.0  7.8  7.6
## [14]  7.4  7.2  7.0  6.8  6.6  6.4  6.2  6.0  5.8  5.6  5.4  5.2  5.0
## [27]  4.8  4.6  4.4  4.2  4.0  3.8  3.6  3.4  3.2  3.0
```

```
class(dec_frac_seq)
```

```
## [1] "numeric"
```

Using the : operator

A short-cut version of the `seq` version can be achieved using the : operator. If we are increasing values by 1 (or -1), we can use the : operator to build our vector:

```
inc_seq <- 98:112
inc_seq
```

```
##  [1]  98  99 100 101 102 103 104 105 106 107 108 109 110 111 112
```

```
dec_seq <- 5:-5
dec_seq
```

```
##  [1]  5  4  3  2  1  0 -1 -2 -3 -4 -5
```

Combining vectors into data frames

If you aren't reading in data from a file and you have some vectors of information you'd like combined into a single data frame, you can use the `data.frame` function to do so:

```
friends <- data.frame(names = friend_names,
                       ages = friend_ages,
                       stringsAsFactors = FALSE)
friends
```

```
##         names ages
## 1    Bertha   25
## 2   Herbert   37
## 3     Alice   22
## 4 Nathaniel   30
```

Here we have created a `names` variable in the `friends` data frame that corresponds to the values in the `friend_names` vector and similarly an `ages` variable in `friends` that corresponds to the values in `friend_ages`.

5.3.3   Factors

If we have a strings vector/variable that has some sort of natural ordering to it, it frequently makes sense to convert that vector/variable into a **factor**. The factor will convert the strings to integers to keep track of which order you'd prefer and also keep track of the original string values as well.

Looking over our `periodic_table` data frame again via `View(periodic_table)`, you can see some good candidates for shifting from `chr` to `Factor`. If you remember your chemistry, you'll know that the natural ordering of the `block` variable is `"s"`, `"p"`, `"d"`, and `"f"`.

By default, R will organize character strings in alphabetical order. To see this, we'll introduce two new features: the `table` function and the `$` operator.

```
table(periodic_table$block)
```

```
##
##  d  f  p  s
## 40 28 36 14
```

We see here a count of the number of elements that appear in each block. But as I said, the ordering is off. You may remember the `$` as appearing before the variable names in the `str` function. That wasn't a coincidence. To access specific variables inside a data frame we can do so by entering the name of the data frame followed by `$` and lastly by the name of the variable. (Note here that spaces in variable names will not work. You'll learn that the hard way as I have more than likely.)

To convert `block` into a factor, we use the aptly named `factor` function. Note that this is "converting" by assigning the result of `factor` back to `block` in `periodic_table`.

```
periodic_table$block <- factor(periodic_table$block,
                        levels = c("s", "p", "d", "f"))
```

```
table(periodic_table$block)
```

```
##
##  s  p  d  f
## 14 36 40 28
```

You'll find that this is an easy way to organize your data whenever you'd like to summarize it or to plot it, but we'll save that discussion for a different time and a different book.

5.4   Vectorized operations

R can work extremely quickly when handed a vector or a collection of vectors like a data frame. Instead of walking through each element to perform an operation that we might need to do in other older programming languages, we can do something like this:

```
five_years_older <- ages_numeric + 5L
five_years_older
```

```
## [1] 30 42 27 35
```

Like that, we have ages that are five more than where we started. This extends to adding two vectors together[1].

5.5   Indexing and subsetting

So we have a big data frame of information about the periodic table, but what if we wanted to extract smaller pieces of the data frame? You already saw that to focus on any specific variable we can use the $ operator.

5.5.1   Using [ ] with a vector/variable

Recall the use of [ ] when a vector was printed to help us better understand where we are in printing a large vector. We will use this same tool to select the tenth to the twentieth elements of the `periodic_table$name` variable:

```
periodic_table$name[10:20]
```

```
##  [1] "Neon"       "Sodium"     "Magnesium"  "Aluminium"  "Silicon"
##  [6] "Phosphorus" "Sulfur"     "Chlorine"   "Argon"      "Potassium"
## [11] "Calcium"
```

Similarly if we'd only like to select a few elements from our `friend_names` vector, we can specify the entries directly:

```
friend_names[c(1, 3)]
```

```
## [1] "Bertha" "Alice"
```

We can also use - to select everything but the elements listed after it:

```
friend_names[-c(2, 4)]
```

```
## [1] "Bertha" "Alice"
```

5.5.2   Using [,] with a data frame

We have now seen how to select specific elements of a vector or a variable but what if we wanted a subset of the values in the larger data frame across both rows (observations) and columns (variables). We can use [,] where the spot before the comma corresponds to rows

[1] Vectors of the same size, of course…well, actually R has a way of dealing with vectors of different sizes and not giving errors, but let's ignore that for now.

and the spot after the comma corresponds to columns. Let's pick rows 40 to 50 and columns 1, 2, and 4 from `periodic_table`:

```
periodic_table[41:50, c(1, 2, 4)]
```

```
##    atomic_number symbol
## 41            41     Nb
## 42            42     Mo
## 43            43     Tc
## 44            44     Ru
## 45            45     Rh
## 46            46     Pd
## 47            47     Ag
## 48            48     Cd
## 49            49     In
## 50            50     Sn
##                                                         name_origin
## 41                 Niobe, daughter of king Tantalus from Greek mythology
## 42                            the Greek molybdos meaning 'lead'
## 43                      the Greek tekhn??tos meaning 'artificial'
## 44                       Ruthenia, the New Latin name for Russia
## 45                      the Greek rhodos, meaning 'rose coloured'
## 46 the then recently discovered asteroid Pallas, considered a planet at the time
## 47                                  English word (argentum in Latin)
## 48                          the New Latin cadmia, from King Kadmos
## 49                                                          indigo
## 50                                  English word (stannum in Latin)
```

5.5.3   Using logicals

As you've seen we can specify directly which elements we'd like to select based on the integer values of the indices of the data frame. Another way to select elements is by using a logical vector:

```
friend_names[c(TRUE, FALSE, TRUE, FALSE)]
```

```
## [1] "Bertha" "Alice"
```

This can be extended to choose specific elements from a data frame based on the values in the "cells" of the data frame. A logical vector like the one above (`c(TRUE, FALSE, TRUE, FALSE)`) can be generated based on our entries:

```
friend_names == "Bertha"
```

```
## [1]  TRUE FALSE FALSE FALSE
```

We see that only the first element is set to `TRUE` as we suspected since `"Bertha"` is the first entry in the vector. We, thus, have another way of subsetting to choose only those names that are `"Bertha"` or `"Alice"`:

```
friend_names[friend_names %in% c("Bertha", "Alice")]
```

```
## [1] "Bertha" "Alice"
```

The `%in%` operator looks element-wise in the `friend_names` vector and then tries to match each entry with the entries in `c("Bertha", "Alice")`.

Now we can think about how to subset an entire data frame using the same sort of creation of two logical vectors (one for rows and one for columns):

```
periodic_table[ (periodic_table$name %in% c("Hydrogen", "Oxygen") ),
                c("atomic_weight", "state_at_stp")]
```

```
##   atomic_weight state_at_stp
## 1      1.008235          Gas
## 8     15.999000          Gas
```

The extra parentheses around `periodic_table$name %in% c("Hydrogen", "Oxygen")` are a good habit to get into as they ensure everything before the comma is used to select specific rows matching that condition. For the columns, we can specify a vector of the column names to focus on only those variables. The resulting table here gives the `atomic_weight` and `state_at_stp` for `"Hydrogen"` and then for `"Oxygen"`.

There are many more complicated ways to subset a data frame and one can use the `subset` function built into R, but in my experience it is even easier to use the `filter` and `select` function in the `dplyr` package whenever you want to do anything more complicated than what we have done here.

5.6   Functions

We've been using **functions** throughout this entire chapter and you might not have even noticed it. The `seq` command we saw earlier is a function. It expects a few arguments: `from`, `to`, `by`, and a few others that we didn't specify. How do I know this and why didn't we specify them?

Recall that you can look up the help documentation on any function by entering `?` and the function name in the R console. If we do this for `seq` with `?seq`, we are given some examples of what to expect under the **Usage** section. R allows for function arguments to take on default values and that's what we see:

```
seq(from = 1, to = 1, by = ((to - from)/(length.out - 1)),
    length.out = NULL, along.with = NULL, ...)
```

By default, the sequence will both start and end at 1 (a not very interesting sequence). The `length.out` and `along.with` arguments are specified to `NULL` by default. `NULL` represents an empty object in R, so they are essentially ignored unless the person using the `seq` function specifies values for them. The `...` argument is beyond the scope of this book but you can read more about it and other useful tips about writing function at the NiceRCode page here.

Not all functions have all default arguments like `seq`. The `mean` function is one such example:

```r
mean()
```

```
Error in mean.default() : argument "x" is missing, with no default
Calls: <Anonymous> ... withVisible -> eval -> eval -> mean -> mean.default
Execution halted
```

```
Exited with status 1.
```

Notice that an error is given here, where one is not given if you don't specify the arguments to `seq`:

```r
seq()
```

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

To fix the error, we'll need to specify which vector/object we'd like to compute the mean of. Recall the `ages_numeric` vector. We can pass that into the `mean` function:

```r
ages_numeric
```

```
## [1] 25 37 22 30
```

```r
mean(x = ages_numeric)
```

```
## [1] 28.5
```

We can also skip specifying the name of the argument if you follow the same order as what is given in **Usage** in the documentation:

```r
mean(ages_numeric)
```

```
## [1] 28.5
```

We can see that R expects the arguments to be `x`, then `trim`, and then `na.rm`. What happens if we try to specify `TRUE` for `na.rm` without specifically saying `na.rm = TRUE`?

```r
mean(ages_numeric, TRUE)
```

```
Error in mean.default(ages_numeric, TRUE) :
  'trim' must be numeric of length one
Calls: <Anonymous> ... withVisible -> eval -> eval -> mean -> mean.default
Execution halted
```

```
Exited with status 1.
```

Since `trim` comes before `na.rm` in the list if we don't specify our name it will think the second argument is for `trim`. `trim` expects a fraction between 0 and 0.5 though so R barks at us that it doesn't understand. It usually is good practice for beginners to enter the name of the argument and then an equals sign and then the value you'd like the argument to take on. Something like what follows is clean and helps with readability:

```r
mean(x = ages_numeric, na.rm = TRUE)
```

```
## [1] 28.5
```

Why do some arguments require quotations and others don't?

As you begin to explore help documentation for different functions, you'll begin to notice that some arguments require quotations around them while others (like those for `mean` don't). This brings us back to the discussion earlier about strings, logicals, and numeric/integer classes.

An example of a function requiring a character string (or vector) is the `install.packages` function, which is at the heart of R's ability to expand on its built-in functionality by importing **packages** that include functions, templates, and data and are written by users of R. If you run `?install.packages` you'll see that the first argument `pkgs` is expected to be a character vector. You'll therefore need to enter the packages you'd like to download and install inside quotes.

Two useful packages that I recommend you install and download are `"ggplot2"` and `"dplyr"`. (Hopefully your instructor/server administrator has already done so for you if you are using an RStudio Server.) This function has a large number of arguments with all but `pkgs` set by default. We'll pick a couple here to specify instead of using the defaults:

```
install.packages(pkgs = c("ggplot2","dplyr"),
                 repos = "http://cran.rstudio.org",
                 dependencies = TRUE,
                 quiet = TRUE)
```

You'll notice looking through the help via `?install.packages` that what type of argument is expected is given:

- `pkgs` expects a character vector
- `repos` expects a character vector
- `dependencies` expects a logical value (`TRUE` or `FALSE`)
- `quiet` expects a logical value

After you've downloaded the packages, you'll need to load the package into your R environment using the `library` function:

```
library("ggplot2")
library("dplyr")
```

5.7 Closing thoughts

There are plenty more advanced analyses to be done with R. This chapter serves as a way to get you started with R without digging in too much. I encourage you to review this chapter frequently as you learn to use R. Quiz yourself on what a specific command does and see if you are correct. Breaking R is a pretty hard thing to do. Play around and try to figure out error messages on your own first for 15 minutes or so. If you are still not sure what is going on, check out some of the help with errors in the next chapter.

6

Deciphering Common R Errors

For references on errors, check out the following two links by Noam Ross here and David Smith here.

6.1 `Error: could not find function`

This error usually occurs when a package has not been loaded into R via `library`. R does not know where to find the specified function. It's a good habit to use the `library` functions on all of the packages you will be using in the top R chunk in your R Markdown file, which is usually given the chunk name `setup`.

6.2 `Error: object not found`

This error usually refers to your R Markdown document having a chunk that refers to an object that has not been defined in an R chunk at or before that chunk. You'll frequently see this when you've forgotten to copy code from your R Console sandbox back into a chunk in R Markdown.

6.3 Misspellings

One of the most frustrating errors you can do in R is by misspelling the name of an object or function. R is not forgiving on this and it won't try to automatically figure out what you are referring to. You'll usually be able to quite easily figure out that you made a typo because you'll receive an `object not found` error.

Remember that R is also case-sensitive so if you called an object `Name` and then try to call it `name` later on without `name` being defined, you'll receive an error.

6.4   Unmatched parenthesis

Another common error is forgetting/neglecting to finish a call to a function with a closing).
An example of this follows:

```r
mean(x = c(1, 5, 10, 52)
```

```
Error in parse(text = x, srcfile = src) :
  <text>:2:0: unexpected end of input
1: mean(x = c(1, 5, 10, 52)
   ^
Calls: <Anonymous> ... evaluate -> parse_all -> parse_all.character -> parse
Execution halted
```

```
Exited with status 1.
```

There needs to be one more parenthesis added at the end of your call to `mean`:

```r
mean(x = c(1, 5, 10, 52))
```

6.5   General guidelines

Try your best to not be intimidated by R errors. Oftentimes, you will find that you are able
to understand what they mean by carefully reading over them. When you can't, carefully look
over your R Markdown file again. You might also want to clear out all of your R environment
and start at the top by running the chunks. Remember to only include what you deem your
reader will need to follow your analysis.

Even people that have worked with R and programmed for years still use Google and support
websites like Stack Overflow asking for help with their R errors or when they aren't sure how
to do something in R. I think you'll be pleasantly surprised at just how much support is out
there.

7
Concluding Remarks

I hope that this book provides a nice gateway into understanding how statisticians, data scientists, and many other professionals use RStudio and R Markdown to simplify their analyses and to ensure that their reports are computationally reproducible. Learning R is nowhere near as intimidating as it used to be and more and more industries are shifting towards free open-source tools like R and RStudio. R Markdown provides an excellent way to document your analyses and share it with others in a variety of formats.

Of course, this book is just the tip of the iceberg in terms of showing you what R can really do. If you'd like to learn more I encourage you to check out Albert Kim and I's online, free, open-source book on using modern data analysis techniques and visualization with R, RStudio, and R Markdown. More details on that project will be coming soon.

Additionally, Garrett Grolemund's "Hands-On Programming with R" (Grolemund, 2014) is an excellent resource and goes into more depth than I do here as to how to work with more complicated objects in R. It also discusses concepts in a project-based framework that is entertaining and easy-to-read.

As always, feel free to send me an email at `chester.ismay@gmail.com` if you'd like any further clarification or if you have suggestions on improvements. Thanks for taking the time to read through this and best wishes to you on your next steps towards reproducible, thoughtful, beautiful analyses!

– Chester

References

Grolemund, G. (2014). *Hands-On Programming with R*. O'Reilly.