R Programming for Data Sciences

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

2019-05-06

Contents

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.

Chapter 1

Data

Data science is a field that intersects with statistics, mathematics, computer science, and a wide range of applied fields, such as marketing, biology, and physics. As such, it is hard to formally define data science, but obviously data is central to data science, and it is useful at the start to consider some types of data that are of interest.

1.1 Baby Crawling Data

When thinking about data, we might initially have in mind a modest-sized and uncomplicated data set consisting primarily of numbers. As an example of such a data set, a study was done to assess the possible relationship between the age at which babies first begin to crawl and the temperature at the time of first crawling. Participants in the study were volunteers.[1] The data set from this study separates the babies by birth month, and reports the birth month, the average age (in weeks) when first crawling for that month, the standard deviation of the crawling ages for that month, the number of infants for that month, and the average temperature during the month when crawling commenced. The data are shown in Table **??** below.[2]

```
> u <- "http://blue.for.msu.edu/FOR875/data/BabyCrawling.tsv"
> BabyCrawling <- read.table(u, header = T)
```

This data set has many simple properties: it is relatively small, there are no missing observations, the variables are easily understood, etc.

1.2 World Bank Data

The World Bank provides data related to the development of countries. A data set was constructed from the World Bank repository. The data set contains data on countries throughout the world for the years 1960 through 2014 and contains, among others, variables representing average life expectancy, fertility rate, and population. Table **??** contains the first five records and then 10 more randomly selected records for these variables in the data set.

Notice that many observations contain missing data for fertility rate and life expectancy. If all the variables were shown, we would see much more missing data. This data set is also substantially larger than the baby crawling age data, with 11880 rows and 15 columns of data in the full data set. (Each column represents one of the variables. Each row represents one country during one year).

[1] More correctly, were volunteered by their parents.

[2] These data were retrieved from http://lib.stat.cmu.edu/DASL/Datafiles/Crawling.html.

Table 1.1: Data on age at crawling

BirthMonth	AvgCrawlingAge	SD	n	temperature
January	29.84	7.08	32	66
February	30.52	6.96	36	73
March	29.70	8.33	23	72
April	31.84	6.21	26	63
May	28.58	8.07	27	52
June	31.44	8.10	29	39
July	33.64	6.91	21	33
August	32.82	7.61	45	30
September	33.83	6.93	38	33
October	33.35	7.29	44	37
November	33.38	7.42	49	48
December	32.32	5.71	44	57

Table 1.2: A small portion of the World Bank data set

country	year	fertility.rate	life.expectancy	population
Andorra	1978			33746
Andorra	1979			34819
Andorra	1977			32769
Andorra	2007	1.180		81292
Andorra	1976			31781
Bulgaria	1969	2.270	70.43	8434172
St. Martin (French part)	1977			6778
Macao SAR, China	2007	0.906	79.06	493206
Iran, Islamic Rep.	1979	6.422	55.27	37465764
Comoros	1997	5.270	57.18	489627
Syrian Arab Republic	1975	7.472	62.78	7564000
Virgin Islands (U.S.)	1967	5.561	66.46	50800
Malta	1982	1.900	73.40	325898
St. Lucia	2013	1.912	74.79	182273
Vanuatu	1976	5.858	56.09	103024

1.3 Email Data

It is estimated that in 2015, 90% of the total 205 billion emails sent were spam.[3] Spam filters use large amounts of data from emails to learn what distinguishes spam messages from non-spam (sometimes called "ham") messages. Below we include one spam message followed by a ham message.[4]

```
From safety33o@l11.newnamedns.com  Fri Aug 23 11:03:37 2002
Return-Path: <safety33o@l11.newnamedns.com>
Delivered-To: zzzz@localhost.example.com
Received: from localhost (localhost [127.0.0.1])
    by phobos.labs.example.com (Postfix) with ESMTP id 5AC994415F
    for <zzzz@localhost>; Fri, 23 Aug 2002 06:02:59 -0400 (EDT)
Received: from mail.webnote.net [193.120.211.219]
    by localhost with POP3 (fetchmail-5.9.0)
    for zzzz@localhost (single-drop); Fri, 23 Aug 2002 11:02:59 +0100 (IST)
Received: from l11.newnamedns.com ([64.25.38.81])
    by webnote.net (8.9.3/8.9.3) with ESMTP id KAA09379
    for <zzzz@example.com>; Fri, 23 Aug 2002 10:18:03 +0100
From: safety33o@l11.newnamedns.com
Date: Fri, 23 Aug 2002 02:16:25 -0400
Message-Id: <200208230616.g7N6GOR28438@l11.newnamedns.com>
To: kxzzzzgxlrah@l11.newnamedns.com
Reply-To: safety33o@l11.newnamedns.com
Subject: ADV: Lowest life insurance rates available!
moode

Lowest rates available for term life insurance! Take a moment
and fill out our online form
to see the low rate you qualify for.
Save up to 70% from regular rates! Smokers accepted!
http://www.newnamedns.com/termlife/

Representing quality nationwide carriers. Act now!
```

```
From rssfeeds@jmason.org  Tue Oct  1 10:37:22 2002
Return-Path: <rssfeeds@example.com>
Delivered-To: yyyy@localhost.example.com
Received: from localhost (jalapeno [127.0.0.1])
    by jmason.org (Postfix) with ESMTP id B277816F16
    for <jm@localhost>; Tue,  1 Oct 2002 10:37:21 +0100 (IST)
Received: from jalapeno [127.0.0.1]
    by localhost with IMAP (fetchmail-5.9.0)
    for jm@localhost (single-drop); Tue, 01 Oct 2002 10:37:21 +0100 (IST)
Received: from dogma.slashnull.org (localhost [127.0.0.1]) by
    dogma.slashnull.org (8.11.6/8.11.6) with ESMTP id g9180YK15357 for
    <jm@jmason.org>; Tue, 1 Oct 2002 09:00:34 +0100
Message-Id: <200210010800.g9180YK15357@dogma.slashnull.org>
To: yyyy@example.com
From: boingboing <rssfeeds@example.com>
Subject: Disney's no-good Park-Czar replaced
Date: Tue, 01 Oct 2002 08:00:34 -0000
Content-Type: text/plain; encoding=utf-8
```

[3] Radicati Group www.radicati.com

[4] These messages both come from the large collection of spam and ham messages at http://spamassassin.apache.org.

```
X-Spam-Status: No, hits=-641.2 required=5.0
    tests=AWL
    version=2.50-cvs
X-Spam-Level:

URL: http://boingboing.net/#85506723
Date: Not supplied

Disney has named a new president of Walt Disney Parks, replacing Paul Pressler,
the exec who did his damnedest to ruin Disneyland, slashing spending (at the
expense of safety and employee satisfaction), building the craptastical
California Adventure, reducing the number of SKUs available for sale in the
Park stores, and so on. The new president, James Rasulo, used to be head of
Euro Disney. Link[1] Discuss[2]

[1] http://reuters.com/news_article.jhtml?type=search&StoryID=1510778
[2] http://www.quicktopic.com/boing/H/rw7cDXT3W44C
```

To implement a spam filter we would have to get the data from these email messages (and thousands of others) into a software package, extract and separate potentially important features such as the `To:` line, the `Subject:` line, the message body, etc., and then compare spam and non-spam messages to find a method to classify new emails correctly. These steps are not simple in this example. In particular, we would need to become skilled at working with *text data*.

1.4 Handwritten Digit Recognition

Correct recognition of handwritten digits by a machine is commonly required in today's world. For example, the postal service must scan and recognize zip codes on handwritten mail. Roughly speaking, a handwritten digit is scanned and converted to a digital image. To keep things simple we will assume the scanning creates a grayscale rather than a color image. When converting an image to a grayscale digital image, a grid of "pixels"" is used to represent the handwritten image, where each pixel has a black intensity value. For concreteness we'll assume that intensities are recorded on a scale from -1 (no black intensity at all) to 1 (maximum black intensity). If the pixel grid is 16 by 16 then the resulting digitized image will contain 256 intensity values, one for each of the $16 \times 16 = 256$ pixels.

For example, here are the data corresponding to one handwritten digit, which happens to be the numeral "6". Figure **??** shows how that digit looks when digitized.

```
-1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -0.631  0.862 -0.167
-1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000
-1.000 -1.000 -0.992  0.297  1.000  0.307 -1.000 -1.000 -1.000 -1.000
-1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -0.410  1.000
 0.986 -0.565 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000
-1.000 -1.000 -1.000 -0.683  0.825  1.000  0.562 -1.000 -1.000 -1.000
-1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -0.938  0.540
 1.000  0.778 -0.715 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000
-1.000 -1.000 -1.000 -1.000  0.100  1.000  0.922 -0.439 -1.000 -1.000
-1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -0.257
 0.950  1.000 -0.162 -1.000 -1.000 -1.000 -0.987 -0.714 -0.832 -1.000
-1.000 -1.000 -1.000 -1.000 -0.797  0.909  1.000  0.300 -0.961 -1.000
-1.000 -0.550  0.485  0.996  0.867  0.092 -1.000 -1.000 -1.000 -1.000
 0.278  1.000  0.877 -0.824 -1.000 -0.905  0.145  0.977  1.000  1.000
 1.000  0.990 -0.745 -1.000 -1.000 -0.950  0.847  1.000  0.327 -1.000
-1.000  0.355  1.000  0.655 -0.109 -0.185  1.000  0.988 -0.723 -1.000
```

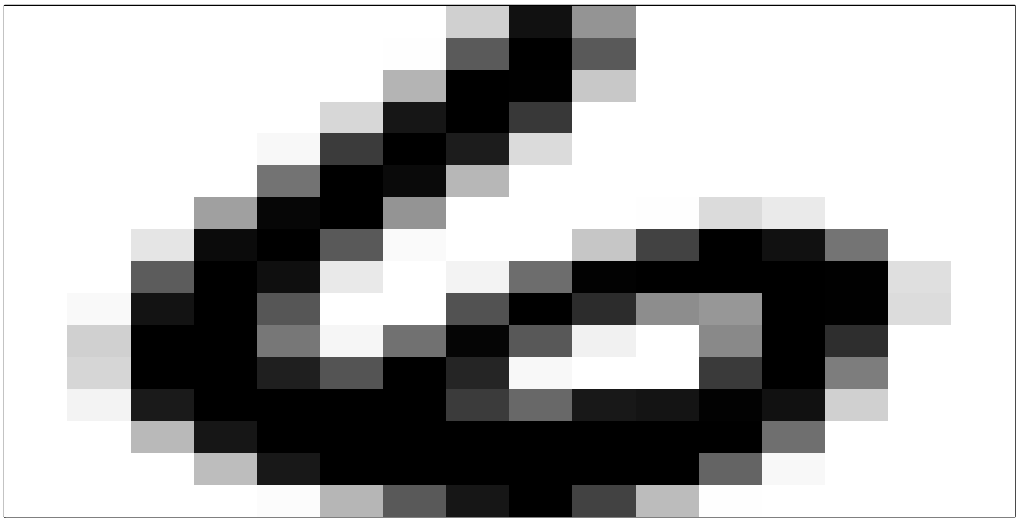


Figure 1.1: A digitized version of a handwritten 6

```
-1.000 -0.630  1.000  1.000  0.068 -0.925  0.113  0.960  0.308 -0.884
-1.000 -0.075  1.000  0.641 -0.995 -1.000 -1.000 -0.677  1.000  1.000
 0.753  0.341  1.000  0.707 -0.942 -1.000 -1.000  0.545  1.000  0.027
-1.000 -1.000 -1.000 -0.903  0.792  1.000  1.000  1.000  1.000  0.536
 0.184  0.812  0.837  0.978  0.864 -0.630 -1.000 -1.000 -1.000 -1.000
-0.452  0.828  1.000  1.000  1.000  1.000  1.000  1.000  1.000  1.000
 0.135 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -0.483  0.813  1.000
 1.000  1.000  1.000  1.000  1.000  0.219 -0.943 -1.000 -1.000 -1.000
-1.000 -1.000 -1.000 -1.000 -0.974 -0.429  0.304  0.823  1.000  0.482
-0.474 -0.991 -1.000 -1.000 -1.000 -1.000
```

Looking at the digitized images, it may seem simple to correctly identify a handwritten numeral. But remember, the machine only has access to the 256 pixel intensities, and must make a decision based on them.

Figure **??** shows the digitized images of the first 25 numerals in the data set, and Figure **??** shows the digitized images of the first 25 numeral sevens in the data set. These give some idea of the variability in how digits are written.[5]

1.5 Looking Forward

The four examples above illustrate a small sample of the wide variety of data sets that may be encountered in data science. Each of these provides its own challenges. The baby crawling data present challenges that are more statistical in nature. For example, how might the study design (which isn't described here) affect methods of analysis and conclusions drawn from the study? Similar challenges are also present within the other data sets, but these data sets also present more substantial challenges prior to (and during) the analysis stage, such as how to work with the missing data in the World Bank data set, or how to effectively and efficiently process the email data to extract features of interest.

This book and associated material introduce tools to tackle some of the challenges in working with real data sets, within the context of the R statistical system. We will focus on important topics such as

[5] Actually, these data were already pre-processed to get the orientation correct. Actual handwritten digits would be even more variable.

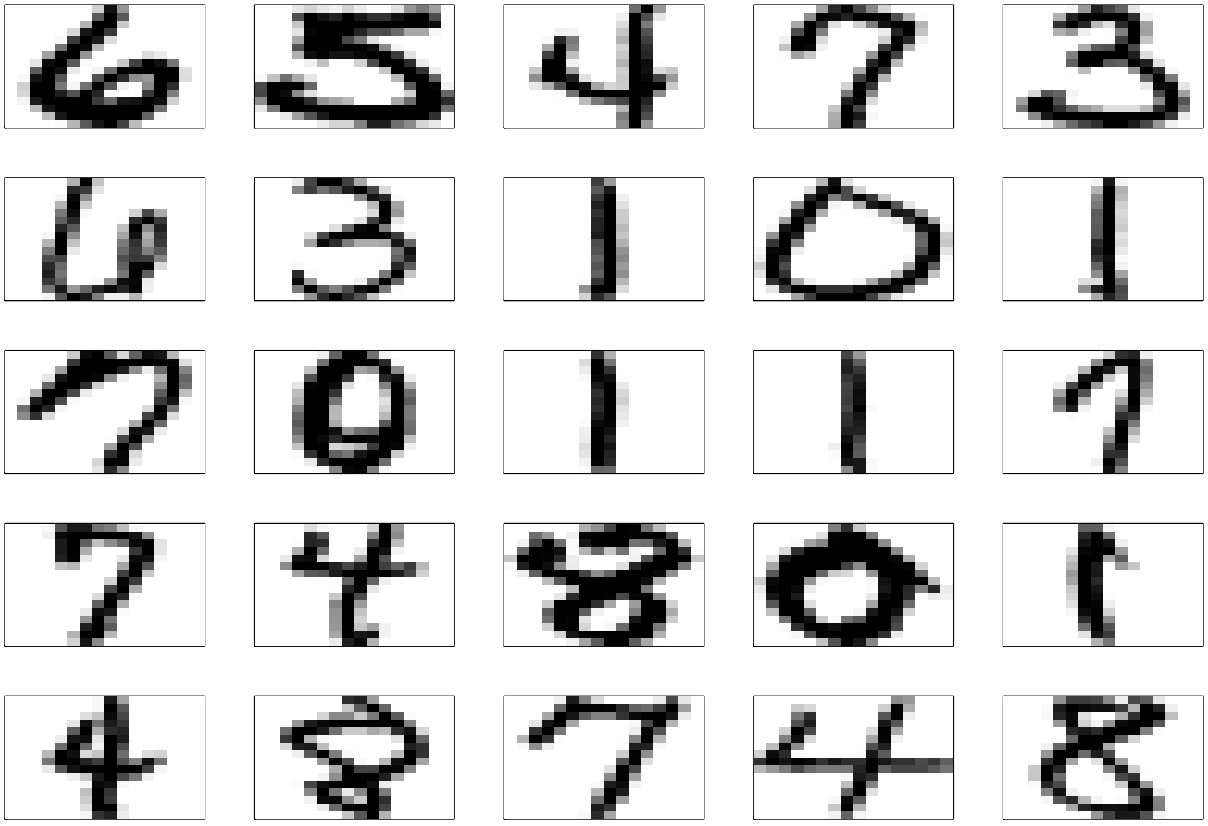


Figure 1.2: The first 25 handwritten numerals, digitized

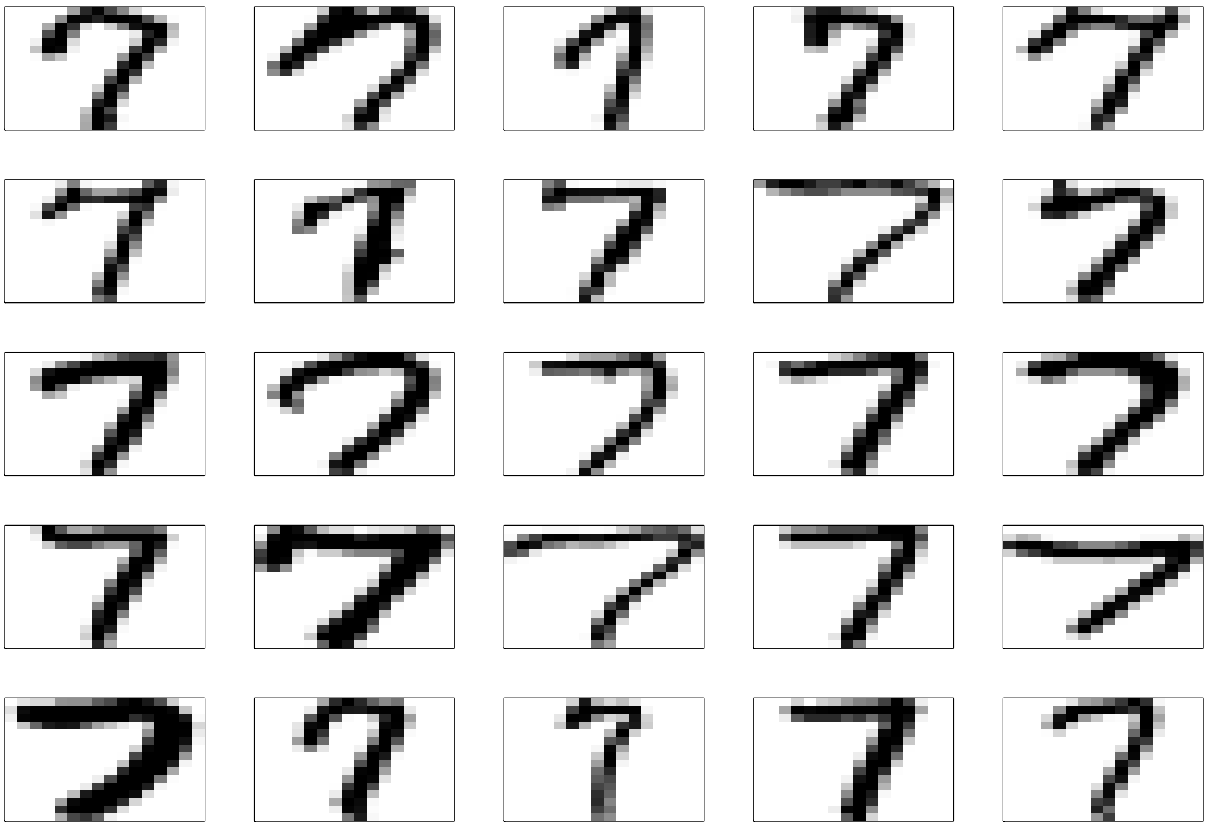


Figure 1.3: The first 25 numeral sevens, digitized

1. Obtaining and manipulating data
2. Graphical tools for exploring and summarizing data
3. Communicating findings about data that support reproducible research
4. Tools for classification problems such as email spam filtering or handwritten digit recognition
5. Programming and wirting functions in R

1.6 How to learn (The most important section in this book!)

There are several ways to engage with the content of this book and associated materials.

One way is not to engage at all. Leave the book closed on a shelf and do something else with your time. That may or may not be a good life strategy, depending on what else you do with your time, but you won't learn much from the book!

Another way to engage is to read through the book "passively", reading all that's written but not reading the book while at your computer, where you could enter the R commands from the book. With this strategy you'll probably learn more than if you leave the book closed on a shelf, but there are better options.

A third way to engage is to read the book while you're at a computer, enter the R commands from the book as you read about them, and work on the practice exercises at the end of each chapter. You'll likely learn more this way.

A fourth strategy is even better. In addition to reading, entering the commands given in the book, and working through the practice exercises, you think about what you're doing, and ask yourself questions (which you then go on to answer). For example after working through some R code computing the logarithm of positive numbers you might ask yourself, "What would R do if I asked it to calculate the logarithm of a negative number? What would R do if I asked it to calculate the logarithm of a really large number such as one trillion?" You could explore these questions easily by just trying things out in the R Console window.

If your goal is to maximize the time you have to binge-watch *Stranger Things* Season 2 on Netflix, the first strategy may be optimal. But if your goal is to learn a lot about computational tools for data science, the fourth strategy is probably going to be best.

Chapter 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] 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

[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.

[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.

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 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
```

```
[1] 234
```

```
> exp(2)  ##The exponential function
```

```
[1] 7.389
```

```
> log10(100)  ##Base 10 logarithm
```

```
[1] 2
```

```
> log(100)  ##Base e logarithm
```

```
[1] 4.605
```

```
> 10^log10(55)
```

```
[1] 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
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
[18] 18 19 20 21 22 23 24 25
```

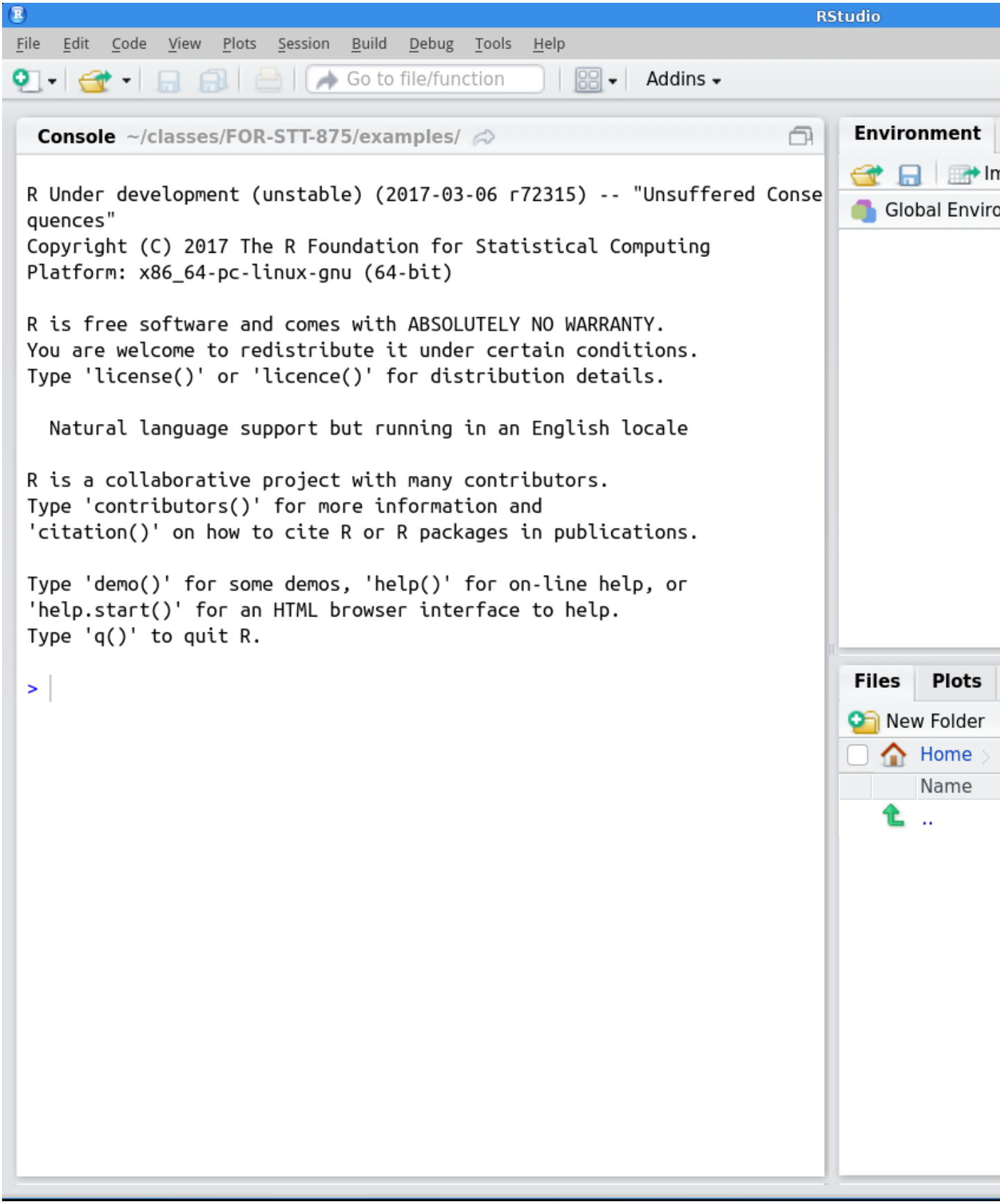


Figure 2.1: The RStudio IDE

```
> log(1:25)  ##The base e logarithm of these integers

 [1] 0.0000 0.6931 1.0986 1.3863 1.6094 1.7918 1.9459
 [8] 2.0794 2.1972 2.3026 2.3979 2.4849 2.5649 2.6391
[15] 2.7081 2.7726 2.8332 2.8904 2.9444 2.9957 3.0445
[22] 3.0910 3.1355 3.1781 3.2189
> 1:25 * 1:25  ##What will this produce?

 [1]   1   4   9  16  25  36  49  64  81 100 121 144
[13] 169 196 225 256 289 324 361 400 441 484 529 576
[25] 625
> 1:25 * 1:5  ##What about this?

 [1]   1   4   9  16  25   6  14  24  36  50  11  24
[13]  39  56  75  16  34  54  76 100  21  44  69  96
[25] 125
> seq(from = 0, to = 1, by = 0.1)  ##A sequence of numbers from 0 to 1

 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
> exp(seq(from = 0, to = 1, by = 0.1))  ##What will this produce?

 [1] 1.000 1.105 1.221 1.350 1.492 1.649 1.822 2.014
 [9] 2.226 2.460 2.718
```

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 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",
```

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

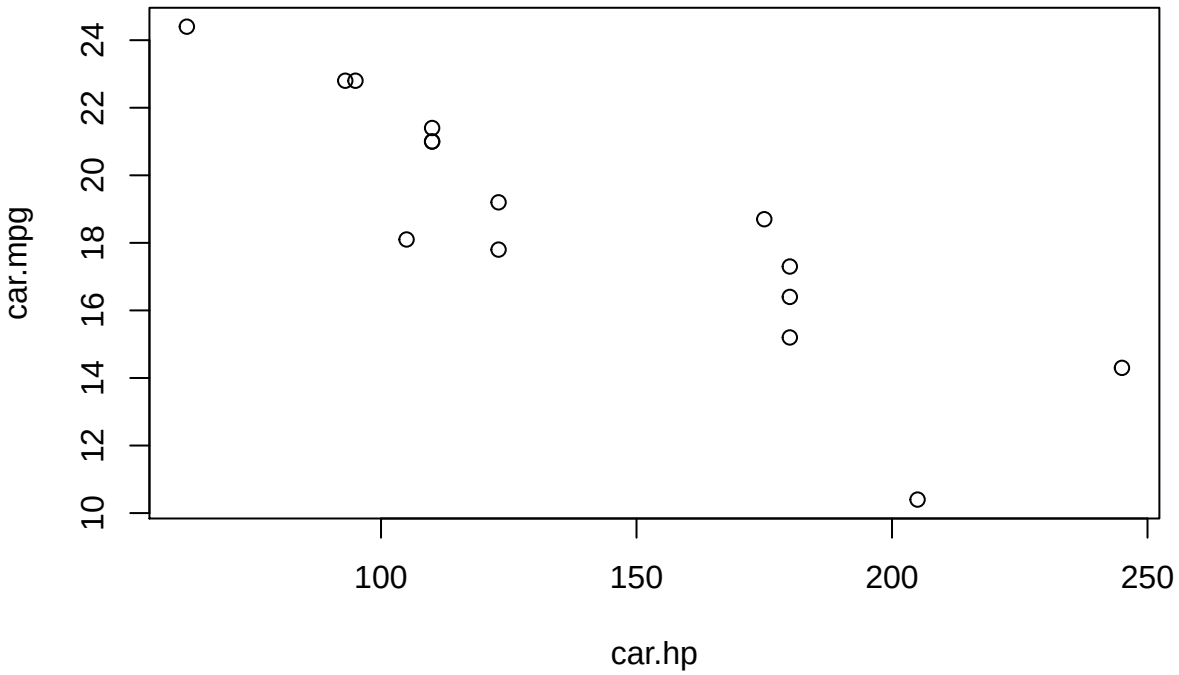
```
+               "Hornet 4 Drive", "Hornet Sportabout", "Valiant",
+               "Duster 360", "Merc 240D", "Merc 230", "Merc 280",
+               "Merc 280C", "Merc 450SE", "Merc 450SL",
+               "Merc 450SLC", "Cadillac Fleetwood")
> car.hp

 [1] 110 110  93 110 175 105 245  62  95 123 123 180
[13] 180 180 205
> car.mpg

 [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2
[11] 17.8 16.4 17.3 15.2 10.4
> car.name

 [1] "Mazda RX4"          "Mazda RX4 Wag"
 [3] "Datsun 710"         "Hornet 4 Drive"
 [5] "Hornet Sportabout"  "Valiant"
 [7] "Duster 360"         "Merc 240D"
 [9] "Merc 230"           "Merc 280"
[11] "Merc 280C"          "Merc 450SE"
[13] "Merc 450SL"         "Merc 450SLC"
[15] "Cadillac Fleetwood"
```

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

```
> mean(car.hp)

[1] 139.7
> sd(car.hp)

[1] 50.78
> summary(car.hp)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
     62     108     123     140     180     245
> mean(car.mpg)

[1] 18.72
> sd(car.mpg)

[1] 3.714
> summary(car.mpg)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   10.4    16.9    18.7    18.7    21.2    24.4
```

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.

[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.

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, which is a search engine focused specifically on R. Both Google and rseek.org 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 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. Below are a few of the key points from the guide that we will use right away.

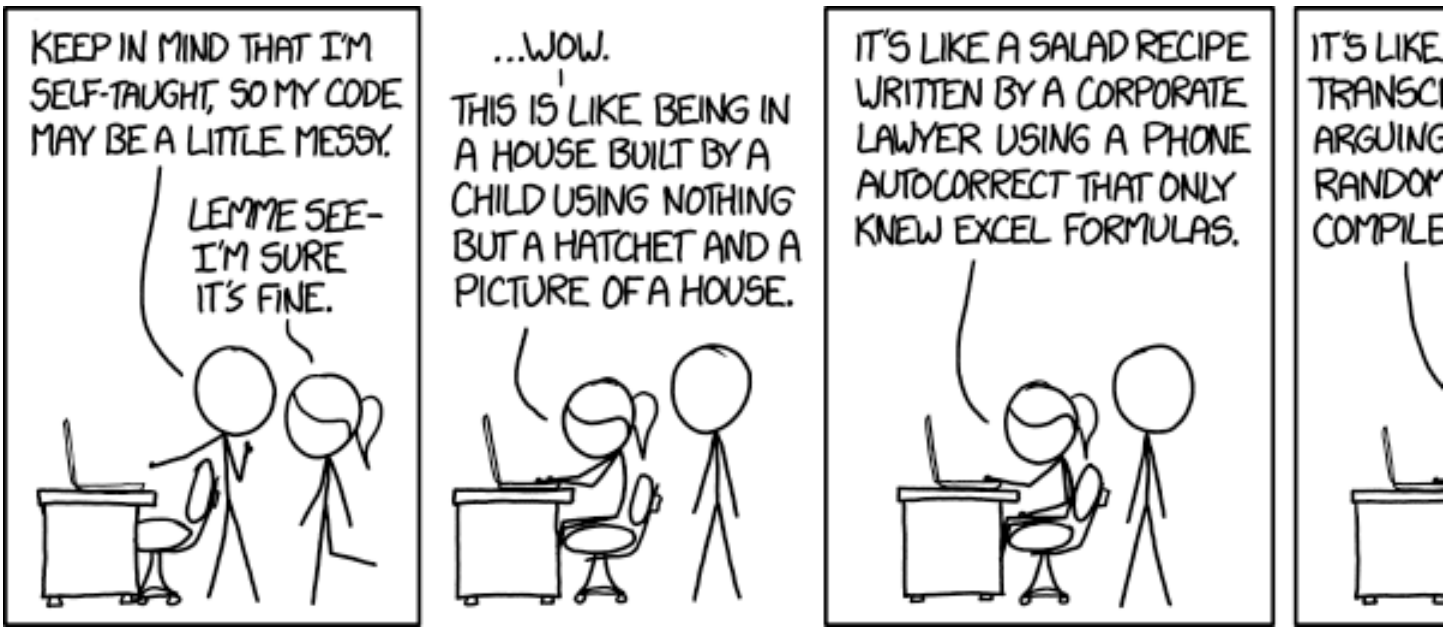


Figure 2.2: Code Quality

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)

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.)

Chapter 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.

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)
```

[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.

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)
```

```
 [1] "iso2c"
 [2] "country"
 [3] "year"
 [4] "fertility.rate"
 [5] "life.expectancy"
 [6] "population"
 [7] "GDP.per.capita.Current.USD"
 [8] "X15.to.25.yr.female.literacy"
 [9] "iso3c"
[10] "region"
[11] "capital"
[12] "longitude"
[13] "latitude"
[14] "income"
[15] "lending"
```

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]
```

```
 [1]    NA 6.928 7.671 4.425 6.186 4.550 7.316 3.109
 [9]    NA 2.690
```

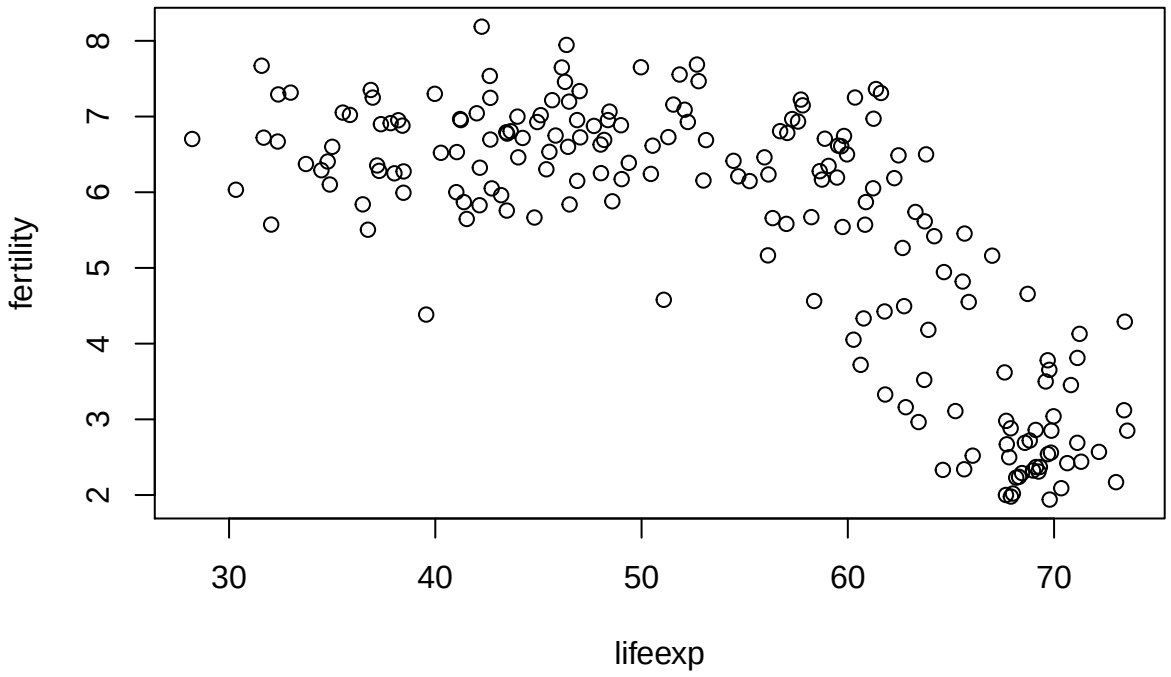
```
> lifeexp[1:10]
```

```
 [1]    NA 52.24 31.58 61.78 62.25 65.86 32.98 65.22
 [9]    NA 68.59
```

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]
```

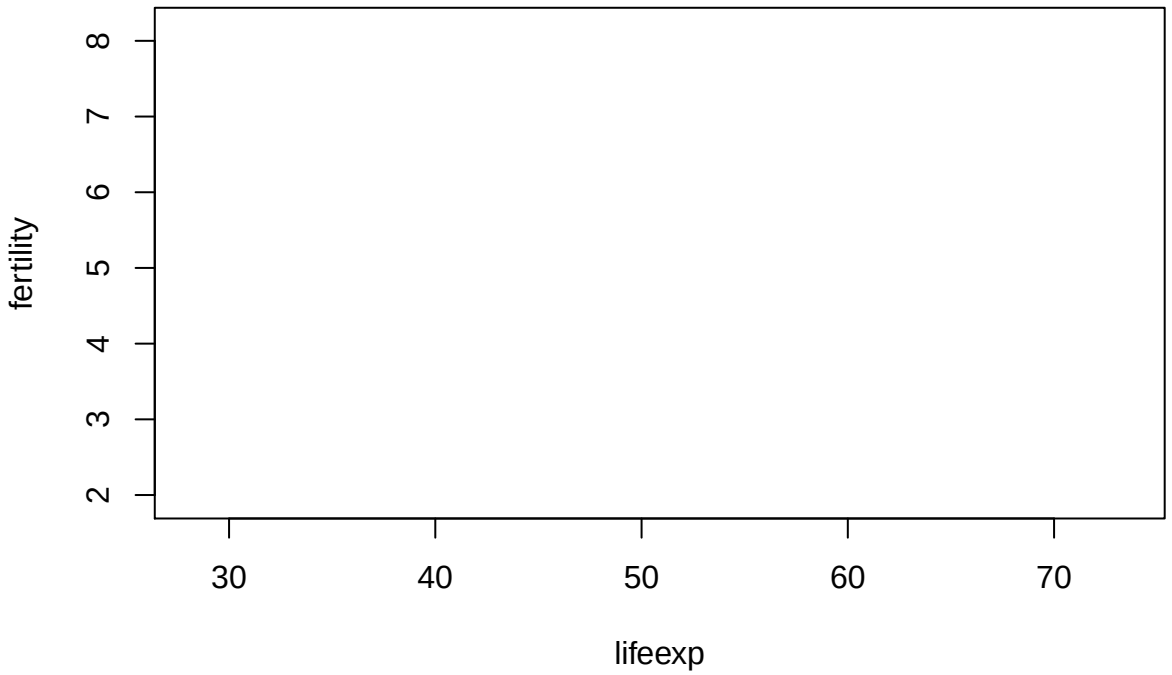
```
 [1]    13414    89608  8774440    54681  1608800
 [6]  1867396  4965988 20623998    20012  7047539
```

```
> region[1:10]
```

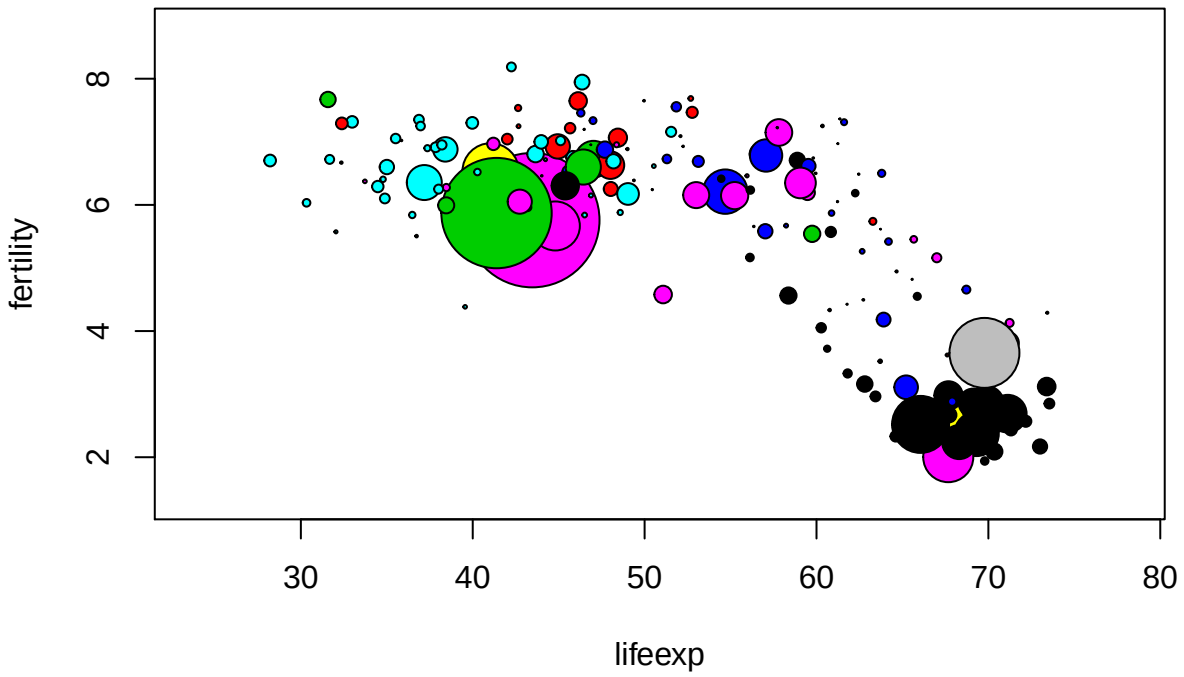
```
 [1] "Europe & Central Asia (all income levels)"
 [2] "Middle East & North Africa (all income levels)"
 [3] "South Asia"
 [4] "Latin America & Caribbean (all income levels)"
 [5] "Europe & Central Asia (all income levels)"
 [6] "Europe & Central Asia (all income levels)"
 [7] "Sub-Saharan Africa (all income levels)"
 [8] "Latin America & Caribbean (all income levels)"
 [9] "East Asia & Pacific (all income levels)"
[10] "Europe & Central Asia (all income levels)"
```

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

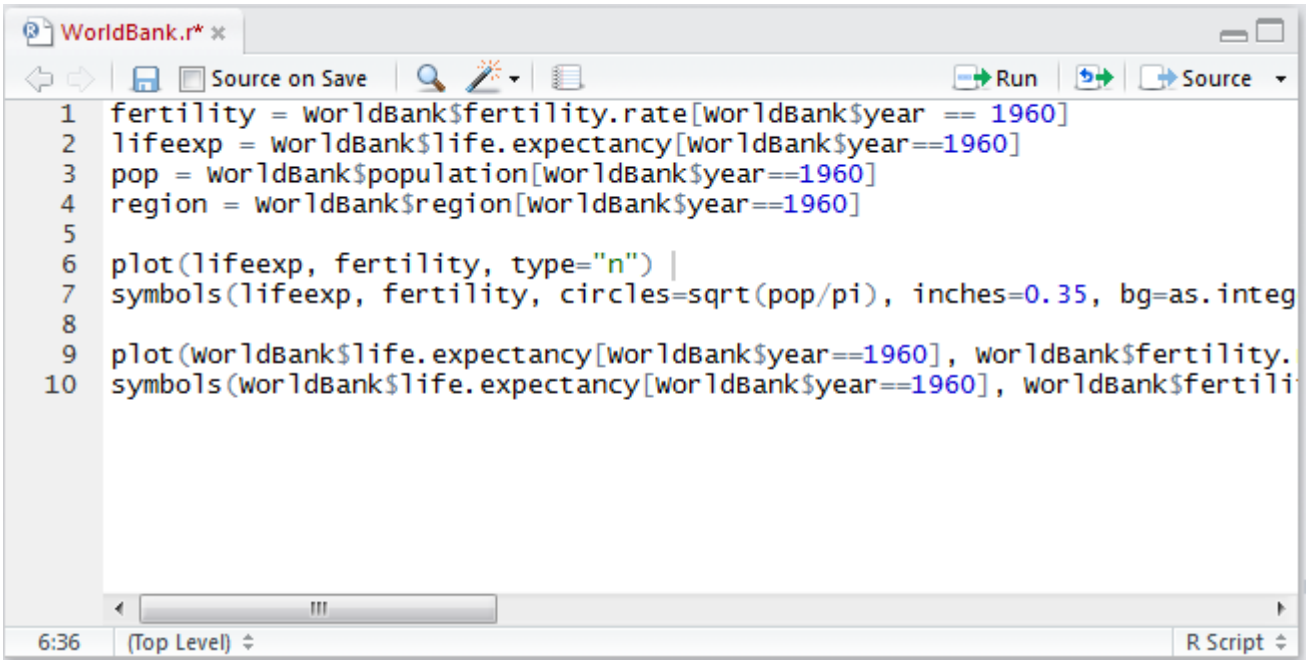


Figure 3.1: A script window in RStudio

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

[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.

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 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] 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.

```
`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.

R Markdown

Andy Finley

April 3, 2017

Basic formatting:

italic

bold

~~strikethrough~~

A code chunk:

```
x <- 1:10
y <- 10:1
mean(x)
```

```
## [1] 5.5
```

```
sd(y)
```

```
## [1] 3.02765
```

Inline code:

10

Inline code not executed:

`5+5`

Figure 3.2: Output from the above R Markdown code

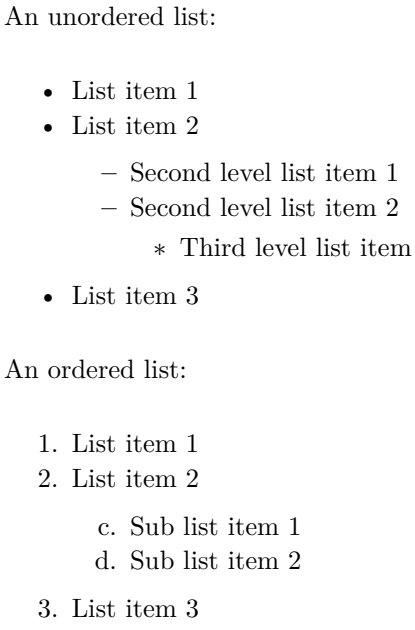


Figure 3.3: R Markdown List Output

- 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
```

```
> 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.

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

$\int_0^1 x^2 dx$

$\frac{x}{y}$

$\sqrt{x}$

$\sqrt[y]{x}$

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

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

Figure 3.4: More R Markdown Output

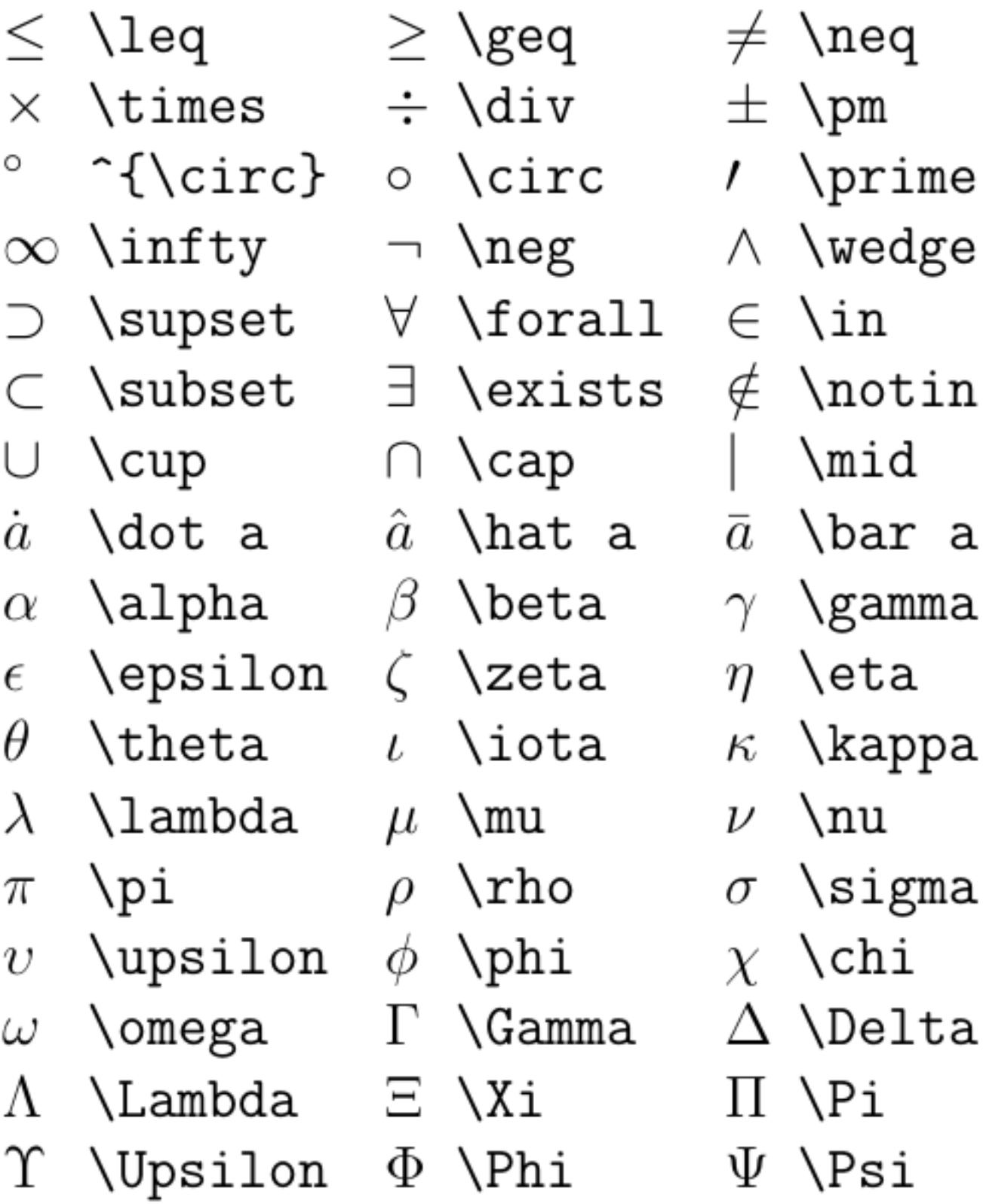


Figure 3.5: Other useful LaTeX expressions and symboles available for use in R Markdown

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 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.

[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/.

No options:

```
x = 1:10
x
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

echo=FALSE:

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

comment=NA:

```
x = 1:10
x
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

comment='#', prompt=TRUE:

```
> x = 1:10
> x
```

```
#  [1]  1  2  3  4  5  6  7  8  9 10
```

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

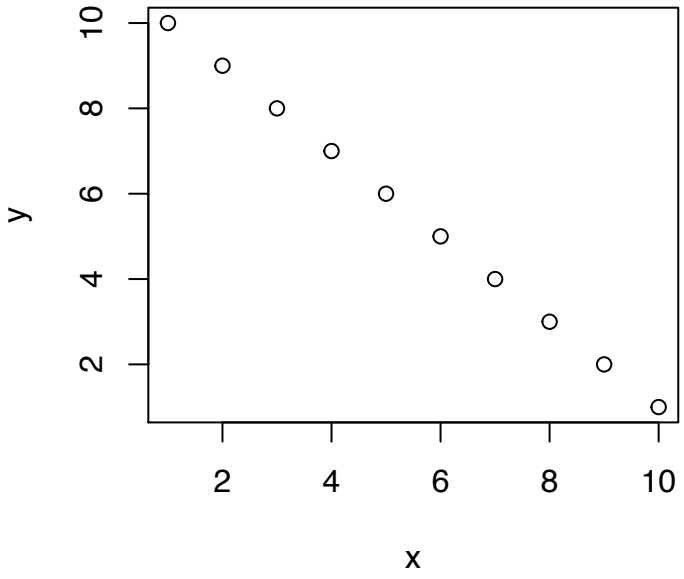


Figure 3.6: Output of Example R Code Chunk Options

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.

Chapter 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)
```

```
 num 1
```

```
> is.vector(a)
```

```
[1] TRUE
```

```
> length(a)
```

Table 4.1: Dimension and type content of base data structures in R

Dimension	Homogeneous	Heterogeneous
1	Atomic vector	List
2	Matrix	Data frame
N	Array	

```
[1] 1
```

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
```

```
[1] 123 157 205 199 223 140 105
```

```
> gender <- c("female", "female", "male", "female", "male",
+             "male", "female")
> gender
```

```
[1] "female" "female" "male"   "female" "male"
[6] "male"   "female"
```

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
```

```
[1] 123 157 205 199 223 140 105 194
```

```
> gender
```

```
[1] "female" "female" "male"   "female" "male"
[6] "male"   "female" "male"
```

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 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.

[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.

```
> typeof(weight)
```

```
[1] "double"
```

```
> typeof(gender)
```

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

```
> bp <- c(FALSE, TRUE, FALSE, FALSE, TRUE, FALSE, TRUE, FALSE)
> bp
```

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

```
> typeof(bp)
```

```
[1] "logical"
```

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
```

```
[1] 123 157 205 199 223 140 105 194
```

```
> typeof(weight.int)
```

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

```
> weight.char <- as.character(weight)
> weight.char
```

```
[1] "123" "157" "205" "199" "223" "140" "105" "194"
```

```
> bp.double <- as.double(bp)
> bp.double
```

```
[1] 0 1 0 0 1 0 1 0
```

```
> gender.oops <- as.double(gender)
```

```
Warning: NAs introduced by coercion
```

```
> gender.oops
```

```
[1] NA NA NA NA NA NA NA NA
```

```
> sum(bp)
```

```
[1] 3
```

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 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)
```

```
[1] TRUE
```

```
> is.character(weight)
```

```
[1] FALSE
```

```
> is.integer(weight.int)
```

```
[1] TRUE
```

```
> is.logical(bp)
```

```
[1] TRUE
```

4.1.1.1 Coercion

Consider the following examples.

```
> xx <- c(1, 2, 3, TRUE)
> xx
```

```
[1] 1 2 3 1
```

```
> yy <- c(1, 2, 3, "dog")
> yy
```

```
[1] "1"   "2"   "3"   "dog"
```

```
> zz <- c(TRUE, FALSE, "cat")
> zz
```

```
[1] "TRUE"  "FALSE" "cat"
```

```
> weight + bp
```

```
[1] 123 158 205 199 224 140 106 194
```

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]`.

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

```
> weight
```

```
[1] 123 157 205 199 223 140 105 194
```

```
> weight[5]
```

```
[1] 223
```

```
> weight[1:3]
```

```
[1] 123 157 205
```

```
> length(weight)
```

```
[1] 8
```

```
> weight[length(weight)]
```

```
[1] 194
```

```
> weight[]
```

```
[1] 123 157 205 199 223 140 105 194
```

```
> weight[3] <- 202
> weight
```

```
[1] 123 157 202 199 223 140 105 194
```

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]
```

```
[1] 123 157 199 223 140 105 194
```

```
> weight[-length(weight)]
```

```
[1] 123 157 202 199 223 140 105
```

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

```
[1] 157 199 140 105 194
```

```
> weight[0]
```

```
numeric(0)
```

```
> weight[c(0, 2, 1)]
```

```
[1] 157 123
```

```
> weight[c(-1, 2)]
```

```
Error in weight[c(-1, 2)]: only 0's may be mixed with negative subscripts
```

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
```

```
[1] "middle age" "senior"     "middle age" "senior"
[5] "senior"     "senior"     "senior"     "middle age"
```

```
> income
```

```
[1] "lower"  "lower"  "upper"  "middle" "upper"
[6] "lower"  "lower"  "middle"
```

```
> age <- factor(age, levels=c("youth", "young adult", "middle age",
+                             "senior"))
> age
```

```
[1] middle age senior     middle age senior
[5] senior     senior     senior     middle age
Levels: youth young adult middle age senior
```

```
> income <- factor(income, levels=c("lower", "middle", "upper"),
+                  ordered = TRUE)
> income
```

```
[1] lower  lower  upper  middle upper  lower  lower
[8] middle
Levels: lower < middle < upper
```

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
```

```
[1] 123 157 202 199 223 140 105 194
```

```
> Weight
```

```
Error in eval(expr, envir, enclos): object 'Weight' not found
```

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
```

```
[1] "dog"   "cat"   NA      "pig"   NA      "horse"
```

```
> is.na(missingCharacter)
```

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

```
> missingCharacter <- c(missingCharacter, "NA")
> missingCharacter
```

```
[1] "dog"   "cat"   NA      "pig"   NA      "horse"
[7] "NA"
```

```
> is.na(missingCharacter)
```

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

```
> allMissing <- c(NA, NA, NA)
> typeof(allMissing)
```

```
[1] "logical"
```

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))
```

```
[1] NA
```

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

```
[1] 2.75
```

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] 0 1 2 3 4
```

```
> 1/x
```

```
[1]    Inf 1.0000 0.5000 0.3333 0.2500
```

```
> x/x
```

```
[1] NaN   1   1   1   1
```

```
> y <- c(10, 1000, 10000)
> 2^y
```

```
[1]  1.024e+03 1.072e+301        Inf
```

`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
```

```
  Weight Gender bp.meds
1    123 female   FALSE
2    157 female    TRUE
3    202   male   FALSE
4    199 female   FALSE
5    223   male    TRUE
6    140   male   FALSE
7    105 female    TRUE
8    194   male   FALSE
```

```
> names(healthData)
```

```
[1] "Weight"  "Gender"  "bp.meds"
```

```
> colnames(healthData)
```

```
[1] "Weight"  "Gender"  "bp.meds"
```

```
> names(healthData) <- c("Wt", "Gdr", "bp")
> healthData
```

```
   Wt    Gdr    bp
1 123 female FALSE
2 157 female  TRUE
3 202   male FALSE
4 199 female FALSE
5 223   male  TRUE
6 140   male FALSE
7 105 female  TRUE
8 194   male FALSE
```

```
> rownames(healthData)
```

```
[1] "1" "2" "3" "4" "5" "6" "7" "8"
```

```
> 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)
```

```
                   mpg cyl disp  hp drat    wt  qsec
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02
Datsun 710        22.8   4  108  93 3.85 2.320 18.61
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02
Valiant           18.1   6  225 105 2.76 3.460 20.22
                  vs am gear carb
Mazda RX4          0  1    4    4
Mazda RX4 Wag      0  1    4    4
Datsun 710         1  1    4    1
Hornet 4 Drive     1  0    3    1
Hornet Sportabout  0  0    3    2
Valiant            1  0    3    1
```

```
> tail(mtcars)
```

```
                mpg cyl  disp  hp drat    wt qsec vs
Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.7  0
Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.9  1
Ford Pantera L 15.8   8 351.0 264 4.22 3.170 14.5  0
Ferrari Dino   19.7   6 145.0 175 3.62 2.770 15.5  0
Maserati Bora  15.0   8 301.0 335 3.54 3.570 14.6  0
Volvo 142E     21.4   4 121.0 109 4.11 2.780 18.6  1
               am gear carb
Porsche 914-2   1    5    2
Lotus Europa    1    5    2
Ford Pantera L  1    5    4
Ferrari Dino    1    5    6
Maserati Bora   1    5    8
Volvo 142E      1    4    2
```

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]
```

```
[1] 110
```

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

```
[1] 160 160 108
```

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

```
              cyl disp
Mazda RX4       6  160
Mazda RX4 Wag   6  160
Datsun 710      4  108
```

```
> mtcars[, 1]
```

```
 [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2
[11] 17.8 16.4 17.3 15.2 10.4 10.4 14.7 32.4 30.4 33.9
[21] 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7
[31] 15.0 21.4
```

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
```

```
 [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2
[11] 17.8 16.4 17.3 15.2 10.4 10.4 14.7 32.4 30.4 33.9
[21] 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7
[31] 15.0 21.4
```

```
> mtcars$cyl
```

```
 [1] 6 6 4 6 8 6 8 4 4 6 6 8 8 8 8 8 8 4 4 4 4 8 8 8 8
[26] 4 4 4 8 6 8 4
```

```
> mpg
```

```
# A tibble: 234 x 11
   manufacturer model displ  year   cyl trans drv
   <chr>        <chr> <dbl> <int> <int> <chr> <chr>
 1 audi         a4      1.8  1999     4 auto~ f
 2 audi         a4      1.8  1999     4 manu~ f
 3 audi         a4      2    2008     4 manu~ f
 4 audi         a4      2    2008     4 auto~ f
 5 audi         a4      2.8  1999     6 auto~ f
 6 audi         a4      2.8  1999     6 manu~ f
 7 audi         a4      3.1  2008     6 auto~ f
 8 audi         a4 q~   1.8  1999     4 manu~ 4
 9 audi         a4 q~   1.8  1999     4 auto~ 4
10 audi         a4 q~   2    2008     4 manu~ 4
# ... with 224 more rows, and 4 more variables:
#   cty <int>, hwy <int>, fl <chr>, class <chr>
```

```
> cyl
```

```
Error in eval(expr, envir, enclos): object 'cyl' not found
```

```
> weight
```

```
[1] 123 157 202 199 223 140 105 194
```

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)
```

```
[1] "list"
```

```
> names(mpgHpLinMod)
```

```
 [1] "coefficients"  "residuals"     "effects"
 [4] "rank"          "fitted.values" "assign"
 [7] "qr"            "df.residual"   "xlevels"
[10] "call"          "terms"         "model"
```

```
> mpgHpLinMod$coefficients
```

```
(Intercept)          hp
   30.09886    -0.06823
```

```
> mpgHpLinMod$residuals
```

```
          Mazda RX4       Mazda RX4 Wag
           -1.59375            -1.59375
         Datsun 710      Hornet 4 Drive
           -0.95363            -1.19375
  Hornet Sportabout             Valiant
            0.54109            -4.83489
          Duster 360           Merc 240D
            0.91707            -1.46871
           Merc 230            Merc 280
           -0.81717            -2.50678
          Merc 280C          Merc 450SE
           -3.90678            -1.41777
         Merc 450SL         Merc 450SLC
           -0.51777            -2.61777
 Cadillac Fleetwood Lincoln Continental
           -5.71206            -5.02978
  Chrysler Imperial            Fiat 128
            0.29364             6.80421
        Honda Civic      Toyota Corolla
            3.84901             8.23598
     Toyota Corona    Dodge Challenger
           -1.98072            -4.36462
       AMC Javelin          Camaro Z28
```

```
          -4.66462           -0.08293
   Pontiac Firebird          Fiat X1-9
           1.04109             1.70421
      Porsche 914-2       Lotus Europa
           2.10991             8.01093
     Ford Pantera L       Ferrari Dino
           3.71340             1.54109
      Maserati Bora         Volvo 142E
           7.75761            -1.26198
```

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 2 vector of coefficients, while another component is the length 32 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
```

```
$first
[1] 123 157 202 199 223 140 105 194

$second
  Weight Gender bp.meds
1    123 female   FALSE
2    157 female    TRUE
3    202   male   FALSE
4    199 female   FALSE
5    223   male    TRUE
6    140   male   FALSE
7    105 female    TRUE
8    194   male   FALSE

$pickle
$pickle$a
 [1]  1  2  3  4  5  6  7  8  9 10

$pickle$b
  Weight Gender bp.meds
1    123 female   FALSE
2    157 female    TRUE
3    202   male   FALSE
4    199 female   FALSE
5    223   male    TRUE
6    140   male   FALSE
7    105 female    TRUE
8    194   male   FALSE
```

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 **??**, 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.

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

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.

```
> temporaryList$first
```

```
[1] 123 157 202 199 223 140 105 194
```

```
> mode(temporaryList$first)
```

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

```
> temporaryList[[1]]
```

```
[1] 123 157 202 199 223 140 105 194
```

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

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

```
> temporaryList[1]
```

```
$first
[1] 123 157 202 199 223 140 105 194
```

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

```
[1] "list"
```

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)]
```

```
$first
[1] 123 157 202 199 223 140 105 194

$second
  Weight Gender bp.meds
1    123 female   FALSE
2    157 female    TRUE
3    202   male   FALSE
4    199 female   FALSE
5    223   male    TRUE
6    140   male   FALSE
7    105 female    TRUE
8    194   male   FALSE
```

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

```
[1] 157
```

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)]]
```

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

```
Error in temporaryList[[c(1, 2, 3)]]: recursive indexing failed at level 2
```

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 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 **??**.

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
```

```
[1] 123 157 202 199 223 140 105 194
```

```
> weight > 200
```

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

```
> gender[weight > 200]
```

```
[1] "male" "male"
```

```
> weight[weight > 200]
```

```
[1] 202 223
```

```
> gender == "female"
```

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

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

```
[1] 123 157 199 105
```

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 female. 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
```

```
[1] 123 157 202 199 223 140 105 194
```

```
> lightweight <- weight[weight < 200]
> lightweight
```

```
[1] 123 157 199 140 105 194
```

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

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

```
> x[x < 5] <- 0
> x
```

```
 [1]  0  0  0  0  5  6  7  8  9 10
```

```
> y <- -3:9
> y
```

```
 [1] -3 -2 -1  0  1  2  3  4  5  6  7  8  9
```

```
> y[y < 0] <- NA
> y
```

```
 [1] NA NA NA  0  1  2  3  4  5  6  7  8  9
```

```
> rm(x)
> rm(y)
```

4.7.2 Logical Subsetting and Data Frames

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

```
> healthData
```

```
  Weight Gender bp.meds
1    123 female   FALSE
```

```
2    157 female    TRUE
3    202   male   FALSE
4    199 female   FALSE
5    223   male    TRUE
6    140   male   FALSE
7    105 female    TRUE
8    194   male   FALSE
```

```
> healthData$Weight[healthData$Gender == "male"]
```

```
[1] 202 223 140 194
```

```
> healthData[healthData$Gender == "female", ]
```

```
  Weight Gender bp.meds
1    123 female   FALSE
2    157 female    TRUE
4    199 female   FALSE
7    105 female    TRUE
```

```
> healthData[healthData$Weight > 190, 2:3]
```

```
  Gender bp.meds
3   male   FALSE
4 female   FALSE
5   male    TRUE
8   male   FALSE
```

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)
```

```
'data.frame':   32 obs. of  11 variables:
 $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
 $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
 $ disp: num  160 160 108 258 360 ...
 $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
 $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
 $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
 $ qsec: num  16.5 17 18.6 19.4 17 ...
 $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
 $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
 $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
 $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

```
> str(temporaryList)
```

```
List of 3
 $ first : num [1:8] 123 157 202 199 223 140 105 194
 $ second:'data.frame': 8 obs. of  3 variables:
  ..$ Weight : num [1:8] 123 157 202 199 223 140 105 194
  ..$ Gender : chr [1:8] "female" "female" "male" "female" ...
```

```
  ..$ bp.meds: logi [1:8] FALSE TRUE FALSE FALSE TRUE FALSE ...
 $ pickle:List of 2
  ..$ a: int [1:10] 1 2 3 4 5 6 7 8 9 10
  ..$ b:'data.frame':   8 obs. of  3 variables:
  .. ..$ Weight : num [1:8] 123 157 202 199 223 140 105 194
  .. ..$ Gender : chr [1:8] "female" "female" "male" "female" ...
  .. ..$ bp.meds: logi [1:8] FALSE TRUE FALSE FALSE TRUE FALSE ...
```

```
> str(WorldBank)
```

```
'data.frame':   11880 obs. of  15 variables:
 $ iso2c                      : chr  "AD" "AD" "AD" "AD" ...
 $ country                    : chr  "Andorra" "Andorra" "Andorra" "Andorra" ...
 $ year                       : int  1978 1979 1977 2007 1976 2011 2012 2008 1980 1972 ...
 $ fertility.rate             : num  NA NA NA 1.18 NA NA NA 1.25 NA NA ...
 $ life.expectancy            : num  NA NA NA NA NA NA NA NA NA NA ...
 $ population                 : num  33746 34819 32769 81292 31781 ...
 $ GDP.per.capita.Current.USD : num  9128 11820 7751 39923 7152 ...
 $ X15.to.25.yr.female.literacy: num  NA NA NA NA NA NA NA NA NA NA ...
 $ iso3c                      : chr  "AND" "AND" "AND" "AND" ...
 $ region                     : chr  "Europe & Central Asia (all income levels)" "Europe & Central Asi
 $ capital                    : chr  "Andorra la Vella" "Andorra la Vella" "Andorra la Vella" "Andorra
 $ longitude                  : num  1.52 1.52 1.52 1.52 1.52 ...
 $ latitude                   : num  42.5 42.5 42.5 42.5 42.5 ...
 $ income                     : chr  "High income: nonOECD" "High income: nonOECD" "High income: nonOE
 $ lending                    : chr  "Not classified" "Not classified" "Not classified" "Not classifie
```

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)
```

```
[1] 216  15
```

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 216 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]
```

```
 [1]    NA 6.512 7.671 3.517 4.933 3.118 7.264 3.104
 [9]    NA 2.200 2.961 2.788 4.479 2.260 2.775 2.949
[17] 6.942 2.210 6.657 2.100 6.293 7.329 6.786    NA
```

```
[25] 5.771
```

```
> !is.na(WorldBank1971$fertility.rate[1:25])
```

```
 [1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
 [9] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[17]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE
[25]  TRUE
```

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

```
[1] 193  15
```

From `dim` we see that there are 193 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
```

```
  V1 V2 V3
1  1 NA  1
2  2  1  2
3  3  4  3
4  4  5  5
5 NA NA  7
```

```
> is.na(temporaryDataFrame)
```

```
        V1    V2    V3
[1,] FALSE  TRUE FALSE
[2,] FALSE FALSE FALSE
[3,] FALSE FALSE FALSE
[4,] FALSE FALSE FALSE
[5,]  TRUE  TRUE FALSE
```

```
> rowSums(is.na(temporaryDataFrame))
```

```
[1] 1 0 0 0 2
```

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)
```

```
[1] 11880    15
```

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

```
[1] 564  15
```

Out of the 564 rows in the original data frame, only 564 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
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

```
> -5:3
```

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

```
> 10:4
```

```
[1] 10  9  8  7  6  5  4
```

```
> pi:7
```

```
[1] 3.142 4.142 5.142 6.142
```

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)
```

```
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

```
> seq(from = 1, to = 5, by = 1/3)
```

```
 [1] 1.000 1.333 1.667 2.000 2.333 2.667 3.000 3.333
 [9] 3.667 4.000 4.333 4.667 5.000
```

```
> seq(from = 3, to = -1, length = 10)
```

```
 [1]  3.0000  2.5556  2.1111  1.6667  1.2222  0.7778
 [7]  0.3333 -0.1111 -0.5556 -1.0000
```

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

```
> rep(c(1, 2, 4), length = 9)
```

```
[1] 1 2 4 1 2 4 1 2 4
```

```
> rep(c(1, 2, 4), times = 3)
```

```
[1] 1 2 4 1 2 4 1 2 4
```

```
> rep(c("a", "b", "c"), times = c(3, 2, 7))
```

```
 [1] "a" "a" "a" "b" "b" "c" "c" "c" "c" "c" "c" "c"
```

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.

Chapter 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 **?** 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.

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)
```

```
'data.frame':   150 obs. of  5 variables:
```

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

```
 $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

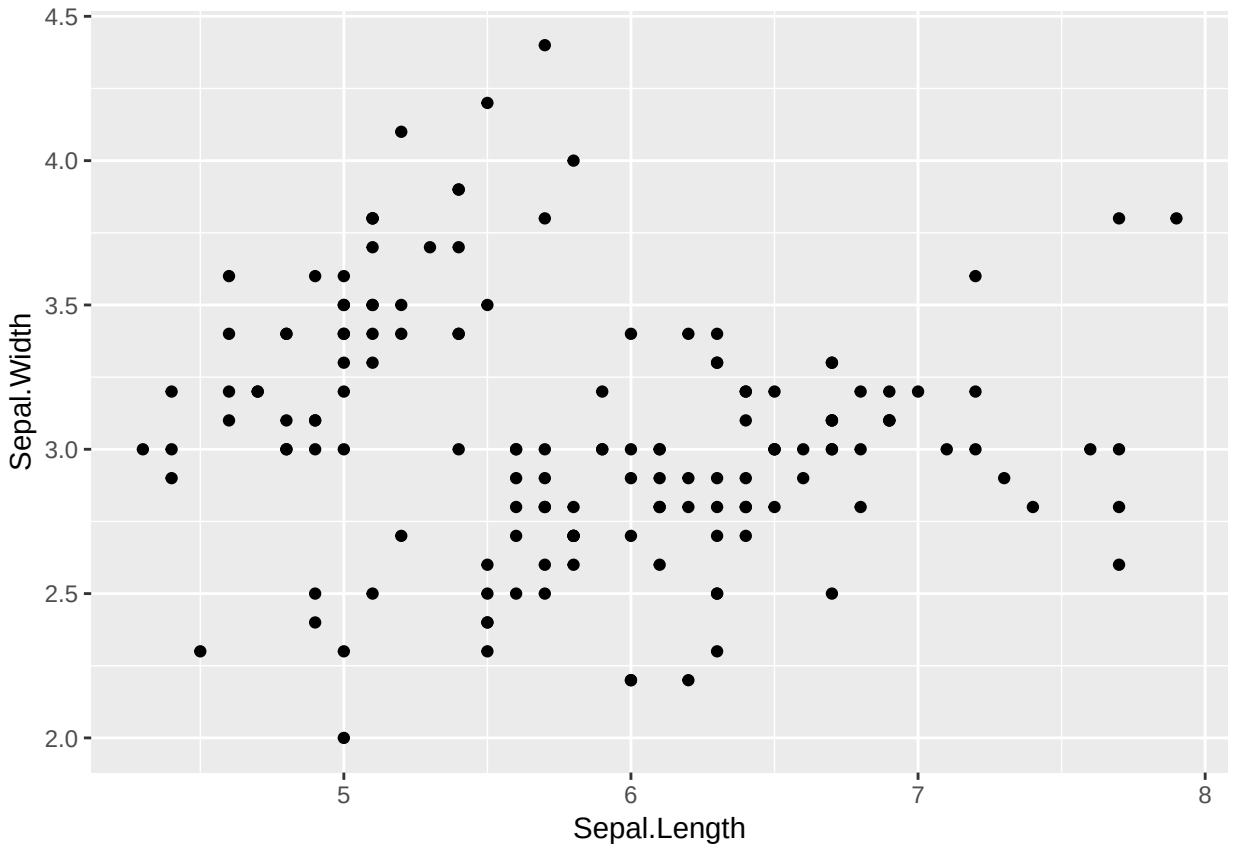
The data contain measurements on petal and sepal length and width for 150 iris plants. The plants are from one of three species, and the species information is also included in the data frame. The data are commonly used to test classification methods, where the goal would be to correctly determine the 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()
```

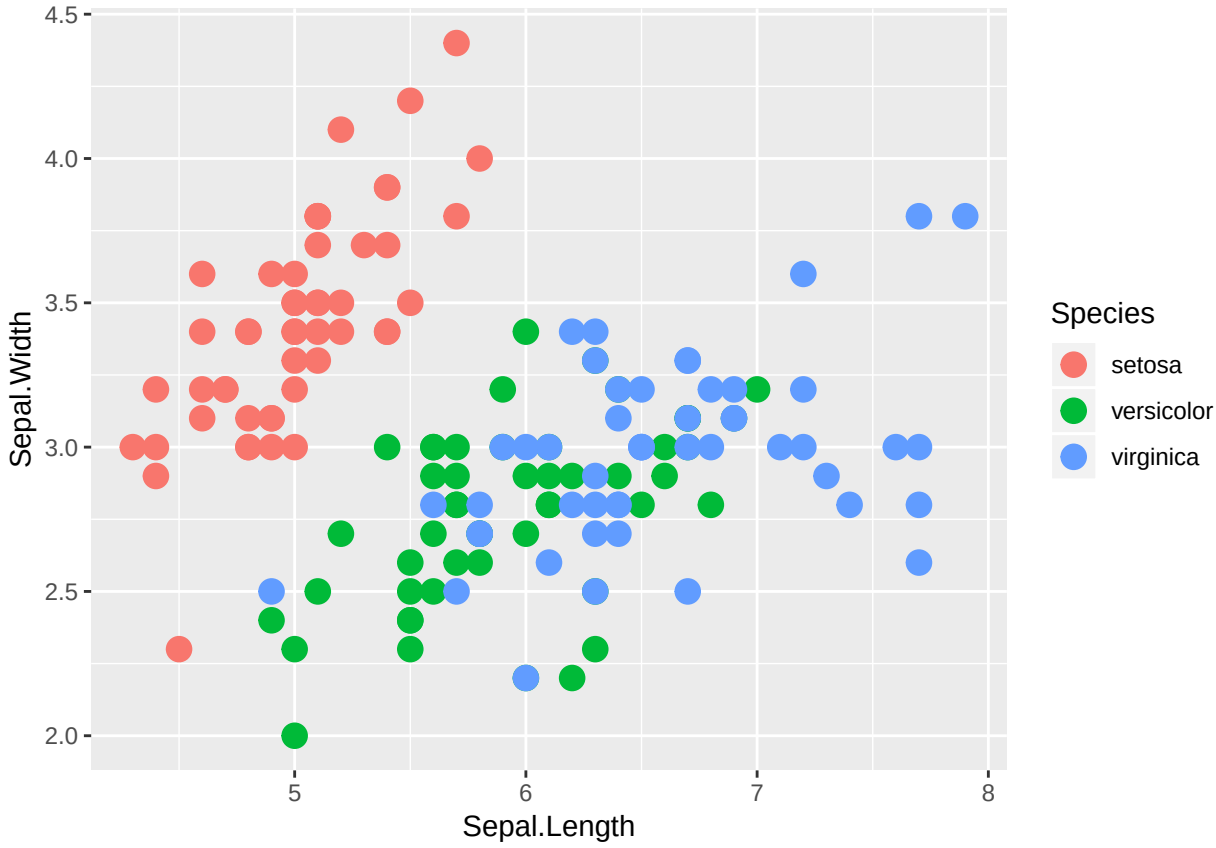


In this case the first argument to the `ggplot` function is the name of the data frame. Second, the `aes` (short for aesthetics) function specifies the mapping to the `x` and `y` axes. By itself the `ggplot` function as written doesn't tell R what sort of graphical display is desired. That is done by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

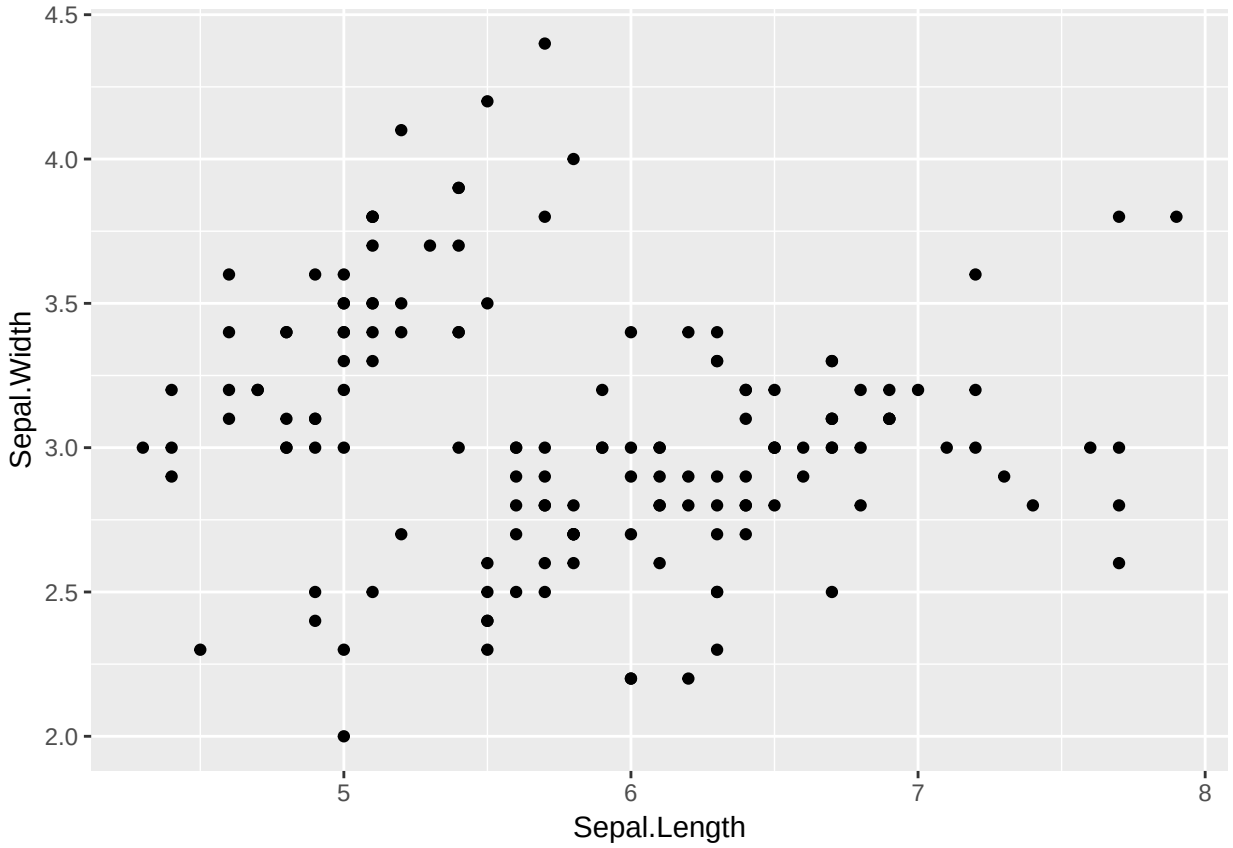
Here we see that the legend automatically changes to include species specific color and shape. The size of the points seems more appropriate.

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()
```

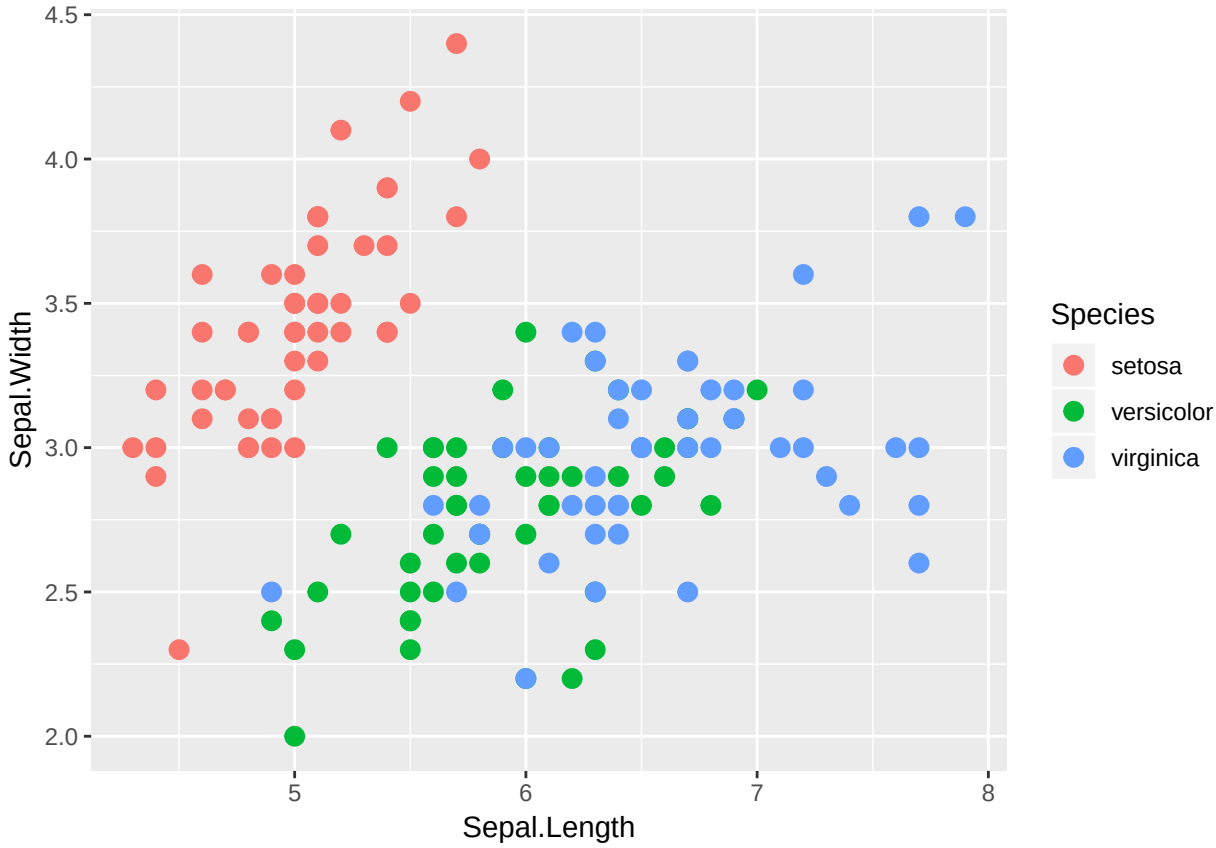



```
iris.p + geom_point(size = 3, aes(color = Species, shape = Species))
```

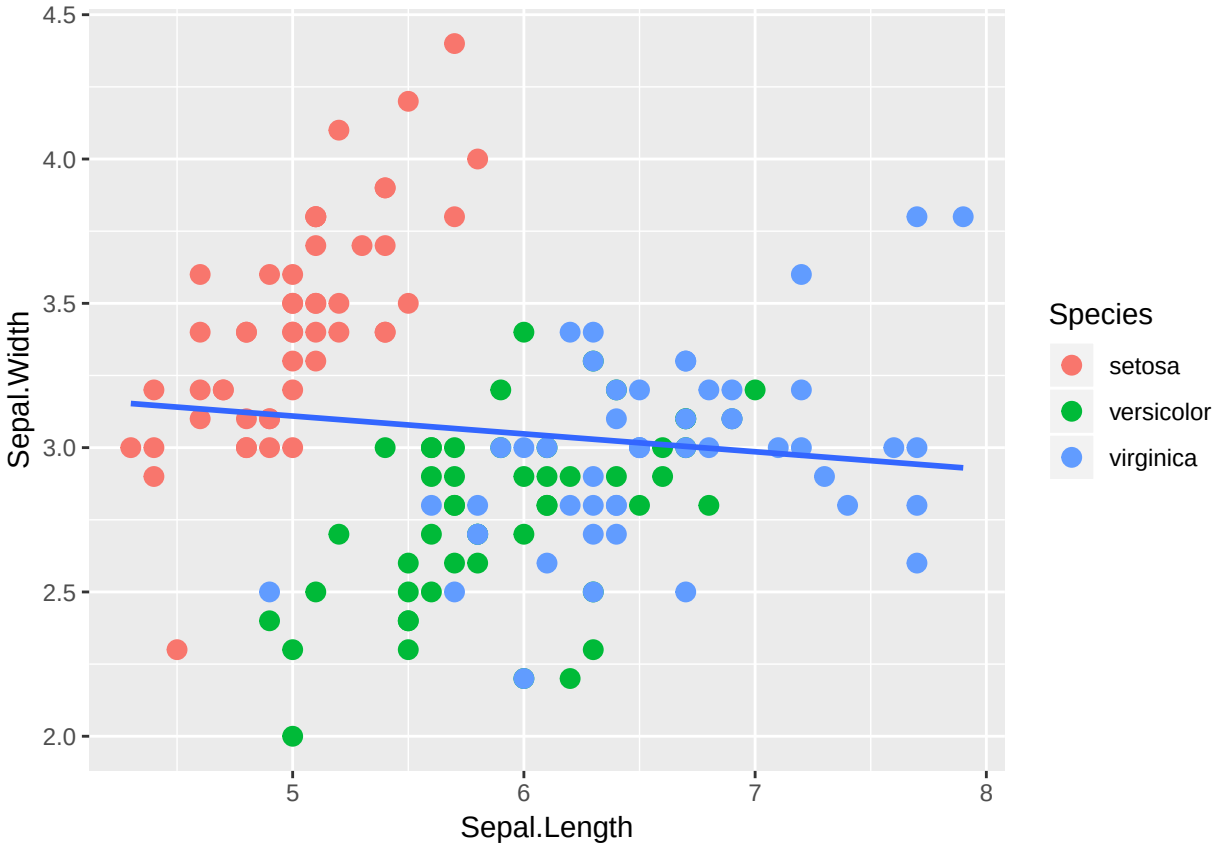
5.1.2 Adding lines to a scatter plot

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))
```

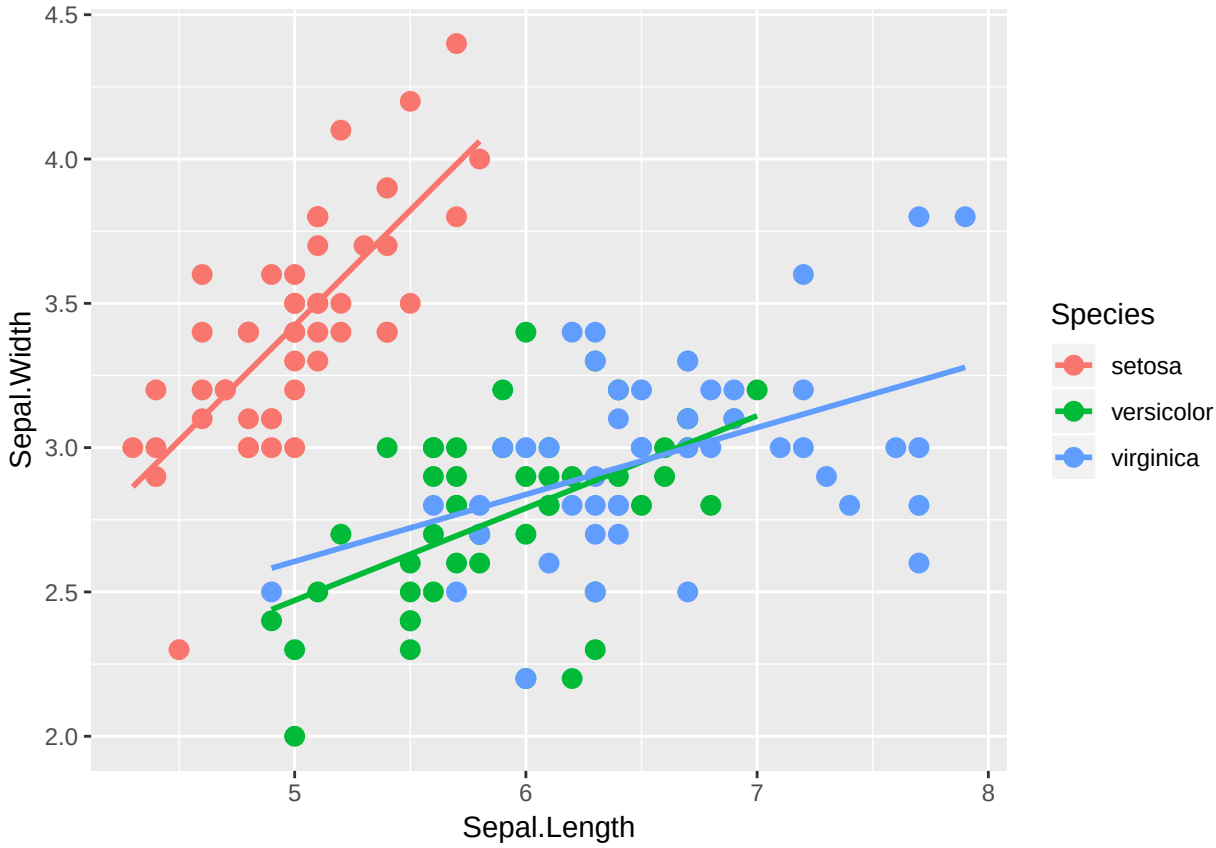



```
ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width)) +
    geom_point(size=3, aes(color=Species)) +
    stat_smooth(method = lm, se=FALSE)
```



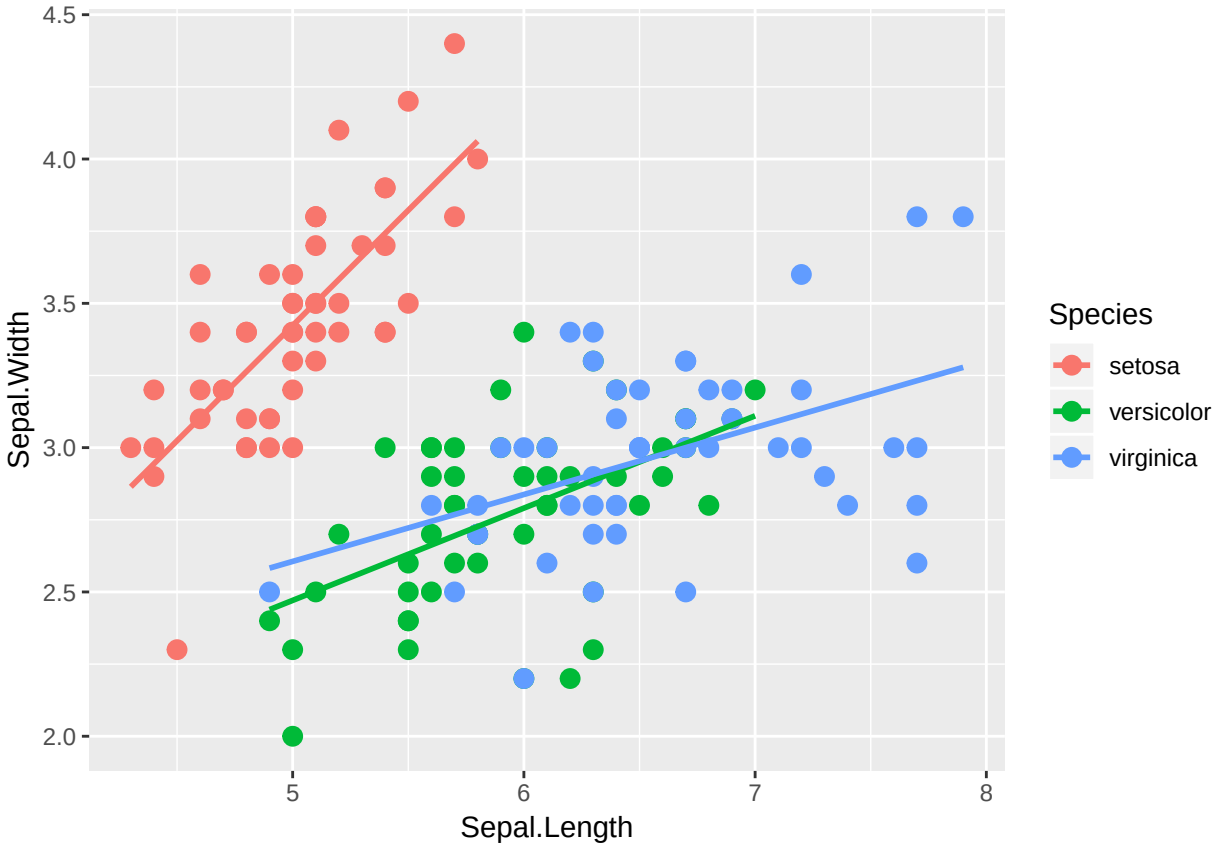
For the iris data, it probably makes more sense to fit separate lines by species. This can be specified using the `aes()` function inside `stat_smooth()`.

```
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))
```



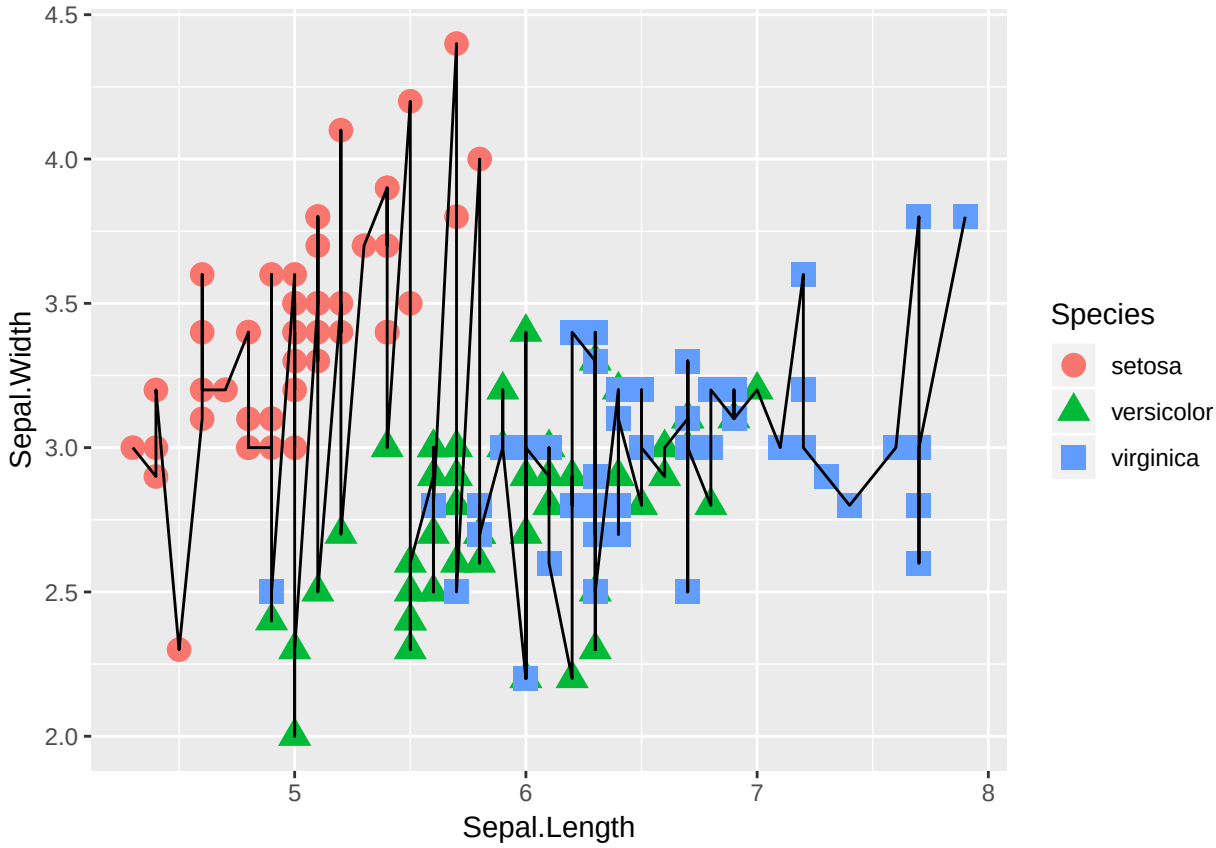
In this case we specified the same color aesthetic for the points and the lines. If we know we want this color aesthetic (colors corresponding to species) for all aspects of the graphic, we can specify it in the main `ggplot()` function:

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

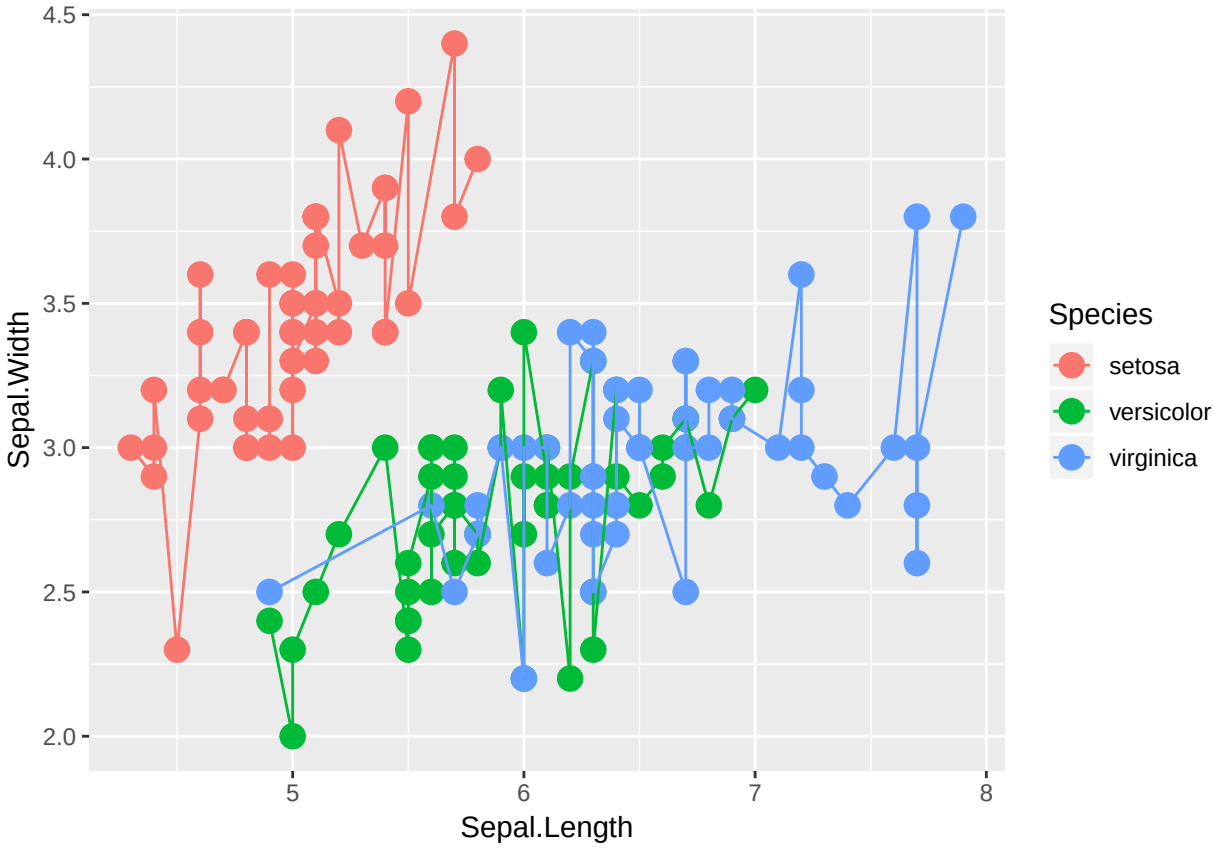


Another common use of line segments in a graphic is to connect the points in order, accomplished via the `geom_line()` function. Although it is not clear why this helps in understanding the iris data, the technique is illustrated next, first doing this for all the points in the graphic, and second doing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

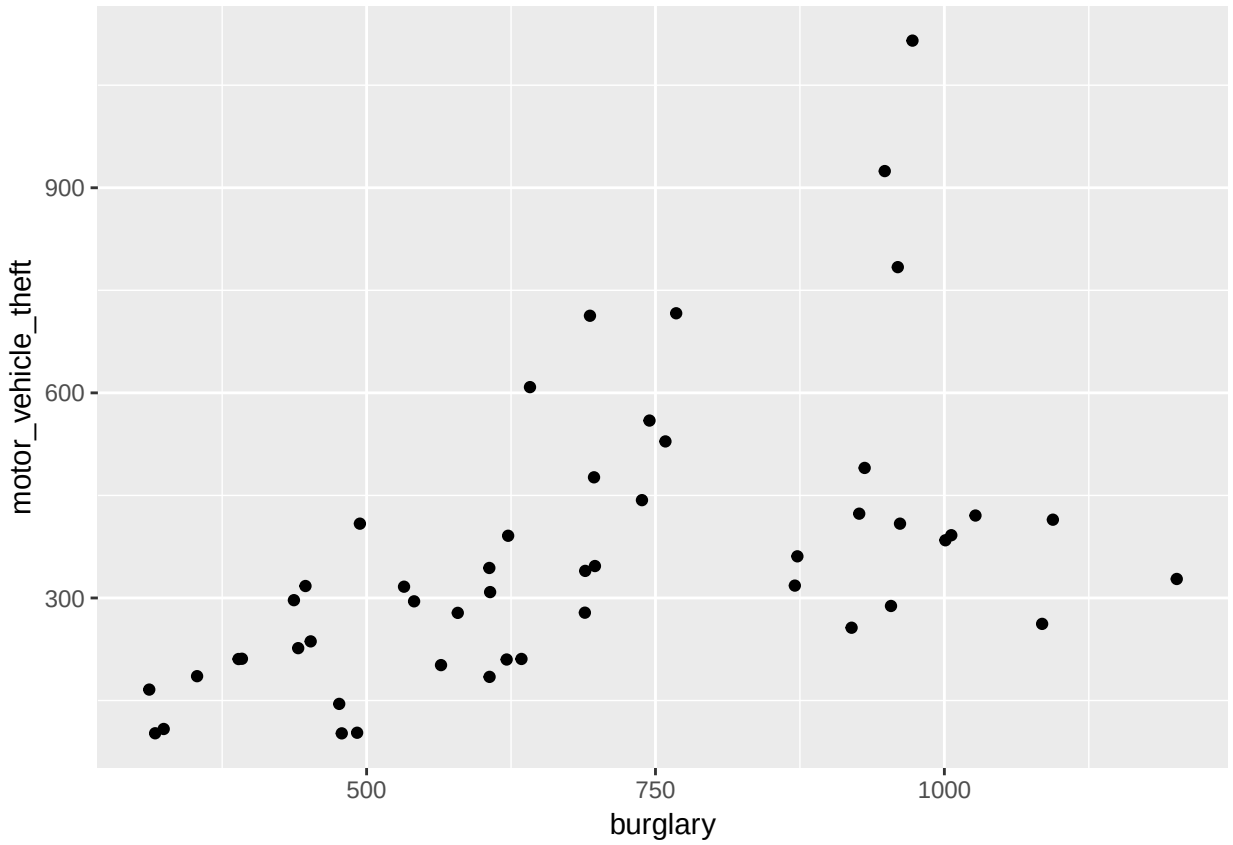
First, read in the data, examine its structure, and produce a simple scatter plot of motor vehicle theft versus burglary.

```
> u.crime <- "http://blue.for.msu.edu/FOR875/data/crimeRatesByState2005.csv"
> crime <- read.csv(u.crime, header=TRUE)
> str(crime)
```

```
'data.frame':   50 obs. of  9 variables:
 $ state              : Factor w/ 50 levels "Alabama ","Alaska ",..: 1 2 3 4 5 6 7 8 9 10 ...
 $ murder             : num  8.2 4.8 7.5 6.7 6.9 3.7 2.9 4.4 5 6.2 ...
 $ Forcible_rate      : num  34.3 81.1 33.8 42.9 26 43.4 20 44.7 37.1 23.6 ...
 $ Robbery            : num  141.4 80.9 144.4 91.1 176.1 ...
 $ aggravated_assult  : num  248 465 327 387 317 ...
 $ burglary           : num  954 622 948 1085 693 ...
```

```
 $ larceny_theft      : num  2650 2599 2965 2711 1916 ...
 $ motor_vehicle_theft: num  288 391 924 262 713 ...
 $ population         : int  4627851 686293 6500180 2855390 36756666 4861515 3501252 873092 18328340 968
```

```
> ggplot(data <- crime, aes(x = burglary, y = motor_vehicle_theft)) +
+     geom_point()
```

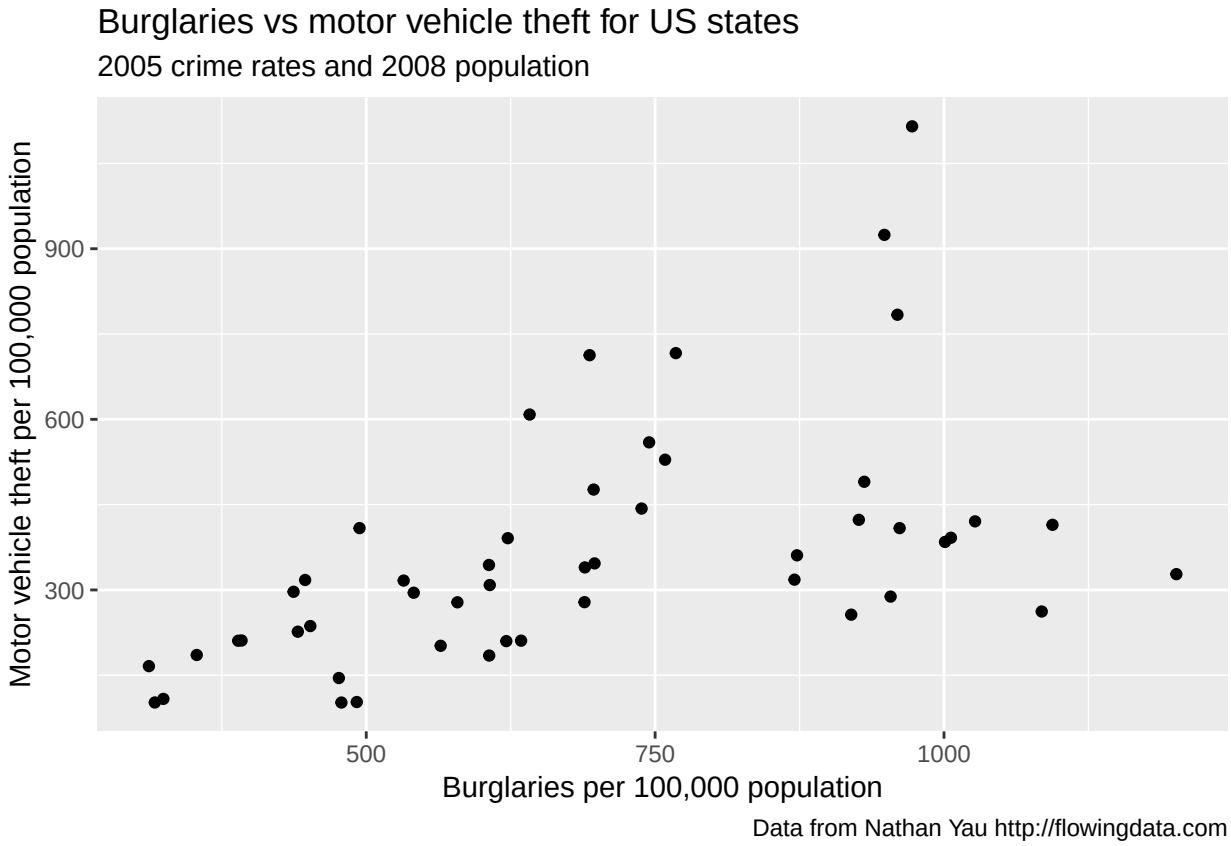


5.2.1 Labels

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"
         )
```

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


Burglaries vs motor vehicle theft for US states
2005 crime rates and 2008 population
Motor vehicle theft per 100,000 population
900
600
300
500
750
1000
Burglaries per 100,000 population
Data from Nathan Yau http://flowingdata.com

Chapter 6

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 altered to start at zero and go to the maximum of the x and y variables.[1] 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)))
```

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