Introduction to R for Natural Resource Scientists

Ben Staton

with contributions by Henry Hershey

Table of Contents

# Overview

This book is intended to be a first course in R programming for natural resource professionals. It is by no means comprehensive (no book about R ever could be), but instead attempts to introduce the main topics needed to get a beginner up and running with applying R to their own work. It is intended to be a companion to in-person workshop sessions, in which each chapter is covered in a 2 hour session, however it can be used as “self-teach” manual as well. Although the examples shown have a natural resource/ecological theme, the general skills presented are general to R users across all scientific disciplines.

## What is Covered?

The book is composed of six chapters intended to cover a suite of topics in introductory R programming. In general, the material builds in complexity from chapter to chapter and earlier chapters can be seen as prerequisites for later chapters.

* **Chapter 1** covers the basics of working in R through RStudio, including the basics of the R coding language and environment.
* **Chapter ??** covers the basics of plotting using the base R graphics functionality.
* **Chapter ??** covers the basics of fitting statistical models using built-in functionality for generalized linear models as well as non-linear models.
* **Chapter ??** covers the basics of simulation modeling in R.
* **Chapter ??** covers the basics of the {dplyr} and {reshape2} packages for manipulating and summarizing large data sets using highly readable code.
* **Chapter ??** covers the basics of producing maps and performing spatial analysis in R. *This chapter was contributed by Henry Hershey*

## Prerequisites

Chapter 1 starts at the first step (installing R) and progresses by assuming no prior knowledge of programming in R or in any other language. In the later chapters, e.g., Chapters ?? and ??, an understanding of statistics at the introductory undergraduate level would be helpful but not strictly essential.

There are, however, some tasks you’ll need to complete before using this book, which are described in the two sections that follow.

### Prepare Your Computer

#### Installation

First off, you will need to get R and RStudio[[1]](#footnote-25) onto your computer. Go to:

* <https://cran.rstudio.com/> to get R and
* <https://www.rstudio.com/products/rstudio/download/> to get RStudio Desktop.

Download the appropriate installation file for your operating system and run that file. All default settings should be fine.

#### Optional Configuration

As a matter of personal preference, you are recommended to configure a few settings. Open up RStudio and go to *Tools > Global Options*, and in the section listed “General”:

* Make sure *Restore .RData into workspace at startup* is **unchecked**
* Make sure *Save workspace to .RData on exit* is set to **Never**
* Make sure *Always save history (even when not saving .RData)* is **unchecked**.

These settings will prevent you from getting a bunch of useless files and dialog boxes every time you open and close R.

#### Create a Book Directory

You should create a devoted folder on your computer for this book. All examples will assume this folder is located here: C:/Users/YOU/Documents/R-Book. Change YOU to be specific for your computer.

#### Data Sets

The data sets[[2]](#footnote-31) used in this book are hosted on a GitHub repository maintained by the author. It is located here: <https://github.com/bstaton1/au-r-workshop-data>.

To acquire the data for this book, you should:

1. Navigate to the GitHub repository
2. click the green *Clone or download* button at the top right,
3. click *Download ZIP*
4. unzip the contents of this folder into the location: C:/Users/YOU/Documents/R-Book/Data

File organization will be very important for your success in learning to use R. This book will assume your R-Book directory is organized as shown below. Notice that there is a separate folder for the data downloaded from GitHub as well as one for each chapter that will house the R code for that chapter. Do not worry about making all of these folders now, you will do this at the appropriate time as you work your way through this book. For now, just make sure there is a Data folder that contains all of the unzipped contents from the GitHub repository your main R-Book directory.

## levelName  
## 1 R-Book   
## 2 ¦--Data   
## 3 ¦ ¦--asl.csv  
## 4 ¦ ¦--...   
## 5 ¦ °--Ch6   
## 6 ¦--Chapter1   
## 7 ¦ ¦--Ch1.R   
## 8 ¦ ¦--Ex1A.R   
## 9 ¦ °--Ex1B.R   
## 10 ¦--Chapter2   
## 11 ¦ ¦--Ch2.R   
## 12 ¦ °--Ex2.R   
## 13 ¦--Chapter3   
## 14 ¦--Chapter4   
## 15 ¦--Chapter5   
## 16 °--Chapter6

## Exercises

Following each chapter, there is a set of exercises. You should attempt and complete them, as they give you an opportunity to practice what you learned while reading and typing along. Solutions are provided at the end of this book, however you are **strongly** recommended to attempt to figure the problems out on your own before looking to how the author would solve them.

Some exercises have bonus questions. These are intended to challenge you with some of the more difficult tasks shown in the chapter or ask you to extend what you learned to a completely different problem. If you can get all of the non-bonus questions without looking at the solutions too much, you can consider yourself to have good understanding of that chapter’s material. If you can complete the bonus questions with little or no help, that means you have mastered that chapter’s material!

## Text Conventions

* Regular text: a description of what you you should do, how some code works, or a general narrative of something.
* monospace: references something in R
  + this() references some function
  + this references some other object
  + {this} references an R package
  + C:/This is a file path
* **Bold** is intended to provide more emphasis to a word or topic. In general, new topics are introduced this way.
* [Links](#notation): this is a link to some other location in this book. External links are provided with a full URL.
* : it is sometimes useful to describe concepts mathematically before showing how to do it in R.
* [[3]](#footnote-35): a footnote containing more information.

## Keyboard Shortcuts

Several parts of this book in this book make reference to keyboard shortcuts. They are never necessary, but can help you be more efficient if you commit them to muscle memory. This book assumes you are using a PC for the keyboard shortcuts. If you are using a Mac, they will be different[[4]](#footnote-37). For a complete list of RStudio’s keyboard shortcuts specific to your operating system, go to *Help > Keyboard Shortcuts Help*.

## Development of this Book

This book represents the third reincarnation of the Auburn R Workshop Series. The first version was written in Fall 2014 using Microsoft Word, but the author found that making even small changes was clunky - each change to code in the document required a copy-paste of code and output from R to Word. Individual session materials (i.e., handout, exercises, solutions, data) were created in separate documents, saved as PDFs and .xlsx files, and uploaded to a wordpress webpage.

The second version was written through R (R Core Team [2018](#ref-R-base)) and RStudio using the R packages {rmarkdown} (Allaire et al. [2018](#ref-R-rmarkdown)) and {knitr} (Xie [2018b](#ref-R-knitr), [2015](#ref-knitr-cite)), which allowed the integration of text, code, and output all into one output file. This version was completed in Fall 2015. Like the first version, individual session materials were created in separate documents, and replaced those previously found on the wordpress site.

This third version was written through R and RStudio but used the R package {bookdown} (Xie [2018a](#ref-R-bookdown)) which allowed for the individual sessions to be combined into one “book” by turning each session into a chapter. This facilitated cross-references to topics covered in previous chapters and allows the reader to only refer to one location when trying to remember how to use a skill. It also allowed for multiple formats to be published including both HTML and PDF versions.

The book is hosted on [GitHub Pages](https://pages.github.com/), and was last built on 10-31-2019.

## About the Author

Ben Staton is a PhD candidate in the School of Fisheries at Auburn University. He studies quantitative methods for assessing fish populations for use in harvest management, with a focus on Pacific salmon in western Alaska. His interests are in population dynamics, Bayesian methods, Monte Carlo methods, and reproducible research. Ben has been using R on a daily basis since the beginning of his graduate work in 2014, and is enthusiastic about helping others learn to use R for their own work.

# Introduction to the R Environment

## Chapter Overview

In this first chapter, you will get familiar with the basics of using R. You will learn:

* how to use R as a basic calculator
* some basic object types
* some basic data classes
* some basic data structures
* how to read in data
* how to produce basic data summaries
* how to write out data
* how to write your own functions

## Before You Begin

Before you start this chapter, you should make sure you have read and done everything in the [Overview](#overview) pertaining to [preparing your computer](#comp-prep) for working through this book.

## The R Studio Interface

When you open up RStudio for the first time, you will see three panes: the left-hand side is the **console** where results from executed commands are printed, and the two panes on the right are for additional information to help you code more efficiently - don’t worry too much about what these are at the moment. For now, focus your attention on the console.

### Write Some Simple Code

To start off, you will use R as a simple calculator. Type these commands (not the lines with ##, those are output[[5]](#footnote-46)) one at a time and hit **CTRL + ENTER** to run it. The spaces don’t matter at all, they are used here for clarity and for styling. To learn more about standard R code styling, check out the section in Wickham ([2015](#ref-adv-r-cite)) on it[[6]](#footnote-47).

3 + 3

## [1] 6

12/4

## [1] 3

Notice that when you run each line, it prints the command and the output to the console.

R is an **object oriented language**, which means that you fill objects with data and do things with them. Make an object called x that stores the result of the calculation 3 + 3 (type this and run using **CTRL + ENTER**):

x = 3 + 3

Notice that running this line did not return a value as before. This is because in that line you are **assigning** a value to the object x. You can view the contents of x by typing its name alone and running just that:

x

## [1] 6

When used this way, the = sign denotes assignment of the value on the right-hand side to an object with the name on the left-hand side. The <- serves this same purpose so in this context the two are interchangeable:

y <- 2 + 5

You can highlight smaller sections of a line to run as well. For example after creating y above, press the **up arrow** to see the line you just ran, highlight just the y, and press **CTRL + ENTER**. From this point forward, the verb “run” means execute some code using **CTRL + ENTER**.

You can use your objects together to make a new object:

z = y - x

**Here are some things to note about object names:**

* Object names can contain any of the following:
  + letters
  + numbers
  + the . or \_ symbols
* Object names must start with a letter, not a number or symbol and cannot contain spaces
* As a general rule, avoid naming your objects things that already have names in R, e.g., data(), mean(), sum(), sd(), etc.
* Capitalization matters: A and a are two different objects
* Keep your names short with abbreviations or shorthand

## Saving Your Code: Scripts

If you closed R at this moment, your work would be lost. Running code in the console like you have just done **does not save a record of your work**. To save R code, you must use what is called a **script**, which is a plain-text file with the extension .R. To create a new script file, go to *File > New File > R Script*, or use the keyboard shortcut **CTRL + SHIFT + N**. A new pane will open called the **source** pane - this is where you will edit your code and save your progress. R Scripts are a key feature of reproducible research with R, given that if they are well-written they can present a complete road map of your statistical analysis and workflow.

## The Working Directory

Keeping things organized is essential to efficient use of R. For this book, you should have a separate subfolder for your scripts in each chapter. Create a subfolder called C:/Users/YOU/Documents/R-Book/Chapter1 and save your Ch1.R script there.

Part of keeping your work organized in R is making sure you know where R is looking for your files. One way to facilitate this is to use a **working directory**. This is the location (i.e., folder) on your computer that your current R session will “talk to” by default. R will read files from and write files to the working directory by default. Because you’ll likely be visiting it often, it should probably be somewhere that is easy to remember and not too deeply buried in your computer’s file system.

To set the working directory to C:/Users/YOU/Documents/R-Book/Chapter1, you have three options:

1. **Go to Session > Set Working Directory > Source File Location**. This will set the working directory to the location of the file that is currently open in your source pane.
2. **Go to Session > Set Working Directory > Choose Directory**. This will open an interactive file selection window to allow you to navigate to the desired directory.
3. **Use code**. In the console, you can type setwd("C:/Users/YOU/Documents/R-Book/Chapter1"). If at any point you want to know where your current working directory is set to, you can either look at the top of the console pane, which shows the full path or by running getwd() in the console. Note the use of / rather than \ for file paths in R. If you are using a Mac, omit C: from your directory name.

**The main benefits of using a working directory are**:

* Files are read from and written to a consistent and predictable place every time
* Everything for your analysis is organized into one place on your computer
* You don’t have to continuously type file paths to your work. If file.txt is a file in your current working directory, you can reference it your R session using "file.txt" rather than with "C:/Users/YOU/Documents/R-Book/Chapter1/file.txt" each time.

Note that while it is generally good practice to keep your data in the working directory, this is not recommended for this book. You will be using the same data files in multiple chapters, so it will help if they are all stored in one location. More details on this later (Section 1.8).

## R Object Types

R has a variety of object types that you will need to become familiar with.

### Functions

Much of your work in R will involve functions. A function is called using the syntax:

fun(arg1 = value1, arg2 = value2)

Here, fun() is the **function name** and arg1 and arg2 are called **arguments**. Functions take input in the form of the arguments, do some task with them, then return some output. The parentheses are a sure sign that fun() is a function.

The syntax above passes the function two arguments by name: all functions have arguments, all arguments have names, and there is always a default order to the arguments. If you memorize the argument order of functions you use frequently, you don’t have to specify the argument names:

fun(value1, value2)

would give the same result as the command above in which the argument names were specified.

Here’s a real example:

print(x = z)

## [1] 1

The function is print(), the argument is x, and the value you have supplied the argument is the object z. The task that print() does is to print the value of z to the console.

R has a ton of built-in documentation to help you learn how to use a function. Take a look at the help file for the mean() function. Run ?mean in the console: a window on the right-hand side of the R Studio interface should open. The help file tells you what goes into a function and what comes out. For more complex functions it also tells you what all of the options (i.e., arguments) can do. Help files can be a bit intimidating to interpret at first, but they are all organized the same and once you learn their layout you will know where to go to find the information you’re looking for.

### Vectors

Vectors are one of the most common data structures. A vector is a set of numbers going in only one dimension. Each position in a vector is called an **element**, and the number of elements is called the **length** of the vector. Here are some ways to make some vectors with different elements, all of length five:

# this is a comment. R will ignore all text on a line after a #  
# the ; means run everything after it on a new line  
  
# count up by 1  
month = 2:6; month

## [1] 2 3 4 5 6

# count up by 2  
day = seq(from = 1, to = 9, by = 2); day

## [1] 1 3 5 7 9

# repeat the same number (repeat 2018 5 times)  
year = rep(2018, 5); year

## [1] 2018 2018 2018 2018 2018

The [1] that shows up is an element position, more on this later (see Section 1.7). If you wish to know how many elements are in a vector, use length():

length(year)

## [1] 5

You can also create a vector “by-hand” using the c() function[[7]](#footnote-54):

# a numeric vector  
number = c(4, 7, 8, 10, 15); number

## [1] 4 7 8 10 15

# a character vector  
pond = c("F11", "S28", "S30", "S8", 'S11'); pond

## [1] "F11" "S28" "S30" "S8" "S11"

Note the difference between the numeric and character vectors. The terms “numeric” and “character”" represent **data classes**, which specify the type of data the vector is holding:

* A **numeric vector** stores numbers. You can do math with numeric vectors
* A **character vector** stores what are essentially letters. You can’t do math with letters. A character vector is easy to spot because the elements will be wrapped with quotes[[8]](#footnote-55).

A vector can only hold one data class at a time:

v = c(1,2,3,"a"); v

## [1] "1" "2" "3" "a"

Notice how all the elements now have quotes around them. The numbers have been **coerced** to characters[[9]](#footnote-56). If you attempt to calculate the sum of your vector:

sum(v)

## Error in sum(v): invalid 'type' (character) of argument

you would find that it is impossible in its current form.

### Matrices

Matrices act just like vectors, but they have two dimensions, i.e., they have both rows and columns. An easy way to make a matrix is by combining vectors you have already made:

# combine vectors by column (each vector will become a column)  
m1 = cbind(month, day, year, number); m1

## month day year number  
## [1,] 2 1 2018 4  
## [2,] 3 3 2018 7  
## [3,] 4 5 2018 8  
## [4,] 5 7 2018 10  
## [5,] 6 9 2018 15

# combine vectors by row (each vector will become a row)  
m2 = rbind(month, day, year, number); m2

## [,1] [,2] [,3] [,4] [,5]  
## month 2 3 4 5 6  
## day 1 3 5 7 9  
## year 2018 2018 2018 2018 2018  
## number 4 7 8 10 15

Just like vectors, matrices can hold only one data class (note the coercion of numbers to characters):

cbind(m1, pond)

## month day year number pond   
## [1,] "2" "1" "2018" "4" "F11"  
## [2,] "3" "3" "2018" "7" "S28"  
## [3,] "4" "5" "2018" "8" "S30"  
## [4,] "5" "7" "2018" "10" "S8"   
## [5,] "6" "9" "2018" "15" "S11"

Each vector should have the same length.

### Data Frames

Many data sets you will work with require storing different data classes in different columns, which would rule out the use of a matrix. This is where **data frames** come in:

df1 = data.frame(month, day, year, number, pond); df1

## month day year number pond  
## 1 2 1 2018 4 F11  
## 2 3 3 2018 7 S28  
## 3 4 5 2018 8 S30  
## 4 5 7 2018 10 S8  
## 5 6 9 2018 15 S11

Notice the lack of quotation marks which indicates that all variables (i.e., columns) are stored as their original data class.

It is important to know what kind of object type you are using, since R treats them differently. For example, some functions can only use a certain object type. The same holds true for data classes (numeric vs. character). You can quickly determine what kind of object you are dealing with by using the class() function:

class(day); class(pond); class(m1); class(df1)

## [1] "numeric"

## [1] "character"

## [1] "matrix"

## [1] "data.frame"

## Factors

At this point, it is worthwhile to introduce an additional data class: factors. Notice the data class of the pond variable in df1:

class(df1$pond)

## [1] "factor"

The character vector pond was coerced to a factor when you placed it in the data frame. A vector with a **factor** class is like a character vector in that you see letters and that you can’t do math on it. However, a factor has additional properties: in particular, it is a grouping variable. See what happens when you print the pond variable:

df1$pond

## [1] F11 S28 S30 S8 S11  
## Levels: F11 S11 S28 S30 S8

A factor has levels, with each level being a subcategory of the factor. You can see the unique levels of your factor by running:

levels(df1$pond)

## [1] "F11" "S11" "S28" "S30" "S8"

Additionally, factor levels have an assigned order (even if the levels are totally nominal), which will become important in Chapter ?? when you learn how to fit linear models to groups of data, in which one level is the “reference” group that all other groups are compared to (see Section ?? for more details).

If you run into errors about R expecting character vectors, it may be because they are actually stored as factors. When you make a data frame, you’ll often have the option to turn off the automatic factor coercion. For example:

data.frame(month, day, year, number, pond, stringsAsFactors = F)  
read.csv("streams.csv", stringsAsFactors = F) # see below for details on read.csv

will result in character vectors remaining that way as opposed to being coerced to factors. This can be preferable if you are doing many string manipulations, as character vectors are often easier to work with than factors.

## Vector Math

R does vectorized calculations. This means that if supplied with two numeric vectors of equal length and a mathematical operator, R will perform the calculation on each pair of elements. For example, if you wanted to add the two vectors day and month, then you would just run:

dm = day + month; dm

## [1] 3 6 9 12 15

Look at the contents of both day and month again to make sure you see what R did. You typically should ensure that the vectors you are doing math with are of equal lengths.

You could do the same calculation to each element of a vector (e.g., divide each element by 2) with:

dm/2

## [1] 1.5 3.0 4.5 6.0 7.5

## Data Subsets/Queries

This is perhaps the most important and versatile skill to know in R. Say you have an object with data in it and you want to use it for analysis, but you don’t want the whole data set: just a few rows or just a few columns, or perhaps you need just a single element from a vector. This section is devoted to ways you can extract certain parts of data objects (the terms **query** and **subset** are often used interchangeably to describe this task). There are three main methods:

1. **By Index** – This method allows you to pull out specific rows/columns by their location in an object. However, you must know exactly where in the object the desired data are. An **index** is a location of an element in a data object, like the element position or the position of a specific row or column. The syntax for subsetting a vector by index is vector[element] and for a matrix it is matrix[row,column]. Here are some examples:

# show all of day, then subset the third element  
day; day[3]

## [1] 1 3 5 7 9

## [1] 5

# show all of m1, then subset the cell in row 1 col 4   
m1; m1[1,4]

## month day year number  
## [1,] 2 1 2018 4  
## [2,] 3 3 2018 7  
## [3,] 4 5 2018 8  
## [4,] 5 7 2018 10  
## [5,] 6 9 2018 15

## number   
## 4

# show all of df1, then subset the entire first column  
df1; df1[,1]

## month day year number pond  
## 1 2 1 2018 4 F11  
## 2 3 3 2018 7 S28  
## 3 4 5 2018 8 S30  
## 4 5 7 2018 10 S8  
## 5 6 9 2018 15 S11

## [1] 2 3 4 5 6

Note this last line: the [,1] says “keep all the rows, but take only the first column”.

Here is another example:

# show m1, then subset the 1st, 2nd, and 4th rows and every column  
m1; m1[c(1,2,4),]

## month day year number  
## [1,] 2 1 2018 4  
## [2,] 3 3 2018 7  
## [3,] 4 5 2018 8  
## [4,] 5 7 2018 10  
## [5,] 6 9 2018 15

## month day year number  
## [1,] 2 1 2018 4  
## [2,] 3 3 2018 7  
## [3,] 5 7 2018 10

Notice how you can pass a vector of row indices here to exclude the 3rd and 5th rows.

**2. By name** – This method allows you to pull out a specific column of data based on what the column name is. Of course, the column must have a name first. The name method uses the $ operator:

df1$month

## [1] 2 3 4 5 6

You can combine these two methods:

df1$month[3]

## [1] 4

# or  
df1[,c("year", "month")]

## year month  
## 1 2018 2  
## 2 2018 3  
## 3 2018 4  
## 4 2018 5  
## 5 2018 6

The $ method is useful because it can be used to add columns to a data frame:

df1$dm = df1$day + df1$month; df1

## month day year number pond dm  
## 1 2 1 2018 4 F11 3  
## 2 3 3 2018 7 S28 6  
## 3 4 5 2018 8 S30 9  
## 4 5 7 2018 10 S8 12  
## 5 6 9 2018 15 S11 15

1. **Logical Subsetting** – This is perhaps the most flexible method, but requires more explaining. It is described in Section 1.11.

## Exercise 1A

Take a break to apply what you’ve learned so far to enter the data found in Table ?? into R by hand and do some basic data subsets.

*The solutions to this exercise are found at the end of this book (*[*here*](#ex1a-answers)*). You are* ***strongly recommended*** *to make a good attempt at completing this exercise on your own and only look at the solutions when you are truly stumped.*

1. Create a new file in your working directory called Ex\_1A.R.
2. Enter these data into vectors. Call the vectors whatever you would like. Should you enter the data as vectors by rows, or by columns? (*Hint: remember the properties of vectors*).
3. Combine your vectors into a data frame. Why should you use a data frame instead of a matrix?
4. Subset all of the data from Small Lake.
5. Subset the area for all of the lakes.
6. Subset the number of fish for Big and Square lakes.
7. You realize that you sampled 209 fish at Square Lake, not 109. Fix the mistake. There are two ways to do this, can you think of them both? Which do you think is better?
8. Save your script. Close R and re-open your script to see that it was saved.

## Read External Data Files

It is rare that you will enter data by hand as you did in Exercise 1A. Often, you have a data set that you wish to analyze or manipulate that is stored in a spreadsheet. R has several ways to read information from data files and in this book, you will be using a common and simple method: reading in .csv files. .csv files are data files that separate columns with commas[[10]](#footnote-64). If your data are in a Microsoft Excel spreadsheet, you can save your spreadsheet file as a .csv file (*File > Save As > Save as Type> CSV (Comma Delimited)*). Several dialog boxes will open asking if you are sure you want to save it as a .csv file.

The syntax for reading in a .csv file is:

dat = read.csv("Path/To/FileName.csv")

Make sure your working directory is set to the location of your current file, which should be in the location C:/Users/YOU/Documents/R-Book/Chapter 1. According to the [instructions](#data-sets) on acquiring the data, you should have placed all of the data files for this book (downloaded from the GitHub repository) in the location C:/Users/YOU/Documents/R-Book/Data. If you have done this already and your working directory is set to the location of the Ch1.R script, you can simply run:

dat = read.csv("../Data/streams.csv")

The ../ tells R to look up one directory from the working directory for a folder called Data, then within that, look for a file called streams.csv.

If you do not get an error, congratulations! However, if you get an error that looks like this:

## Warning in file(file, "rt"): cannot open file 'streams.csv': No such file  
## or directory

## Error in file(file, "rt"): cannot open the connection

then fear not. This must be among the most common errors encountered by R users world-wide. It simply means the file you told R to look for doesn’t exist where you told R to find it. Here is a trouble-shooting guide to this error:

1. The exact case and spelling matters, as do the quotes and .csv at the end. Ensure the file name is typed correctly.
2. Check what files are in the path you are specifying: run dir("../Data"). This will return a vector with the names of the files located in the Data directory (which is one folder up from your working directory). Is the file you told R was there truly in there? Is your working directory set to where you thought it was?

A simpler method is to put the data file in your working directory, in which case it can be read into R using dat = read.csv("streams.csv"). This method is not recommended for this book, because you will be using the same data files in multiple chapters, and it will help if they are all located in the same location.

If you did not get any errors, then the data are in the object you named (dat) and that object is a data frame. Do not proceed until you are able to get read.csv to run successfully.

**A few things to note about reading in .csv files**:

* R will assume the first row are column headers by default.
* If there is a space in one of the header cells, a "." will be inserted. For example, the column header Total Length would become Total.Length.
* R brings in .csv files in as data frames by default.
* If a record (i.e., cell) is truly missing and you want R to treat it that way (i.e., as an NA), you have three options:
  + Hard code an NA into that cell in your .csv file
  + Leave that cell completely empty in your .csv file
  + Enter in some other character (e.g., ".") alone in all cells that are meant to be coded as NA in R and use the na.strings = "." argument of read.csv().
* If at some point you did “Clear Contents” in Microsoft Excel to delete rows or columns from your .csv file, these “deleted” rows/columns will be read in as all NAs, which can be annoying. To remove this problem, open the .csv file in Excel, then highlight and **delete** the rows/columns in question and save the file. Read it back into R again using read.csv().
* If even a single character is found in a numeric column in FileName.csv, the *entire column* will be coerced to a character/factor data class after it is read in (i.e., no more math with data on that column until you remove the character). A common error is to have a #VALUE! record left over from an invalid Excel function result. You must remove all of these occurrences in order to use that column as numeric. Characters include anything other than a number ([0-9]) and a period when used as a decimal. None of these characters: !?[]\/@#$%^&\*()<>\_-+=[a-z];[A-Z] should never be found in a column you wish to do math with (e.g., take the mean of that column). **This is an incredibly common problem!**

## Explore the Data Set

Have a look at the data. You could just run dat to view the contents of the object, but it will show the whole thing, which may be undesirable if the data set is large. To view the first handful of rows, run head(dat) or the last handful of rows with tail(dat).

You will now use some basic functions to explore the simulated streams data before any analysis. The summary() function is very useful for getting a coarse look at how R has interpreted the data frame:

summary(dat)

## state stream\_width flow   
## Alabama :5 Min. :17.65 Min. : 28.75   
## Florida :5 1st Qu.:46.09 1st Qu.: 65.50   
## Georgia :5 Median :61.80 Median : 95.64   
## Tennessee:5 Mean :60.88 Mean : 91.49   
## 3rd Qu.:79.34 3rd Qu.:120.55   
## Max. :94.65 Max. :149.54   
## NA's :1

You can see the spread of the numeric data and see the different levels of the factor (state) as well as how many records belong to each level. Note that there is one NA in the variable called flow.

To count the number of elements in a variable (or any vector), remember the length() function:

length(dat$stream\_width)

## [1] 20

Note that R counts missing values as elements as well:

length(dat$flow)

## [1] 20

To get the dimensions of an object with more than one dimension (i.e., a data frame or matrix) you can use the dim() function. This returns a vector with two elements: the first number is the number of rows and the second is the number of columns. If you only want one of these, use the nrow() or ncol() functions (but remember, only for objects with more than one dimension; vectors don’t have rows or columns!).

dim(dat); nrow(dat); ncol(dat)

## [1] 20 3

## [1] 20

## [1] 3

You can extract the names of the variables (i.e., columns) in the data frame using colnames():

colnames(dat)

## [1] "state" "stream\_width" "flow"

Calculate the mean of all the stream\_width records:

mean(dat$stream\_width)

## [1] 60.8845

Calculate the mean of all of the flow records:

mean(dat$flow)

## [1] NA

mean() returned an NA because there is an NA in the data for this variable. The way to tell R to ignore this NA is by including the argument na.rm = TRUE in the mean() function (separate arguments are always separated by commas). This is a **logical** argument, meaning that it asks a question. It says “do you want to remove NAs before calculating the mean?” TRUE means “yes” and FALSE means “no.” TRUE and FALSE can be abbreviated as T and F, respectively. Many of R’s functions have the na.rm argument (e.g. mean(), sd(), var(), min(), max(), sum(), etc. - most anything that collapses a vector into one number).

mean(dat$flow, na.rm = T)

## [1] 91.49053

which is the same as (i.e., the definition of the mean with the NA removed):

sum(dat$flow, na.rm = T)/(nrow(dat) - 1)

## [1] 91.49053

What if you need to apply a function to more than one variable at a time? One of the easiest ways to do this (though as with most things in R, there are many) is by using the apply() function. This function applies the same summary function to individual subsets of a data object at a time then returns the individual summaries all at once:

apply(dat[,c("stream\_width", "flow")], 2, FUN = var, na.rm = T)

## stream\_width flow   
## 581.1693 1337.3853

The first argument is the data object to which you want to apply the function. The second argument (the number 2) specifies that you want to apply the function to columns, 1 would tell R to apply it to rows. The FUN argument specifies what function you wish to apply to each of the columns; here you are calculating the variance which takes the na.rm = T argument. This use of apply() alone is very powerful and can help you get around having to write the dreaded for() loop (introduced in Chapter ??).

There is a whole family of -apply() functions, the base apply() is the most basic but a more sophisticated one is tapply(), which applies a function based on some grouping variable (a factor). Calculate the mean stream width **separated by state**:

tapply(dat$stream\_width, dat$state, mean)

## Alabama Florida Georgia Tennessee   
## 53.664 63.588 54.996 71.290

The first argument is the variable to which you want to apply the mean function, the second is the grouping variable, and the third is what function you wish to apply. Try to commit this command to memory given this is a pretty common task.

If you want a data frame as output, you can use the aggregate function to do the same thing:

aggregate(dat$stream\_width, by = list(state = dat$state), mean)

## state x  
## 1 Alabama 53.664  
## 2 Florida 63.588  
## 3 Georgia 54.996  
## 4 Tennessee 71.290

## Logical/Boolean Operators

To be an efficient and capable programmer in any language, you will need to become familiar with how to implement numerical logic, i.e., the Boolean operators. These are very useful because they always return a TRUE or a FALSE, off of which program-based decisions can be made (e.g., whether to operate a given subroutine, whether to keep certain rows, whether to print the output, etc.).

Define a simple object: x = 5 . Note that this will write over what was previously stored in the object x. Suppose you want to ask some questions of the new object x and have the answer printed to the console as a TRUE for “yes” and a FALSE for “no”. Below are the common Boolean operators and their usage in R.

#### Equality

To ask if x is exactly equal to 5, you run:

x == 5

## [1] TRUE

Note the use of the double == to denote equality as opposed to the single = as used in assignment (e.g., x = 5) or in argument specification (e.g., mean(dat$flow, na.rm = T)).

#### Inequalities

To ask if x is not equal to 5, you run:

x != 5

## [1] FALSE

The ! is the logical **not** in R, and can be used to flip the direction of any T to a F or *vice versa*.

To ask if x is less than 5, you run:

x < 5

## [1] FALSE

To ask if x is less than *or equal to* 5, you run:

x <= 5

## [1] TRUE

Greater than works the same way, though with the > symbol replaced.

#### In

Suppose you want to ask which elements of one vector are also found within another vector:

y = c(1,2,3)  
x %in% y

## [1] FALSE

# or  
y = c(4,5,6)  
x %in% y

## [1] TRUE

# or  
y %in% x

## [1] FALSE TRUE FALSE

%in% is a very useful operator in R that is not one of the traditional Boolean operators.

#### And

Suppose you have two conditions, and you want to know if **both are met**. For this you would use **and** by running:

x > 4 & x < 6

## [1] TRUE

which asks if x is between 4 and 6.

#### Or

Suppose you have two conditions, and you want to know if **either are met**. For this you would use **or** by running:

x <= 5 | x > 5

## [1] TRUE

which asks if x is less than or equal to 5 **or** greater than 5 - you would be hard-pressed to find a real number that did not meet these conditions!

## Logical Subsetting

A critical use of logical/Boolean operators is in the subsetting of data objects. You can use a logical vector (i.e., one made of only TRUE and FALSE elements) to tell R to extract only those elements corresponding to the TRUE records. For example:

# here's logical vector: TRUE everywhere condition met  
dat$stream\_width > 60

## [1] TRUE FALSE FALSE FALSE FALSE FALSE TRUE FALSE TRUE TRUE FALSE  
## [12] FALSE TRUE TRUE TRUE FALSE FALSE TRUE TRUE TRUE

# insert it to see only the flows for the TRUE elmements  
dat$flow[dat$stream\_width > 60]

## [1] 120.48 123.78 95.64 95.82 120.06 135.63 120.61 111.34 149.54 131.22

gives all of the flow values for which stream\_width is greater than 60.

To extract all of the data from Alabama, you would run:

dat[dat$state == "Alabama",]

## state stream\_width flow  
## 1 Alabama 81.68 120.48  
## 2 Alabama 57.76 85.90  
## 3 Alabama 48.32 73.38  
## 4 Alabama 31.63 46.55  
## 5 Alabama 48.93 80.91

To exclude all of the data from Alabama, you would run:

dat[dat$state != "Alabama",]

You will be frequently revisiting this skill throughout the workshop.

## if(), else, and ifelse()

You can tell R to do something if the result of a question is TRUE. This is a typical if-then statement:

if (x == 5) print("x is equal to 5")

## [1] "x is equal to 5"

This says “if x equals 5, then print the phrase ‘x is equal to 5’ to the console”. If the logical returns a FALSE, then this command does nothing. To see this, change the == to a != and re-run:

if (x != 5) print("x is equal to 5")

Notice that because x does equal 5, running this line has no effect.

You can tell R to do multiple things if the logical is TRUE by using curly braces:

if (x == 5) {  
 print("x is equal to 5")  
 print("you dummy, x is supposed to be 6")  
}

## [1] "x is equal to 5"  
## [1] "you dummy, x is supposed to be 6"

You can always use curly braces to extend code across multiple lines whereas it may have been intended to go on one line.

If you want R to do something if the logical is FALSE, you would use the else command:

if (x > 5) print("x is greater than 5") else print("x is not greater than 5")

## [1] "x is not greater than 5"

Or extend this same thing to multiple lines:

if (x > 5) {  
 print("x is greater than 5")  
} else {  
 print("x is not greater than 5")  
}

The if() function is useful, but it can only respond to one question at a time. If you supply it with a vector of length greater than 1, it will give a warning:

# vector from -5 to 5, excluding zero  
xs = c(-5:-1, 1:5)  
  
# attempt a logical decision  
if (xs < 0) print("negative") else print("positive")

## Warning in if (xs < 0) print("negative") else print("positive"): the  
## condition has length > 1 and only the first element will be used

## [1] "negative"

Warnings are different than errors in that something still happens, but it tells you that it might not be what you wanted, whereas an error stops R altogether. In short, this warning is telling you that you passed if() a logical vector with more than 1 element, and that it can only use one element so it’s picking the first one. Because the first element of xs is -5, xs < 0 evaluated to TRUE, and you got a "negative" printed along with the warning.

To respond to multiple questions at once, you must use ifelse(). This function is similar, but it combines the if() and else syntax into one useful function function:

ifelse(xs > 0, "positive", "negative")

## [1] "negative" "negative" "negative" "negative" "negative" "positive"  
## [7] "positive" "positive" "positive" "positive"

The syntax is ifelse(condition, do\_if\_TRUE, do\_if\_FALSE). You can cbind() the output with xs to verify it worked:

cbind(  
 xs,  
 ifelse(xs > 0, "positive", "negative")  
)

## xs   
## [1,] "-5" "negative"  
## [2,] "-4" "negative"  
## [3,] "-3" "negative"  
## [4,] "-2" "negative"  
## [5,] "-1" "negative"  
## [6,] "1" "positive"  
## [7,] "2" "positive"  
## [8,] "3" "positive"  
## [9,] "4" "positive"  
## [10,] "5" "positive"

Use ifelse() to create a new variable in dat that indicates whether a stream is big or small depending on whether stream\_width is greater or less than 50:

dat$size\_cat = ifelse(dat$stream\_width > 50, "big", "small"); head(dat)

## state stream\_width flow size\_cat  
## 1 Alabama 81.68 120.48 big  
## 2 Alabama 57.76 85.90 big  
## 3 Alabama 48.32 73.38 small  
## 4 Alabama 31.63 46.55 small  
## 5 Alabama 48.93 80.91 small  
## 6 Georgia 39.42 57.63 small

This says “make a new variable in the data frame dat called size\_cat and assign each row a ‘big’ if stream\_width is greater than 50 and a ‘small’ if less than 50”.

One neat thing about ifelse() is that you can nest multiple statements inside another[[11]](#footnote-75). What if you wanted three categories: ‘small’, ‘medium’, and ‘large’?

dat$size\_cat\_fine = ifelse(dat$stream\_width <= 40, "small",  
 ifelse(dat$stream\_width > 40 & dat$stream\_width <= 70, "medium", "big")); head(dat)

## state stream\_width flow size\_cat size\_cat\_fine  
## 1 Alabama 81.68 120.48 big big  
## 2 Alabama 57.76 85.90 big medium  
## 3 Alabama 48.32 73.38 small medium  
## 4 Alabama 31.63 46.55 small small  
## 5 Alabama 48.93 80.91 small medium  
## 6 Georgia 39.42 57.63 small small

If the first condition is TRUE, then it will give that row a “small”. If not, it will start another ifelse() to ask if the stream\_width is greater than 40 *and* less than or equal to 70. If so, it will give it a “medium”, if not it will get a “big”. Not all function nesting examples are this complex, but this is a neat example. Without ifelse(), you would have to use as many if() statements as there are elements in dat$stream\_width.

## Writing Output Files

### .csv Files

Now that you have made some new variables in your data frame, you may want to save this work in the form of a new .csv file. To do this, you can use the write.csv function:

write.csv(dat, "updated\_streams.csv", row.names = F)

The first argument is the data frame (or matrix) to write, the second is what you want to call it (don’t forget the .csv!), and row.names = F tells R to not include the row names (because they are just numbers in this case). R puts the file in your working directory unless you tell it otherwise. To put it somewhere else, type in the path with the new file name at the end (e.g., C:/Users/YOU/Documents/R-Book/Data/updated\_streams.csv.

### Saving R Objects

If all you care about is the data frame dat as interpreted by R (not a share-able file like the .csv method), then you can save the object dat (in its current state) then load it in to a future R session. You can save the new data frame using:

save(dat, file = "updated\_streams")

Then try removing the dat object from your current session (rm(dat)) and loading it back in using:

rm(dat); head(dat) # should give error  
load(file = "updated\_streams")  
head(dat) # should show first 6 rows

## User-Defined Functions

Sometimes you may want R to carry out a specific task, but there is no built-in function to do it. In these cases, you can write your own functions. Function writing makes R incredibly flexible, though you will only get a small taste of this topic here. You will see more examples in later Chapters, particularly in Chapter ??.

First, you must think of a name for your function (e.g., myfun). Then, you specify that you want the object myfun() to be a function by using using the function() function. Then, in parentheses, you specify any arguments that you want to use within the function to carry out the specific task. Open and closed curly braces specify the start and end of your function body, i.e., the code that specifies how it uses the arguments to do its job.

Here’s the general syntax for specifying your own function:

myfun = function(arg1) {  
 # function body goes here  
 # use arg1 to do something  
   
 # return something as last step  
}

As an example, write a general function to take any number x to any power y:

power = function(x, y){  
 x^y  
}

After typing and running the function code (power() is an object that must be assigned), try using it:

power(x = 5, y = 3)

## [1] 125

Remember, you can nest or embed functions:

power(power(5,2),2)

## [1] 625

This is the equivalent of .

## Exercise 1B

In this exercise, you will be using what you learned in Chapter 1 to summarize data from a hypothetical pond experiment.

Pretend that you added nutrients to mesocosoms and counted the densities of four different zooplankton taxa. In this experiment, there were two ponds, two treatments per pond, and five replicates of each treatment. The data are located in the data file ponds.csv (see the [instructions](#data-sets) on acquiring and organizing the data files for this book. There is one error in the data set. After you download the data and place it in the appropriate directory, make sure you open this file and fix it *before* you bring it into R. Refer back to the information about reading in data (Section 1.8) to make sure you find the error.

Create a new R script in your working directory for this chapter called Ex1B.R and use that script to complete this exercise.

*The solutions to this exercise are found at the end of this book (*[*here*](#ex1b-answers)*). You are* ***strongly recommended*** *to make a good attempt at completing this exercise on your own and only look at the solutions when you are truly stumped.*

1. Read in the data to R and assign it to an object.
2. Calculate some basic summary statistics of your data using the summary() function.
3. Calculate the mean chlorophyll *a* for each pond (*Hint: pond is a grouping variable*)
4. Calculate the mean number of *Chaoborus* for each treatment in each pond using tapply(). (*Hint: You can group by two variables with:* tapply(dat$var, list(dat$grp1, dat$grp2), fun).
5. Use the more general apply() function to calculate the variance for each zooplankton taxa found only in pond S-28.
6. Create a new variable called prod in the data frame that represents the quantity of chlorophyll *a* in each replicate. If the chlorophyll *a* in the replicate is greater than 30 give it a “high”, otherwise give it a “low”. (*Hint: are you asking R to respond to one question or multiple questions? How should this change the strategy you use?*)

### Exercise 1B Bonus

1. Use ?table to figure out how you can use table() to count how many observations of high and low there were in each treatment (*Hint: table will have only two arguments.*).
2. Create a new function called product() that multiplies any two numbers you specify.
3. Modify your function to print a message to the console and return the value if() it meets a condition and to print another message and not return the value if it doesn’t.

Allaire, JJ, Yihui Xie, Jonathan McPherson, Javier Luraschi, Kevin Ushey, Aron Atkins, Hadley Wickham, Joe Cheng, and Winston Chang. 2018. *Rmarkdown: Dynamic Documents for R*. <https://CRAN.R-project.org/package=rmarkdown>.

R Core Team. 2018. *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. <https://www.R-project.org/>.

Wickham, Hadley. 2015. *Advanced R*. 2nd ed. Boca Raton, Florida: Chapman; Hall/CRC. <https://adv-r.hadley.nz/>.

Xie, Yihui. 2015. *Dynamic Documents with R and Knitr*. 2nd ed. Boca Raton, Florida: Chapman; Hall/CRC. <http://yihui.name/knitr/>.

———. 2018a. *Bookdown: Authoring Books and Technical Documents with R Markdown*. <https://CRAN.R-project.org/package=bookdown>.

———. 2018b. *Knitr: A General-Purpose Package for Dynamic Report Generation in R*. <https://CRAN.R-project.org/package=knitr>.

1. While it is possible to run R on its own, it is clunky. You are strongly advised to use the RStudio IDE (integrated development environment) given its compactness, neat features, code tools (like syntax and parentheses highlighting). This workshop will assume you are using RStudio [↑](#footnote-ref-25)
2. Many of the data sets used in this book were simulated by the author. Cases in which the data set used was not simulated are noted and a citation to the data source is provided. More details on the individual data sets can be found on the GitHub repository. [↑](#footnote-ref-31)
3. This is a footnote. If you’re viewing this on GitHub Pages, click the arrow to the right to return to the text [↑](#footnote-ref-35)
4. For some keyboard shortcuts, you may just need to swap out the **CTRL** keystroke for the **CMD** keystroke for a Mac computer [↑](#footnote-ref-37)
5. The formatting used here includes ## on output to denote code and output separately. You won’t see the ## show up in your console. [↑](#footnote-ref-46)
6. Hadley Wickham’s R code style guide: <http://adv-r.had.co.nz/Style.html> [↑](#footnote-ref-47)
7. The c stands for **concatenate**, which basically means combine many smaller objects into one larger object [↑](#footnote-ref-54)
8. " " or ' ' both work as long as you use the same on the front and end of the element [↑](#footnote-ref-55)
9. The coercion works this way because numbers can be expressed as characters, but a letter cannot be unambiguously expressed as a number. [↑](#footnote-ref-56)
10. Note that if your computer is configured for a Spanish-speaking country, Microsoft Excel might convert decimals to commas. This can really mess with reading in data - I would suggest changing the language of Excel if you find this to be the case. [↑](#footnote-ref-64)
11. You can nest **ALL** R functions, by the way. [↑](#footnote-ref-75)