Andrew O. Finley, Vince Melfi, Jeffrey W. Doser

R Programming for Data Sciences

To my son,

without whom I should have finished this book two years earlier

Contents

List of Tables

List of Figures

Course Description

This book serves as an introduction to programming in R and the use of associated open source tools. We address practical issues in documenting workflow, data management, and scientific computing.

1 Introduction

Now unplug your Internet cable, and start doing some serious work.

We have a nice figure in Figure **??**, and also a table in Table **??**.

```
> par(mar = c(4, 4, 1, 0.1))
> plot(cars, pch = 19)
```

```
> knitr::kable(head(iris), caption = "The boring iris data.",
+   booktabs = TRUE)
```

More chapters to come in `02-foo.Rmd`, `03-bar`.Rmd, …

2

Introduction to R and RStudio

Various statistical and programming software environments are used in data science, including R, Python, SAS, C++, SPSS, and many others. Each has strengths and weaknesses, and often two or more are used in a single project. This book focuses on R for several reasons:

1. R is free
2. It is one of, if not the, most widely used software environments in data science
3. R is under constant and open development by a diverse and expert core group
4. It has an incredible variety of contributed packages
5. A new user can (relatively) quickly gain enough skills to obtain, manage, and analyze data in R

Several enhanced interfaces for R have been developed. Generally such interfaces are referred to as *integrated development environments (IDE)*. These interfaces are used to facilitate software development. At minimum, an IDE typically consists of a source code editor and build automation tools. We will use the RStudio IDE, which according to its developers "is a powerful productive user interface for R."[1] RStudio is widely used, it is used increasingly in the R community, and it makes learning to use R a bit simpler. Although we will use RStudio, most of what is presented in this book can be accomplished in R (without an added interface) with few or no changes.

2.1 Obtaining and Installing R

It is simple to install R on computers running Microsoft Windows, macOS, or Linux. For other operating systems users can compile the source code directly.[2]

[1] `http://www.rstudio.com/`

[2] Windows, macOS, and Linux users also can compile the source code directly, but for most it is a better idea to install R from already compiled binary distributions.

Here is a step-by-step guide to installing R for Microsoft Windows.[3] macOS and Linux users would follow similar steps.

1. Go to `http://www.r-project.org/`
2. Click on the `CRAN` link on the left side of the page
3. Choose one of the mirrors.[4]
4. Click on `Download R for Windows`
5. Click on `base`
6. Click on `Download R 3.5.0 for Windows`
7. Install R as you would install any other Windows program

2.2 Obtaining and Installing RStudio

You must install R prior to installing RStudio. RStudio is also simple to install:

1. Go to `http://www.rstudio.com`
2. Click on the link `RStudio` under the `Products` tab, then select the `Desktop` option
3. Click on the `Desktop` link
4. Choose the `DOWNLOAD RSTUDIO DESKTOP` link in the `Open Source Edition` column
5. On the ensuing page, click on the `Installer` version for your operating system, and once downloaded, install as you would any program

2.3 Using R and RStudio

Start RStudio as you would any other program in your operating system. For example, under Microsoft Windows use the Start Menu or double click on the shortcut on the desktop (if a shortcut was created in the installation process). A (rather small) view of RStudio is displayed in Figure **??**.

Initially the RStudio window contains three smaller windows. For now our main focus will be the large window on the left, the `Console` window, in which R statements are typed. The next few sections give simple examples of the use of R. In these sections we will focus on small and non-complex data sets, but

[3] New versions of R are released regularly, so the version number in Step 6 might be different from what is listed below.

[4] The `http://cran.rstudio.com/` mirror is usually fast. Otherwise choose a mirror in Michigan.

of course later in the book we will work with much larger and more complex sets of data. Read these sections at your computer with R running, and enter the R commands there to get comfortable using the R console window and RStudio.

2.3.1 R as a Calculator

R can be used as a calculator. Note that `#` is the comment character in R, so R ignores everything following this character. Also, you will see that R prints `[1]` before the results of each command. Soon we will explain its relevance, but ignore this for now. The command prompt in R is the greater than sign `>`.

```
> 34 + 20 * sqrt(100)  ## +,-,*,/ have the expected meanings
> exp(2)  ##The exponential function
> log10(100)  ##Base 10 logarithm
> log(100)  ##Base e logarithm
> 10^log10(55)
```

Most functions in R can be applied to vector arguments rather than operating on a single argument at a time. A `vector` is a data structure that contains elements of the same data type (i.e. integers).

```
> 1:25  ##The integers from 1 to 25
> log(1:25)  ##The base e logarithm of these integers
> 1:25 * 1:25  ##What will this produce?
> 1:25 * 1:5  ##What about this?
> seq(from = 0, to = 1, by = 0.1)  ##A sequence of numbers from 0 to 1
> exp(seq(from = 0, to = 1, by = 0.1))  ##What will this produce?
```

Now the mysterious square bracketed numbers appearing next to the output make sense. R puts the position of the beginning value on a line in square brackets before the line of output. For example if the output has 40 values, and 15 values appear on each line, then the first line will have `[1]` at the left, the second line will have `[16]` to the left, and the third line will have `[31]` to the left.

2.3.2 Basic descriptive statistics and graphics in R

It is easy to compute basic descriptive statistics and to produce standard graphical representations of data in R. First we create three variables with horsepower, miles per gallon, and names for 15 cars.[5] In this case with a

[5]These are from a relatively old data set, with 1974 model cars.

small data set we enter the data “by hand” using the `c()` function, which concatenates its arguments into a vector. For larger data sets we will clearly want an alternative. Note that character values are surrounded by quotation marks.

A style note: R has two widely used methods of assignment: the left arrow, which consists of a less than sign followed immediately by a dash: `<-` and the equals sign: `=`. Much ink has been used debating the relative merits of the two methods, and their subtle differences. Many leading R style guides (e.g., the Google style guide at https://google.github.io/styleguide/Rguide.xml and the Bioconductor style guide at http://www.bioconductor.org/developers/how-to/coding-style/) recommend the left arrow `<-` as an assignment operator, and we will use this throughout the book.

Also you will see that if a command has not been completed but the ENTER key is pressed, the command prompt changes to a `+` sign.

```
> car.hp <- c(110, 110, 93, 110, 175, 105, 245, 62, 95, 123,
+ 123, 180, 180, 180, 205)
> car.mpg <- c(21.0, 21.0, 22.8, 21.4, 18.7, 18.1, 14.3, 24.4, 22.8,
+              19.2, 17.8, 16.4, 17.3, 15.2, 10.4)
> car.name <- c("Mazda RX4", "Mazda RX4 Wag", "Datsun 710",
+               "Hornet 4 Drive", "Hornet Sportabout", "Valiant",
+               "Duster 360", "Merc 240D", "Merc 230", "Merc 280",
+               "Merc 280C", "Merc 450SE", "Merc 450SL",
+               "Merc 450SLC", "Cadillac Fleetwood")
> car.hp
> car.mpg
> car.name
```

Next we compute some descriptive statistics for the two numeric variables (`car.hp` and `car.mpg`)

```
> mean(car.hp)
> sd(car.hp)
> summary(car.hp)
> mean(car.mpg)
> sd(car.mpg)
> summary(car.mpg)
```

Next, a scatter plot of `cars.mpg` versus `cars.hp`:

```
> plot(car.hp, car.mpg)
```

Unsurprisingly as horsepower increases, mpg tends to decrease. This relationship can be investigated further using linear regression, a statistical procedure that involves fitting a linear model to a data set in order to further understand the relationship between two variables.

2.3.3 An Initial Tour of RStudio

When you created the `car.hp` and other vectors in the previous section, you might have noticed the vector name and a short description of its attributes appear in the top right `Global Environment` window. Similarly, when you called `plot(car.hp,car.mpg)` the corresponding plot appeared in the lower right `Plots` window.

A comprehensive, but slightly overwhelming, cheatsheet for RStudio is available here `https://www.rstudio.com/wp-content/uploads/2016/01/rstudio-IDE-cheatsheet.pdf`. As we progress in learning R and RStudio, this cheatsheet will become more useful. For now you might use the cheatsheet to locate the various windows and functions identified in the coming chapters.

2.4 Getting Help

There are several free (and several not free) ways to get R help when needed.

Several help-related functions are built into R. If there's a particular R function of interest, such as `log`, `help(log)` or `?log` will bring up a help page for that function. In RStudio the help page is displayed, by default, in the `Help` tab in the lower right window.[6] The function `help.start` opens a window which allows browsing of the online documentation included with R. To use this, type `help.start()` in the console window.[7] The `help.start` function also provides several manuals online and can be a useful interface in addition to the built in help.

Search engines provide another, sometimes more user-friendly, way to receive answers for R questions. A Google search often quickly finds something written by another user who had the same (or a similar) question, or an online tutorial that touches on the question. More specialized is rseek.org[8], which is a search

[6] There are ways to change this default behavior.

[7] You may wonder about the parentheses after `help.start`. A user can specify arguments to any R function inside parentheses. For example `log(10)` asks R to return the logarithm of the argument 10. Even if no arguments are needed, R requires empty parentheses at the end of any function name. In fact if you just type the function name without parentheses, R returns the definition of the function. For simple functions this can be illuminating.

[8] `http://rseek.org`

engine focused specifically on R. Both Google and rseek.org[9] are valuable tools, often providing more user-friendly information then R's own help system.

In addition, R users have written many types of contributed documentation. Some of this documentation is available at `http://cran.r-project.org/other-docs.html`. Of course there are also numerous books covering general and specialized R topics available for purchase.

2.5 Workspace, Working Directory, and Keeping Organized

The *workspace* is your R session working environment and includes any objects you create. Recall these objects are listed in the `Global Environment` window. The command `ls()`, which stands for list, will also list all the objects in your workspace (note, this is the same list that is given in the `Global Environment` window). When you close RStudio, a dialog box will ask you if you want to save an image of the current workspace. If you choose to save your workspace, RStudio saves your session objects and information in a `.RData` file (the period makes it a hidden file) in your *working directory*. Next time you start R or RStudio it checks if there is a `.RData` in the working directory, loads it if it exists, and your session continues where you left off. Otherwise R starts with an empty workspace. This leads to the next question—what is a working directory?

Each R session is associated with a working directory. This is just a directory from which R reads and writes files, e.g., the `.RData` file, data files you want to analyze, or files you want to save. On Mac when you start RStudio it sets the working directory to your home directory (for me that's `/Users/andy`). If you're on a different operating system, you can check where the default working directory is by typing `getwd()` in the console. You can change the default working directory under RStudio's `Global Option` dialog found under the `Tools` dropdown menu. There are multiple ways to change the working directory once an R session is started in RStudio. One method is to click on the `Files` tab in the lower right window and then click the `More` button. Alternatively, you can set the session's working directory using the `setwd()` in the console. For example, on Windows `setwd("C:/Users/andy/for875/exercise1")` will set the working directory to `C:/Users/andy/for875/exercise1`, assuming that file path and directory exist (Note: Windows file path uses a backslash, `\`, but in R the backslash is an escape character, hence specifying file paths in R on Windows uses the forward slash, i.e., `/`). Similarly on Mac you

[9] `http://rseek.org`

can use `setwd("/Users/andy/for875/exercise1")`. Perhaps the most simple method is to click on the `Session` tab at the top of your screen and click on the `Set Working Directory` option. Later on when we start reading and writing data from our R session, it will be very important that you are able to identify your current working directory and change it if needed. We will revisit this in subsequent chapters.

As with all work, keeping organized is the key to efficiency. It is good practice to have a dedicated directory for each R project or exercise.

2.6 Quality of R code

Writing well-organized and well-labeled code allows your code to be more easily read and understood by another person. (See xkcd's take on code quality in Figure **??**.) More importantly, though, your well-written code is more accessible to you hours, days, or even months later. We are hoping that you can use the code you write in this class in future projects and research.

Google provides style guides for many programming languages. You can find the R style guide here[10]. Below are a few of the key points from the guide that we will use right away.

2.6.1 Naming Files

File names should be meaningful and end in `.R`. If we write a script that analyzes a certain species distribution:

- GOOD: `african_rhino_distribution.R`
- GOOD: `africanRhinoDistribution.R`
- BAD: `speciesDist.R` (too ambiguous)
- BAD: `species.dist.R` (too ambiguous and two periods can confuse operating systems' file type auto-detect)
- BAD: `speciesdist.R` (too ambiguous and confusing)

2.6.2 Naming Variables

- GOOD: `rhino.count`
- GOOD: `rhinoCount`
- GOOD: `rhino_count` (We don't mind the underscore and use it quite often, although Google's style guide says it's a no-no for some reason)
- BAD: `rhinocount` (confusing)

[10] https://google.github.io/styleguide/Rguide.xml

2.6.3 Syntax

- Keep code lines under 80 characters long.
- Indent your code with two spaces. (RStudio does this by default when you press the TAB key.)

3

Scripts and R Markdown

Doing work in data science, whether for homework, a project for a business, or a research project, typically involves several iterations. For example creating an effective graphical representation of data can involve trying out several different graphical representations, and then tens if not hundreds of iterations when fine-tuning the chosen representation. Furthermore, each of these representations may require several R commands to create. Although this all could be accomplished by typing and re-typing commands at the R Console, it is easier and more effective to write the commands in a *script file*, which then can be submitted to the R console either a line at a time or all together.[1]

In addition to making the workflow more efficient, R scripts provide another large benefit. Often we work on one part of a homework assignment or project for a few hours, then move on to something else, and then return to the original part a few days, months, or sometimes even years later. In such cases we may have forgotten how we created the graphical display that we were so proud of, and will need to again spend a few hours to recreate it. If we save a script file, we have the ingredients immediately available when we return to a portion of a project.[2]

Next consider the larger scientific endeavor. Ideally a scientific study will be reproducible, meaning that an independent group of researchers (or the original researchers) will be able to duplicate the study. Thinking about data science, this means that all the steps taken when working with the data from a study should be reproducible, from the selection of variables to formal data analysis. In principle, this can be facilitated by explaining, in words, each step of the work with data. In practice, it is typically difficult or impossible to reproduce a full data analysis based on a written explanation. Much more effective is to include the actual computer code which accomplished the data work in the report, whether the report is a homework assignment or a research paper. Tools in R such as *R Markdown* facilitate this process.

[1]Unsurprisingly it is also possible to submit several selected lines of code at once.

[2]In principle the R history mechanism provides a similar record. But with history we have to search through a lot of other code to find what we're looking for, and scripts are a much cleaner mechanism to record our work.

3.1 Scripts in R

As noted above, scripts help to make work with data more efficient and provide a record of how data were managed and analyzed. Below we describe an example. This example uses features of R that we have not yet discussed, so don't worry about the details, but rather about how it motivates the use of a script file.

First we read in a data set containing data on (among other things) fertility rate and life expectancy for countries throughout the world, for the years 1960 through 2014.

```
> u <- "http://blue.for.msu.edu/FOR875/data/WorldBank.csv"
> WorldBank <- read.csv(u, header = TRUE, stringsAsFactors = FALSE)
```

Next we print the names of the variables in the data set. Don't be concerned about the specific details. Later we will learn much more about reading in data and working with data sets in R.

```
> names(WorldBank)
```

We will try to create a scatter plot of fertility rate versus life expectancy of countries for the year 1960. To do this we'll first create variables containing the values of fertility rate and life expectancy for 1960[3], and print out the first ten values of each variable.

```
> fertility <- WorldBank$fertility.rate[WorldBank$year ==
+   1960]
> lifeexp <- WorldBank$life.expectancy[WorldBank$year == 1960]
> fertility[1:10]
> lifeexp[1:10]
```

We see that some countries do not have data for 1960. R represents missing data via `NA`. Of course at some point it would be good to investigate which countries' data are missing and why. The `plot()` function in R will just omit missing values, and for now we will just plot the non-missing data. A scatter plot of the data is drawn next.

```
> plot(lifeexp, fertility)
```

[3]This isn't necessary, but it is convenient

The scatter plot shows that as life expectancy increases, fertility rate tends to decrease in what appears to be a nonlinear relationship. Now that we have a basic scatter plot, it is tempting to make it more informative. We will do this by adding two features. One is to make the points' size proportional to the country's population, and the second is to make the points' color represent the region of the world the country resides in. We'll first extract the population and region variables for 1960.

```
> pop <- WorldBank$population[WorldBank$year == 1960]
> region <- WorldBank$region[WorldBank$year == 1960]
> pop[1:10]
> region[1:10]
```

To create the scatter plot we will do two things. First we will create the axes, labels, etc. for the plot, but not plot the points. The argument `type="n"` tells R to do this. Then we will use the `symbols()` function to add symbols, the `circles` argument to set the sizes of the points, and the `bg` argument to set the colors. Don't worry about the details! In fact, later in the book we will learn about an R package called `ggplot2` that provides a different way to create such plots. You'll see two plots below, first the "empty" plot which is just a building block, then the plot including the appropriate symbols.

```
> plot(lifeexp, fertility, type="n")
> symbols(lifeexp, fertility, circles=sqrt(pop/pi), inches=0.35,
+          bg=match(region, unique(region)))
```

Of course we should have a key which tells the viewer which region each color represents, and a way to determine which country each point represents, and a lot of other refinements. For now we will resist such temptations.

Some of the process leading to the completed plot is shown above, such as reading in the data, creating variables representing the 1960 fertility rate and life expectancy, an intermediate plot that was rejected, and so on. A lot of the process isn't shown, simply to save space. There would likely be mistakes (either minor typing mistakes or more complex errors). Focusing only on the `symbols()` function that was used to add the colorful symbols to the scatter plot, there would likely have been a substantial number of attempts with different values of the `circles`, `inches`, and `bg` arguments before settling on the actual form used to create the plot. This is the typical process you will soon discover when producing useful data visualizations.

Now imagine trying to recreate the plot a few days later. Possibly someone saw the plot and commented that it would be interesting to see some similar plots, but for years in the 1970s when there were major famines in different countries of the world. If all the work, including all the false starts and refinements, were done at the console, it would be hard to sort things out and would take longer than necessary to create the new plots. This would be especially true if a few months had passed rather than just a few days.

But with a script file, especially a script file with a few well-chosen comments, creating the new scatter plots would be much easier. Fortunately it is quite easy to create and work with script files in RStudio.[4] Just choose `File > New File > R script` and a script window will open up in the upper left of the full RStudio window.

An example of a script window (with some R code already typed in) is shown in Figure **??**. From the script window the user can, among other things, save the script (either using the `File` menu or the icon near the top left of the window) and run one or more lines of code from the window (using the `run` icon in the window, or by copying and pasting into the console window). In addition, there is a `Source on Save` checkbox. If this is checked, the R code in the script window is automatically read into R and executed when the script file is saved.

```
> knitr::include_graphics("../figures/ScriptWindow.PNG")
```

3.2 R Markdown

People typically work on data with a larger purpose in mind. Possibly the purpose is to understand a biological system more clearly. Possibly the purpose is to refine a system that recommends movies to users in an online streaming movie service. Possibly the purpose is to complete a homework assignment and demonstrate to the instructor an understanding of an aspect of data analysis. Whatever the purpose, a key aspect is communicating with the desired audience, for example, fellow researchers or an instructor.

One possibility, which is somewhat effective, is to write a document using software such as Microsoft Word[5] and to include R output such as computations and graphics by cutting and pasting into the main document. One drawback to this approach is similar to what makes script files so useful: If the document must be revised it may be hard to unearth the R code that created graphics or analyses.[6] A more subtle but possibly more important drawback is that

[4]It is also easy in R without RStudio. Just use `File > New script` to create a script file, and save it before exiting R.

[5]Or possibly LaTeX if the document is more technical

[6]Organizing the R code using script files and keeping all the work organized in a well-thought-out directory structure can help here, but this requires a level of forethought and organization that most people do not possess...including myself.

the reader of the document will not know precisely how analyses were done, or how graphics were created. And over time even the author(s) of the paper will forget the details. A verbal description in a "methods" section of a paper can help here, but typically these do not provide all the details of the analysis, but rather might state something like, "All analyses were carried out using R version 3.6.0."

RStudio's website provides an excellent overview of R Markdown capabilities for reproducible research. At minimum, follow the `Get Started` link at http://rmarkdown.rstudio.com/ and watch the introduction video.

Among other things, R Markdown provides a way to include R code that reads in data, creates graphics, or performs analyses. This is performed in a single document which is processed to create a research paper, homework assignment, or other written product. The R Markdown file is a plain text file containing text the author wants to have shown in the final document, simple commands to indicate how the text should be formatted (i.e. boldface, italic, or a bulleted list), and R code which creates output (including graphics) on the fly. Perhaps the simplest way to get started is to see an R Markdown file and the resulting document that is produced after the R Markdown document is processed. Below we code that would comprise a very simple R Markdown file, and Figure **??** shows the resulting output. In this case the output created is an HTML file, but there are other possible output formats such as Microsoft Word or PDF.

````
---
title: "R Markdown"
author: "Andy Finley"
date: "April 3, 2017"
output: html_document
---

Basic formatting:

*italic*

**bold**

~~strikethrough~~
````

A code chunk:

~~~
```{r}
x <- 1:10
y <- 10:1
mean(x)
sd(y)
~~~

~~~
```
~~~

Inline code:

```
`r 5+5`
```

Inline code not executed:

```
`5+5`
```

At the top of the input R Markdown file are some lines with `---` at the top and bottom. These lines are not needed, but give a convenient way to specify the title, author, and date of the article that are then typeset prominently at the top of the output document. For now, don't be concerned with the lines following `output:`. These can be omitted (or included as shown).

Next are a few lines showing some of the ways that font effects such as italics, boldface, and strikethrough can be achieved. For example, an asterisk before and after text sets the text in *italics*, and two asterisks before and after text sets the text in *boldface*.

More important for our purposes is the ability to include R code in the R Markdown file, which will be executed with the output appearing in the output document. Bits of R code included this way are called *code chunks*. The beginning of a code chunk is indicated with three backticks and an "r" in curly braces: ```` ```{r} ````. The end of a code chunk is indicated with three backticks: ```` ``` ````. For example, the R Markdown file described above has one code chunk:

~~~
```{r}
x <- 1:10
y <- 10:1
mean(x)
sd(y)
```
~~~

In this code chunk two vectors `x` and `y` are created, and the mean of `x` and the standard deviation of `y` are computed. In the output in Figure **??** the R code is reproduced, and the output of the two lines of code asking for the mean and standard deviation is shown.

3.2.1 Creating and Processing R Markdown Documents

RStudio has features which facilitate creating and processing R Markdown documents. Choose `File > New File > R  Markdown...`. In the ensuing dialog box make sure that `Document` is highlighted on the left, enter the title and author (if desired), and choose the Default Output Format (HTML is good to begin). Then click OK. A document will appear in the upper left of the RStudio window. It is an R Markdown document, and the title and author you chose will show up, delimited by `---` at the top of the document. A generic body of the document will also be included.

For now just keep this generic document as is. To process it to create the HTML output, click the `Knit HTML` button at the top of the R Markdown window. You'll be prompted to choose a filename for the R Markdown file. Use `.Rmd` as the extension for this file. Once you've saved the file, RStudio will process the file, create the HTML output, and open this output in a new window. The HTML output file will also be saved to your working directory. This file can be shared with others, who can open it using a web browser such as Chrome or Firefox.

There are many options which allow customization of R Markdown documents. Some of these affect formatting of text in the document, while others affect how R code is evaluated and displayed. The RStudio web site contains a useful summary of many R Markdown options at www.rstudio.com/wp-content/uploads/2015/03/rmarkdown-reference.pdf. A different, but mind-numbingly busy, cheatsheet is at www.rstudio.com/wp-content/uploads/2016/03/rmarkdown-cheatsheet-2.0.pdf. Some of the more commonly used R Markdown options are described next.

3.2.1.1 Text: Lists and Headers

Unordered (sometimes called bulleted) lists and ordered lists are easy in R Markdown. Below we illustrate the creation of unordered and ordered lists.

- For an unordered list, either an asterisk, a plus sign, or a minus sign may precede list items. Use a space after these symbols before including the list text. To have second-level items (sub-lists) indent four spaces before indicating the list item. This can also be done for third-level items.
- For an ordered list use a numeral followed by a period and a space (1. or 2. or 3. or …) to indicate a numbered list, and use a letter followed by a period and a space (a. or b. or c. or …) to indicate a lettered list. The same four space convention is used to designate sub lists.
- For an ordered list, the first list item will be labeled with the number or letter that you specify, but subsequent list items will be numbered sequentially. This will become clear through the following example. Consider the R Markdown input below and the subsequent output in Figure **??**:

```
An unordered list:

* List item 1
* List item 2
    + Second level list item 1
    + Second level list item 2
        + Third level list item
* List item 3
```

```
An ordered list:

1. List item 1
2. List item 2
    c. Sub list item 1
    q. Sub list item 2
17. List item 3
```

```
> knitr::include_graphics("../figures/ListExamples.pdf")
```

In those examples notice that for the ordered list, although the first-level numbers given in the R Markdown code are 1, 2, and 17, the numbers printed in the output are 1, 2, and 3. Similarly the letters given in the R Markdown code are c and q, but the output file prints c and d.

R Markdown does not give substantial control over font size. Different "header" levels, which provide different font sizes, are available. Put one or more hash marks (#) in front of text to specify different header levels. Other font choices such as subscripts and superscripts are possible, by surrounding the text either by tildes or carets, respectively. More sophisticated mathematical displays are also possible, and are surrounded by dollar signs. The actual mathematical expressions are specified using a language called LaTeX. See the examples below for further information for working with headers in R Markdown and LaTeX commands.

```
# A first *level* ~~header~~

## A second level header

### A third level header

Text subscripts and superscripts:

x~2~ + y~2~

10^3^ = 1000

Mathematics examples:

$x_a$

$x^a$

$\int_0^1 x^2 dx$
```

`$\frac{x}{y}$`

`$\sqrt{x}$`

`$\sqrt[n]{x}$`

`$\sum_{k=1}^n$`

`$\prod_{k=1}^n$`

3.2.1.2 Code Chunks

R Markdown provides a large number of options to vary the behavior of code chunks. In some contexts it is useful to display the output but not the R code leading to the output. In some contexts it is useful to display the R prompt and in others it is not. Perhaps it is useful to configure the size of figures created by graphics. And so on. These code chunk options and many more are described in www.rstudio.com/wp-content/uploads/2015/03/rmarkdown-reference.pdf.

Code chunk options are specified in the curly braces near the beginning of a code chunk. For example the option `echo=FALSE` would be specified via ```` ```{r, echo=FALSE} ````. Below are descriptions of a few of the more commonly used options. The use of these options is illustrated in Figures~**??** and~**??**.

1. `echo=FALSE` specifies that the R code should not be printed (but any output of the R code should be printed) in the resulting document.
2. `include=FALSE` specifies that neither the R code nor the output should be printed. However, the objects created by the code chunk will be available for use in later code chunks.
3. `eval=FALSE` specifies that the R code should not be evaluated. The code will be printed unless, for example, `echo=FALSE` is also given as an option.
4. `error=FALSE` and `warning=FALSE` specify that, respectively, error messages and warning messages generated by the R code should not be printed.
5. The `comment` option allows a specified character string to be prepended to each line of results. By default this is set to `comment = '##'` which explains the two hash marks preceding results in Figure **??** for example. Setting `comment = NA` presents output without any character string prepended. That is done in most code chunks in this book.
6. `prompt=TRUE` specifies that R prompt `>` will be prepended to each

line of R code shown in the document. `prompt = FALSE` specifies that command prompts should not be included.

7. `fig.height` and `fig.width` specify the height and width of figures generated by R code. These are specified in inches, so for example `fig.height=4` specifies a four inch high figure.

The below R Markdown input and **??** (printed output) give examples of the use of code chunk options.

“‘

No options:

```
x <- 1:10
x
```

echo=FALSE: `{r, echo = FALSE} x <- 1:10 x`

comment=NA: `{r, comment = NA} x <- 1:10 x` comment=‘#’, prompt=TRUE: `{r, comment = '#', prompt = TRUE} x <- 1:10 x`

echo=FALSE, fig.height=4, fig.width=4:

```
{r, echo = FALSE, fig.height = 4, fig.width = 4} y <- 10:1
plot(x,y)
```

“‘

3.2.2 Output Formats other than HTML

It is possible to use R Markdown to produce documents in formats other than HTML, including Word and PDF documents, among others. Next to the `Knit HTML` button is a down arrow. Click on this and choose `Knit Word` to produce a Microsoft word output document. Although there is also a `Knit PDF` button, PDF output requires additional software called TeX in addition to RStudio.[7]

3.2.3 LaTeX, `knitr`, and `bookdown`

While basic R Markdown provides substantial flexibility and power, it lacks features such as cross-referencing, fine control over fonts, etc. If this is desired, a variant of R Markdown called `knitr`, which has very similar syntax to R Markdown for code chunks, can be used in conjunction with the typesetting system LaTeX to produce documents. Another option is to use the R package `bookdown` which uses R Markdown syntax and some additional features to allow for writing more technical documents. In fact this book was initially

[7]It isn’t particularly hard to install TeX software. For a Microsoft Windows system, MiKTeX is convenient, and is available from miktex.org. For a Mac system, MacTeX is available from www.tug.org/mactex/.

created using `knitr` and LaTeX, but the simplicity of markdown syntax and the additional intricacies provided by the `bookdown` package convinced us to write the book in R Markdown using `bookdown`. For simpler tasks, basic R Markdown is plenty sufficient, and very easy to use.

3.3 Practice Exercises

3.4 Homework

Exercise 1 Learning objectives: practice setting up a working directory and read in data; explore the workspace within RStudio and associated commands; produce basic descriptive statistics and graphics.

Exercise 2 Learning objectives: practice working within RStudio; create a R Markdown document and resulting html document in RStudio; calculate descriptive statistics and produce graphics.

4

Data Structures

A data structure is a format for organizing and storing data. The structure is designed so that data can be accessed and worked with in specific ways. Statistical software and programming languages have methods (or functions) designed to operate on different kinds of data structures.

This chapter's focus is on data structures. To help initial understanding, the data in this chapter will be relatively modest in size and complexity. The ideas and methods, however, generalize to larger and more complex data sets.

The base data structures in R are vectors, matrices, arrays, data frames, and lists. The first three, vectors, matrices, and arrays, require all elements to be of the same type or homogeneous, e.g., all numeric or all character. Data frames and lists allow elements to be of different types or heterogeneous, e.g., some elements of a data frame may be numeric while other elements may be character. These base structures can also be organized by their dimensionality, i.e., 1-dimensional, 2-dimensional, or N-dimensional, as shown in Table **??**.

R has no scalar types, i.e., 0-dimensional. Individual numbers or strings are actually vectors of length one.

An efficient way to understand what comprises a given object is to use the `str()` function. `str()` is short for structure and prints a compact, human-readable description of any R data structure. For example, in the code below, we prove to ourselves that what we might think of as a scalar value is actually a vector of length one.

```
> a <- 1
> str(a)
> is.vector(a)
> length(a)
```

Here we assigned `a` the scalar value one. The `str(a)` prints `num 1`, which says `a` is numeric of length one. Then just to be sure we used the function `is.vector()` to test if `a` is in fact a vector. Then, just for fun, we asked the length of `a`, which again returns one. There are a set of similar logical tests for the other base data structures, e.g., `is.matrix()`, `is.array()`, is.data.frame(), and is.list(). These will all come in handy as we encounter different R objects.

4.1 Vectors

Think of a vector[1] as a structure to represent one variable in a data set. For example a vector might hold the weights, in pounds, of 7 people in a data set. Or another vector might hold the genders of those 7 people. The c() function in R is useful for creating (small) vectors and for modifying existing vectors. Think of c as standing for "combine".

```
> weight <- c(123, 157, 205, 199, 223, 140, 105)
> weight
> gender <- c("female", "female", "male", "female", "male",
+              "male", "female")
> gender
```

Notice that elements of a vector are separated by commas when using the c() function to create a vector. Also notice that character values are placed inside quotation marks.

The c() function also can be used to add to an existing vector. For example, if an eighth male person was included in the data set, and his weight was 194 pounds, the existing vectors could be modified as follows.

```
> weight <- c(weight, 194)
> gender <- c(gender, "male")
> weight
> gender
```

4.1.1 Types, Conversion, Coercion

Clearly it is important to distinguish between different types of vectors. For example, it makes sense to ask R to calculate the mean of the weights stored in weight, but does not make sense to ask R to compute the mean of the genders stored in gender. Vectors in R may have one of six different "types": character, double, integer, logical, complex, and raw. Only the first four of these will be of interest below, and the distinction between double and integer

[1]Technically the objects described in this section are "atomic" vectors (all elements of the same type), since lists, to be described below, also are actually vectors. This will not be an important issue, and the shorter term vector will be used for atomic vectors below.

will not be of great import. To illustrate logical vectors, imagine that each of the eight people in the data set was asked whether he or she was taking blood pressure medication, and the responses were coded as `TRUE` if the person answered yes, and `FALSE` if the person answered no.

```
> typeof(weight)
> typeof(gender)
> bp <- c(FALSE, TRUE, FALSE, FALSE, TRUE, FALSE, TRUE, FALSE)
> bp
> typeof(bp)
```

It may be surprising to see that the variable `weight` is of `double` type, even though its values all are integers. By default R creates a double type vector when numeric values are given via the `+c()` function.

When it makes sense, it is possible to convert vectors to a different type. Consider the following examples.

```
> weight.int <- as.integer(weight)
> weight.int
> typeof(weight.int)
> weight.char <- as.character(weight)
> weight.char
> bp.double <- as.double(bp)
> bp.double
> gender.oops <- as.double(gender)
> gender.oops
> sum(bp)
```

The integer version of `weight` doesn't look any different, but it is stored differently, which can be important both for computational efficiency and for interfacing with other languages such as `C++`. As noted above, however, we will not worry about the distinction between integer and double types. Converting `weight` to character goes as expected: The character representations of the numbers replace the numbers themselves. Converting the logical vector `bp` to double is pretty straightforward too: `FALSE` is converted to zero, and `TRUE` is converted to one. Now think about converting the character vector `gender` to a numeric double vector. It's not at all clear how to represent "female" and "male" as numbers. In fact in this case what R does is to create a character vector, but with each element set to `NA`, which is the representation of missing data.[2] Finally consider the code `sum(bp)`. Now `bp` is a logical vector, but when R sees that we are asking to sum this logical vector, it automatically converts

[2]Missing data will be discussed in more detail later in the chapter.

it to a numerical vector and then adds the zeros and ones representing `FALSE` and `TRUE`.

R also has functions to test whether a vector is of a particular type.

```
> is.double(weight)
> is.character(weight)
> is.integer(weight.int)
> is.logical(bp)
```

4.1.1.1 Coercion

Consider the following examples.

```
> xx <- c(1, 2, 3, TRUE)
> xx
> yy <- c(1, 2, 3, "dog")
> yy
> zz <- c(TRUE, FALSE, "cat")
> zz
> weight + bp
```

Vectors in R can only contain elements of one type. If more than one type is included in a `c()` function, R silently *coerces* the vector to be of one type. The examples illustrate the hierarchy—if any element is a character, then the whole vector is character. If some elements are numeric (either integer or double) and other elements are logical, the whole vector is numeric. Note what happened when R was asked to add the numeric vector `weight` to the logical vector `bp`. The logical vector was silently coerced to be numeric, so that FALSE became zero and TRUE became one, and then the two numeric vectors were added.

4.1.2 Accessing Specific Elements of Vectors

To access and possibly change specific elements of vectors, refer to the position of the element in square brackets. For example, `weight[4]` refers to the fourth element of the vector `weight`. Note that R starts the numbering of elements at 1, i.e., the first element of a vector `x` is `x[1]`.

```
> weight
> weight[5]
> weight[1:3]
> length(weight)
> weight[length(weight)]
> weight[]
> weight[3] <- 202
> weight
```

Note that including nothing in the square brackets results in the whole vector being returned.

Negative numbers in the square brackets tell R to omit the corresponding value. And a zero as a subscript returns nothing (more precisely, it returns a length zero vector of the appropriate type).

```
> weight[-3]
> weight[-length(weight)]
> lessWeight <- weight[-c(1, 3, 5)]
> lessWeight
> weight[0]
> weight[c(0, 2, 1)]
> weight[c(-1, 2)]
```

Note that mixing zero and other nonzero subscripts is allowed, but mixing negative and positive subscripts is not allowed.

What about the (usual) case where we don't know the positions of the elements we want? For example possibly we want the weights of all females in the data. Later we will learn how to subset using logical indices, which is a very powerful way to access desired elements of a vector.

4.2 Factors

Categorical variables such as `gender` can be represented as character vectors. In many cases this simple representation is sufficient. Consider, however, two other categorical variables, one representing age via categories `youth`, `young adult`, `middle age`, `senior`, and another representing income via categories `lower`, `middle`, and `upper`. Suppose that for the small health data set, all the people are either middle aged or senior citizens. If we just represented the variable via a character vector, there would be no way to know that there are two other categories, representing youth and senior citizens, which happen not to be present in the data set. And for the income variable, the character vector representation does not explicitly indicate that there is an ordering of the levels.

Factors in R provide a more sophisticated way to represent categorical variables. Factors explicitly contain all possible levels, and allow ordering of levels.

```
> age <- c("middle age", "senior", "middle age", "senior",
+          "senior", "senior", "senior", "middle age")
> income <- c("lower", "lower", "upper", "middle", "upper",
+             "lower", "lower", "middle")
> age
> income
> age <- factor(age, levels=c("youth", "young adult", "middle age",
+                             "senior"))
> age
> income <- factor(income, levels=c("lower", "middle", "upper"),
+                  ordered = TRUE)
> income
```

In the factor version of `age` the levels are explicitly listed, so it is clear that the two included levels are not all the possible levels. And in the factor version of income, the ordering is explicit.

In many cases the character vector representation of a categorical variable is sufficient and easier to work with. In this book, factors will not be used extensively. It is important to note that R often by default creates a factor when character data are read in, and sometimes it is necessary to use the argument `stringsAsFactors = FALSE` to explicitly tell R not to do this. This is shown later in the chapter when data frames are introduced.

4.3 Names of Objects in R

There are few hard and fast restrictions on the names of objects (such as vectors) in R. In addition to these restrictions, there are certain good practices, and many things to avoid as well.

From the help page for `make.names` in R, the name of an R object is "syntactically valid" if the name "consists of letters, numbers and the dot or underline characters and starts with a letter or the dot not followed by a number" and is not one of the "reserved words" in R such as `if`, `TRUE`, `function`, etc. For example, `c45t.le_dog` and `.ty56` are both syntactically valid (although not very good names) while `4cats` and `log#@gopher` are not.

A few important comments about naming objects follow:

1. It is important to be aware that names of objects in R are case-sensitive, so `weight` and `Weight` do not refer to the same object.

```
> weight
> Weight
```

2. It is unwise to create an object with the same name as a built in R object such as the function `c` or the function `mean`. In earlier versions of R this could be somewhat disastrous, but even in current versions, it is definitely not a good idea!
3. As much as possible, choose names that are informative. When creating a variable you may initially remember that `x` contains heights and `y` contains genders, but after a few hours, a few days, or a few weeks, you probably will forget this. Better options are `Height` and `Gender`.
4. As much as possible, be consistent in how you name objects. In particular, decide how to separate multi-word names. Some options include:
   - Using case to separate: `BloodPressure` or `bloodPressure` for example
   - Using underscores to separate: `blood_pressure` for example
   - Using a period to separate: `blood.pressure` for example

4.4 Missing Data, Infinity, etc.

Most real-world data sets have variables where some observations are missing. In a longitudinal study participants may drop out. In a survey, participants may decide not to respond to certain questions. Statistical software should be able to represent missing data and to analyze data sets in which some data are missing.

In R, the value `NA` is used for a missing data value. Since missing values may occur in numeric, character, and other types of data, and since R requires that a vector contain only elements of one type, there are different types of `NA` values. Usually R determines the appropriate type of `NA` value automatically. It is worth noting that the default type for `NA` is logical, and that `NA` is NOT the same as the character string `"NA"`.

```
> missingCharacter <- c("dog", "cat", NA, "pig", NA, "horse")
> missingCharacter
> is.na(missingCharacter)
> missingCharacter <- c(missingCharacter, "NA")
> missingCharacter
> is.na(missingCharacter)
> allMissing <- c(NA, NA, NA)
> typeof(allMissing)
```

How should missing data be treated in computations, such as finding the mean or standard deviation of a variable? One possibility is to return `NA`. Another is to remove the missing value(s) and then perform the computation.

```
> mean(c(1, 2, 3, NA, 5))
> mean(c(1, 2, 3, NA, 5), na.rm = TRUE)
```

As this example shows, the default behavior for the `mean()` function is to return `NA`. If removal of the missing values and then computing the mean is desired, the argument `na.rm` is set to `TRUE`. Different R functions have different default behaviors, and there are other possible actions. Consulting the help for a function provides the details.

4.4.1 Infinity and NaN

What happens if R code requests division by zero, or results in a number that is too large to be represented? Here are some examples.

```
> x <- 0:4
> x
> 1/x
> x/x
> y <- c(10, 1000, 10000)
> 2^y
```

`Inf` and `-Inf` represent infinity and negative infinity (and numbers which are too large in magnitude to be represented as floating point numbers). `NaN` represents the result of a calculation where the result is undefined, such as dividing zero by zero. All of these are common to a variety of programming languages, including R.

4.5 Data Frames

Commonly, data is rectangular in form, with variables as columns and cases as rows. Continuing with the (contrived) data on weight, gender, and blood pressure medication, each of those variables would be a column of the data set, and each person's measurements would be a row. In R, such data are represented as a *data frame*.

```
> healthData <- data.frame(Weight = weight, Gender=gender, bp.meds = bp,
+                          stringsAsFactors=FALSE)
> healthData
> names(healthData)
> colnames(healthData)
> names(healthData) <- c("Wt", "Gdr", "bp")
> healthData
> rownames(healthData)
> names(healthData) <- c("Weight", "Gender", "bp.meds")
```

The `data.frame` function can be used to create a data frame (although it's more common to read a data frame into R from an external file, something that will be introduced later). The names of the variables in the data frame are given as arguments, as are the vectors of data that make up the variable's values. The argument `stringsAsFactors=FALSE` asks R not to convert character vectors into factors, which R does by default, to the dismay of many users. Names of the columns (variables) can be extracted and set via either `names` or `colnames`. In the example, the variable names are changed to `Wt, Gdr, bp` and then changed back to the original `Weight, Gender, bp.meds` in this way. Rows can be named also. In this case since specific row names were not provided, the default row names of `"1", "2"` etc. are used.

In the next example a built-in dataset called `mtcars` is made available by the `data` function, and then the first and last six rows are displayed using `head` and `tail`.

```
> data(mtcars)
> head(mtcars)
> tail(mtcars)
```

Note that the `mtcars` data frame does have non-default row names which give the make and model of the cars.

4.5.1 Accessing Specific Elements of Data Frames

Data frames are two-dimensional, so to access a specific element (or elements) we need to specify both the row and column.

```
> mtcars[1, 4]
> mtcars[1:3, 3]
> mtcars[1:3, 2:3]
> mtcars[, 1]
```

Note that `mtcars[,1]` returns ALL elements in the first column. This agrees with the behavior for vectors, where leaving a subscript out of the square brackets tells R to return all values. In this case we are telling R to return all rows, and the first column.

For a data frame there is another way to access specific columns, using the `$` notation.

```
> mtcars$mpg
> mtcars$cyl
> mpg
> cyl
> weight
```

Notice that typing the variable name, such as `mpg`, without the name of the data frame (and a dollar sign) as a prefix, does not work. This is sensible. There may be several data frames that have variables named `mpg`, and just typing `mpg` doesn't provide enough information to know which is desired. But if there is a vector named `mpg` that is created outside a data frame, it will be retrieved when `mpg` is typed, which is why typing `weight` does work, since `weight` was created outside of a data frame, although ultimately it was incorporated into the `healthData` data frame.

4.6 Lists

The third main data structure we will work with is a list. Technically a list is a vector, but one in which elements can be of different types. For example a list may have one element that is a vector, one element that is a data frame, and another element that is a function. Consider designing a function that fits a simple linear regression model to two quantitative variables. We might want that function to compute and return several things such as

- The fitted slope and intercept (a numeric vector with two components)

- The residuals (a numeric vector with n components, where n is the number of data points)
- Fitted values for the data (a numeric vector with n components, where n is the number of data points)
- The names of the dependent and independent variables (a character vector with two components)

In fact R has a function, `lm`, which does this (and much more).

```
> mpgHpLinMod <- lm(mpg ~ hp, data = mtcars)
> mode(mpgHpLinMod)
> names(mpgHpLinMod)
> mpgHpLinMod$coefficients
> mpgHpLinMod$residuals
```

The `lm` function returns a list (which in the code above has been assigned to the object `mpgHpLinMod`).[3] One component of the list is the length `length(mpgHpLinMod$coefficients)` vector of coefficients, while another component is the length `length(mpgHpLinMod$residuals)` vector of residuals. The code also illustrates that named components of a list can be accessed using the dollar sign notation, as with data frames.

The `list` function is used to create lists.

```
> temporaryList <- list(first=weight, second=healthData,
+                       pickle=list(a = 1:10, b=healthData))
> temporaryList
```

Here, for illustration, I assembled a list to hold some of the R data structures we have been working with in this chapter. The first list element, named `first`, holds the `weight` vector we created in Section 4.1, the second list element, named `second`, holds the `healthData` data frame, and the third list element, named `pickle`, holds a list with elements named `a` and `b` that hold a vector of values 1 through 10 and another copy of the `healthData` data frame, respectively. As this example shows, a list can contain another list.

4.6.1 Accessing Specific Elements of Lists

We already have seen the dollar sign notation works for lists. In addition, the square bracket subsetting notation can be used. There is an added, somewhat subtle wrinkle—using either single or double square brackets.

[3]The `mode` function returns the type or storage mode of an object.

```
> temporaryList$first
> mode(temporaryList$first)
> temporaryList[[1]]
> mode(temporaryList[[1]])
> temporaryList[1]
> mode(temporaryList[1])
```

Note the dollar sign and double bracket notation return a numeric vector, while the single bracket notation returns a list. Notice also the difference in results below.

```
> temporaryList[c(1, 2)]
> temporaryList[[c(1, 2)]]
```

The single bracket form returns the first and second elements of the list, while the double bracket form returns the second element in the first element of the list. Generally, do not put a vector of indices or names in a double bracket, you will likely get unexpected results. See, for example, the results below.[4]

```
> temporaryList[[c(1, 2, 3)]]
```

So, in summary, there are two main differences between using the single bracket `[]` and double bracket `[[]]`. First, the single bracket will return a list that holds the object(s) held at the given indices or names placed in the bracket, whereas the double brackets will return the actual object held at the index or name placed in the innermost bracket. Put differently, a single bracket can be used to access a range of list elements and will return a list, and a double bracket can only access a single element in the list and will return the object held at the index.

4.7 Subsetting with Logical Vectors

Consider the `healthData` data frame. How can we access only those weights which are more than 200? How can we access the genders of those whose weights are more than 200? How can we compute the mean weight of males and the mean weight of females? Or consider the `mtcars` data frame. How can we obtain the miles per gallon for all six cylinder cars? Both of these data sets

[4] Try this example using only single brackets... it will return a list holding elements `first`, `second`, and `pickle`.

are small enough that it would not be too onerous to extract the values by hand. But for larger or more complex data sets, this would be very difficult or impossible to do in a reasonable amount of time, and would likely result in errors.

R has a powerful method for solving these sorts of problems using a variant of the subsetting methods that we already have learned. When given a logical vector in square brackets, R will return the values corresponding to `TRUE`. To begin, focus on the `weight` and `gender` vectors created in Section 4.1.

The R code `weight > 200` returns a `TRUE` for each value of `weight` which is more than 200, and a `FALSE` for each value of `weight` which is less than or equal to 200. Similarly `gender == "female"` returns `TRUE` or `FALSE` depending on whether an element of `gender` is equal to `female`.

```
> weight
> weight > 200
> gender[weight > 200]
> weight[weight > 200]
> gender == "female"
> weight[gender == "female"]
```

Consider the lines of R code one by one.

- `weight` instructs R to display the values in the vector `weight`.
- `weight > 200` instructs R to check whether each value in `weight` is greater than 200, and to return `TRUE` if so, and `FALSE` otherwise. +The next line, `gender[weight > 200]`, does two things. First, inside the square brackets, it does the same thing as the second line, namely, returning `TRUE` or `FALSE` depending on whether a value of `weight` is or is not greater than 200. Second, each element of `gender` is matched with the corresponding `TRUE` or `FALSE` value, and is returned if and only if the corresponding value is `TRUE`. For example the first value of `gender` is `gender[1]`. Since the first `TRUE` or `FALSE` value is `FALSE`, the first value of `gender` is not returned. Only the third and fifth values of `gender`, both of which happen to be `male`, are returned. Briefly, this line returns the genders of those people whose weight is over 200 pounds.
- The fourth line of code, `weight[weight > 200]`, again begins by returning `TRUE` or `FALSE` depending on whether elements of `weight` are larger than 200. Then those elements of `weight` corresponding to `TRUE` values, are returned. So this line returns the weights of those people whose weights are more than 200 pounds.
- The fifth line returns `TRUE` or `FALSE` depending on whether elements of `gender` are equal to `female` or not.
- The sixth line returns the weights of those whose gender is `female`.

There are six comparison operators in R, `>, <, >=, <=, ==, !=`. Note that to test for equality a "double equals sign" is used, while `!=` tests for inequality.

4.7.1 Modifying or Creating Objects via Subsetting

The results of subsetting can be assigned to a new (or existing) R object, and subsetting on the left side of an assignment is a common way to modify an existing R object.

```
> weight
> lightweight <- weight[weight < 200]
> lightweight
> x <- 1:10
> x
> x[x < 5] <- 0
> x
> y <- -3:9
> y
> y[y < 0] <- NA
> y
> rm(x)
> rm(y)
```

4.7.2 Logical Subsetting and Data Frames

First consider the small and simple `healthData` data frame.

```
> healthData
> healthData$Weight[healthData$Gender == "male"]
> healthData[healthData$Gender == "female", ]
> healthData[healthData$Weight > 190, 2:3]
```

The first example is really just subsetting a vector, since the `$` notation creates vectors. The second two examples return subsets of the whole data frame. Note that the logical vector subsets the rows of the data frame, choosing those rows where the gender is female or the weight is more than 190. Note also that the specification for the columns (after the comma) is left blank in the first case, telling R to return all the columns. In the second case the second and third columns are requested explicitly.

Next consider the much larger and more complex `WorldBank` data frame. Recall, the `str` function displays the "structure" of an R object. Here is a look at the structure of several R objects.

```
> str(mtcars)
> str(temporaryList)
> str(WorldBank)
```

First we see that `mtcars` is a data frame which has 32 observations (rows) on each of 11 variables (columns). The names of the variables are given, along with their type (in this case, all numeric), and the first few values of each variable is given.

Second we see that `temporaryList` is a list with three components. Each of the components is described separately, with the first few values again given.

Third we examine the structure of `WorldBank`. It is a data frame with 11880 observations on each of 15 variables. Some of these are character variables, some are numeric, and one (`year`) is integer. Looking at the first few values we see that some variables have missing values.

Consider creating a data frame which only has the observations from one year, say 1971. That's relatively easy. Just choose rows for which `year` is equal to 1971.

```
> WorldBank1971 <- WorldBank[WorldBank$year == 1971, ]
> dim(WorldBank1971)
```

The `dim` function returns the dimensions of a data frame, i.e., the number of rows and the number of columns. From `dim` we see that there are `dim(WorldBank1971)[1]` cases from 1971.

Next, how can we create a data frame which only contains data from 1971, and also only contains cases for which there are no missing values in the fertility rate variable? R has a built in function `is.na` which returns `TRUE` if the observation is missing and returns `FALSE` otherwise. And `!is.na` returns the negation, i.e., it returns `FALSE` if the observation is missing and `TRUE` if the observation is not missing.

```
> WorldBank1971$fertility.rate[1:25]
> !is.na(WorldBank1971$fertility.rate[1:25])
> WorldBank1971 <- WorldBank1971[!is.na(WorldBank1971$fertility.rate),
+   ]
> dim(WorldBank1971)
```

From `dim` we see that there are `dim(WorldBank1971)[1]` cases from 1971 with non-missing fertility rate data.

Return attention now to the original `WorldBank` data frame with data not only from 1971. How can we extract only those cases (rows) which have NO missing data? Consider the following simple example:

```
> temporaryDataFrame <- data.frame(V1 = c(1, 2, 3, 4, NA),
+                                  V2 = c(NA, 1, 4, 5, NA),
+                                  V3 = c(1, 2, 3, 5, 7))
> temporaryDataFrame
> is.na(temporaryDataFrame)
> rowSums(is.na(temporaryDataFrame))
```

First notice that `is.na` will test each element of a data frame for missingness. Also recall that if R is asked to sum a logical vector, it will first convert the logical vector to numeric and then compute the sum, which effectively counts the number of elements in the logical vector which are `TRUE`. The `rowSums` function computes the sum of each row. So `rowSums(is.na(temporaryDataFrame))` returns a vector with as many elements as there are rows in the data frame. If an element is zero, the corresponding row has no missing values. If an element is greater than zero, the value is the number of variables which are missing in that row. This gives a simple method to return all the cases which have no missing data.

```
> dim(WorldBank)
> WorldBankComplete <- WorldBank[rowSums(is.na(WorldBank)) ==
+   0, ]
> dim(WorldBankComplete)
```

Out of the `dim(WorldBankComplete)[1]` rows in the original data frame, only `dim(WorldBankComplete)[1]` have no missing observations!

4.8 Patterned Data

Sometimes it is useful to generate all the integers from 1 through 20, to generate a sequence of 100 points equally spaced between 0 and 1, etc. The R functions `seq()` and `rep()` as well as the “colon operator” `:` help to generate such sequences.

The colon operator generates a sequence of values with increments of 1 or −1.

```
> 1:10
> -5:3
> 10:4
> pi:7
```

The `seq()` function generates either a sequence of pre-specified length or a sequence with pre-specified increments.

```
> seq(from = 0, to = 1, length = 11)
> seq(from = 1, to = 5, by = 1/3)
> seq(from = 3, to = -1, length = 10)
```

The `rep()` function replicates the values in a given vector.

```
> rep(c(1, 2, 4), length = 9)
> rep(c(1, 2, 4), times = 3)
> rep(c("a", "b", "c"), times = c(3, 2, 7))
```

4.9 Practice Exercises

4.10 Homework

Exercise 3 Learning objectives: create, subset, and manipulate vector contents and attributes; summarize vector data using R `table()` and other functions; generate basic graphics using vector data.

Exercise 4 Learning objectives: use functions to describe data frame characteristics; summarize and generate basic graphics for variables held in data frames; apply the subset function with logical operators; illustrate `NA`, `NaN`, `Inf`, and other special values occur; recognize the implications of using floating point arithmetic with logical operators.

Exercise 5 Learning objectives: practice with lists, data frames, and associated functions; summarize variables held in lists and data frames; work with R's linear regression `lm()` function output; review logical subsetting of vectors for partitioning and assigning of new values; generate and visualize data from mathematical functions.

5 Graphics in R

R can be used to create a vast array of graphical representations of data. Creating "standard" graphical displays is straightforward, but a main strength of R is the ability to customize graphical displays to create either non-standard graphics or to modify more standard graphical displays to create publication-ready versions.

There are several packages available in R for creating graphics. The two leading packages are the `graphics` package, which comes with your base installation of R, and the `ggplot2` package, which must be installed and made available by the user.[1] Knowing how to use both the `graphics` and `ggplot2` packages is worthwhile. For beginners `ggplot2` has somewhat simpler syntax, and also produces excellent graphics without much tinkering, so the focus in this book will be on `ggplot2`.

The `gg` in `ggplot2` stands for *Grammar of Graphics*. The package provides a unified and logical way to describe graphical displays such as scatter plots, histograms, bar charts, and many other types of graphics. The grammar describes the mapping from data to the graphical display's aesthetic attributes (color, shape, size) of geometric objects (points, lines, bars). As will become obvious, once this grammar is mastered for a particular type of plot, such as a scatter plot, it is easy to transfer this knowledge to other types of graphics.

Once you work through this chapter, the best place to learn more about `ggplot2` is from the package's official book Wickham and Sievert (2016) by Hadley Wickham. It is available on-line in digital format from MSU's library. The book goes into much more depth on the theory underlying the grammar and syntax, and has many examples on solving practical graphical problems. In addition to the free on-line version available through MSU, the book's source code is available at `https://github.com/hadley/ggplot2-book`.

Another useful resource is the `ggplot2` extensions guide `http://www.ggplot2-exts.org`. This site lists packages that extend `ggplot2`. It's a good place to start if you're trying to do something that seems hard with `ggplot2`. We'll explore a few of these extension packages toward the end of this chapter.

[1]Other graphics packages include `lattice` and `grid`

5.1 Scatter Plots

Scatter plots are a workhorse of data visualization and provide a good entry point to the `ggplot2` system. Begin by considering a simple and classic data set sometimes called *Fisher's Iris Data*. These data are available in R.

```
> data(iris)
> str(iris)
```

The data contain measurements on petal and sepal length and width for `dim(iris)[1]` iris plants. The plants are from one of three species, and the species information is also included in the data frame. The data are commonly used to test classification methods, where the goal would be to correctly determine the species based on the four length and width measurements. To get a preliminary sense of how this might work, we can draw some scatter plots of length versus width. Recall that `ggplot2` is not available by default, so we first have to download and install the package.

```
> install.packages("ggplot2")
```

Once this is done the package is installed on the local hard drive, and we can use the `library` function to make the package available during the current R session.

Next a basic scatter plot is drawn. We'll keep the focus on sepal length and width, but of course similar plots could be drawn using petal length and width. The prompt is not displayed below, since the continuation prompt `+` can cause confusion.

```
library(ggplot2)
ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width)) +
    geom_point()
```

In this case the first argument to the `ggplot` function is the name of the data frame. Second, the `aes` (short for aesthetics) function specifies the mapping to the `x` and `y` axes. By itself the `ggplot` function as written doesn't tell R what sort of graphical display is desired. That is done by adding a `geom` (short for geometry) specification, in this case `geom_point`.

Looking at the scatter plot and thinking about the focus of finding a method to classify the species, two thoughts come to mind. First, the plot might be improved by increasing the size of the points. And second, using different colors for the points corresponding to the three species would help.

```
ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width)) +
    geom_point(size = 4, aes(color=Species))
```

Notice that a legend showing what the colors represent is automatically generated and included in the graphic. Next, the size of the points seems a bit big now, and it might be helpful to use different shapes for the different species.

```
ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width)) +
    geom_point(size = 3, aes(color=Species, shape=Species))
```

Here we see that the legend automatically changes to include species specific color and shape. The size of the points seems more appropriate.

5.1.1 Structure of a Typical ggplot

The examples above start with the function `ggplot()`, which takes as arguments the data frame containing the data to be plotted as well as a mapping from the data to the axes, enclosed by the `aes()` function. Next a `geom` function, in the above case `geom_point()`, is added. It might just specify the geometry, but also might specify aspects such as size, color, or shape.

Typically many graphics are created and discarded in the search for an informative graphic, and often the initial specification of data and basic aesthetics from `ggplot()` stays the same in all the attempts. In such a case it can be helpful to assign that portion of the graphic to an R object, both to minimize the amount of typing and to keep certain aspects of all the graphics constant. Here's how that could be done for the graphics above.

```
iris.p <- ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width))
iris.p + geom_point()
iris.p + geom_point(size = 3, aes(color = Species, shape = Species))
```

5.1.2 Adding lines to a scatter plot

To add a fitted least squares line to a scatter plot, use `stat_smooth`, which adds a smoother (possibly a least squares line, possibly a smooth curve fit to the data, etc.). The argument `method = lm` specifies a line fitted by least squares, and the argument `se = FALSE` suppresses the default display of a confidence band around the line or curve which was fit to the data.

```
ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width)) +
    geom_point(size=3, aes(color=Species))
```

```
ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width)) +
    geom_point(size=3, aes(color=Species)) +
    stat_smooth(method = lm, se=FALSE)
```

For the iris data, it probably makes more sense to fit separate lines by species. This can be specified using the `aes()` function inside `stat_smooth()`.

```
ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width)) +
    geom_point(size=3, aes(color=Species)) +
    stat_smooth(method = lm, se=FALSE, aes(color=Species))
```

In this case we specified the same color aesthetic for the points and the lines. If we know we want this color aesthetic (colors corresponding to species) for all aspects of the graphic, we can specify it in the main `ggplot()` function:

```
ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width, color = Species)) +
    geom_point(size=3) + stat_smooth(method = lm, se=FALSE)
```

Another common use of line segments in a graphic is to connect the points in order, accomplished via the `geom_line()` function. Although it is not clear why this helps in understanding the iris data, the technique is illustrated next, first doing this for all the points in the graphic, and second doing this separately for the three species.

```
ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width)) +
    geom_point(size = 4, aes(color=Species, shape = Species)) +
    geom_line()
```

```
ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width)) +
    geom_point(size = 4, aes(color=Species)) +
    geom_line(aes(color=Species))
```

5.2 Labels, Axes, Text, etc.

The default settings of `ggplot2` often produce excellent graphics, but once a graphic is chosen for dissemination, the user will likely want to customize things like the title, axes, etc. In this section some tools for customization are presented. Most will be illustrated in the context of a data set on crime rates in the 50 states in the United States. These data were made available by Nathan Yau at http://flowingdata.com/2010/11/23/how-to-make-bubble-charts/. The data include crime rates per 100,000 people for various crimes such as murder and robbery, and also include each state's population. The crime rates are from the year 2005, while the population numbers are from the year 2008, but the difference in population between the years is not great, and the exact population is not particularly important for what we'll do below.

First, read in the data, examine its structure, and produce a simple scatter plot of motor vehicle theft versus burglary.

```
> u.crime <- "http://blue.for.msu.edu/FOR875/data/crimeRatesByState2005.csv"
> crime <- read.csv(u.crime, header=TRUE)
> str(crime)
> ggplot(data <- crime, aes(x = burglary, y = motor_vehicle_theft)) +
+     geom_point()
```

5.2.1 Labels

By default axis and legend labels are the names of the relevant columns in the data frame. While convenient, we often want to customize these labels. Here we use `labs()` to change the x and y axis labels and other descriptive text.[2]

```
ggplot(data = crime, aes(x = burglary, y = motor_vehicle_theft)) +
    geom_point() +
    labs(x = "Burglaries per 100,000 population",
         y = "Motor vehicle theft per 100,000 population",
         title = "Burglaries vs motor vehicle theft for US states",
         subtitle = "2005 crime rates and 2008 population",
         caption = "Data from Nathan Yau http://flowingdata.com"
         )
```

5.3 Customizing Axes

`ggplot` also provides default axis extents (i.e., limits) and other axis features. These, and other axis features such as tick marks, labels, and transformations, can be changed using the scale functions. Here the range of the x and y axis is

[2]Axis and legend labels can also be set in the individual scales, see the subsequent sections.

altered to start at zero and go to the maximum of the x and y variables.[3] Here too, axis labels are specified within the scale function, which is an alterative to using the `labs()` function.

```
ggplot(data = crime, aes(x = burglary, y = motor_vehicle_theft)) +
    geom_point() +
    scale_x_continuous(name="Burglaries per 100,000 population",
                       limits=c(0,max(crime$burglary))) +
    scale_y_continuous(name="Motor vehicle theft per 100,000 population",
                       limits = c(0, max(crime$motor_vehicle_theft)))
```

5.3.1 Text, Point Size, and Color

Next we make point size proportional to population, change the color, and add a state label. Note, in the `ggplot()` call I scaled `population` by 100,000 to help with the interpretability of the legend. Accordingly, I also changed the "population" label on the legend to "Population\n(100,000)" using the `labs()` function[4]. We use the `geom_label()` function to add the label, which provides an outline around the label text and allows you to control the box characteristics, e.g., I make the boxes slightly transparent using the `alpha` argument.[5]

```
ggplot(data = crime, aes(x = burglary, y = motor_vehicle_theft,
           size=population/100000)) +
    geom_point(color = "blue") +
    geom_label(aes(label = state), alpha = 0.5) +
    scale_x_continuous(name="Burglaries per 100,000 population",
                       limits=c(0,max(crime$burglary))) +
    scale_y_continuous(name="Motor vehicle theft per 100,000 population",
                       limits = c(0, max(crime$motor_vehicle_theft))) +
    labs(size="Population\n(100,000)")
```

The labels are helpful but just too cluttered. There are some additional arguments that can go into `geom_label()` that allow for label offset; however, this won't help us much here. Instead, we can try the `ggrepel` package by Kamil Slowikowski. This useful package will automatically adjust labels so that they don't overlap. First we need to download and add the package using either RStudio's install package buttons or via `install.packages("ggrepel")`. Next to make all of `ggrepel`'s functions

[3] `ggplot2` makes the axes extend slightly beyond the given range, since typically this is what the user wants.

[4] The \n is the line break and puts "(100,000)" below "Population".

[5] You can also add labels via `geom_text()` function or a `label` argument in the `ggplot()` call.

available we can call `library(ggrepel)` function or, if we know which function we want, we can load only that particular function using the `::` operators. I use `::` below to make clear which function is coming from `ggrepel` and which is coming from `ggplot2`.

```
ggplot(data = crime, aes(x = burglary, y = motor_vehicle_theft,
           size=population/100000)) +
    geom_point(color = "blue") +
    scale_x_continuous(name="Burglaries per 100,000 population",
                       limits=c(0,max(crime$burglary))) +
    scale_y_continuous(name="Motor vehicle theft per 100,000 population",
                       limits = c(0, max(crime$motor_vehicle_theft))) +
    labs(size="Population\n(100,000)") +
    ggrepel::geom_label_repel(aes(label = state), alpha = 0.5)
```

This looks a bit better. We'll resist making additional improvements to the figure for now.

5.4 Other Types of Graphics

Scatter and line plots, which have just been presented, are common but certainly not the only graphical displays in common use. Histograms, boxplots, and bar graphs, as well as more "mathematical" displays such as the graph of a function, are commonly used to represent data. Examples of each are presented below.

5.4.1 Histograms

Simon Newcomb conducted several experiments to estimate the speed of light by measuring the time it took for light to travel from his laboratory to a mirror at the base of the Washington Monument, and then back to his lab. This is a distance of 7.44373 km, and by dividing this distance by the measured time, an estimate for the speed of light is obtained.

The times are of course quite small, and to avoid working with very small numbers, the data are recoded to be the deviation from 24800 nanoseconds. For example an observation coded as 28 represents a time of 24828 nanoseconds, while an observation coded as -44 represents a time of 24756 nanoseconds.

```
> u.newcomb <- "http://blue.for.msu.edu/FOR875/data/Newcomb.csv"
> Newcomb <- read.csv(u.newcomb, header = TRUE)
> head(Newcomb)
```

```
ggplot(Newcomb, aes(x = Time)) + geom_histogram()
```

The software has an algorithm to calculate bin widths for the histogram. Sometimes the algorithm makes choices that aren't suitable (hence the R message above), and these can be changed by specifying a `binwidth`. In addition, the appearance of the bars also can be changed.

```
ggplot(Newcomb, aes(x = Time)) +
    geom_histogram(binwidth = 5, color = "black", fill = "blue" )
```

5.4.2 Boxplots

Next we consider some data from the gap minder data set to construct some box plots. These data are available in the `gapminder` package, which might need to be installed via `install.packages("gapminder")`.

```
library(gapminder)
ggplot(data = subset(gapminder,  year == 2002),
       aes(x = continent, y = gdpPercap)) +
    geom_boxplot(color = "black", fill = "lightblue")
```

Here's the same set of boxplots, but with different colors, different axis labels, and the boxes plotted horizontally rather than vertically.

```
ggplot(data = subset(gapminder,  year == 2002),
       aes(x = continent, y = gdpPercap)) +
    geom_boxplot(color = "red", fill = "lightblue") +
    scale_x_discrete(name = "Continent") +
    scale_y_continuous(name = "Per Capita GDP") + coord_flip()
```

5.4.3 Bar Graphs

As part of a study, elementary school students were asked which was more important to them: good grades, popularity, or athletic ability. Here is a brief look at the data.

```
> u.goals <- "http://blue.for.msu.edu/FOR875/data/StudentGoals.csv"
> StudentGoals <- read.csv(u.goals, header = TRUE)
> head(StudentGoals)
```

First, a simple bar graph of the most important goal chosen is drawn, followed by a stacked bar graph which also includes the student's gender. We then add a side by side bar graph that includes the student's gender.

```
ggplot(StudentGoals, aes(x = Goals)) + geom_bar()
ggplot(StudentGoals, aes(x = Goals, fill = Gender)) + geom_bar()
ggplot(StudentGoals, aes(x = Goals, fill = Gender)) +
    geom_bar(position = "dodge")
```

In this example R counted the number of students who had each goal and used these counts as the height of the bars. Sometimes the data contain the bar heights as a variable. For example, we create a bar graph of India's per capita GDP with separate bars for each year in the data[6].

```
ggplot(subset(gapminder, country == "India"), aes(x = year, y = gdpPercap)) +
    geom_bar(stat = "identity", color = "black", fill = "steelblue2") +
    ggtitle("India's per-capita GDP")
```

5.4.4 Graphs of Functions

One way to create a plot of a mathematical function f is to create a data frame with x values in one column and $f(x)$ values in another column, and then draw a line plot.

```
x <- seq(-pi, pi, len = 1000)
sin.data <- data.frame(x = x, y = sin(x))
ggplot(data = sin.data, aes(x = x, y = y)) + geom_line() +
    scale_y_continuous(name = "sin(x)")
```

This method works well, but with a better understanding of functions in R we will be able to plot mathematical functions in a simpler and more natural way.

[6]R offers a large color palette, run `colors()` on the console to see a list of color names.

5.5 Themes

The theme defines non-data aspects of the plot's characteristics such as background color, axes, and grid lines. Default themes include: `theme_bw()`, `theme_classic()`, `theme_dark()`, `theme_gray()`, `theme_light()`, `theme_linedraw()`, `theme_minimal()`, and `theme_void()`. Changing the theme is as easy as adding it to your initial `ggplot()` call. Here I replace the default implicit `theme_bw()` theme with the classic theme.

```
ggplot(data = sin.data, aes(x = x, y = y)) + geom_line() +
    scale_y_continuous(name = "sin(x)") +
    theme_classic()
```

The `ggthemes` add-on package [https://github.com/jrnold/ggthemes] by Jeffrey Arnold provides a large selection of themes beyond the eight themes that come with `ggplot2`.

5.6 Saving Graphics

We often want to export our graphics to use in an external document or share with colleagues. There are several ways to save graphics in a variety of file formats. The `ggsave()` function will allow you to save your most recent `ggplot()` to a variety of vector (e.g., "eps", "ps", "pdf", "svg") or raster (e.g., "jpeg", "tiff", "png", "bmp", "wmf") formats [7]. The subsequent call to `ggsave()` saves the `sin.data` plot to a pdf file called "sin-plot.pdf".

```
ggplot(filename = "sin-plot.pdf", device="pdf")
```

The `ggplot()` function takes additional arguments to control scale, measurement units, and raster plot resolution, i.e., dots per inch (dpi).

[7] Vector files comprise lines and curves known as paths, whereas raster files are comprised of pixels. Vector images are often preferred for publication quality graphics because they can be edited, scale well, and provide crisper detail.

5.7 More Resources

In summary, `ggplot2` provides a fairly intuitive[8] framework for developing an enormous variety of graphics. In addition to the resources mentioned at the beginning of this chapter, there are numerous online `ggplot2` resources and galleries to get ideas for creating beautiful graphics to convey the stories in your data. See, for example,

- `http://docs.ggplot2.org`
- `http://www.r-graph-gallery.com/portfolio/ggplot2-package`
- `http://www.ggplot2-exts.org/gallery`
- `http://www.cookbook-r.com/Graphs`
- and of course www.google.com

While the built-in `ggplot2` package documentation (accessible via the help tab in RStudio) is helpful, the official online documentation at `http://docs.ggplot2.org` is particularly useful because it provides example plots and easy navigation between related topics. The large number number of functions and syntax in `ggplot2` can be daunting. RStudio provides some handy cheat-sheets to help you along www.rstudio.com/resources/cheatsheets or direct link www.rstudio.com/wp-content/uploads/2016/11/ggplot2-cheatsheet-2.1.pdf.

`ggplot2` also has an active mailing list at `http://groups.google.com/group/ggplot2`. The list is an excellent resource for users at all stages of experience. Another useful resource is stackoverflow, `http://stackoverflow.com`. There is an active `ggplot2` community on stackoverflow, and many common questions have already been asked and answered. When posting questions on any programming mailing list, it is best to provide a minimal reproducible example of your issue. The `reprex` `https://github.com/jennybc/reprex` package by Jenny Bryan provides a convenient way to do this, and also includes advice on creating a good example. The more information you provide about your issue, the more likely the community is to help you.

[8]Like everything else in this book, it takes practice to get used to the syntax.

5.8 Practice Exercises

5.9 Homework

Exercise 6 Learning objectives: practice using `ggplot2` functions; summarize variables using graphics; introduce `ggplot2` facets.

6
Working with Data

Bringing data into R, exporting data from R in a form that is readable by other software, cleaning and reshaping data, and other data manipulation tasks are an important and often overlooked component of data science. The book Spector (2008), while a few years old, is still an excellent reference for data-related issues. And the *R Data Import/Export* manual, available online at `https://cran.r-project.org/doc/manuals/r-release/R-data.html`, is an up-to-date (and free) reference on importing a wide variety of datasets into R and on exporting data in various forms.

6.1 Reading Data into R

Data come in a dizzying variety of forms. It might be in a proprietary format such as an .xlsx[1] Excel file, a .sav[2] SPSS file, or a .mtw[3] Minitab file. It might be structured using a relational model[4] comprising many tables that need to be connected via key-value pairs. It might be a data-interchange format such as JSON[5] (JavaScript Object Notation), or a markup language such as XML[6] (Extensible Markup Language), perhaps with specialized standards for describing ecological information, see EML[7] (Ecological Metadata Language). Both XML and EML are common data metadata formats (i.e., data that provides information about other data). Fortunately many datasets are (or can be) saved as plain text files, and most software can both read and write such files, so our initial focus will be on reading plain text files into R and saving data from R in plain text format. RStudio provides a handy data import cheat sheet[8] for many of the read functions detailed in this section.

The `foreign` R package provides functions to directly read data saved in

[1]https://en.wikipedia.org/wiki/Microsoft_Excel
[2]https://en.wikipedia.org/wiki/SPSS
[3]https://en.wikipedia.org/wiki/Minitab
[4]https://en.wikipedia.org/wiki/Relational_model%7D%7Brelational%20model
[5]http://www.json.org/
[6]https://en.wikipedia.org/wiki/XML
[7]https://en.wikipedia.org/wiki/Ecological_Metadata_Language
[8]http://www.rstudio.com/resources/cheatsheets

some of the proprietary formats into R, which is sometimes unavoidable, but if possible it is good to save data from another package as plain text and then read this plain text file into R. In Chapter **??** methods for reading web-based data sets into R will be discussed.

The function `read.table()` and its offshoots such as `read.csv()` are used to read in rectangular data from a text file. For example, the file `BrainAndBody.csv` contains data[9] on the brain weight, body weight, and name of some terrestrial animals. Here are the first few lines of that file:

```
body,brain,name
1.35,8.1,Mountain beaver
465,423,Cow
36.33,119.5,Grey wolf
27.66,115,Goat
1.04,5.5,Guinea pig
```

As is evident, the first line of the file contains the names of the three variables, separated (delimited) by commas. Each subsequent line contains the body weight, brain weight, and name of a specific terrestrial animal.

This file is accessible at the url http://blue.for.msu.edu/FOR875/data/BrainAndBody.csv. The `read.table()` function is used to read these data into an R data frame.

```
> u.bb <- "http://blue.for.msu.edu/FOR875/data/BrainAndBody.csv"
> BrainBody <- read.table(file = u.bb, header = TRUE, sep = ",",
+                          stringsAsFactors = FALSE)
> head(BrainBody)
```

The arguments used in this call to `read.table()` include:

1. `file = u.bb`, which tells R the location of the file. In this case the string `http://blue.for.msu.edu/FOR875/data/BrainAndBody.csv` giving the location is rather long, so it was first assigned to the object `u.bb`.
2. `header = TRUE`, which tells R the first line of the file gives the names of the variables.
3. `sep = ","`, which tells R that a comma separates the fields in the file.
4. `stringsAsFactors = FALSE` which tells R not to convert character vectors to factors.

The function `read.csv()` is the same as `read.table()` except the default

[9]These data come from the `MASS` R library.

separator is a comma, whereas the default separator for `read.table()` is whitespace.

The file `BrainAndBody.tsv` contains the same data, except a tab is used in place of a comma to separate fields. The only change needed to read in the data in this file is in the `sep` argument (and of course the `file` argument, since the data are stored in a different file):

```
> u.bb <- "http://blue.for.msu.edu/FOR875/data/BrainAndBody.tsv"
> BrainBody2 <- read.table(file = u.bb, header = TRUE, sep = "\t",
+                          stringsAsFactors = FALSE)
> head(BrainBody2)
```

File extensions, e.g., `.csv` or `.tsv`, are naming conventions only and are there to remind us how the columns are separated. In other words, they have no influence on R's file read functions.

A third file, `BrainAndBody.txt`, contains the same data, but also contains a few lines of explanatory text above the names of the variables. It also uses whitespace rather than a comma or a tab as a separator. Here are the first several lines of the file.

```
This file contains data
on brain and body
weights of several terrestrial animals

"body" "brain" "name"
1.35 8.1 "Mountain beaver"
465 423 "Cow"
36.33 119.5 "Grey wolf"
27.66 115 "Goat"
1.04 5.5 "Guinea pig"
11700 50 "Dipliodocus"
2547 4603 "Asian elephant"
```

Notice that in this file the values of `name` are put inside of quotation marks. This is necessary since instead R would (reasonably) assume the first line contained the values of four variables, the values being `1.35`, `8.1`, `Mountain`, and `beaver` while in reality there are only three values desired, with `Mountain beaver` being the third.

To read in this file we need to tell R to skip the first five lines and to use whitespace as the separator. The `skip` argument handles the first, and the `sep` argument the second. First let's see what happens if we don't use the `skip` argument.

```
> u.bb <- "http://blue.for.msu.edu/FOR875/data/BrainAndBody.txt"
> BrainBody3 <- read.table(u.bb, header = TRUE, sep = " ",
+                          stringsAsFactors = FALSE)
```

R assumed the first line of the file contained the variable names, since `header = TRUE` was specified, and counted four including `This`, `file`, `contains`, and `data`. So in the first line of actual data, R expected four columns containing data plus possibly a fifth column containing row names for the data set, and complained that "line 1 did not have 5 elements." The error message is somewhat mysterious, since it starts with "Error in scan." This happens because `read.table()` actually uses a more basic R function called `scan()` to do the work.

Here's how to read in the file correctly.

```
> u.bb <- "http://blue.for.msu.edu/FOR875/data/BrainAndBody.txt"
> BrainBody3 <- read.table(u.bb, header = TRUE, sep = " ",
+                          stringsAsFactors = FALSE, skip = 4)
> BrainBody3[1:10,]
```

6.2 Reading Data with Missing Observations

Missing data are represented in many ways. Sometimes a missing data point is just that, i.e., the place where it should be in the file is blank. Other times specific numbers such as −9999 or specific symbols are used. The `read.table()` function has an argument `na.string` that allows the user to specify how missing data is indicated in the source file.

The site `http://www.wunderground.com/history/` makes weather data available for locations around the world from dates going back to 1945. The file `WeatherKLAN2014.csv` contains weather data for Lansing, Michigan for the year 2014. Here are the first few lines of that file:

```
EST,Max TemperatureF,Min TemperatureF, Events
1/1/14,14,9,Snow
1/2/14,13,-3,Snow
1/3/14,13,-11,Snow
1/4/14,31,13,Snow
1/5/14,29,16,Fog-Snow
1/6/14,16,-12,Fog-Snow
1/7/14,2,-13,Snow
```

```
1/8/14,17,-1,Snow
1/9/14,21,2,Snow
1/10/14,39,21,Fog-Rain-Snow
1/11/14,41,32,Fog-Rain
1/12/14,39,31,
```

Look at the last line, and notice that instead of an `Event` such as `Snow` or `Fog-Snow` there is nothing after the comma. This observation is missing, but rather than using an explicit code such as `NA`, the site just leaves that entry blank. To read these data into R we will supply the argument `na.string = ""` which tells R the file indicates missing data by leaving the appropriate entry blank.

```
> u.weather <- "http://blue.for.msu.edu/FOR875/data/WeatherKLAN2014.csv"
> WeatherKLAN2014 <- read.csv(u.weather, header=TRUE,
+                             stringsAsFactors = FALSE, na.string = "")
> WeatherKLAN2014[1:15,]
```

6.3 Summarizing Data Frames

Some common data tasks include variable summaries such as means or standard deviations, transforming an existing variable, and creating new variables. As with many tasks, there are several ways to accomplish each of these.

6.3.1 Column (and Row) Summaries

The file `WeatherKLAN2014Full.csv` contains a more complete set of weather data variables than `WeatherKLAN2014.csv`, from the same source, http://www.wunderground.com/history.

```
> u.weather <- "http://blue.for.msu.edu/FOR875/data/WeatherKLAN2014Full.csv"
> WeatherKLAN2014Full <- read.csv(u.weather, header=TRUE,
+                                 stringsAsFactors = FALSE,
+                                 na.string = "")
> names(WeatherKLAN2014Full)
```

How can we compute the mean for each variable? One possibility is to do this a variable at a time:

```
> mean(WeatherKLAN2014Full$Mean.TemperatureF)
> mean(WeatherKLAN2014Full$Min.TemperatureF)
> mean(WeatherKLAN2014Full$Max.TemperatureF)
> ## Et Cetera
```

This is pretty inefficient. Fortunately there is a `colMeans()` function which computes the mean of each column (or a specified number of columns) in a data frame. Some columns in the current data frame are not numeric, and obviously we don't want to ask R to compute means for these columns. We use `str()` to investigate.

```
> str(WeatherKLAN2014Full)
```

It isn't surprising that `EST` and `Events` are not numeric, but is surprising that `PrecipitationIn`, which measures precipitation in inches, also is not numeric, but is character. Let's investigate further.

```
> WeatherKLAN2014Full$PrecipitationIn[1:50]
```

Now it's more clear. The original data file included `T` in the precipitation column to represent a "trace" of precipitation, which is precipitation greater than 0 but less than 0.01 inches. One possibility would be to set all these values to `"0"`, and then to convert the column to numeric. For now we will just leave the `PrecipitationIn` column out of the columns for which we request the mean.

```
> colMeans(WeatherKLAN2014Full[, c(2:19, 21, 23)])
```

6.3.2 The `apply()` Function

R also has functions `rowMeans()`, `colSums()`, and `rowSums()`. But what if we want to compute the median or standard deviation of columns of data, or some other summary statistic? For this the `apply()` function can be used. This function applies a user-chosen function to either the rows or columns (or both) of a data frame. The arguments are:

1. `X`: the data frame of interest
2. `MARGIN`: specifying either rows (`MARGIN = 1`) or columns (`MARGIN = 2`)
3. `FUN`: the function to be applied.

```
> apply(X = WeatherKLAN2014Full[, c(2:19, 21, 23)], MARGIN = 2,
+   FUN = sd)
```

As with any R function the arguments don't need to be named as long as they are specified in the correct order, so

```
> apply(WeatherKLAN2014Full[, c(2:19, 21, 23)], 2, sd)
```

has the same result.

6.3.3 Saving Typing Using `with()`

Consider calculating the mean of the maximum temperature values for those days where the cloud cover is less than 4 and when the maximum humidity is over 85. We can do this using subsetting.

```
> mean(WeatherKLAN2014Full$Max.TemperatureF[
+                              WeatherKLAN2014Full$CloudCover < 4 &
+                              WeatherKLAN2014Full$Max.Humidity > 85])
```

While this works, it requires a lot of typing, since each time we refer to a variable in the data set we need to preface its name by `WeatherKLAN2014Full$`. The `with()` function tells R that we are working with a particular data frame, and we don't need to keep typing the name of the data frame.

```
> with(WeatherKLAN2014Full, mean(Max.TemperatureF[CloudCover <
+   4 & Max.Humidity > 85]))
```

6.4 Transforming a Data Frame

Variables are often added to, removed from, changed in, or rearranged in a data frame. The subsetting features of R make this reasonably easy. We will investigate this in the context of the `gapminder` data frame. If the `gapminder` library is not yet installed, use `install.packages("gapminder")` to install it locally.

```
> library(gapminder)
> str(gapminder)
```

6.4.1 Adding Variables

The data frame contains per capita GDP and population, and it might be interesting to create a variable that gives the total GDP by multiplying these two variables. (If we were interested in an accurate value for the total GDP we would probably be better off getting this information directly, since it is likely that the per capita GDP values in the data frame are rounded substantially.)

```
> gapminder$TotalGDP <- gapminder$gdpPercap * gapminder$pop
> str(gapminder)
```

Analogous to the `with()` function, there is a function `within()` which can simplify the syntax. Whereas `with()` does not change the data frame, `within()` can. Note, below I first remove the altered gapminder dataframe using `rm()` then bring a clean copy back in by reloading the `gapminder` package.

```
> rm(gapminder)
> library(gapminder)
> str(gapminder)
> gapminder <- within(gapminder, TotalGDP <- gdpPercap * pop)
> str(gapminder)
```

A nice feature of `within()` is its ability to add more than one variable at a time to a data frame. In this case the two or more formulas creating new variables must be enclosed in braces.

```
> gapminder <- within(gapminder, {TotalGDP <- gdpPercap * pop
+     lifeExpMonths <- lifeExp * 12})
> str(gapminder)
```

6.4.2 Removing Variables

After reflection we may realize the new variables we added to the `gapminder` data frame are not useful, and should be removed.

```
> str(gapminder)
> gapminder <- gapminder[1:6]
> str(gapminder)
```

The same result could be obtained via `gapminder <- gapminder[, 1:6]`. The first method uses the fact that a data frame is also a list, and uses list subsetting methods. It is slightly preferable, since even if only one variable is retained, the object will still be a data frame, while the other method can return a vector in this case. Note this difference in the resulting `x` variable below (again this behavior can be frustrating at times if it is not anticipated).

```
> a <- data.frame(x = 1:3, y = c("dog", "cat", "pig"), z = seq(from = 1,
+   to = 2, length = 3))
> a
> a <- a[1]
> a
> a <- data.frame(x = 1:3, y = c("dog", "cat", "pig"), z = seq(from = 1,
+   to = 2, length = 3))
> a
> a <- a[, 1]
> a
```

One can also use a negative sign in front of the variable number(s). For example, `a[-(2:3)]` would drop the first two columns of `a`. Some care is needed when removing variables using the negative sign—what happens if you write `a[-2:3]` instead of `a[-(2:3)]` (why are the parentheses important here)?

An alternative approach is to set the variables you'd like to remove to `NULL`. For example, `a[c("y","z")] <- NULL` and `a[,2:3] <- NULL` produce the same result as above.

6.4.3 Transforming Variables

Consider the gapminder data again. Possibly we don't want to add a new variable that gives life expectancy in months, but rather want to modify the existing variable to measure life expectancy in months. Here are two ways to accomplish this.

```
> rm(gapminder)
> library(gapminder)
> gapminder$lifeExp[1:5]
> gapminder$lifeExp <- gapminder$lifeExp * 12
> gapminder$lifeExp[1:5]
> rm(gapminder)
> library(gapminder)
> gapminder$lifeExp[1:5]
> gapminder <- within(gapminder, lifeExp <- lifeExp * 12)
> gapminder$lifeExp[1:5]
```

6.5 Rearranging Variables

Consider the full weather data set again.

```
> u.weather <- "http://blue.for.msu.edu/FOR875/data/WeatherKLAN2014Full.csv"
> WeatherKLAN2014Full <- read.csv(u.weather, header=TRUE,
+                                stringsAsFactors = FALSE,
+                                na.string = "")
> names(WeatherKLAN2014Full)
```

If we want the wind speed variables to come right after the date, we can again use subsetting.

```
> WeatherKLAN2014Full <- WeatherKLAN2014Full[c(1, 17, 18,
+   19, 2:16, 20:23)]
> names(WeatherKLAN2014Full)
```

6.6 Reshaping Data

A data set can be represented in several different formats. Consider a (fictitious) data set on incomes of three people during three different years. Here is one representation of the data:

```
> yearlyIncomeWide
```

Here is another representation of the same data:

```
> yearlyIncomeLong
```

For hopefully obvious reasons, the first representation is called a *wide* representation of the data, and the second is called a *long* representation. Each has its merits. The first representation is probably easier for people to read, while the second is often the form needed for analysis by statistical software such as R. There are of course other representations. For example the rows and columns could be interchanged to create a different wide representation, or the long representation, which currently groups data by year, could group by name instead.

Whatever the relative merits of wide and long representations of data, transforming data from wide to long or long to wide is often required. As with many tasks, there are several ways to accomplish this in R. We will focus on a library called `tidyr` written by Hadley Wickham that performs the transformations and more.

6.6.1 `tidyr`

The R library `tidyr` has functions for converting data between formats. To illustrate its use, we examine a simple data set that explores the relationship between religion and income in the United States. The data come from a Pew survey, and are used in the `tidyr` documentation to illustrate transforming data from wide to long format.

```
> u.rel <- "http://blue.for.msu.edu/FOR875/data/religion2.csv"
> religion <- read.csv(u.rel, header = TRUE, stringsAsFactors = FALSE)
> head(religion)
```

As given, the columns include religion and income level, and there are counts for each of the combinations of religion and income level. For example, there are 27 people who are Agnostic and whose income is less than 10 thousand dollars, and there are 617 people who are Catholic and whose income is between 10 and 20 thousand dollars.

The `gather()` function can transform data from wide to long format.

```
> library(tidyr)
> religionLong <- gather(data = religion, key = IncomeLevel,
+                        value = Frequency, 2:11)
> head(religionLong)
> tail(religionLong)
```

To use `gather()` we specified the data frame (`data = religion`), the name we want to give to the column created from the income levels (`key = IncomeLevel`), the name we want to give to the column containing the frequency values (`value = Frequency`) and the columns to gather (`2:11`).

Columns to be gathered can be specified by name also, and we can also specify which columns should be omitted using a negative sign in front of the name(s). So the following creates an equivalent data frame:

```
> religionLong <- gather(data = religion, key = IncomeLevel,
+                        value = Frequency, -religion)
> head(religionLong)
> religionWide <- spread(data = religionLong, key = IncomeLevel,
+   value = Frequency)
> head(religionWide)
```

Here we specify the data frame (`religionLong`), the column (`IncomeLevel`) to be spread, and the column of values (`Frequency`) to be spread among the newly created columns. As can be seen, this particular call to `spread()` yields the original data frame.

`tidyr` provides two other useful functions to separate and unite variables based on some deliminator. Consider again the `yearlyIncomeWide` table. Say we want to split the `name` variable into first and last name. This can be done using the `separate()` function.

```
> firstLast <- separate(data = yearlyIncomeLong, col = name,
+                       into = c("first", "last"), sep="\\s")
> print(firstLast)
```

Now say, you're not happy with that and you want to combine the name column again, but this time separate the first and last name with a underscore. This is done using the `unite()` function.

```
> unite(firstLast, col = name, first, last, sep = "_")
```

6.7 Manipulating Data with `dplyr` #{dplyr}

Much of the effort (a figure of 80% is sometimes suggested) in data analysis is spent cleaning the data and getting it ready for analysis. Having effective tools for this task can save substantial time and effort. The R package `dplyr` written by Hadley Wickham is designed, in Hadley's words, to be "a grammar of data manipulation, providing a consistent set of verbs that help you solve the most common data manipulation challenges." Casting data analysis tasks in terms of "grammar" should be familiar from our work with the `ggplot2` package, which was also authored by Hadley. Functions provided by `dplyr` do in fact capture key data analysis actions (i.e., verbs). These functions include

- `mutate()` adds new variables that are functions of existing variables
- `select()` picks variables based on their names
- `filter()` picks cases based on their values
- `summarize()` reduces multiple values down to a single summary

- `arrange()` changes the ordering of the rows.

These all combine naturally with a `group_by()` function that allows you to perform any operation grouped by values of one or more variables. All the tasks done using `dplyr` can be accomplished using tools already covered in this text; however, `dplyr`'s functions provide a potentially more efficient and convenient framework to accomplish these tasks. RStudio provides a convenient data wrangling cheat sheet[10] that covers many aspects of the `tidyr` and `dplyr` packages.

This somewhat long section on `dplyr` adapts the nice introduction by Jenny Bryan, available at `http://stat545-ubc.github.io/block010_dplyr-end-single-table.html`.

6.7.1 Improved Data Frames

The `dplyr` package provides two functions that offer improvements on data frames. First, `data_frame()` is a trimmed down version of the `data.frame()` function that is somewhat more user friendly, and won't be discussed here. Second, `tbl_df()` creates a tibble[11]. A tibble has two advantages over a data frame. First, when printing, it only prints the first ten rows and the columns that fit on the page, as well as some additional information about the table's dimension, data type of variables, and non-printed columns. Second, recall that subsetting a data frame can sometimes return a vector rather than a data frame (if only one row or column is the result of the subset), a tibble does not have this behavior. Here is an example using the `religionWide` data frame.[13]

```
> library(dplyr)
> head(religionWide)
> religionWide[, 1]
> religionWideTbl <- tbl_df(religionWide)
> head(religionWideTbl)
> religionWideTbl[, 1]
```

As seen above, note that once the data frame is reduced to one dimension by subsetting to one column, it is no longer a data frame and has been *simplified* to a vector. This might not seem like a big deal; however, it can be very

[10]`https://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf`

[11]Reminds me of The Trouble with Tribbles[12]

[13]The text printed immediately after `library(dplyr)` means the `stats` and `base` packages, which are automatically loaded when you start R, have functions with the same name as functions in `dplyr`. So, for example, if you call the `filter()` or `lag()` functions, R will use `library(dplyr)`'s functions. Use the `::` operator to explicity identify which packages' function you want to use, e.g., if you want `stats`'s `lag()` then call `stats::lag()`.

frustrating and potentially break your code when you expect an object to behave like a data frame and it doesn't because it's now a vector. Alternatively, once we convert `religionWide` to a `tbl_df` via the `tbl_df()` function the object remains a data frame even when subsetting down to one dimension (there is no automatic simplification). Converting data frames using `tbl_df()` is not required for using `dplyr` but is convenient. Also, it is important to note that `tbl_df` is simply a wrapper around a data frame that provides some additional behaviors. The newly formed `tbl_df` object will still behave like a data frame (because it technically still is a data frame) but will have some added niceties (some of which are illustrated below).

6.7.2 Filtering Data by Row

Recall the `gapminder` data. These data are available in tab-separated format in `gapminder.tsv`, and can be read in using `read.delim()` (or the related read functions described previously). The `read.delim()` function defaults to `header = TRUE` so this doesn't need to be specified explicitly. In this section we will be working with the `gapminder` data often, so we will use a short name for the data frame to save typing.

```
> u.gm <- "http://blue.for.msu.edu/FOR875/data/gapminder.tsv"
> gm <- read.delim(u.gm)
> gm <- tbl_df(gm)
> str(gm)
> head(gm)
```

Filtering helps us to examine subsets of the data such as data from a particular country, from several specified countries, from certain years, from countries with certain populations, etc. Some examples:

```
> filter(gm, country == "Brazil")
> filter(gm, country %in% c("Brazil", "Mexico"))
> filter(gm, country %in% c("Brazil", "Mexico") & year %in%
+   c(1952, 1972))
> filter(gm, pop > 3e+08)
> filter(gm, pop > 3e+08 & year == 2007)
```

Notice the full results are not printed. For example, when we asked for the data for Brazil and Mexico, only the first ten rows were printed. This is an effect of using the `tbl_df()` function. Of course if we wanted to analyze the results (as we will below) the full set of data would be available.

6.7.3 Selecting variables by column

Continuing with the `gapminder` data, another common task is to restrict attention to some subset of variables in the data set. The `select()` function does this.

```
> dplyr::select(gm, country, year, lifeExp)
> dplyr::select(gm, 2:4)
> dplyr::select(gm, -c(2, 3, 4))
> dplyr::select(gm, starts_with("c"))
```

Notice a few things. Variables can be selected by name or column number. As usual, a negative sign tells R to leave something out. And there are special functions such as `starts_with` that provide ways to match part of a variable's name.

6.7.4 Pipes

Consider selecting the country, year, and population for countries in Asia or Europe. One possibility is to nest a `filter()` function inside a `select()` function.

```
> dplyr::select(filter(gm, continent %in% c("Asia", "Europe")),
+   country, year, pop)
```

Even a two-step process like this becomes hard to follow in this nested form, and often we will want to perform more than two operations. There is a nice feature in `dplyr` that allows us to "feed" results of one function into the first argument of a subsequent function. Another way of saying this is that we are "piping" the results into another function. The `%>%` operator does the piping. Here we again restrict attention to country, year, and population for countries in Asia or Europe[14].

```
> gm %>%
+   filter(continent %in% c("Asia", "Europe")) %>%
+   dplyr::select(country, year, pop)
```

It can help to think of `%>%` as representing the word "then". The above can be read as, "Start with the data frame `gm` *then* filter it to select data from the continents Asia and Europe *then* select the variables country, year, and population from these data".

[14]Notice the indentation used in the code. This is not necessary, as the code could be all on one line, but I often find it easier to read in this more organized format

The pipe operator `%>%` is not restricted to functions in `dplyr`. In fact the pipe operator itself was introduced in another package called `magrittr`, but is included in `dplyr` as a convenience.

6.7.5 Arranging Data by Row

By default the `gapminder` data are arranged by country and then by year.

```
> head(gm, 15)
```

Possibly arranging the data by year and then country would be desired. The `arrange()` function makes this easy. We will again use pipes.

```
> gm %>% arrange(year, country)
```

How about the data for Rwanda, arranged in order of life expectancy.

```
> gm %>%
+   filter(country == "Rwanda") %>%
+   arrange(lifeExp)
```

Possibly we want these data to be in decreasing (descending) order. Here, `desc()` is one of many `dplyr` helper functions.

```
> gm %>%
+   filter(country == "Rwanda") %>%
+   arrange(desc(lifeExp))
```

Possibly we want to include only the year and life expectancy, to make the message more stark.

```
> gm %>%
+   filter(country == "Rwanda") %>%
+   dplyr::select(year, lifeExp) %>%
+   arrange(desc(lifeExp))
```

For analyzing data in R, the order shouldn't matter. But for presentation to human eyes, the order is important.

6.7.6 Renaming Variables

The `dplyr` package has a `rename` function that makes renaming variables in a data frame quite easy.

```
> gm <- rename(gm, population = pop)
> head(gm)
```

6.7.7 Data Summaries and Grouping

The `summarize()` function computes summary statistics using user provided functions for one or more columns of data in a data frame.

```
> summarize(gm, meanpop = mean(population), medpop = median(population))
> ##or
> gm %>%
+   summarize(meanpop = mean(population), medpop = median(population))
```

Often we want summaries for specific components of the data. For example, we might want the median life expectancy for each continent separately. One option is subsetting:

```
> median(gm$lifeExp[gm$continent == "Africa"])
> median(gm$lifeExp[gm$continent == "Asia"])
> median(gm$lifeExp[gm$continent == "Europe"])
> median(gm$lifeExp[gm$continent == "Americas"])
> median(gm$lifeExp[gm$continent == "Oceania"])
```

The `group_by()` function makes this easier, and makes the output more useful.

```
> gm %>%
+   group_by(continent) %>%
+   summarize(medLifeExp = median(lifeExp))
```

Or if we want the results ordered by the median life expectancy:

```
> gm %>%
+   group_by(continent) %>%
+   summarize(medLifeExp = median(lifeExp)) %>%
+   arrange(medLifeExp)
```

As another example, we calculate the number of observations we have per continent (using the `n()` helper function), and then, among continents, how many distinct countries are represented (using `n_distinct()`).

```
> gm %>%
+   group_by(continent) %>%
+   summarize(numObs = n())
> gm %>%
+   group_by(continent) %>%
+   summarize(n_obs = n(), n_countries = n_distinct(country))
```

Here is a bit more involved example that calculates the minimum and maximum life expectancies for countries in Africa by year.

```
> gm %>%
+   filter(continent == "Africa") %>%
+   group_by(year) %>%
+   summarize(min_lifeExp = min(lifeExp), max_lifeExp = max(lifeExp))
```

This is interesting, but the results don't include the countries that achieved the minimum and maximum life expectancies. Here is one way to achieve that. We will start with the minimum life expectancy. Note the rank of the minimum value will be 1.

```
> gm %>%
+   dplyr::select(country, continent, year, lifeExp) %>%
+   group_by(year) %>%
+   arrange(year) %>%
+   filter(rank(lifeExp) == 1)
```

Next we add the maximum life expectancy. Here we need to better understand the `desc()` function, which will transform a vector into a numeric vector which will be sorted in descending order. Here are some examples.

```
> desc(1:5)
> desc(c(2, 3, 1, 5, 6, -4))
> desc(c("a", "c", "b", "w", "e"))
```

We now use this to extract the maximum life expectancy. Recall that `|` represents "or". Also by default only the first few rows of a `tbl_df` object will be printed. To see all the rows we pipe the output to `print(n = 24)` to ask for all 24 rows to be printed.

```
> gm %>%
+   dplyr::select(country, continent, year, lifeExp) %>%
+   group_by(year) %>%
+   arrange(year) %>%
+   filter(rank(lifeExp) == 1 | rank(desc(lifeExp)) == 1) %>%
+   print(n=24)
```

6.7.8 Creating New Variables

The `$` notation provides a simple way to create new variables in a data frame. The `mutate()` function provides another, sometimes cleaner way to do this. We will use `mutate()` along with the `lag()` function to investigate changes in life expectancy over five years for the `gapminder` data. We'll do this in a few steps. First, we create a variable that measures the change in life expectancy and remove the population and GDP variables that are not of interest. We have to be careful to first group by country, since we want to calculate the change in life expectancy by country.

```
> gm %>%
+   group_by(country) %>%
+   mutate(changeLifeExp = lifeExp - lag(lifeExp, order_by = year)) %>%
+   dplyr::select(-c(population, gdpPercap))
```

Next, summarize by computing the largest drop in life expectancy.

```
> gm %>%
+   group_by(country) %>%
+   mutate(changeLifeExp = lifeExp - lag(lifeExp, order_by = year)) %>%
+   dplyr::select(-c(population, gdpPercap)) %>%
+   summarize(largestDropLifeExp = min(changeLifeExp))
```

Oops. We forgot that since we don't have data from before 1952, the first drop will be `NA`. Let's try again.

```
> gm %>%
+   group_by(country) %>%
+   mutate(changeLifeExp = lifeExp - lag(lifeExp, order_by = year)) %>%
+   select(-c(population, gdpPercap)) %>%
+   summarize(largestDropLifeExp = min(changeLifeExp, na.rm = TRUE))
```

That's not quite what we wanted. We could arrange the results by the life expectancy drop, but it would be good to have both the continent and year printed out also. So we'll take a slightly different approach, by arranging the results in increasing order.

```
> gm %>%
+   group_by(country) %>%
+   mutate(changeLifeExp = lifeExp - lag(lifeExp, order_by = year)) %>%
+   dplyr::select(-c(population, gdpPercap)) %>%
+   arrange(changeLifeExp)
```

That's still not quite right. Because the data are grouped by country, R did the ordering within group. If we want to see the largest drops overall, we need to remove the grouping.

```
> gm %>% group_by(country) %>% mutate(changeLifeExp = lifeExp -
+   lag(lifeExp, order_by = year)) %>% dplyr::select(-c(population,
+   gdpPercap)) %>% ungroup() %>% arrange(changeLifeExp) %>%
+   print(n = 20)
```

6.8 Practice Exercises

1. In Section 6.3.2 we computed the standard deviation across all numeric columns using the following line of code:

```
> apply(WeatherKLAN2014Full[, c(2:19, 21, 23)], 2, sd)
```

Notice the output value of `NA` for the column `Max.Gust.SpeedMPH`. Why does this happen? Figure out a way to make the `apply()` function return a numeric value for this column. *Hint*: check out the `sd()` definition.

6.9 Homework

Exercise 7 Learning objectives: introduce `with()`, `tapply()`, and `cut()` functions; summarize data using the `table()` function with logical subsetting; practice using factor data types.

Exercise 8 Learning objectives: work with messy data; import data from an external spreadsheet; practice using functions in `tidyr` and `ggplot2`.}

Exercise 9 {Learning objectives: work with several key **dplyr** functions; manipulate data frames (actually tibbles); summarize and visualize data from large data files.}

7

Functions and Programming

We have been working with a wide variety of R functions, from simple functions such as `mean()` and `sd()` to more complex functions such as `ggplot()` and `apply()`. Gaining a better understanding of existing functions and the ability to write your own functions dramatically increases what we can do with R. Learning about R's programming capabilities is an important step in gaining facility with functions.

7.1 R Functions

Data on the yield (pounds per acre) of two types of corn seeds (regular and kiln dried) were collected. Each of the 11 plots of land was split into two subplots, and one of the subplots was planted in regular corn while the other was planted in kiln dried corn. These data were analyzed in a famous paper authored by William Gosset. Here are the data.

```
> u.corn <- "http://blue.for.msu.edu/FOR875/data/corn.csv"
> corn <- read.csv(u.corn, header = TRUE)
> corn
```

A paired t test, or a confidence interval for the mean difference, may be used to assess the difference in yield between the two varieties. Of course R has a function `t.test` that performs a paired t test and computes a confidence interval, but we will perform the test without using that function. We will focus for now on testing the hypotheses $H_0 : \mu_d = 0$ versus $H_a : \mu_d \neq 0$ and on a two-sided confidence interval for μ_d. Here μ_d represents the population mean difference.

The paired t statistic is defined by

$$t = \frac{\overline{d}}{S_d/\sqrt{n}} \tag{7.1}$$

where $\overline{d}$ is the mean of the differences, S_d is the standard deviation of the differences, and n is the sample size. The p-value is twice the area to the right of $|t_{\text{obs}}|$, where t_{obs} is the observed t statistic, and a confidence interval is given by

$$\overline{d} \pm t^*(S_d/\sqrt{n}). \tag{7.2}$$

Here t^* is an appropriate quantile of a t distribution with $n-1$ degrees of freedom.

```
> dbar <- mean(corn$kiln_dried - corn$regular)
> n <- length(corn$regular)
> S_d <- sd(corn$kiln_dried - corn$regular)
> t_obs <- dbar/(S_d/sqrt(n))
> t_obs
>
> pval <- 2 * (1 - pt(abs(t_obs), n - 1))
> pval
>
> margin <- qt(0.975, n - 1) * (S_d/sqrt(n))
> lcl <- dbar - margin
> ucl <- dbar + margin
> lcl
> ucl
```

With a few lines of R code we have calculated the t statistic, the p-value, and the confidence interval. Since paired t tests are pretty common, however, it would be helpful to automate this procedure. One obvious reason is to save time and effort, but another important reason is to avoid mistakes. It would be easy to make a mistake (e.g., using n instead of $n-1$ as the degrees of freedom) when repeating the above computations.

Here is a first basic function which automates the computation.

```
> paired_t <- function(x1, x2) {
+   n <- length(x1)
+   dbar <- mean(x1 - x2)
+   s_d <- sd(x1 - x2)
+   tstat <- dbar/(s_d/sqrt(n))
+   pval <- 2 * (1 - pt(abs(tstat), n - 1))
+   margin <- qt(0.975, n - 1) * s_d/sqrt(n)
+   lcl <- dbar - margin
+   ucl <- dbar + margin
+   return(list(tstat = tstat, pval = pval, lcl = lcl, ucl = ucl))
+ }
```

And here is the function in action

```
> paired_t(x1 = corn$kiln_dried, x2 = corn$regular)
```

An explanation and comments on the function are in order.

- `paired_t <- function(x1, x2)` assigns a function of two variables, `x1` and `x2`, to an R object called `paired_t`.
- The *compound expression*, i.e., the code that makes up the body of the function, is enclosed in curly braces `{}`.
- `return(list(tstat = tstat, pval = pval, lcl=lcl, ucl=ucl))` indicates the object(s) returned by the function. In this case the function returns a list with four components.
- The body of the function basically mimics the computations required to compute the t statistic, the p-value, and the confidence interval.
- Several objects such as `n` and `dbar` are created in the body of the function. These objects are NOT available outside the function. We will discuss this further when we cover environments and scope in R.

Our function has automated the basic calculations. But it is still somewhat limited in usefulness. For example, it only computes a 95% confidence interval, while a user may want a different confidence level. And the function only performs a two-sided test, while a user may want a one-sided procedure. We modify the function slightly to allow the user to specify the confidence level next.

```
> paired_t <- function(x1, x2, cl = 0.95) {
+   n <- length(x1)
+   dbar <- mean(x1 - x2)
+   s_d <- sd(x1 - x2)
+   tstat <- dbar/(s_d/sqrt(n))
+   pval <- 2 * (1 - pt(abs(tstat), n - 1))
+   pctile <- 1 - (1 - cl)/2
+   margin <- qt(pctile, n - 1) * s_d/sqrt(n)
+   lcl <- dbar - margin
+   ucl <- dbar + margin
+   return(list(tstat = tstat, pval = pval, lcl = lcl, ucl = ucl))
+ }
```

```
> paired_t(corn$kiln_dried, corn$regular)
> paired_t(corn$kiln_dried, corn$regular, cl = 0.99)
```

Two things to note. First, arguments do not have to be named. So

`paired_t(corn$kiln_dried, corn$regular)`

and

```
paired_t(x1 = corn$kiln_dried, x2 = corn$regular)
```

are equivalent. But we need to be careful if we do not name arguments because then we have to know the ordering of the arguments in the function declaration.

Second, in the declaration of the function, the third argument `cl` was given a default value of `0.95`. If a user does not specify a value for `cl` it will silently be set to `0.95`. But of course a user can override this, as we did in

```
paired_t(corn$kiln_dried, corn$regular, cl = 0.99)
```

7.1.1 Creating Functions

Creating very short functions at the command prompt is a reasonable strategy. For longer functions, one option is to write the function in a script and then submit the whole function. Or a function can be written in any text editor, saved as a plain text file (possibly with a `.r` extension), and then read into R using the `source()` function.

7.2 Programming: Conditional Statements

The `paired_t` function is somewhat useful, but could be improved in several ways. For example, consider the following:

```
> paired_t(1:5, 1:4)
```

The user specified data had different numbers of observations in `x1` and `x2`, which of course can't be data tested by a paired t test. Rather than stopping and letting the user know that this is a problem, the function continued and produced meaningless output.

Also, the function as written only allows testing against a two-sided alternative hypothesis, and it would be good to allow one-sided alternatives.

First we will address some checks on arguments specified by the user. For this we will use an `if()` function and a `stop()` function.

```
> paired_t <- function(x1, x2, cl = 0.95) {
+   if (length(x1) != length(x2)) {
+     stop("The input vectors  must have the same length")
+   }
+   n <- length(x1)
+   dbar <- mean(x1 - x2)
+   s_d <- sd(x1 - x2)
+   tstat <- dbar/(s_d/sqrt(n))
+   pval <- 2 * (1 - pt(abs(tstat), n - 1))
+   pctile <- 1 - (1 - cl)/2
+   margin <- qt(pctile, n - 1) * s_d/sqrt(n)
+   lcl <- dbar - margin
+   ucl <- dbar + margin
+   return(list(tstat = tstat, pval = pval, lcl = lcl, ucl = ucl))
+ }
> paired_t(1:5, 1:4)
```

The argument to the `if()` function is evaluated. If the argument returns `TRUE` the ensuing code is executed. Otherwise, the ensuing code is skipped and the rest of the function is evaluated. If a `stop()` function is executed, the function is exited and the argument of `stop()` is printed.

To better understand and use `if()` statements, we need to understand comparison operators and logical operators.

7.2.1 Comparison and Logical Operators

We have made use of some of the comparison operators in R. These include

- Equal: `==`
- Not equal: `!=`
- Greater than: `>`
- Less than: `<`
- Greater than or equal to: `>=`
- Less than or equal to: `<=`

Special care needs to be taken with the `==` and `!=` operators because of how numbers are represented on computers, see Section 7.3.

There are also three logical operators, with two variants of the "and" operator and the "or" operator.

- and: Either `&` or `&&`
- or: Either `|` or `||`
- not: `!`

The "double" operators `&&` and `||` just examine the first element of the two vectors, whereas the "single" operators `&` and `|` compare element by element.

```
> c(FALSE, TRUE, FALSE) || c(TRUE, FALSE, FALSE)
> c(FALSE, TRUE, FALSE) | c(TRUE, FALSE, FALSE)
> c(FALSE, TRUE, FALSE) && c(TRUE, TRUE, FALSE)
> c(FALSE, TRUE, FALSE) & c(TRUE, TRUE, FALSE)
```

We can use the logical operators to check whether a user-specified confidence level is between 0 and 1.

```
> paired_t <- function(x1, x2, cl = 0.95) {
+   if (length(x1) != length(x2)) {
+     stop("The input vectors  must have the same length")
+   }
+   if (cl <= 0 || cl >= 1) {
+     stop("The confidence level must be between 0 and 1")
+   }
+   n <- length(x1)
+   dbar <- mean(x1 - x2)
+   s_d <- sd(x1 - x2)
+   tstat <- dbar/(s_d/sqrt(n))
+   pval <- 2 * (1 - pt(abs(tstat), n - 1))
+   pctile <- 1 - (1 - cl)/2
+   margin <- qt(pctile, n - 1) * s_d/sqrt(n)
+   lcl <- dbar - margin
+   ucl <- dbar + margin
+   return(list(tstat = tstat, pval = pval, lcl = lcl, ucl = ucl))
+ }
> paired_t(1:5, 2:6, cl = 15)
```

7.2.2 If else statements

The `if()` statement we have used so far has the form

```
if (condition) {
  expression
}
```

Often we want to evaluate one expression if the condition is true, and evaluate a different expression if the condition is false. That is accomplished by the `if else` statement. Here we determine whether a number is positive, negative, or zero.

```
> Sign <- function(x) {
+   if (x < 0) {
+     print("the number is negative")
+   } else if (x > 0) {
```

```
+     print("the number is positive")
+   } else {
+     print("the number is zero")
+   }
+ }
> Sign(3)
> Sign(-3)
> Sign(0)
```

Notice the “different expression” for the first `if` statement was itself an `if` statement.

Next we modify the `paired_t` function to allow two-sided and one-sided alternatives.

```
> paired_t <- function(x1, x2, cl = 0.95, alternative = "not.equal") {
+   if (length(x1) != length(x2)) {
+     stop("The input vectors must be of the same length")
+   }
+   if (cl <= 0 || cl >= 1) {
+     stop("The confidence level must be between 0 and 1")
+   }
+   n <- length(x1)
+   dbar <- mean(x1 - x2)
+   s_d <- sd(x1 - x2)
+   tstat <- dbar/(s_d/sqrt(n))
+   if (alternative == "not.equal") {
+     pval <- 2 * (1 - pt(abs(tstat), n - 1))
+     pctile <- 1 - (1 - cl)/2
+     margin <- qt(pctile, n - 1) * s_d/sqrt(n)
+     lcl <- dbar - margin
+     ucl <- dbar + margin
+   } else if (alternative == "greater") {
+     pval <- 1 - pt(tstat, n - 1)
+     margin <- qt(cl, n - 1) * s_d/sqrt(n)
+     lcl <- dbar - margin
+     ucl <- Inf
+   } else if (alternative == "less") {
+     pval <- pt(tstat, n - 1)
+     margin <- qt(cl, n - 1) * s_d/sqrt(n)
+     lcl <- -Inf
+     ucl <- dbar + margin
+   }
+
+   return(list(tstat = tstat, pval = pval, lcl = lcl, ucl = ucl))
+ }
> paired_t(corn$kiln_dried, corn$regular)
> paired_t(corn$kiln_dried, corn$regular, alternative = "less")
> paired_t(corn$kiln_dried, corn$regular, alternative = "greater")
```

7.3 Computer Arithmetic

Like most software, R does not perform exact arithmetic. Rather, R follows the IEEE 754 floating point standards. This can have profound effects on how computational algorithms are implemented, but is also important when considering things like comparisons.

Note first that computer arithmetic does not follow some of the rules of ordinary arithmetic. For example, it is not associative.

```
> 2^-30
> 2^-30 + (2^30 - 2^30)
> (2^-30 + 2^30) - 2^30
```

Computer arithmetic is not exact.

```
> 1.5 - 1.4
> 1.5 - 1.4 == 0.1
> (1.5 - 1.4) - 0.1
```

So for example an `if` statement that uses an equality test may not give the expected answer. One way to avoid this problem is to test "near equality" using `all.equal()`. The function takes as arguments two objects to be compared, and a tolerance. If the objects are within the tolerance of each other, the function returns `TRUE`. The tolerance has a default value of about 1.5×10^{-8}, which works well in many cases.

```
> all.equal((1.5 - 1.4), 0.1)
```

7.4 Loops

Loops are an important component of any programming language, including R. Vectorized calculations and functions such as `apply()` make loops a bit less central to R than to many other languages, but an understanding of the three looping structures in R is still quite important.

We will investigate loops in the context of computing what is sometimes called the "machine epsilon." Because of the inexact representation of numbers in R (and other languages) sometimes R cannot distinguish between the numbers `1` and `|1 + x|` for small values of `x`. The smallest value of `x` such that `1` and `|1+x|` are not declared equal is the machine epsilon.

```
> 1 == 1 + 10^-4
> 1 == 1 + 10^-50
```

Clearly the machine epsilon is somewhere between 10^{-4} and 10^{-50}. How can we find its value exactly? Since floating point numbers use a binary representation, we know that the machine epsilon will be equal to $1/2^k$ for some value of k. So to find the machine epsilon, we can keep testing whether 1 and $1 + 1/2^k$ are equal, until we find a value k where the two are equal. Then the machine epsilon will be $1/2^{k-1}$, since it is the smallest value for which the two are NOT equal.

```
> 1 == 1 + 1/2
> 1 == 1 + 1/2^2
> 1 == 1 + 1/2^3
```

Testing by hand like this gets tedious quickly. A loop can automate the process. We will do this with two R loop types, `repeat` and `while`.

7.4.1 A Repeat Loop

A `repeat` loop just repeats a given expression over and over again until a `break` statement is encountered.

```
> k <- 1
> repeat {
+   if (1 == 1 + 1/2^k) {
+     break
+   } else {
+     k <- k + 1
+   }
+ }
> k
> 1/2^(k - 1)
```

This code initializes `k` at 1. The body of the loop initially checks whether 1 and $1+1/2^k$ are equal. If they are equal, the `break` statement is executed and control is transferred outside the loop. If they are not equal, `k` is increased by 1, and we return to the beginning of the top of the body of the loop.

7.4.2 A While Loop

A `while` loop has the form

```
while (condition) {
  expression
}
```

As long as the `condition` is `TRUE` the `expression` is evaluated. Once the `condition` is `FALSE` control is transferred outside the loop.

```
> k <- 1
> while (1 != 1 + 1/2^k) {
+   k <- k + 1
+ }
> k
> 1/2^(k - 1)
```

7.4.3 A For Loop

A `for` loop has the form

```
for (variable in vector) {
  expression
}
```

The `for` loop sets the `variable` equal to each element of the `vector` in succession, and evaluates the `expression` each time. Here are two different ways to use a `for` loop to calculate the sum of the elements in a vector.

```
> x <- 1:10
> S <- 0
> for (i in 1:length(x)) {
+   S <- S + x[i]
+ }
> S
> S <- 0
> for (value in x) {
+   S <- S + value
+ }
> S
```

In the first case we loop over the positions of the vector elements, while in the second case we loop over the elements themselves.

7.5 Efficiency Considerations

In many contexts R and modern computers are fast enough that the user does not need to worry about writing efficient code. There are a few simple ways to write efficient code that are easy enough, and provide enough speed-up, that they are worth following as often as possible. The R function `system.time()` reports how long a set of R code takes to execute, and we will use this function to compare different ways to accomplish objectives in R.

7.5.1 Growing Objects

Consider two ways to create a sequence of integers from 1 to n, implemented in functions `f1` and `f2`.

1. Start with a zero-length vector, and let it grow:

```
> f1 <- function(n) {
+   x <- numeric(0)
+   for (i in 1:n) {
+     x <- c(x, i)
+   }
+   x
+ }
```

2. Start with a vector of length n and fill in the values:

```
> f2 <- function(n) {
+   x <- numeric(n)
+   for (i in 1:n) {
+     x[i] <- i
+   }
+   x
+ }
```

Here are the two functions in action, with $n = 100000$.

```
> n <- 1e+05
> system.time(f1(n))
> system.time(f2(n))
```

It is *much* more efficient to start with a full-length vector and then fill in values.[1]

Of course another way to create a vector of integers from 1 to n is to use `1:n`. Let's see how fast this is.

```
> system.time(1:n)
> n <- 1e+06
> system.time(1:n)
```

For $n = 100000$ this is so fast the system time is very close to zero. Even when n is 1000000 the time is very small. So another important lesson is to use built-in R functions whenever possible, since they have had substantial development focused on efficiency, correctness, etc.

7.5.2 Vectorization

Next consider calculating the sum of the squared entries in each column of a matrix. For example with the matrix M,

$$M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix},$$

the sums would be $1^2+4^2 = 17$, $2^2+5^2 = 29$, and $3^2+6^2 = 45$. One possibility is to have an outer loop that traverses the columns and an inner loop that traverses the rows within a column, squaring the entries and adding them together.

[1]Roughly speaking, the first option is slower because each time the vector is increased in size, R must resize the vector and re-allocate memory.

```
> test_matrix <- matrix(1:6, byrow = TRUE, nrow = 2)
> test_matrix
> ss1 <- function(M) {
+   n <- dim(M)[1]
+   m <- dim(M)[2]
+   out <- rep(0, m)
+   for (j in 1:m) {
+     for (i in 1:n) {
+       out[j] <- out[j] + M[i, j]^2
+     }
+   }
+   return(out)
+ }
> ss1(test_matrix)
```

Another possibility eliminates the inner loop, using the `sum()` function to compute the sum of the squared entries in the column directly.

```
> ss2 <- function(M) {
+   m <- dim(M)[2]
+   out <- numeric(m)
+   for (j in 1:m) {
+     out[j] <- sum(M[, j]^2)
+   }
+   return(out)
+ }
> ss2(test_matrix)
```

A third possibility uses the `colSums()` function.

```
> ss3 <- function(M) {
+   out <- colSums(M^2)
+   return(out)
+ }
> ss3(test_matrix)
```

Here is a speed comparison, using a 1000×10000 matrix.

```
> mm <- matrix(1:1e+07, byrow = TRUE, nrow = 1000)
> system.time(ss1(mm))
> system.time(ss2(mm))
> system.time(ss3(mm))
> rm(mm)
```

7.6 More on Functions

Understanding functions deeply requires a careful study of R's scoping rules, as well as a good understanding of environments in R. That's beyond the scope of this book, but we will briefly discuss some issues that are most salient. For a more in-depth treatment, see "Advanced R" by Hadley Wickham, especially the chapters on functions and environments.

7.7 Calling Functions

When using a function, the functions arguments can be specified in three ways:

1. By the full name of the argument.
2. By the position of the argument.
3. By a partial name of the argument.

```
> tmp_function <- function(first.arg, second.arg, third.arg,
+   fourth.arg) {
+   return(c(first.arg, second.arg, third.arg, fourth.arg))
+ }
> tmp_function(34, 15, third.arg = 11, fou = 99)
```

Positional matching of arguments is convenient, but should be used carefully, and probably limited to the first few, and most commonly used, arguments in a function. Partial does have pitfalls. A partially specified argument must unambiguously match exactly one argument—a requirement that's not met below.

```
> tmp_function <- function(first.arg, fourth.arg) {
+   return(c(first.arg, fourth.arg))
+ }
> tmp_function(1, f = 2)
```

7.7.1 The ... argument

In defining a function, a special argument denoted by ... can be used. Sometimes this is called the “ellipsis” argument, sometimes the “three dot” argument, sometimes the “dot dot dot” argument, etc. The R language definition https://cran.r-project.org/doc/manuals/r-release/R-lang.html describes the argument in this way:

> The special type of argument ‘…’ can contain any number of supplied arguments. It is used for a variety of purposes. It allows you to write a function that takes an arbitrary number of arguments. It can be used to absorb some arguments into an intermediate function which can then be extracted by functions called subsequently.

Consider for example the `sum()` function.

```
> sum(1:5)
> sum(1:5, c(3, 4, 90))
> sum(1, 2, 3, c(3, 4, 90), 1:5)
```

Think about writing such a function. There is no way to predict in advance the number of arguments a user might specify. So the function is defined with ... as the first argument:

```
> sum
```

This is true of many commonly-used functions in R such as `c()` among others.

Next, consider a function that calls another function in its body. For example, suppose that a collaborator always supplies comma delimited files that have five lines of description, followed by a line containing variable names, followed by the data. You are tired of having to specify `skip = 5`, `header = TRUE`, and `sep = ","` to `read.table()` and want to create a function `my.read()` which uses these as defaults.

```
> my.read <- function(file, header = TRUE, sep = ",", skip = 5,
+   ...) {
+   read.table(file = file, header = header, sep = sep,
+     skip = skip, ...)
+ }
```

The ... in the definition of `my.read()` allows the user to specify other arguments, for example, `stringsAsFactors = FALSE`. These will be passed on to the `read.table()` function. In fact, that is how `read.csv()` is defined.

```
> read.csv
```

7.7.2 Lazy Evaluation

Arguments to R functions are not evaluated until they are needed, sometimes called *lazy* evaluation.

```
> f <- function(a, b) {
+   print(a^2)
+   print(a^3)
+   print(a * b)
+ }
> f(a = 3, b = 2)
> f(a = 3)
```

The first call specified both of the arguments `a` and `b`, and produced the expected output. In the second call the argument `b` was not specified. Since it was not needed until the third `print` statement, R happily executed the first two `print` statements, and only reported an error in the third statement, when `b` was needed to compute `a*b`.

```
> f <- function(a, b = a^3) {
+   return(a * b)
+ }
> f(2)
> f(2, 10)
```

In the first call, since `b` was not specified, it was computed as `a^3`. In the second call, `b` was specified, and the specified value was used.

7.8 Homework

Exercise 10 Learning objectives: translate statistical notation into coded functions; learn about tools for checking validity of function arguments; practice writing functions that return multiple objects.

8

Spatial Data Visualization and Analysis

8.1 Overview

Recall, a data structure is a format for organizing and storing data. The structure is designed so that data can be accessed and worked with in specific ways. Statistical software and programming languages have methods (or functions) designed to operate on different kinds of data structures.

This chapter focuses on spatial data structures and some of the R functions that work with these data. Spatial data comprise values associated with locations, such as temperature data at a given latitude, longitude, and perhaps elevation. Spatial data are typically organized into **vector** or **raster** data types. (See Figure **??**).

- Vector data represent features such as discrete points, lines, and polygons.
- Raster data represent surfaces as a rectangular matrix of square cells or pixels.

Whether or not you use vector or raster data depends on the type of problem, the type of maps you need, and the data source. Each data structure has strengths and weaknesses in terms of functionality and representation. As you gain more experience working with spatial data, you will be able to determine which structure to use for a particular application.

There is a large set of R packages available for working with spatial (and space-time) data. These packages are described in the Cran Task View: Analysis of Spatial Data[1]. The CRAN task view attempts to organize the various packages into categories, such as *Handling spatial data*, *Reading and writing spatial data*, *Visualization*, and *Disease mapping and areal data analysis*, so users can quickly identify package options given their project needs.

Exploring the extent of the excellent spatial data tools available in R is beyond the scope of this book. Rather, we would point you to subject texts like *Applied Spatial Data Analysis with R* by Bivand et al. (2013) (available for free via the MSU library system), and numerous online tutorials on pretty much any aspect of spatial data analysis with R. These tools make R a full-blown

[1]https://CRAN.R-project.org/view=Spatial

Geographic Information System[2] (GIS) capable of spatial data manipulation and analysis on par with commercial GIS systems such as ESRI's ArcGIS[3].

8.1.1 Some Spatial Data Packages

This chapter will focus on a few R packages for manipulating and visualizing spatial data. Specifically we will touch on the following packages

- `sp`: spatial data structures and methods
- `rgdal`: interface to the C/C++ spatial data Geospatial Data Abstraction Library
- `ggmap`: extends `ggplot2` language to handle spatial data
- `leaflet`: generates dynamic online maps

8.2 Motivating Data

We motivate the topics introduced in this chapter using some forestry data from the Penobscot Experimental Forest[4] (PEF) located in Maine (which you've previously seen throughout the course). The PEF is a long-term experimental forest that is used to understand the effects of silviculture (i.e., science of tending trees) treatments on forest growth and composition. The PEF is divided into non-overlapping management units that receive different harvesting treatments. Within each management unit is a series of observation point locations (called forest inventory plots) where forest variables have been measured. Ultimately, we want to summarize the inventory plots measurements by management unit and map the results.

8.3 Reading Spatial Data into R

Spatial data come in a variety of file formats. Examples of popular vector file formats for points, lines, and polygons, include ESRI's shapefile[5] and

[2] https://en.wikipedia.org/wiki/Geographic_information_system
[3] http://www.esri.com/arcgis/about-arcgis
[4] https://www.nrs.fs.fed.us/ef/locations/me/penobscot%7D%7BPenobscot%20Experimental%20Forest
[5] https://en.wikipedia.org/wiki/Shapefile

open standard GeoJSON[6]. Common raster file formats include GeoTIFF[7] and netCDF[8][9].

The `rgdal` function `readOGR` will read a large variety of vector data file formats (there is also a `writeOGR()` for writing vector data files). Raster data file formats can be read using the `rgdal` function `readGDAL` (yup, also a `writeGDAL()`) or read functions in the `raster` package. All of these functions automatically cast the data into an appropriate R spatial data object (i.e., data structure), which are defined in the `sp` or `raster` packages. Table **??** provides an abbreviated list of these R spatial objects[10]. The *Without attributes* column gives the `sp` package's spatial data object classes for points, lines, polygons, and raster pixels that do not have data associated with the spatial objects (i.e., without attributes in GIS speak). `DataFrame` is appended to the object class name once data, in the form of variables, are added to the spatial object.

You can create your own spatial data objects in R. Below, for example, we create a `SpatialPoints` object consisting of four points. Then add some data to the points to make it a `SpatialPointsDataFrame`.

```
> library(sp)
> library(dplyr)
>
> x <- c(3, 2, 5, 6)
> y <- c(2, 5, 6, 7)
>
> coords <- cbind(x, y)
>
> sp.pts <- SpatialPoints(coords)
>
> class(sp.pts)
>
> some.data <- data.frame(var.1 = c("a", "b", "c", "d"), var.2 = 1:4)
>
> sp.pts.df <- SpatialPointsDataFrame(sp.pts, some.data)
>
> class(sp.pts.df)
```

[6]https://en.wikipedia.org/wiki/GeoJSON

[7]https://en.wikipedia.org/wiki/GeoTIFF

[8]https://en.wikipedia.org/wiki/NetCDF

[9]A longer list of spatial data file formats is available at https://en.wikipedia.org/wiki/GIS_file_formats.

[10]A more complete list of the `sp` package's spatial data classes and methods is detailed in the package's vignette https://cran.r-project.org/web/packages/sp/vignettes/intro_sp.pdf.

If, for example, you already have a data frame that includes the spatial coordinate columns and other variables, then you can promote it to a `SpatialPointsDataFrame` by indicating which columns contain point coordinates. You can extract or access the data frame associated with the spatial object using `@data`. You can also access individual variables directly from the spatial object using `$` or by name or column number to the right of the comma in `[,]` (analogues to accessing variables in a data frame).

```
> df <- data.frame(x = c(3, 2, 5, 6), y = c(2, 5, 6, 7), var.1 = c("a",
+   "b", "c", "d"), var.2 = 1:4)
> class(df)
>
> # promote to a SpatialPointsDataFrame
> coordinates(df) <- ~x + y
>
> class(df)
>
> # access entire data frame
> df@data
>
> class(df@data)
>
> # access columns directly
> df$var.1
>
> df[, c("var.1", "var.2")]
>
> df[, 2]
>
> # get the bounding box
> bbox(df)
```

Here, the data frame `df` is promoted to a `SpatialPointsDataFrame` by indicating the column names that hold the longitude and latitude (i.e., `x` and `y` respectively) using the `coordinates` function. Here too, the `@data` is used to retrieve the data frame associated with the points. We also illustrate how variables can be accessed directly from the spatial object. The `bbox` function is used to return the bounding box that is defined by the spatial extent of the point coordinates. The other spatial objects noted in Table **??** can be created, and their data accessed, in a similar way[11].

More than often we find ourselves reading existing spatial data files into R. The code below uses the `downloader` package to download all of the PEF

[11] This cheatsheet(www.maths.lancs.ac.uk/~rowlings/Teaching/UseR2012/cheatsheet.html) written by Barry Rowlingson is a useful reference www.maths.lancs.ac.uk/~rowlings/Teaching/UseR2012/cheatsheet.html

data we'll use in this chapter. The data are compressed in a single zip file, which is then extracted into the working directory using the `unzip` function. A look into the PEF directory using `list.files` shows nine files[12]. Those named `MU-bounds.*` comprise the shapefile that holds the PEF's management unit boundaries in the form of polygons. Like other spatial data file formats, shapefiles are made up of several different files (with different file extensions) that are linked together to form a spatial data object. The `plots.csv` file holds the spatial coordinates and other information about the PEF's forest inventory plots. The `roads.*` shapefile holds roads and trails in and around the PEF.

```
> library(downloader)
>
> download("http://blue.for.msu.edu/FOR875/data/PEF.zip",
+          destfile="./PEF.zip", mode="wb")
>
> unzip("PEF.zip", exdir = ".")
>
> list.files("PEF")
```

Next we read the MU-bounds shapefile into R using `readOGR()`[13] and explore the resulting `mu` object. Notice that when we read a shapefile into R, we do not include a file extension with the shapefile name because a shapefile is always composed of multiple files.

```
> library(rgdal)
> mu <- readOGR("PEF", "MU-bounds")
```

When called, the `readOGR` function provides a bit of information about the object being read in. Here, we see that it read the MU-bounds shapefile from PEF directory and the shapefile had 40 features (i.e., polygons) and 1 field (i.e., field is synonymous with column or variable in the data frame).

You can think of the resulting `mu` object as a data frame where each row corresponds to a polygon and each column holds information about the polygon[14]. More specifically, the `mu` object is a `SpatialPolygonsDataFrame`.

[12]The `list.files` function does not read data into R; it simply prints the contents of a directory.

[13]The authors of the `rgdal` library decided to have some information about the version of GDAL and other software specifics be printed when the library is loaded. Don't let it distract you.

[14]Much of the actual spatial information is hidden from you in other parts of the data structure, but is available if you ask nicely for it (see subsequent sections).

```
> class(mu)
```

As illustrated using the made-up point data in the example above, you can access the data frame associated with the polygons using `@data`.

```
> class(mu@data)
> dim(mu@data)
> head(mu@data)
```

Above, a call to `class()` confirms we have accessed the data frame, `dim()` shows there are 40 rows (one row for each polygon) and one column, and `head()` shows the first six values of the column named `mu_id`. The `mu_id` values are unique identifiers for each management unit polygon across the PEF.

8.4 Coordinate Reference Systems

One of the more challenging aspects of working with spatial data is getting used to the idea of a coordinate reference system. A *coordinate reference system* (CRS) is a system that uses one or more numbers, or coordinates, to uniquely determine the position of a point or other geometric element (e.g., line, polygon, raster). For spatial data, there are two common coordinate systems:

1. Spherical coordinate system, such as latitude-longitude, often referred to as a *geographic coordinate system*.
2. Projected coordinate system based on a map projection, which is a systematic transformation of the latitudes and longitudes that aims to minimize distortion occurring from projecting maps of the earth's spherical surface onto a two-dimensional Cartesian coordinate plane. Projected coordinate systems are sometimes referred to as *map projections*.

There are numerous map projections[15]. One of the more frustrating parts of working with spatial data is that it seems like each data source you find offers its data in a different map projection and hence you spend a great deal of time *reprojecting* (i.e., transforming from one CRS to another) data into a common

[15]See partial list of map projections at `https://en.wikipedia.org/wiki/List/_of/_map/_projections`. See a humorous discussion of map projections at `http://brilliantmaps.com/xkcd/`.

CRS such that they overlay correctly. Reprojecting is accomplished using the sp package's spTransform function as demonstrated in Section 8.5.

In R, a spatial object's CRS is accessed via the sp package proj4string function. The code below shows the current projection of mu.

```
> proj4string(mu)
```

The cryptic looking string returned by proj4string() is a set of directives understood by the proj.4[16] C library, which is part of sp, and used to map geographic longitude and latitude coordinates into the projected Cartesian coordinates. This CRS tells us the mu object is in Universal Transverse Mercator (UTM)[17] zone 19 coordinate system.[18]

8.5 Illustration using ggmap

Let's begin by making a map of PEF management unit boundaries over top of a satellite image using the ggmap package. Given an address, location, or bounding box, the ggmap package's get_map function will query Google Maps, OpenStreetMap, Stamen Maps, or Naver Map servers for a user-specified map type. The get_map function requires the location or bounding box coordinates be in a geographic coordinate system (i.e., latitude-longitude). This means we need to reproject mu from UTM zone 19 to latitude-longitude geographic coordinates, which is defined by the '"proj=longlat +datum=WGS84"' proj.4 string. As seen below, the first argument in spTransform function is the spatial object to reproject and the second argument is a CRS object created by passing a proj.4 string into the CRS function.

```
> mu <- spTransform(mu, CRS("+proj=longlat +datum=WGS84"))
> proj4string(mu)
```

Unfortunately, we cannot just feed the SpatialPolygonsDataFrame mu into ggplot (perhaps some day soon this will possible). Rather, we need to first convert the SpatialPolygonsDataFrame into a specially formatted data frame us-

[16]http://proj4.org/
[17]https://en.wikipedia.org/wiki/Universal_Transverse_Mercator_coordinate_system
[18]If you start dealing with a lot of spatial data and reprojecting, http://spatialreference.org is an excellent resources for finding and specifying coordinate reference systems.

ing the `fortify` function that is part of the `ggplot2` package[19]. The `fortify` function will also need a unique identifier for each polygon specified using the `region` argument, which for `mu` is the `mu_id`.

```
> library(ggmap)
> mu.f <- fortify(mu, region = "mu_id")
> head(mu.f)
```

Notice the `id` column in the fortified version of `mu` holds each polygon's `mu_id` value (this will be important later when we link data to the polygons).

Next, we query the satellite imagery used to underlay the management units (we'll generally refer to this underlying map as the basemap). As of October 2018, Google now requires you to set up a Google API account in order to run the following maps. This is free, but it does require a credit card to obtain the API Key that is required to make the `ggmap` package work. Here I provide you with an API key for a project I created for this class that should allow you to run the following function if you desire. If you are interested in obtaining your own API key, see the page here[20] for learning about how to use Google maps web services.

```
> register_google(key = "AIzaSyBPAwSY5x8vQqlnG-QwiCAWQW12U3CTLZY")
> mu.bbox <- bbox(mu)
>
> basemap <- get_map(location=mu.bbox, zoom = 14, maptype="satellite")
>
> ggmap(basemap) +
+     geom_polygon(data=mu.f, aes(x = long, y = lat, group=group),
+                  fill=NA, size=0.2, color="orange")
```

Looks pretty good! Take a look at the `get_map` function manual page and try different options for `maptype` (e.g., `maptype="terrain"`).

Next we'll add the forest inventory plots to the map. Begin by reading in the PEF forest inventory plot data held in "plots.csv". Recall, foresters have measured forest variables at a set of locations (i.e., inventory plots) within each management unit. The following statement reads these data and displays the resulting data frame structure.

```
> plots <- read.csv("PEF/plots.csv", stringsAsFactors = FALSE)
> str(plots)
```

[19] `ggmap` depends on `ggplot2` so `ggplot2` will be automatically loaded when you call `library(ggmap)`.

[20] https://developers.google.com/maps/documentation/geocoding/get-api-key

In `plots` each row is a forest inventory plot and columns are:

- `mu_id` identifies the management unit within which the plot is located
- `plot` unique plot number within the management unit
- `easting` longitude coordinate in UTM zone 19
- `northing` latitude coordinate in UTM zone 19
- `biomass_mg_ha` tree biomass in metric ton (per hectare basis)
- `stocking_stems_ha` number of tree (per hectare basis)
- `diameter_cm` average tree diameter measured 130 cm up the tree trunk
- `basal_area_m2_ha` total cross-sectional area at 130 cm up the tree trunk (per hectare basis)

There is nothing inherently spatial about this data structure—it is simply a data frame. We make `plots` into a spatial object by identifying which columns hold the coordinates. This is done below using the `coordinates` function, which promotes the `plots` data frame to a `SpatialPointsDataFrame`.

```
> coordinates(plots) <- ~easting+northing
>
> class(plots)
```

Although `plots` is now a `SpatialPointsDataFrame`, it does not know to which CRS the coordinates belong; hence, the `NA` when `proj4string(plots)` is called below. As noted in the `plots` file description above, `easting` and `northing` are in UTM zone 19. This CRS is set using the second call to `proj4string` below.

```
> proj4string(plots)
>
> proj4string(plots) <- CRS("+proj=utm +zone=19 +datum=NAD83 +units=m
+                               +no_defs +ellps=GRS80 +towgs84=0,0,0")
```

Now let's reproject `plots` to share a common CRS with `mu`

```
> plots <- spTransform(plots, CRS("+proj=longlat +datum=WGS84"))
```

Note, because `mu` is already in the projection we want for `plots`, we could have replaced the second argument in the `spTransform` call above with `proj4string(mu)` and saved some typing.

We're now ready to add the forest inventory plots to the existing basemap with management units. Specifically, let's map the `biomass_mg_ha` variable to show changes in biomass across the forest. No need to fortify `plots`, `ggplot` is happy to take `geom_point`'s `data` argument as a data frame (although we do need to convert `plots` from a `SpatialPointsDataFrame` to a data frame using the as.data.frame function). Check out the scale_color_gradient function in your favorite ggplot2 reference to understand how the color scale is set.

```
> ggmap(basemap) +
+     geom_polygon(data=mu.f, aes(x = long, y = lat, group=group),
+                  fill=NA, size=0.2, color="orange") +
+     geom_point(data=as.data.frame(plots),
+                aes(x = easting, y = northing, color=biomass_mg_ha)) +
+     scale_color_gradient(low="white", high="darkblue") +
+     labs(color = "Biomass (mg/ha)")
```

There is something subtle and easy to miss in the code above. Notice the aes function arguments in geom_points take geographic longitude and latitude, x and y respectively, from the points data frame (but recall easting and northing were in UTM zone 19). This works because we applied spTransform to reproject the points SpatialPointsDataFrame to geographic coordinates. sp then replaces the values in easting and northing columns with the reprojected coordinate values when converting a SpatialPointsDataFrame to a data frame via as.data.frame().

Foresters use the inventory plot measurements to estimate forest variables within management units, e.g., the average or total management unit biomass. Next we'll make a plot with management unit polygons colored by average biomass_mg_ha.

```
> mu.bio <- plots@data %>% group_by(mu_id) %>%
+     summarize(biomass_mu = mean(biomass_mg_ha))
> print(mu.bio)
```

Recall from Section 6.7.4 this one-liner can be read as "get the data frame from plots's SpatialPointsDataFrame *then* group by management unit *then* make a new variable called biomass_mu that is the average of biomass_mg_ha and assign it to the mu.bio tibble."

The management unit specific biomass_mu can now be joined to the mu polygons using the common mu_id value. Remember when we created the fortified version of mu called mu.f? The fortify function region argument was mu_id which is the id variable in the resulting mu.f. This id variable in mu.f can be linked to the mu_id variable in mu.bio using dplyr's left_join function as illustrated below.

```
> head(mu.f, n = 2)
>
```

```
> mu.f <- left_join(mu.f, mu.bio, by = c(id = "mu_id"))
>
> head(mu.f, n = 2)
```

The calls to `head()` show the first few rows of `mu.f` pre- and post-join. After the join, `mu.f` includes `biomass_mu`, which is used used below for `geom_polygon`'s `fill` argument to color the polygons accordingly.

```
> ggmap(basemap) +
+     geom_polygon(data=mu.f, aes(x = long, y = lat,
+                                        group=group, fill=biomass_mu),
+                 size=0.2, color="orange") +
+     scale_fill_gradient(low="white", high="darkblue",
+                        na.value="transparent") +
+     labs(fill="Biomass (mg/ha)")
```

Let's add the roads and some more descriptive labels as a finishing touch. The roads data include a variable called `type` that identifies the road type. To color roads by type in the map, we need to join the `roads` data frame with the fortified roads `roads.f` using the common variable `id` as a road segment specific identifier. Then `geom_path`'s `color` argument gets this `type` variable as a factor to create road-specific color. The default coloring of the roads blends in too much with the polygon colors, so we manually set the road colors using the `scale_color_brewer` function. The `palette` argument in this function accepts a set of key words, e.g., `"Dark2"`, that specify sets of diverging colors chosen to make map object difference optimally distinct (see, the manual page for `scale_color_brewer`, http://colorbrewer2.org, and blog here[21].)[22]

```
> roads <- readOGR("PEF", "roads")
>
> roads <- spTransform(roads, proj4string(mu))
>
> roads.f <- fortify(roads, region="id")
> roads.f <- left_join(roads.f, roads@data, by = c('id' = 'id'))
>
> ggmap(basemap) +
+     geom_polygon(data=mu.f, aes(x = long, y = lat, group=group,
```

[21] https://www.r-bloggers.com/r-using-rcolorbrewer-to-colour-your-figures-in-r

[22] Install the `RColorBrewer` package and run `library(RColorBrewer); display.brewer.all()` to get a graphical list of available palettes.

```
+                                fill=biomass_mu),
+                 size=0.2, color="orange") +
+     geom_path(data=roads.f, aes(x = long, y = lat,
+                                group=group, color=factor(type))) +
+     scale_fill_gradient(low="white", high="darkblue",
+                        na.value="transparent") +
+     scale_color_brewer(palette="Dark2") +
+     labs(fill="Biomass (mg/ha)", color="Road type", xlab="Longitude",
+          ylab="Latitude", title="PEF forest biomass")
```

The second, and more cryptic, of the two warnings from this code occurs because some of the roads extend beyond the range of the map axes and are removed (nothing to worry about).

8.6 Illustration using `leaflet`

Leaflet is one of the most popular open-source JavaScript libraries for interactive maps. As noted on the official R leaflet website[23], it's used by websites ranging from *The New York Times* and *The Washington Post* to GitHub and Flickr, as well as by GIS specialists like OpenStreetMap, Mapbox, and CartoDB.

The R leaflet website[24] is an excellent resource to learn leaflet basics, and should serve as a reference to gain a better understanding of the topics we briefly explore below.

You create a leaflet map using these basic steps:

1. Create a map by calling `leaflet()`
2. Add data layers to the map using layer functions such as, `addTiles()`, `addMarkers()`, `addPolygons()`, `addCircleMarkers()`, `addPolylines()`, `addRasterImage()` and other `add...` functions
3. Repeat step 2 to add more layers to the map
4. Print the map to display it

Here's a brief example.

[23]https://rstudio.github.io/leaflet
[24]https://rstudio.github.io/leaflet

```
> library(leaflet)
>
> m <- leaflet() %>%
+       addTiles() %>%  # Add default OpenStreetMap map tiles
+       addMarkers(lng=-84.482004, lat=42.727516,
+                  popup="<b>Here I am!</b>") # Add a clickable marker
> m  # Print the map
```

There are a couple things to note in the code. First, we use the pipe operator `%>%` just like in `dplyr` functions. Second, the `popup` argument in `addMarkers()` takes standard HTML and clicking on the marker makes the text popup. Third, the html version of this text provides the full interactive, dynamic map, so we encourage you to read and interact with the html version of this textbook for this section. The PDF document will simply display a static version of this map and will not do justice to how awesome `leaflet` truly is!

As seen in the `leaflet()` call above, the various `add...` functions can take longitude (i.e., `lng`) and latitude (i.e., `lat`). Alternatively, these functions can extract the necessary spatial information from `sp` objects, e.g., Table **??**, when passed to the data argument (which greatly simplifies life compared with map making using `ggmap`).

8.7 Subsetting Spatial Data

You can imagine that we might want to subset spatial objects to map specific points, lines, or polygons that meet some criteria, or perhaps extract values from polygons or raster surfaces at a set of points or geographic extent. These, and similar types, of operations are easy in R (as long as all spatial objects are in a common CRS). Recall from Chapter **??** how handy it is to subset data structures, e.g., vectors and data frames, using the `[]` operator and logical vectors? Well it's just as easy to subset spatial objects, thanks to the authors of `sp`, `raster`, and other spatial data packages.

8.7.1 Fetching and Cropping Data using `raster`

In order to motivate our exploration of spatial data subsetting and to illustrate some useful functionality of the `raster` package, let's download some elevation data for the PEF. The `raster` package has a rich set of functions for manipulating raster data as well as functions for downloading data from open source repositories. We'll focus on the package's `getData` function, which, given a location in geographic longitude and latitude or location name, will download data from GADM[25], Shuttle Radar Topography Mission[26], Global Climate Data[27], and other sources commonly used in spatial data applications.

Let's download SRTM surface elevation data for the PEF, check the resulting object's class and CRS, and display it using the `raster` package's `image` function along with the PEF forest inventory plots.

```
> library(raster)
>
> pef.centroid <- as.data.frame(plots) %>%
+     summarize(mu.x = mean(easting), mu.y = mean(northing))
>
> srtm <- getData("SRTM", lon = pef.centroid[, 1], lat = pef.centroid[, 2])
>
> srtm <- raster("srtm_23_04.tif")
> proj4string(srtm) <- "+proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0"
>
> class(srtm)
>
> proj4string(srtm)
>
> image(srtm)
> plot(plots, add = TRUE)
```

A few things to notice in the code above. First the `getData` function needs the longitude `lon` and latitude `lat` to identify which SRTM raster tile to return (SRTM data come in large raster tiles that cover the globe). As usual, look at the `getData` function documentation for a description of the arguments. To estimate the PEF's centroid coordinate, we averaged the forest inventory plots' latitude and longitude then assigned the result to `pef.centroid`. Second, there is currently a bug with downloading `SRTM` data using the `getData()` function. All the data are downloaded into your current directory, but the function does not properly load them into `R`. If you run this line yourself and get an error, continue going through the code we have. In the next line we load the data in ourselves using the call to `raster("srtm_23_04.tiff")`. I also manually change the coordinate system using the `proj4string` function. The `srtm` object result from our code to get around the bug is a `RasterLayer`, see Table **??**. Third, `srtm` is in a longitude latitude geographic CRS (same as our other PEF data). Finally, the image shows SRTM elevation along the coast of Maine, the PEF plots are those tiny specks of black in the northwest quadrant, and the white region of the image is the Atlantic Ocean.

[25]http://www.gadm.org/%7D%7BGlobal%20Administrative%20Boundaries
[26]https://www2.jpl.nasa.gov/srtm/%7D%7BShuttle%20Radar%20Topography%20Mission
[27]http://www.worldclim.org/

Okay, this is a start, but it would be good to crop the SRTM image to the PEF's extent. This is done using `raster`'s `crop` function. This function can use many different kinds of spatial objects in the second argument to calculate the extent at which to crop the object in the first argument. Here, I set `mu` as the second argument and save the resulting SRTM subset over the larger tile (the `srtm` object).

```
> srtm <- crop(srtm, mu)
>
> image(srtm)
> plot(mu, add = TRUE)
```

The `crop` is in effect doing a spatial setting of the raster data. We'll return to the `srtm` data and explore different kinds of subsetting in the subsequent sections.

8.7.2 Logical, Index, and Name Subsetting

As promised, we can subset spatial objects using the `[]` operator and a logical, index, or name vector. The key is that `sp` objects behave like data frames, see Section **??**. A logical or index vector to the left of the comma in `[,]` accesses points, lines, polygons, or pixels. Similarly, a logical, index, or name vector to the right of the comma accesses variables.

For example, say we want to map forest inventory plots with more than 10,000 stems per hectare, `stems_ha` (the `min()` was added below to double check that the subset worked correctly.

```
> min(plots$stems_ha)
>
> plots.10k <- plots[plots$stems_ha > 10000, ]
>
> min(plots.10k$stems_ha)
```

You can also add new variables to the spatial objects.

```
> plots$diameter_in <- plots$diameter_cm/2.54
>
> head(plots)
```

8.7.3 Spatial Subsetting and Overlay

A spatial overlay retrieves the indexes or variables from object A using the location of object B. With some spatial objects this operation can be done using the [] operator. For example, say we want to select and map all management units in mu, i.e., A, that contain plots with more than 10,000 stems per ha, i.e., B.

```
> mu.10k <- mu[plots.10k, ]  ## A[B,]
>
> mu.10k.f <- fortify(mu.10k, region = "mu_id")
>
> ggmap(basemap) + geom_polygon(data = mu.10k.f, aes(x = long,
+   y = lat, group = group), fill = "transparent", size = 0.2,
+   color = "orange") + geom_point(data = as.data.frame(plots.10k),
+   aes(x = easting, y = northing), color = "white")
```

More generally, however, the over function offers consistent overlay operations for sp objects and can return either indexes or variables from object A given locations from object B, i.e., over(B, A) or, equivalently, B%over%A. The code below duplicates the result from the preceding example using over.

```
> mu.10k <- mu[mu$mu_id %in% unique(over(plots.10k, mu)$mu_id),
+   ]
```

Yes, this requires more code but over provides a more flexible and general purpose function for overlays on the variety of sp objects. Let's unpack this one-liner into its five steps.

```
> i <- over(plots.10k, mu)
> ii <- i$mu_id
> iii <- unique(ii)
> iv <- mu$mu_id %in% iii
> v <- mu[iv, ]
```

i. The over function returns variables for mu's polygons that coincide with the nrow(plots.10k@data) points in plots.10k. No points fall outside the polygons and the polygons do not overlap, so i should be a data frame with nrow(plots.10k@data) rows. If polygons did overlap and a point fell within the overlap region, then variables for the coinciding polygons are returned.
ii. Select the unique mu identifier mu_id (this step is actually not necessary here because mu only has one variable).

iii. Because some management units contain multiple plots there will be repeat values of `mu_id` in *ii*, so apply the `unique` function to get rid of duplicates.
iv. Use the `%in%` operator to create a logical vector that identifies which polygons should be in the final map.
v. Subset `mu` using the logical vector created in *iv*.

Now let's do something similar using the `srtm` elevation raster. Say we want to map elevation along trails, winter roads, and gravel roads across the PEF. We could subset `srtm` using the `roads` `SpatialLinesDataFrame`; however, mapping the resulting pixel values along the road segments using `ggmap` requires a bit more massaging. So, to simplify things for this example, `roads` is first coerced into a `SpatialPointsDataFrame` called `roads.pts` that is used to extract spatially coinciding `srtm` pixel values which themselves are coerced from `raster`'s `RasterLayer` to `sp`'s `SpatialPixelsDataFrame` called `srtm.sp` so that we can use the `over` function. We also choose a different basemap just for fun.

```
> hikes <- roads[roads$type %in% c("Trail", "Winter", "Gravel"),]
>
> hikes.pts <- as(hikes, "SpatialPointsDataFrame")
> srtm.sp <- as(srtm, "SpatialPixelsDataFrame")
>
> hikes.pts$srtm <- over(hikes.pts, srtm.sp)
>
> basemap <- get_map(location=mu.bbox,  zoom = 14, maptype="terrain")
>
> color.vals <- srtm@data@values[1:length(hikes.pts)]
>
> ggmap(basemap) +
+     geom_point(data=as.data.frame(hikes.pts),
+                aes(x = coords.x1, y = coords.x2, color = color.vals)) +
+     scale_color_gradient(low="green", high="red") +
+     labs(color = "Hiking trail elevation\n(m above sea level)",
+          xlab="Longitude", ylab="Latitude")
```

In the call to `geom_point` above, `coords.x1 coords.x2` are the default names given to longitude and latitude, respectively, when `sp` coerces `hikes` to `hikes.pts`. These points represent the vertices along line segments. I create the vector `color.vals` that contains the values from `srtm` that I use in the map argument `color`. Normally, I would be able to simply use the argument `color = srtm` in the graph, but since there is a bug in the `getData` function I mentioned earlier, we need to do another workaround here.

Overlay operations involving lines and polygons over polygons require the rgeos package which provides an interface to the Geometry Engine - Open Source[28] (GEOS) C++ library for topology operations on geometries. We'll leave it to you to explore these capabilities.

8.7.4 Spatial Aggregation

We have seen aggregation operations before when using dplyr's summarize function. The summarize function is particularly powerful when combined with group_by(), which can apply a function specified in summarize() to a variable partitioned using a grouping variable. The aggregate function in sp works in a similar manner, except groups are delineated by the spatial extent of a thematic variable. In fact, the work we did to create mu.bio using dplyr functions can be accomplished with aggregate(). Using aggregate() will, however, require a slightly different approach for joining the derived average biomass_mg_ha to the fortified mu. This is because the aggregate function will apply the user specified function to all variables in the input object, which, in our case, results in an NA for the linking variable mu_id as demonstrated below.

```
> mu.ag <- aggregate(plots[, c("mu_id", "biomass_mg_ha")],
+   by = mu, FUN = mean)
>
> head(mu.ag@data, n = 2)
```

With mu_id rendered useless, we do not have a variable that uniquely identifies each polygon for use in fortify's region argument; hence no way to subsequently join the unfortified and fortified versions of mu.bio.ag. Here's the work around. If the region is not specified, fortify() uses an internal unique polygon ID that is part of the sp data object and accessed via row.names()[29] So, the trick is to add this unique polygon ID to the aggregate() output prior to calling fortify() as demonstrated below.

```
> mu.ag$id <- row.names(mu.ag)
>
> mu.ag.f <- fortify(mu.ag)
>
> mu.ag.f <- left_join(mu.ag.f, mu.ag@data)
>
```

[28] https://trac.osgeo.org/geos/

[29] With other data, there is a chance the row names differ from the unique polygon IDs. Therefore a more reliable approach to getting a unique ID is to use sapply(slot(mu.ag, 'polygons'), function(x) slot(x, 'ID')), but replace mu.ag with your SpatialPolygonsDataFrame. Also, this approach will work with other sp objects in the right column of Table ??.

```
> ggmap(basemap) + geom_polygon(data = mu.ag.f, aes(x = long,
+   y = lat, group = group, fill = biomass_mg_ha), size = 0.2,
+   color = "orange") + scale_fill_gradient(low = "white",
+   high = "darkblue", na.value = "transparent") + labs(fill = "Biomass (mg/ha)")
```

The `aggregate()` function will work with all `sp` objects. For example let's map the variance of pixel values in `srtm.sp` by management unit. Notice that `aggregate()` is happy to take a user-specified function for `FUN`.

```
> mu.srtm <- aggregate(srtm.sp, by=mu,
+                      FUN=function(x){sqrt(var(x))})
>
> mu.srtm$id <- row.names(mu.srtm)
>
> mu.srtm.f <- fortify(mu.srtm)
>
> mu.srtm.f <- left_join(mu.srtm.f, mu.srtm@data)
>
> ggmap(basemap) +
+     geom_polygon(data=mu.srtm.f, aes(x = long, y = lat, group=group,
+                                     fill=srtm_23_04),
+                  size=0.2, color="orange") +
+     scale_fill_gradient(low="green", high="red") +
+         labs(fill = "Elevation standard deviation\n(m above sea level)",
+          xlab="Longitude", ylab="Latitude")
```

8.8 Where to go from here

This chapter just scratches the surface of R's spatial data manipulation and visualization capabilities. The basic ideas we presented here should allow you to take a deeper look into `sp`, `rgdal`, `rgeos`, `ggmap`, `leaflet`, and a myriad of other excellent user-contributed R spatial data packages. A good place to start is with Edzer Pebesma's excellent vignette on the use of the map overlay and spatial aggregation, available here[30], as well as *Applied Spatial Data Analysis with R* by Bivand et al. (2013).

[30] https://cran.r-project.org/web/packages/sp/vignettes/over.pdf

Bibliography

Bivand, R. S., Pebesma, E., and Gomez-Rubio, V. (2013). *Applied spatial data analysis with R, Second edition.* Springer, NY.

Spector, P. (2008). *Data Manipulation With R.* Use R!

Wickham, H. and Sievert, C. (2016). *ggplot2: Elegant Graphics for Data Analysis.* Springer, Cham, 2nd edition.