raw - R Actuarial Workshops

Brian A. Fannin, ACAS

2016-09-11

Contents

Introduction

> "R isn't software. It's a community." — John Chambers

Hello! Very happy to have you here and I hope you find this useful. This book is meant to serve as a companion to any of the R training sessions that I'm involved in. A few quick notes before we proceed.

Why does this book exist?

For over three years now, I've joined other actuaries in teaching R at events sponsored by the Casualty Actuarial Society[1]. I've learned a lot about what questions get asked, where folks get stuck and what content matters most. We've reached the place where, to be honest, attendees can get more out of the live sessions if they come in having done some preliminary work. That should give us more opportunity for hands on instruction. At a minimum, this book should serve as a handy reference before, during and after a live training to reinforce what we're trying to teach.

That understood, this book is hardly the only game in town. There are loads of good books about R and I can easily recommend many of them. [Mat11] is a great one. Go check them out. I have.

This is an organic book

You'll not find this in Barnes & Nobel or Amazon and I'll strongly suggest that you resist the temptation to print this. I fully expect that there will corrections, additions and updates as the technology changes. The book will live on the internet as long as I can support it and it's probably best to check it out there. By all means, download the PDF if you'd like a local copy, but do check back for updates.

You don't need to read this from start to finish

Though I've done my best to give this book a clear flow, I've had to make a few sacrifices in order to get in all the material that I needed. This means that there are some, sorry, boring bits like a page about data types or how to write loops and such. Some people will find this stuff fascinating, some people will find this … necessary. Feel free to treat this like a reference text and not like a Michael Chabon novel and you're likely to get more out of it.

So, that's all the preliminaries. Away we go!

[1] The CAS has not sponsored this publication and no one should construe my involvement with the CAS as constituting their endorsement of the material presented here.

First Steps

The first part of this book will get you started using R. Unless you've been using R comfortably for several months, resist the tempation to skip through this quickly. We've found that one of the biggest hurdles to getting users comfortable with R is getting the software installed and running with a minimum of hassle. The next step is getting folks reconciled to the idea that R is very different from Excel. You can like that, or you can hate that, but you can't change that. Hopefully, by the end of this section, you'll be open to the idea that R can accomplish some tasks and it's something you can add to your tool kit.

Chapter 1

Setup

By the end of this chapter, you will have done the following:

- Install R
- Install RStudio
- Install the `raw` package

1.1 Installing R

1.1.1 Operating systems

Although R was developed primarily on Unix-based operating systems, it may be used on many different platforms. As I write these words, there are three major systems in use: Windows, Mac OS and Linux. I've used R and RStudio on all three and the experience is pretty much the same. This is one of the fantastic features of the software. It's meant to be as widely used and portable as possible to maximize its use.

There are one or two operating system quirks, but in general I won't need to refer to OS differences again, apart from one preliminary note. When refrring to keystroke combinationsn, I will only refer to the CTRL key. Mac OS users will understand that this key is CMD on their keyboards.

If you're curious, the version of R and system architecture being used to write this book are noted below:

```
sessionInfo()$platform
#> [1] "x86_64-pc-linux-gnu (64-bit)"
```

In each case, what you'll do is download a file from the internet and then follow the standard process you go through to install software on whichever system you're using. For the most part, installation is quick and painless, but there may be limitations placed on you by your IT department. I have a few suggestions which I hope can help overcome any difficulties you might experience.

The first place to look for installation is [cran.r-project.org]. From there, you will see links to downloads for Windows, Mac and Linux. Clicking on the appropriate link will take you to the page that's relevant for your operating system. You may see lots of bizarre, arcane language around binaries, source and tarballs. If those words (in this context) mean nothing to you, don't panic. Some folk like to build their own version of R directly from the source code. If you're reading these instructions, you're probably not one of those people.

I recommend getting familiar with the CRAN website and reading the documentation there. If you get totally lost, try the links below which should take you directly to the download site for Windows and Mac. (If you're running Linux, I can't imagine you need my help.)

- Windows install: http://cran.revolutionanalytics.com/bin/windows/base/
- Mac install: http://cran.revolutionanalytics.com/bin/macosx/

It's possible that you'll be asked to identify a "mirror". R is hosted on a number of servers throughout the world. It's all the same R, but distributing it in this way helps to minimized load on servers which host the files. There's nothing much to decide here - the internet is pretty fast.

1.1.2 Bitness

You may be asked to decide between 32 and 64 bit R. The numbers refer to the width of an address in memory. Software will look for instructions or data in memory using one of those two memory addressing schemes. This young, ungrateful millenial generation wasn't around when it happened, but I can remember when we moved from 16 to 32 bit software. It was a big deal. This is one is less so. You'll probably want to use 64 bit R. Know that there's a chance that you'll run into problems working with other software, particularly the Microsoft Access database or the Java virtual machine. This could result if another program uses 32 bit memory addressing. I'm not going to pretend to be sensitive to all the technical nuances. Pretend that you're from the deep south trying to have a conversation with someone in Scottland. Although you're speaking the same language, it's possible that you're not able to have a conversation.

If this happens, you may be able to use 32 bit R, presuming you've installed it. I won't walk through how to implement that here.

1.2 Installing R Studio

Installing R is most of the battle. Depending on the sort of person you are, it may even be all of the battle (see the following section on environments). R comes with a fairly spartan user interface, which is sufficient to get work done. However, most folk find that they enjoy using an Integrated Development Environment (IDE). This allows one to work on several source files at the same time, read help, observe console output, see what variables are loaded in memory, etc. There are a few options, but I've not yet found anything better than RStudio.

RStudio's main website may be found at www.rstudio.com. At the time of writing, the download page may be found at http://www.rstudio.com/products/rstudio/download/. Here you will find links to specific systems. The browser will even attempt to detect what operating system you're using and suggest a link for you. Cool, huh?

1.2.1 IT

I don't think I've ever met anyone who's made it through a white-collar existence without at least one or two frustrating exchanges with a corporate IT department. If you work for a large- or even small- company, you likely have a staff of folks who keep the network running and handle software requests from every user in the company. To ensure that your company's network is free from malicious attack or well-intentioned, but careless or imperfect users, most computers have sensitive areas restricted. This means that if you want to install software, you need an administrator to do it for you.

What this also means is that your IT department might not be as delighted as you are to install open-source software on the company laptop. This might be a problem that they're not inclined to solve for you and you may find your interaction with IT folks to a bit frustrating and they may seem as though they're not at all helpful.

The first thing to bear in mind is that, despite any appearance to the contrary, your IT staff is there to help you. Moreover, they're people. They have families who love them, possibly small children who think their moms and dads are awesome, pets who miss them and lives outside of work. They have to deal with

ridiculous hours to accommodate you and they get far more complaints than they do praise. Be nice to them and you may be surprised how supportive they can be.

With that understood, there are several situations you may find yourself in.:

You have- or can talk your way into- admin rights to your computer

Lucky you. Also lucky me as this is the happy situation that I enjoy. How do you handle this situation? Don't blow it! Be careful what you download, don't greedily consume bandwidth, server space or any of your company's other scarce resources and be VERY NICE to your IT staff. Acknowledge that you're grateful to have been given such trust and pledge not to do anything to have it removed.

You don't have admin rights to your computer

What to do? Request to be given admin rights. Explain why, in detail and don't be evasive or vague. Trust and mutual respect help. Talk to other folks in your department and get them on board. Present a strong business case for why use of this software will permit you and your department to work more efficiently. Show them this book and underline the parts where I tell folk to be nice and respectful towards IT staff.

IT won't give you admin rights to your computer

In this case, you may ask them to install it for you. Pool your resources. Talk to other actuaries and analysts in your company. Talk to your boss.

IT won't install the software.

Solution? Install the software to a memory stick. Yes, it is often (but not always!) possible to do this. This is is obviously not a preferred option, but it will get you up and running and enable you to attend the workshop.

Memory sticks are locked down

In this case, your IT department really wants you all to be running terminals. OK. Suggest an install of RStudio Server. This enables R to run on a server with controlled user access. This is quite a lot more work for your IT staff and you'll need to make a strong use case for it. If you're a large organization which has a predictive modelling or analytics area, they'll likely want this software. This won't allow you to use R remotely, so getting the most out of the workshop will be tough. However, there's still one more option.

The nuclear option

Your IT staff won't run R on a server, won't give you a laptop with R installed. They're really against this software. I'd like to advise you to get another job, but that's defeatist. This is where we reach the nuclear option, which is to use your own computer. This will drive folks at your company nuts. Now you're transferring data from a secure machine to one which you use for personal e-mail, Facebook, sports, personal finance and other activities that we needn't dwell on here. This is an absolute last resort and the overheard of moving stuff from one device to another will obviate most of the efficiency gains that open source software will provide. Here's how to make it work: produce work that is ONLY POSSIBLE using R, or Python or any of the tools which we will discuss. Show a killer visual and then patiently explain to your boss why it can't be done in Excel and why you can't share it with other departments and why it can't be done every quarter. This is a tall order, but it just might get someone's attention.

1.3 Conclusion

Beg, borrow or steal, make friends, vanquish enemies, do whatever you need to do and get the software installed. It may be easy and you'll wonder why I'm making such a fuss. I hope that's how it goes down. If it's difficult, just know that it'll all be worth it. There are a host of crazy issues that you may have to resolve that you'd never see with Excel or Outlook or whatever. Hang in there. With every problem that you solve, you're getting closer and closer to data/stats nirvana. If you emerge from installation with a few battle scars, wear them like badges of honor. One day, you'll be knocking back a beer with Brian Ripley, Hadley Wickham or Dirk Eddelbeutel and speaking their language.

Chapter 2

Getting started

This chapter will give you a short tour through R. We'll assume that you've installed R and Rstudio. By the end of this chapter you will be able to:

- enter some basic commands,
- draw some pictures,
- create your first statistical model,
- save and reload a script,
- understand what a package is and how to install them.

2.1 The Operating Environment

Right. So, you've got R installed. Now what? Among the first differences you'll encounter relative to Excel is that you now have several different options when it comes to using R. R is an engine designed to process R commands. Where you store those commands and how you deal with that output is something over which you have a great deal of control. Terrible, frighening control. Here are those options in a nutshell:

- Command-line interface (CLI)
- RGui
- RStudio
- Others

2.1.1 Command-line interface

R, like S before it, presumed that users would interact with the program from the command line. And, if you invoke the R command from a terminal, that's exactly what you'll get. The image below is from my

Throughout this book, I will assume that you're using RStudio. You don't have to, but I will strongly recommend it. Why?

- Things are easier with RStudio
- RStudio, keeps track of all the variables in memory
- Everyone else is using it[1].

[1] OK, not much of an argument. This is the exact opposite of the logic our parents used to try and discourage us from smoking. However, in this case, it makes sense. When you're talking with other people and trying to reproduce your problem or share your awesome code, they're probably using RStudio. Using the same tool reduces the amount of effort needed to communicate.

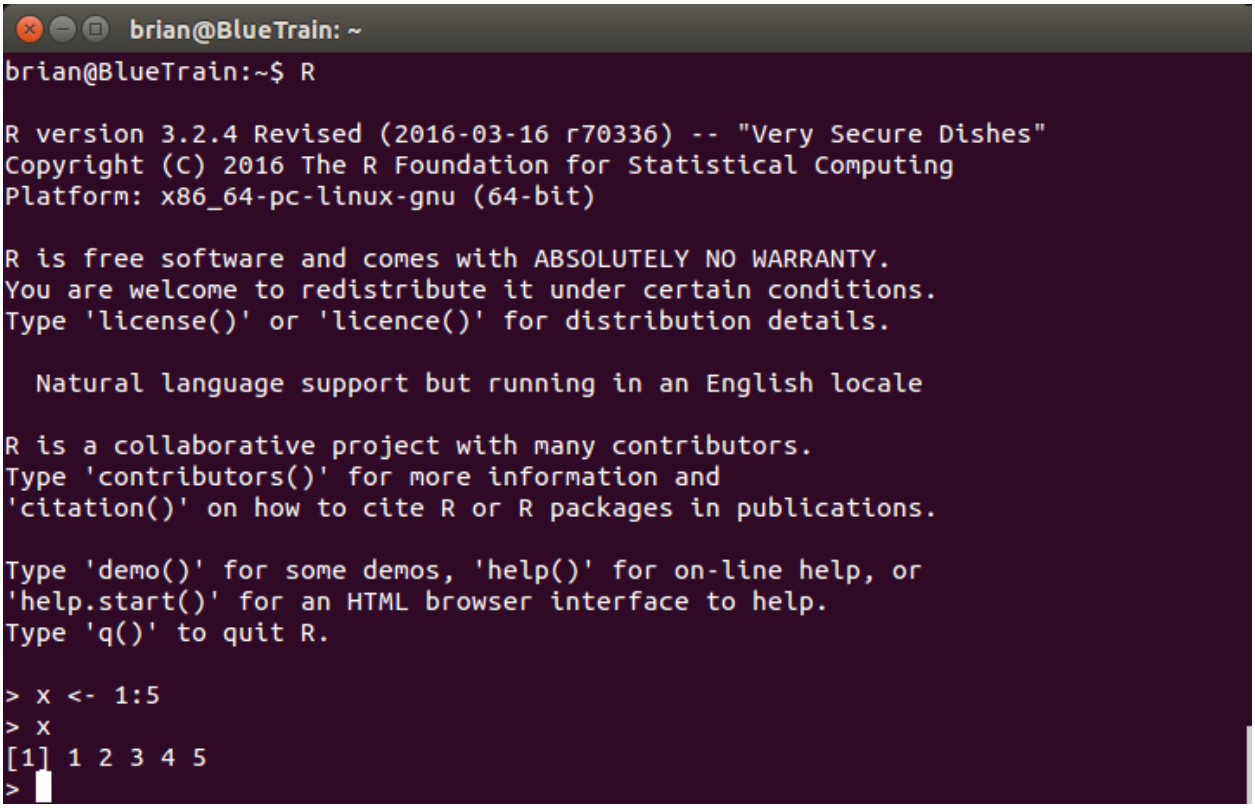


Figure 2.1: R at the command-line

2.2 Entering Commands

Now that you've got an environment, you're ready to go. That cursor is blinking and waiting for you to tell it what to do! So what's the first thing you'll accomplish?

In RStudio, the console may be reached by pressing CTRL-2 (Command-2 on Mac).

Well, not much. We'll get into more fun stuff soon, but for now let's play it safe. You can use R a basic calculator, so take a few minutes to enter some basic mathematical expressions.

```
1 + 1
#> [1] 2

pi
#> [1] 3.141593

2*pi*4^2
#> [1] 100.531
```

2.3 Your first script

Typing, editing and debugging at the command line will get tedious quickly.

A source file (file extension .R) contains a sequence of commands.

Analogous to the formulae entered in a spreadsheet (but so much more powerful!)

```
N <- 100
B0 <- 5
B1 <- 1.5

set.seed(1234)

e <- rnorm(N, mean = 0, sd = 1)
X1 <- rep(seq(1,10),10)

Y <- B0 + B1 * X1 + e

myFit <- lm(Y ~ X1)
```

Save this file.

CTRL-S on Windows/Linux, CMD-S on Mac.

2.3.1 Executing a script

Either:

1. Open the file and execute the lines one at a time, or
2. Use the "source" function.

```
source("SomefileName.R")
```

Within RStudio, you may also click the "Source" button in the upper right hand corner.

2.4 Getting help

```
?plot

??cluster
```

Within RStudio, the TAB key will autocomplete

2.5 The working directory

The source of much frustration when starting out.

Where am I?

```
getwd()
#> [1] "/home/brian/Projects/books/raw"
```

How do I get somewhere else?

```
setwd("~/SomeNewDirectory/SomeSubfolder")
```

Try to stick with relative pathnames. This makes work portable.

2.5.1 Directory paths

R prefers Unix style directories[2]. This means "/", **not** "\". Windows prefers "\". All things being equal, this isn't much of a big deal; it's just an arbitrary convention, like deciding that electricity flows from negative to positive rather than the other way around[3]. R is designed to be deployed on both Windows and Unix, so its internal functions will use whatever convention applies on the target operating system. However, there's a catch: "\" is an "escape" character, used for things like tabs and newline characters. To get a single slash, you need to type it twice.

[2]The Mac OS was based on Unix and adopts many of its conventions, among them file path separators.
[3]Benjamin Franklin reference here.

Chapter 3

Elements of the Language

There are certain concepts common to virtually all programming languages, which tell the computer how to behave. Those elements are: variables, functions and operators. This chapter will discuss what those are and how they're implemented in R. By the end of this chapter, you will be able to answer the following:

- What is a variable and how do I create and modify them?
- How do functions work?
- How can I use logic to control what commends get executed?

If you're familiar with other languages like Visual Basic, Python or Java Script, you may be tempted to skip this section. If you do, you'll survive, but I'd suggest giving it a quick read. You may learn something about how R differs from those other languages.

3.1 Variables

Programming languages work by assigning values to space in your computer's memory. Those values are then available for computation. Because the value of what's stored in memory may "vary", we call these things "variables". Think of a cell in a spreadsheet. Before we put something in it, it's just an empty box. We can fill it with whatever we like, be it a person's name, their birthdate, their age, whatever.

3.1.1 Assignment

Assignment will create a variable which contains a value. This value may be used later.

```
r <- 4

r + 2
#> [1] 6
```

Both "<-" and "=" will work for assignment.

3.2 Operators

3.2.1 Mathematical Operators

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Exponentiation

3.2.2 Logical Operators

Operator	Operation
&	and
|	or
!	not
==	equal
!=	not equal
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal
xor()	exclusive or
&&	non-vector and
||	non-vector or

3.3 Functions

Functions in R are very similar to functions in a spreadsheet. The function takes in arguments and returns a result.

```
sqrt(4)
#> [1] 2
```

Functions may be composed by having the output of one serve as the input to another.

$\sqrt{e^{sin\pi}}$

```
sqrt(exp(sin(pi)))
#> [1] 1
```

3.3.1 A few mathematical functions

```
?S3groupGeneric
```

- abs, sign
- floor, ceiling, trunc, round, signif
- sqrt, exp, log
- cos, sin, tan (and many others)
- lgamma, gamma, digamma, trigamma

3.4 Comments

R uses the hash/pound character "#" to indicate comments.

SQL or C++ style multiline comments are not supported.

Comment early and often!

Comments should describe "why", not "what".

3.4.1 Bad comment

```
# Take the ratio of loss to premium to determine the loss ratio

lossRatio <- Losses / Premium
```

3.4.2 Good comment

```
# Because this is a retrospective view of
# profitability, these losses have been
# developed, but not trended to a future
# point in time
lossRatio <- Losses / Premium
```

3.5 Exercises

- What is the area of a cylinder with radius = e and height = pi?
- What arguments are listed for the "plot" function?
- Find the help file for a generalized linear model
- Create a script which calculates the area of a cylinder. From a new script, assign the value 4 to a variable and source the other file. Assign the value 8 to your variable and source again. What happened?

Data

Yay, data! The good stuff! Well, pretty good, but we've got a lot to cover. Some of this may not make much sense on a first read. Don't get bogged down, press on and refer to this later if you feel you need to.

Chapter 4

Data types

To a human, the difference between something numeric- like a person's age- and something textual - like their name - isn't a big deal. To a computer, however, this matters a lot. In order to ensure that there is sufficient memory to store the information and to ensure that it may be used in an operation, the computer needs to know what type of data it's working with. In other words: 5 + "Steve" = Huh?

In this chapter, we'll talk through the various primitive data types that R supports. By the end of this chapter, you will be able to answer the following:

- What are the different data types?
- When and how is one type converted to another?
- How can I work with dates?
- What the heck is a factor?

4.1 Data types

R supports four "primitive" data types as shown below:

- logical
- integer
- double
- character

To know what type of data you're working with, you use the (wait for it) `typeof` function. If you want to test for a specific data type, you can use the suite of `is.` functions. Have a look at the example below. Note that when we want something to be an integer, we type the letter "L" after the number.

```
x <- 6
y <- 3L
z <- TRUE
typeof(x)
#> [1] "double"
typeof(y)
#> [1] "integer"
typeof(z)
#> [1] "logical"
is.logical(x)
#> [1] FALSE
```

```
is.double(x)
#> [1] TRUE
```

4.2 Data conversion

It's possible to convert from one type to another. Most of the time, this happens implicitly as part of an operation. R will alter data in order for calculations to take place. For example, let's say that I'm adding together `x` and `y` from the code snippet above. We know that an integer and a real number will add together easily, but the computer needs to convert the integer before the operation can take place.

```
typeof(x + y)
#> [1] "double"
```

Implicit conversion will change data types in the order shown below. Note that all data types for an operation will be converted to the most complex number involved in the calculation.

logical -> integer -> double -> character

Note that implicit conversion can't always help us. Let's try the example from the start of this chapter.

```
5 + 'Steve'
#> Error in 5 + "Steve": non-numeric argument to binary operator
```

Here, R is telling us that it doesn't know how to add a number and a word. I don't either.

For explicit conversion, use the `as.*` functions. When explicit conversion is used to convert a value to a simpler data type - double to integer, say - that there will likely be loss of information.

```
# Implicit conversion
w <- TRUE
x <- 4L
y <- 5.8
z <- w + x + y
typeof(z)
#> [1] "double"

# Explicit conversion. Note loss of data.
as.integer(z)
#> [1] 10
```

In addition to `typeof` there are two other functions which will return basic information about an object.

The `mode` of an object will return a value indicating how the object is meant to be stored. This will generally mirror the output produced by `typeof` except that double and integers both have a mode of "numeric". This function has never improved my life and it won't be discussed any further.

A `class` of an object is a very special kind of metadata. (We'll get more into metadata in the next chapter Vectors.) When we get beyond primitive data types, this starts to become important. We'll see two examples in just a moment when we talk about dates and factors. The class of a basic type will be equal to its type apart from 'double', whose class is 'numeric' for reasons I don't pretend to understand.

```
class(TRUE)
#> [1] "logical"
class(pi)
#> [1] "numeric"
class(4L)
#> [1] "integer"
```

Table 4.1: Key similarities and differences between vectors and lists

Function	Returns
typeof	The type of the object
mode	Storage mode of the object
class	The class(es) of the object
inherits	Whether the object is a particular class
is.	Whether the object is a particular type

```
class(Sys.Date())
#> [1] "Date"
```

The table below summarizes most of the ways we can sort out what sort of data we're working with.

4.3 Dates and times

Dates in R can be tricky. There are two basic classes: `Date` and `POSIXt`. The `Date` class does not get more granular than days. The `POSIXt` class can handle seconds, milliseconds, etc. My recommendation is to stick with the "Date" class. Introducing times means introducing time zones and the possibility for confusion or error. Actuaries rarely need to measure things in minutes.

```
x <- as.Date('2010-01-01')
class(x)
#> [1] "Date"
typeof(x)
#> [1] "double"
```

By default, dates don't follow US conventions. Much like avoiding the metric system, United Statesians are sticking with a convention that doesn't have a lot of logical support. If you want to preserve your sanity, stick with year, month, day order.

```
# Don't do this:
x <- as.Date('06-30-2010')
#> Error in charToDate(x): character string is not in a standard unambiguous format

# But this is just fine:
x <- as.Date('30-06-2010')

# Year, month, day is your friend
x <- as.Date('2010-06-30')
```

To get the date and time of the computer, use the either `Sys.Date()` or `Sys.time()`. Note that `Sys.time()` will return both the day AND the time as a POSIXct object.

```
x <- Sys.Date()
y <- Sys.time()
```

It's worth reading the documentation about dates. Measuring time periods is a common task for actuaries. It's easy to make huge mistakes by getting dates wrong.

The `lubridate` package has some nice convenience functions for setting month and day and reasoning about time periods. It also enables you to deal with time zones, leap days and leap seconds. This is probably more than most folks need, but it's worth looking into.

The `mondate` package was written by Daniel Murphy (an actuary) and supports handling time periods in terms of months. This is a very good thing. You'll quickly learn that the base functions don't like dealing with time periods as measured in months. Why? Because they're all different lengths. It's not clear how to add "one month" to a set of dates. And yet, we very often want to do this. An easy example is adding a set of months to the last day in a month. The close of a quarter is a common task in financial circles. The code below will produce the end of the quarter for a single year. [1]

```
library(mondate)
add(mondate("2010-03-31"), c(0, 3, 6, 9), units = "months")
#> mondate: timeunits="months"
#> [1] 2010-03-31 2010-06-30 2010-09-30 2010-12-31
```

The items below are all worth reading.

- Date class: https://stat.ethz.ch/R-manual/R-devel/library/base/html/Dates.html
- lubridate: http://www.jstatsoft.org/v40/i03/paper
- Ripley and Hornik: http://www.r-project.org/doc/Rnews/Rnews_2001-2.pdf
- mondate: (https://code.google.com/p/mondate/)

4.4 Factors

Factors are a pretty big gotcha. They were necessary many years ago when data collection and storage were expensive. A factor maps a character string to an integer, so that it takes up less space. The code below will illustrate the difference between a factor and a compararble character vector[2].

```
myColors <- c("Red", "Blue", "Green", "Red", "Blue", "Red")
myFactor <- factor(myColors)
myColors
#> [1] "Red"   "Blue"  "Green" "Red"   "Blue"  "Red"
myFactor
#> [1] Red   Blue  Green Red   Blue  Red
#> Levels: Blue Green Red
typeof(myFactor)
#> [1] "integer"
class(myFactor)
#> [1] "factor"
is.character(myFactor)
#> [1] FALSE
is.character(myColors)
#> [1] TRUE
```

Note that when we printed the value of `myFactor` we got the list of colors, but without the quotes around them. We are also told that our object has "Levels". This is important as it defines the set of possible values for the factor. This is rather useful if you have a data set where the permissible values are constrained to a closed set, like gender, education, smoker/non-smoker, etc.

So, what happens if we want to add a new element to our factor?

```
# This probably won't give you what you expect
myOtherFactor <- c(myFactor, "Orange")
myOtherFactor
#> [1] "3"      "1"      "2"      "3"      "1"      "3"      "Orange"
```

[1] You can also use the `quarter` function to achieve much the same thing.

[2] We haven't covered vectors yet, but we're getting there. If this code is confusing, just skip this section for now and come back after you've read up on vectors.

```
# And this will give you an error
myFactor[length(myFactor)+1] <- "Orange"
#> Warning in `[<-.factor`(`*tmp*`, length(myFactor) + 1, value = "Orange"):
#> invalid factor level, NA generated

# Must do things in two steps
myOtherFactor <- factor(c(levels(myFactor), "Orange"))
myOtherFactor[length(myOtherFactor)+1] <- "Orange"
```

Ugh. In the first instance, R recognizes that it can't append a new item to the factor. So, it converts the values to a string and then appends the string "Orange". But note that the items are string values of integers. That's because the underlying data of a factor *is* an integer. In the second instance, we first have to change the levels of the factor and then we can append our new data element.

Often when creating a data frame, R's default behavior is to convert character values into a factor. When we get to creating data frames and importing data, you'll often see us use code like the following:

```
mojo <- read.csv("myFile.csv", stringsAsFactors = FALSE)
```

Now that you know what they are, you can spend the next few months avoiding factors. When R was created, there were compelling reasons to include factors and they still have some utility. More often than not, though, they're a confusing hindrance. If characters aren't behaving the way you expect them to, check the variables with `class` or `is.factor`. Convert them with `as.character` and you'll be back on the road to happiness.

4.5 Exercises

- Create a logical, integer, double and character variable.
- Can you create a vector with both logical and character values?
- What happens when you try to add a logical to an integer? An integer to a double?

4.5.1 Answers

```
myLogical <- TRUE
myInteger <- 1:4
myDouble <- 3.14
myCharacter <- "Hello!"

y <- myLogical + myInteger
typeof(y)
#> [1] "integer"
y <- myInteger + myDouble
typeof(y)
#> [1] "double"
```

Chapter 5

Vectors

In this chapter, we're going to learn about vectors, which are one of the key building blocks of R programming. By the end of this chapter, you will know the following:

- What is a vector?
- How are vectors created?
- What is metadata?
- How can I summarize a vector?

5.1 What is a vector?

Enter a value at the console and hit enter. What do we see?

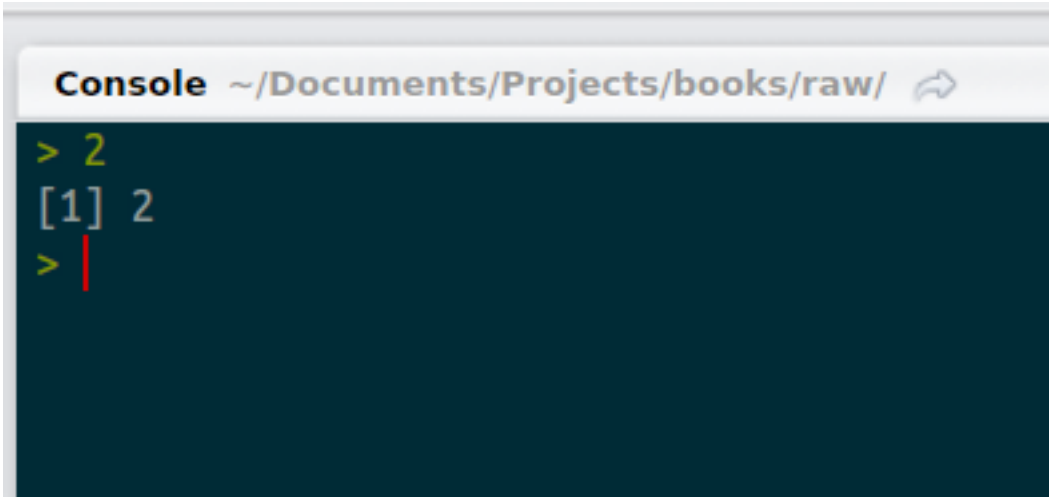


Figure 5.1: Console returning one value

By now, this should make sense. We entered 2 and we got back 2. But what's that 1 in brackets? Things get weirder when we ask R to return more than one value. Type "letters" (without the quotes) and have a look.

Now there's not only a 1 in brackets, there's also a 16 on the second line. (Note that your console may appear a bit different than mine.) You're clever and have probably figured out that the numbers in brackets have something to do with the amount of output generated. In the second case, "p" is the 16th letter of the

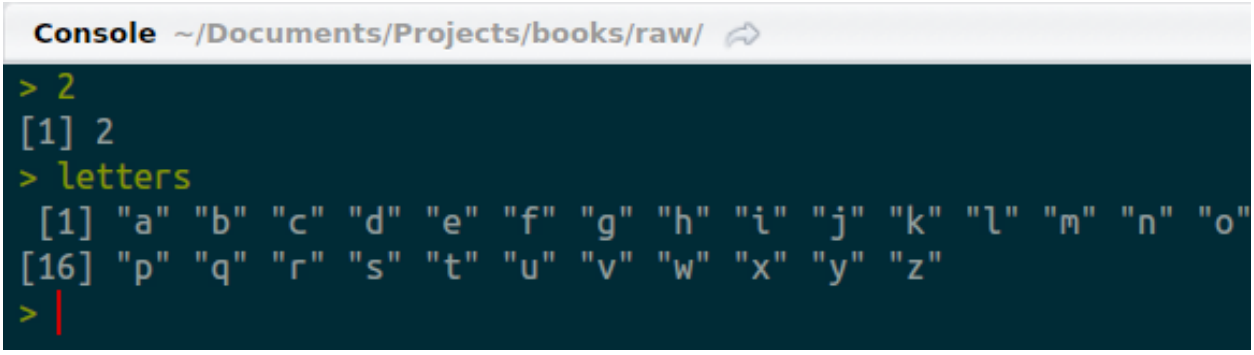


Figure 5.2: Console returning more than one value

alphabet and the bracketed 16 helps us know where we are in the sequence when it spills onto multiple lines. So, the bracketed figures are there to indicate how many numbers have been returned.

OK, cool. So what?

So what? So everything! In R, every variable is a vector. When we entered the number 2 at the console, we were creating (briefly) a vector which had a length of 1. The next console entry - "letters" - is a vector with 26 elements. Vectors can have many different values, but they're all associated with one thing. They allow us to reason about a lot of data at once. Let's say that again, because it's very important. **Vectors allow us to reason about a lot of data at once.** The translation of a large volume of data into fewer values, and/or simple decision rules is the essense of statistics. This is why R - which was designed by statisticians - place vectors front and center in how the language is constructed. Statisticians -and actuaries!- are accustomed to writing mathematical formulas. Our interactions with our calculation engines should exploit that.

OK, enough cheerleading. What is the practical benefit? Well, here's something for openers:

```
x <- 1:100
b <- 1.5
y <- x * b
```

To generate 100 new values, I just applied a multiplication operation in a single line. This is similar to applying the same function to a set of contiguous cells in a spreadsheet. However, in this case, I don't have to make 100 assignments to the variable `y`. I don't even need to worry about how many times the command needs to be repeated. Vectors can grow and shrink automatically. No need to move cells around on a sheet. No need to copy formulas or change named ranges. R just did it. Moreover, there's no chance that I'll miss a cell due to operator error.

5.1.1 Vector properties

So, how will I know a vector when I see one? For starters, it will have the following basic properties:

1. Every element in a vector must be the same basic data type. (Refer to Chapter 4: Data Types for a review of basic data types.)
2. Vectors are ordered.
3. Vectors have one dimension. Higher dimensions are possible via matrices and arrays.
4. Vectors may have metadata like names for each element.

A quick word about dimensionality: there are folks who will make a big fuss about the difference between a vector and a matrix and an array. Further, they might insist that there's a difference between a 100 x 1 matrix and a one dimensional vector. I'm not one of those people. To me, anything that forms a countable set of elements is an array. Splitting it into dimensions is just a matter of imposing some structure to the set. I can sort my bag of M&Ms by color, put them in rows, etc. But don't listen to me. When working with

R, you will almost invariably see a unidimensional set of data referred to as a "vector". Anything with two dimensions is a "matrix" and it contains the specific notion of "rows" and "columns". Anything of higher dimensions is an array.

You needn't spend too much time sweating over that last paragraph. I'm just getting across the idea that dimensionality is a bit arbitrary and will always be mutable. At this point, the key thing to bear in mind is that *all of the data is of the same type.*

5.2 Vector construction

So how do I create a vector? No real trick here. You'll be constructing vectors whether you want to or not. But let's talk about a few core functions for vector construction and manipulation. These are ones that you're likely to see and use often.

5.2.1 `seq`

The first functions we'll look at are `seq` and its first cousin the `:` operator[1]. You've already seen `:` where we've used it generate a sequence of integers. `seq` is much more flexible: you can specify the starting point, the ending point, the length of the interval and the number of elements to output. You'll need to provide at least three of the parameters and R will figure out everything else. We can think of the four use cases as being associated with which element we opt to leave out.

```
# Leave out the interval
someXs <- seq(from = 0, to = 1, length.out = 300)
# Leave out the ending point
pies = seq(from = 0, by = pi, length.out = 5)
# Leave out the length
someYs <- seq(from = 5, to = 15, by = 4)
# Leave out the start
someZ <- seq(to = 100, by = 3, length.out = 20)
```

5.2.2 `rep`

Whereas the `seq` function will generate unique values, the `rep` function will replicate its input. Tell it what you want repeated and how many times. Let's look at 100 pies:

```
i <- rep(pi, 100)
```

Note that the `rep` function also has an argument called `length.out`. This will cause the `times` argument to be ignored. Note that because you're replicating a vector, you might not get every value replicated the same number of times. That will only happen if `length.out` is a multiple of the input vector length. Have a go at the following line of code.

```
rep(1:4, length.out = 9)
#> [1] 1 2 3 4 1 2 3 4 1
```

There's nothing stopping us from combining functions ######

[1] Gotcha: `:` has a different meaning when constructing a formula. In that context, it refers to the interaction of variables.

5.2.3 c

For just a single letter, c packs quite a punch. The c is short for "concatenate" and you'll be happy about the reduced keystrokes. You'll be using this function a LOT. c will join two or more vectors into one vector. Remember the first basic property of vectors: every element must be the same type. If you try to concatenate vectors which have different data types, R will convert them to a common data type. For more on this, see the chapter on data types.

```
i <- c(1, 2, 3, 4, 5)
j <- c(6, 7, 8, 9, 10)
k <- c(i, j)
```

Watch what happens when we try to concatenate two vectors which have different data types:

```
i <- 1:5
i
#> [1] 1 2 3 4 5
j <- letters
j
#>  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
#> [18] "r" "s" "t" "u" "v" "w" "x" "y" "z"
k <- c(i, j)
k
#>  [1] "1" "2" "3" "4" "5" "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l"
#> [18] "m" "n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

5.2.4 paste

Another gotcha: in many other languages, "concatenation" is an operation which joins multiple strings to generate a single string. However, in R, the c function will concatenate multiple vectors (or other objects) into one object. So, how do you concatenate a string? With the paste function and its close relative paste0.

```
firstName <- "Brian"
lastName <- "Fannin"
fullName <- paste(lastName, firstName, sep = ", ")
```

paste0 is a shortcut for the common use case where we want to join characters together without anything separating them:

5.2.5 sample

The sample function will generate a random sample. This is a great one to use for randomizing a vector.

```
months <- c("January", "February", "March", "April"
            , "May", "June", "July", "August"
            , "September", "October", "November", "December")

set.seed(1234)
mixedMonths <- sample(months)
head(mixedMonths)
#> [1] "February" "July"     "November" "June"     "October"  "May"
```

By altering the size parameter and possibly setting the replace parameter to TRUE, we can get lots of values.

```
set.seed(1234)
lotsOfMonths <- sample(months, size = 100, replace = TRUE)
```

`sample` may also be used within the indexing of the vector itself:

```
set.seed(1234)
moreMonths <- months[sample(1:12, replace=TRUE, size=100)]
head(moreMonths)
#> [1] "February" "August"   "August"   "August"   "November" "August"

# Cleaner with sample.int
set.seed(1234)
evenMoreMonths <- months[sample.int(length(months), size=100, replace=TRUE)]
head(evenMoreMonths)
#> [1] "February" "August"   "August"   "August"   "November" "August"
```

5.2.6 sort

Remember how I said that the elements of a vector have an order? Well, they do. Although this may seem obvious, this is not a property that's observed in other data constructs like a relational database. In a system like Oracle or SQL Server, records aren't stored in order. Further, there's no guarantee that when you run a `SELECT` statement that you'll get rows in the back in the same order when you run the same statement a second time[2]. That's done to optimize insertion and deletion in fixed storage. For R, all of your data is in RAM, so this is not as important. So, we can sort our vector and know that this arrangement of elements won't change until we say so.

```
set.seed(1234)
i <- sample(1:5)
i
#> [1] 1 3 2 4 5
sort(i)
#> [1] 1 2 3 4 5
```

5.2.7 vector

The `vector` function will create a vector. Sounds pretty fundamental. So why didn't I show this as the first method to generate a new vector? Because you probably won't use it very often. The `vector` function will create a new, empty vector with whatever data type and length you'd like. Until you fill it with data, this has fairly limited utility. The strongest use case for `vector` is to preallocate storage in memory. This is a performance issue related to the way that R handles the modification of data objects. The subject is beyond the scope of this text, so I'll refer you to [Wic].

```
x <- vector(mode = "numeric", length = 10)
```

5.2.8 Recycling

R will "recycle" vectors until there are enough elements to perform an operation. Everything gets as "long" as the longest vector in the operation. For scalar operations on a vector this doesn't involve any drama. Try the following code:

[2]This might not be true if the table is indexed. However, in general, a table's index is stored on different physical space from the associated table. The table itself remains unordered.

```
vector1 = 1:10
vector2 = 1:5
scalar = 3

print(vector1 + scalar)
#>  [1]  4  5  6  7  8  9 10 11 12 13
print(vector2 + scalar)
#> [1] 4 5 6 7 8
print(vector1 + vector2)
#>  [1]  2  4  6  8 10  7  9 11 13 15
```

5.3 Vector access

Vector access is something that we'll be doing all the time. Here, we're subsetting the elements in our vector to get something useful. This is critical when we want to isolate bits of our data, either for analysis or to emphasize particularly important points.

Vectors may be accessed in one of two ways: by position[3], or via logical subsetting. In the first case, we're asking R to return the elements at particular positions. In the second, we form a vector of logical values of the same length of the vector we're accessing.

5.3.1 `head/tail`

The `head` and `tail` functions will return the first or last few elements of a vector. They're

```
set.seed(1234)
e <- rnorm(100)
e[1]
#> [1] -1.207066
e[1:4]
#> [1] -1.2070657  0.2774292  1.0844412 -2.3456977
e[c(1,3)]
#> [1] -1.207066  1.084441
```

5.3.2 Vector access - logical access

Vectors may be accessed logically. This may be done by passing in a logical vector, or a logical expression.

```
i = 5:9
i[c(TRUE, FALSE, FALSE, FALSE, TRUE)]
#> [1] 5 9
i[i > 7]
#> [1] 8 9
b = i > 7
b
#> [1] FALSE FALSE FALSE  TRUE  TRUE
i[b]
#> [1] 8 9
```

[3] I told you vectors were ordered.

5.3.3 which

The `which` function returns indices that match a logical expression.

```
i <- 11:20
which(i > 15)
#> [1] 6 7 8 9 10
i[which(i > 15)]
#> [1] 16 17 18 19 20
```

As with other functions that return indices, remember that you can store the indices in another variable, or use the indices to extract elements from a different vector.

5.3.4 order

I've put `order` here as it feels like a close companion to `which` and the idea of ordinal vector access. The `order` function will return the indices of the vector in order. This is a key difference from `sort` which alter the contents of the vector itself.

```
set.seed(1234)
x <- sample(1:5)
order(x)
#> [1] 1 3 2 4 5
y <- 3 * x
y[order(x)]
#> [1] 3 6 9 12 15
```

Note that changing the `decreasing` parameter from FALSE to TRUE will change the sort order.

5.4 Set theory

Vectors adhere to all the set theory operations that you would expect.

```
x <- 1:5
y <- 1:10

union(x, y)
#> [1] 1 2 3 4 5 6 7 8 9 10
intersect(x, y)
#> [1] 1 2 3 4 5

setdiff(x, y)
#> integer(0)
setdiff(y, x)
#> [1] 6 7 8 9 10

is.element(4, x)
#> [1] TRUE
is.element(11, x)
#> [1] FALSE
is.element(x, y)
#> [1] TRUE TRUE TRUE TRUE TRUE

setequal(1:5, sample(1:5))
```

```
#> [1] TRUE
identical(1:5, sample(1:5))
#> [1] FALSE
```

The `%in%` operator will return a logical vector indicating whether or not an element of the first set is contained in the second set.

```
x <- 1:10
y <- 5:20
x %in% y
#>  [1] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
is.element(x, y)
#>  [1] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

5.5 Assignment

Assignment works the same as access, but in the opposite direction. Here, we're not extracting a subset of a vector, we're

5.5.1 Growth by assignment

A Assigning a value beyond a vector's limits will automatically grow the vector. Interim values are assigned `NA`.

```
i <- 1:10
i[30] = pi
i
#>  [1]  1.000000  2.000000  3.000000  4.000000  5.000000  6.000000  7.000000
#>  [8]  8.000000  9.000000 10.000000        NA        NA        NA        NA
#> [15]        NA        NA        NA        NA        NA        NA        NA
#> [22]        NA        NA        NA        NA        NA        NA        NA
#> [29]        NA  3.141593
```

5.6 Metadata

Metadata is data about data. Simple vectors don't have much need for metadata, but there is at least one property - `name` - that you'll see often.

```
i <- 1:4
names(i) <- letters[1:4]
i
#> a b c d
#> 1 2 3 4
```

In R, most metadata is set and accessed by the `attr` function and metadata are called "attributes". Some pre-defined attributes like `names` have their own assignment and reference function. Those are: `class`, `comment`, `dim`, `dimnames`, `row.names` and `tsp`. We'll hear more about `dim`, `dimnames` and `row.names`, as well as a few other pre-defined properties, when we talk about matrices later.

In general, an attribute is set using the `<-` assignment operator along with the function name. The attribute is referenced by simply typing the function.

```
comment(i) <- "This is a comment."
comment(i)
#> [1] "This is a comment."
j <- 5:8
comment(j)
#> NULL
```

```
attributes(i)
#> $names
#> [1] "a" "b" "c" "d"
#>
#> $comment
#> [1] "This is a comment."
```

5.6.1 `length`, `dim`

`length` will return the number of elements in a vector.

5.7 Summarization

There are loads of functions which take vector input and return scalar output. Translation of a large set of numbers into a few, informative values is one of the cornerstones of statistics.

```
x = 1:50
sum(x)
mean(x)
max(x)
length(x)
var(x)
```

5.8 Matrices

A matrix is a vector with higher dimensions. In R, when we talk about a matrix, we will always mean something with two dimensions. A matrix may be constructed in two ways:

1. Use the `matrix` function.
2. Change the dimensions of a `vector`.

Note

```
myVector <-
myMatrix <- matrix(1:100, nrow=10, ncol=10)

myOtherMatrix <- myVector
dim(myOtherMatrix) <- c(10,10)

identical(myMatrix, myOtherMatrix)
#> [1] TRUE
```

```
myMatrix <- matrix(nrow=10, ncol=10)
```

5.8.1

```
dim(myMatrix) <- c(25, 4)
```

5.8.2 Matrix metadata

Possible to add metadata. This is typically a name for the columns or rows.

```
myMatrix <- matrix(nrow=10, ncol=10, data = sample(1:100))
colnames(myMatrix) <- letters[1:10]
head(myMatrix, 3)
#>       a  b  c  d  e  f  g  h  i  j
#> [1,] 84 73 61 36 95 70 78 98 11 66
#> [2,] 29 47 16 89 30 79 88 37 72 64
#> [3,] 27 81 21 33 44 25 69  9 43 28
rownames(myMatrix) <- tail(letters, 10)
head(myMatrix, 3)
#>    a  b  c  d  e  f  g  h  i  j
#> q 84 73 61 36 95 70 78 98 11 66
#> r 29 47 16 89 30 79 88 37 72 64
#> s 27 81 21 33 44 25 69  9 43 28
```

5.8.3 Matrix access

Matrix access is similar to vector, but with additional dimensions. For two-dimensional matrices, the order is row first, then column.

```
myMatrix[2, ]
#>  a  b  c  d  e  f  g  h  i  j
#> 29 47 16 89 30 79 88 37 72 64
myMatrix[, 2]
#>  q  r  s  t  u  v  w  x  y  z
#> 73 47 81 90 92 39 96 26 42 93
```

Single index will return values by indexing along only one dimension.

```
myMatrix[2]
#> [1] 29
myMatrix[22]
#> [1] 16
```

5.8.4 Matrix summarizatioon

```
sum(myMatrix)
#> [1] 5050
colSums(myMatrix)
#>   a   b   c   d   e   f   g   h   i   j
#> 351 679 536 410 463 563 520 473 548 507
rowSums(myMatrix)
#>   q   r   s   t   u   v   w   x   y   z
#> 672 551 380 538 690 489 494 548 356 332
colMeans(myMatrix)
#>    a    b    c    d    e    f    g    h    i    j
#> 35.1 67.9 53.6 41.0 46.3 56.3 52.0 47.3 54.8 50.7
```

5.9 Arrays

An array has more than two dimensions. Do you like data with more than two dimensions? Shine on you crazy diamond. In a geometric space, I can hang with three dimensions, but in other contexts and in higher dimensions, I lose cognition pretty quick. In those cases, I often find my data is easier to express as a data frame, which we'll get to. In the meantime, here's ordinal access for any mathechists who love using 3 or more sets of natural numbers to reference data elements.

```
myArray <- 1:100
dim(myArray) <- c(10, 2, 5)

myArray[1, 2, 3]
#> [1] 51
```

Sorting out why that's 51 hurts my brain.

5.10 Exercises

1. Create a vector of length 10, with years starting from 1980.
2. Create a vector with values from 1972 to 2012 in increments of four (1972, 1976, 1980, etc.)

For the next few questions, use the following vectors:

```
FirstName <- c("Richard", "James", "Ronald", "Ronald"
               , "George", "William", "William", "George"
               , "George", "Barack", "Barack")
LastName <- c("Nixon", "Carter", "Reagan", "Reagan"
              , "Bush", "Clinton", "Clinton", "Bush"
              , "Bush", "Obama", "Obama")
ElectionYear <- seq(1972, 2012, 4)
```

3. List the last names in alphabetical order
4. List the years in order by first name.
5. Create a vector of years when someone named "George" was elected.
6. How many Georges were elected before 1996?
7. Generate a random sample of 100 presidents.

5.11 Answers

```
LastName[order(LastName)]
#>  [1] "Bush"    "Bush"    "Bush"    "Carter"  "Clinton" "Clinton" "Nixon"
#>  [8] "Obama"   "Obama"   "Reagan"  "Reagan"
ElectionYear[order(FirstName)]
#>  [1] 2008 2012 1988 2000 2004 1976 1972 1980 1984 1992 1996
ElectionYear[FirstName == 'George']
#> [1] 1988 2000 2004
myLogical <- (FirstName == 'George') & (ElectionYear < 1996)
length(which(myLogical))
#> [1] 1
sum(myLogical)
#> [1] 1
```

```
sample(LastName, 100, replace = TRUE)
#>   [1] "Obama"   "Obama"   "Reagan"  "Carter"  "Bush"    "Bush"    "Obama"
#>   [8] "Obama"   "Obama"   "Clinton" "Reagan"  "Reagan"  "Clinton" "Clinton"
#>  [15] "Reagan"  "Obama"   "Clinton" "Carter"  "Bush"    "Obama"   "Clinton"
#>  [22] "Obama"   "Clinton" "Clinton" "Obama"   "Clinton" "Obama"   "Reagan"
#>  [29] "Clinton" "Reagan"  "Clinton" "Bush"    "Clinton" "Obama"   "Clinton"
#>  [36] "Bush"    "Reagan"  "Nixon"   "Obama"   "Reagan"  "Obama"   "Clinton"
#>  [43] "Obama"   "Bush"    "Clinton" "Bush"    "Clinton" "Bush"    "Reagan"
#>  [50] "Nixon"   "Bush"    "Bush"    "Nixon"   "Bush"    "Reagan"  "Bush"
#>  [57] "Clinton" "Bush"    "Bush"    "Reagan"  "Bush"    "Bush"    "Clinton"
#>  [64] "Carter"  "Reagan"  "Clinton" "Reagan"  "Clinton" "Nixon"   "Obama"
#>  [71] "Nixon"   "Obama"   "Clinton" "Reagan"  "Bush"    "Bush"    "Obama"
#>  [78] "Carter"  "Obama"   "Bush"    "Obama"   "Obama"   "Bush"    "Obama"
#>  [85] "Bush"    "Bush"    "Clinton" "Reagan"  "Bush"    "Clinton" "Bush"
#>  [92] "Reagan"  "Bush"    "Reagan"  "Obama"   "Clinton" "Reagan"  "Reagan"
#>  [99] "Bush"    "Clinton"
```

5.12 Conclusion

Vectors are like atoms. If you understand vectors- how to create them, how to manipulate them, how to access the elements, you're well on your way to grasping how to handle other objects in R. Vectors may combine to form molecules, but fundamentally, *everything* in R is a vector.

Having sorted out vectors, we're next going to turn out attention to lists. Lists are similar to vectors in that they enable us to bundle large amounts of data in a single construct. However, they're far more flexible and require a bit more thought.

Chapter 6

Lists

In this chapter, we're going to learn about lists. Lists can be a bit confusing the first time you begin to use them. Heaven knows it took me ages to get comfortable with them. However, they're a very powerful way to structure data and, once mastered, will give you all kinds of control over pretty much anything the world can throw at you. If vectors are R's atoms, lists are molecules.

By the end of this chapter, you will know the following:

- What is a list and how are they created?
- What is the difference between a list and vector?
- When and how do I use `lapply`?

6.1 Lists Overview

A list is a bit like a vector in that it is a container for many elements of data. However, unlike a vector, the elements of a list may have different data types. In addition, lists may store data *recursively*. This means that a list may contain list, which contains another list and so on. Partially for this reason, access and assignment will use two new operators: `[[]]` and `$`. Confusing? Sorta. But don't worry, we'll walk through why and how you'll work with lists.

The table below outlines some of the similarities and differences between vectors and lists.

6.2 List construction

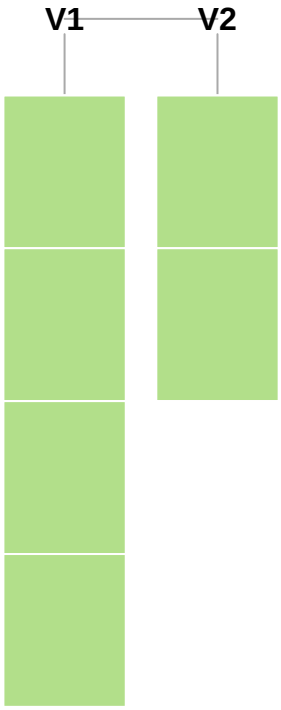
The `list` function will create a list. Have a look at the code below and try it out yourself.

Table 6.1: Key similarities and differences between vectors and lists

Vectors	Lists
Ordered	Also ordered
All elements have the same data type	Elements may be any data type, even other lists
One dimension	Doesn't apply in this context
Access and assignment via []	Access and assignment via [[ ]] and $
May contain metadata	May contain metadata

```
x <- list(c("This", "is", "a", "list")
          , c(pi, exp(1)))
typeof(x)
#> [1] "list"
summary(x)
#>      Length Class  Mode
#> [1,] 4      -none- character
#> [2,] 2      -none- numeric
str(x)
#> List of 2
#>  $ : chr [1:4] "This" "is" "a" "list"
#>  $ : num [1:2] 3.14 2.72
```

Visually, here's what that looks like:



We said earlier that the concept of dimension doesn't really apply to lists. We mean it and so does R. If you ask for the dimension of a list, you'll get `NULL`. I'm not mathy enough to have anything intelligent to say abouut the dimensionality of a single construct composed of elements which each have their own dimension.

Note that you *can* ask for the length. This will return the number of top-most elements in the list.

```
dim(x)
#> NULL
length(x)
#> [1] 2
```

Note that, weirdly - and a little confusingly - you can create a list by using the `vector` function.

```
myList <- vector(mode = "list", length = 5)
```

As we said in the Vectors chapter, you'll likely only do this to improve memory management performance. Don't worry about that at this stage. However, if you'd like to play with access and assignment, `vector` may be useful.

6.2.1 Recursive storage

Lists can contain other lists as elements. And these lists may contain other lists and so on. It sounds complex, but it's no stranger than your file system, e.g. a folder contains a folder, which contains another folder.

```
trey <- list("Trey Anastasio", "guitar", 1964)
page <- list("Page McConnell", "piano", 1963)
jon <- list("Jon Fishman", "drums", 1965)
mike <- list("Mike Gordon", "bass", 1965)
phish_members <- list(trey, page, jon, mike)
phish_albums <- c("Junta", "Rift", "Hoist")
phish <- list(phish_members, phish_albums)
```

This will be familiar to anyone who uses XML, JSON or YAML.

6.2.2 List metadata

Again, metadata will typically be names. However, these become very important for lists as names are handled with the special `$` operator. We'll talk about `$` shortly. We can also assign a name as the list is being constructed.

```
names(phish) <- c("Members", "Albums")
phish <- list(Members = phish_members, Albums = phish_albums)
```

6.3 Access and assignment

Because list elements can be arbitrarily complex, access and assignment get new operators. We'll use the `[[ ]]` operator when we want to access a single element of a list by name or position. `$` will work for named arguments only.

```
phish[["Albums"]]
#> [1] "Junta" "Rift"  "Hoist"
phish[[2]]
#> [1] "Junta" "Rift"  "Hoist"
phish$Albums
#> [1] "Junta" "Rift"  "Hoist"
```

6.3.1 [vs. [[

`[ ]` may also be used to access elements of a list. When first learning R, I found the distinction betwee `[ ]` and `[[ ]]` particularly vexing. After using lists for a while, I finally decided that this was the best way to distinguish between them:

- `[` is used to set and return an element of the same type as the *containing* object.
- `[[` is used to set and return an element of the same type as the *contained* object.

This is why `[ ]` will return a list when applied to a list. This is also why it may be used to return more than one element of list. Have a look at the code snippet below:

```
typeof(phish["Albums"])
#> [1] "list"
typeof(phish[["Albums"]])
#> [1] "character"
```

Don't worry if this doesn't make sense yet. It's difficult for most R programmers.

6.3.2 Assignment

As with vectors, assignment may be thought of as access in reverse.

```
phish[["Albums"]] <- c("Lawn Boy", "A Picture of Nectar")
```

6.4 Summary functions

Because lists are arbitrary, we can't expect functions like `sum` or `mean` to work. Instead, we use functions like `lapply` to summarize particular list elements. `lapply` will apply the same function to each element of a list. In the example below, we'll generate some statistics for three different vectors stored in a list.

```
myList <- list(firstVector = c(1:10)
               , secondVector = c(89, 56, 84, 298, 56)
               , thirdVector = c(7,3,5,6,2,4,2))
lapply(myList, mean)
#> $firstVector
#> [1] 5.5
#>
#> $secondVector
#> [1] 116.6
#>
#> $thirdVector
#> [1] 4.142857
lapply(myList, median)
#> $firstVector
#> [1] 5.5
#>
#> $secondVector
#> [1] 84
#>
#> $thirdVector
#> [1] 4
lapply(myList, sum)
#> $firstVector
#> [1] 55
#>
#> $secondVector
#> [1] 583
#>
#> $thirdVector
#> [1] 29
```

Why `lapply`? Two reasons:

1. It's expressive. A loop is a lot of code which does little to clarify intent. `lapply` indicates that we want to apply the same function to each element of a list and it does it in only one line of code. Think of a formula that exists as a column in a spreadsheet.
2. It's easier to type at an interactive console. In its very early days, `S` was fully interactive. Typing a `for` loop at the console is a tedius and unnecessary task.

Note that we can also use `lapply` on structures like a vector.

6.5 Exercises

- Create a list with two elements. Have the first element be a vector with 100 numbers. Have the second element be a vector with 100 dates. Give your list the names: "Claim" and "AccidentDate".
- What is the average value of a claim?

6.6 Answers

```
myList <- list()
myList$Claims <- rlnorm(100, log(10000))
myList$AccidentDate <- sample(seq.Date(as.Date('2000-01-01'), as.Date('2009-12-31'), length.out = 1000)
mean(myList$Claims)
#> [1] 16349.04
```

Chapter 7

Data Frames

This is the big one! All of that stuff about vectors and lists was prologue to this. The data frame is a seminal concept in R. Most statistical and predictive models expect one and they are the most common way to pass data in and out of R. Although critical to understand, data frames are very, very easy to get. What's a data frame? It's a table. That's it. No, really, that's it.

By the end of this chapter you will know the following:

- What's a data frame and how do I create one?
- How can I access and assign elements to and from a data frame?
- How do I read and write external data?

7.1 What's a data frame?

Underneath the hood, a data frame is actually a mashup of lists and vectors. Every data frame is a list, but it's constrained so that each list element is a vector with the same length. We can see how this exploits some of the fundamental properties of lists and vectors. Because each vector must have the same data type, there's no danger that I'll get character data when I only want dates or integers or whatever. At the same time, the list's flexibility means that we can store different data types in a single data construct.

7.1.1 Creating a data frame

Although there are many functions that will return a data frame, let's construct one from scratch with the `data.frame` function. We'll first create some vectors and then join them together.

```
set.seed(1234)
State <- rep(c("TX", "NY", "CA"), 10)
EarnedPremium <- rlnorm(length(State), meanlog = log(50000), sdlog=1)
Loss <- rlnorm(length(State), meanlog = log(50000 * 0.6), sdlog=1)

df <- data.frame(State, EarnedPremium, Loss, stringsAsFactors=FALSE)
```

Note that I've set the "stringsAsFactors" argument to FALSE. If I hadn't, the column "State" would be a factor, rather than a character. See Data Types: factors for some reasons why we might not want our data to be a factor.

We didn't have to create the vectors before constructing the data frame. If we like, we can pass them in as the result of function calls within the call to `data.frame`.

```
set.seed(1234)
df <- data.frame(State = rep(c("TX", "NY", "CA"), 10)
                 , EarnedPremium = round(rlnorm(length(State), meanlog = log(50000), sdlog=1), 3)
                 , stringsAsFactors = FALSE)
```

When constructing a data frame, there will always be some implicit metadata. For instance, every column must be named; if you don't provide one, names will be provided for you. R will use variable names by default. If there aren't any to be found, you'll get the ugliness you see in the second example below.

```
df <- data.frame(State, EarnedPremium)
names(df)
#> [1] "State"         "EarnedPremium"
df <- data.frame(rep(c("TX", "NY", "CA"), 10)
                 , round(rlnorm(length(State), meanlog = log(50000), sdlog=1), 3))
names(df)
#> [1] "rep.c..TX....NY....CA....10."
#> [2] "round.rlnorm.length.State...meanlog...log.50000...sdlog...1..."
```

Data frames will also have dimensional attributes set automatically.

```
dim(df)
#> [1] 30  2
```

This example is small, but it doesn't have to be. We could have a lot more rows and lot more columns. I lose focus beyond about 50 columns (i.e. once we've moved past column "BA" in a spreadsheet), but I've worked with data frames having more than one million rows on a number of occasions. Is that big data? I couldn't say, but I'm not all that fussed about whether my data is "big" or not. The basic concept of a data frame scales incredibly well. I'll use the same code to construct a model whether my data has 10 observations or 10 million.

7.1.2 Basic properties of a data frame

Once created, it's straightforward to get some basic information about the data we have.

```
summary(df)
#>  rep.c..TX....NY....CA....10.
#>  CA:10
#>  NY:10
#>  TX:10
#>
#>
#>
#>  round.rlnorm.length.State...meanlog...log.50000...sdlog...1...
#>  Min.   :  5652
#>  1st Qu.: 16503
#>  Median : 23835
#>  Mean   : 50519
#>  3rd Qu.: 35784
#>  Max.   :259781
str(df)
#> 'data.frame':    30 obs. of  2 variables:
#>  $ rep.c..TX....NY....CA....10.                                  : Factor w/ 3 levels "CA","NY","TX"
#>  $ round.rlnorm.length.State...meanlog...log.50000...sdlog...1...: num  150554 31076 24596 30288 980
head(df)
#>   rep.c..TX....NY....CA....10.
```

```
#> 1                            TX
#> 2                            NY
#> 3                            CA
#> 4                            TX
#> 5                            NY
#> 6                            CA
#>   round.rlnorm.length.State...meanlog...log.50000...sdlog...1...
#> 1                                                     150553.808
#> 2                                                      31075.817
#> 3                                                      24595.979
#> 4                                                      30288.404
#> 5                                                       9805.364
#> 6                                                      15555.336
tail(df)
#>    rep.c..TX....NY....CA....10.
#> 25                           TX
#> 26                           NY
#> 27                           CA
#> 28                           TX
#> 29                           NY
#> 30                           CA
#>    round.rlnorm.length.State...meanlog...log.50000...sdlog...1...
#> 25                                                      42508.90
#> 26                                                      87801.52
#> 27                                                     259781.39
#> 28                                                      23073.15
#> 29                                                     249119.48
#> 30                                                      15708.70
```

We can also query metadata about our data frame. Note that, for a data frame, `names` and `colnames` will return the same result. `dim` will give the number of rows and columns, or you can use `nrow` and `ncol` directly. In particular, note what result is returned by the `length` function. If you think about the fact that a data frame is actually a list, you may be able to guess why `length` returns the value it does.

```
names(df)
#> [1] "rep.c..TX....NY....CA....10."
#> [2] "round.rlnorm.length.State...meanlog...log.50000...sdlog...1..."
colnames(df)
#> [1] "rep.c..TX....NY....CA....10."
#> [2] "round.rlnorm.length.State...meanlog...log.50000...sdlog...1..."
dim(df)
#> [1] 30  2
length(df)
#> [1] 2
nrow(df)
#> [1] 30
ncol(df)
#> [1] 2
```

7.1.3 Combining data frames

If you've got more than one data frame, it's possible to combine them in several different ways: `rbind`, `cbind` and `merge`.

`rbind` will append rows to the data frame. New rows must have the same number of columns and data types.

```
dfA = df[1:10,]
dfB = df[11:20, ]
rbind(dfA, dfB)
```

`cbind` must have the same number of rows as the data frame.

```
dfC = dfA[, 1:2]
cbind(dfA, dfC)
#>    rep.c..TX....NY....CA....10.
#> 1                            TX
#> 2                            NY
#> 3                            CA
#> 4                            TX
#> 5                            NY
#> 6                            CA
#> 7                            TX
#> 8                            NY
#> 9                            CA
#> 10                           TX
#>    round.rlnorm.length.State...meanlog...log.50000...sdlog...1...
#> 1                                                       150553.808
#> 2                                                        31075.817
#> 3                                                        24595.979
#> 4                                                        30288.404
#> 5                                                         9805.364
#> 6                                                        15555.336
#> 7                                                         5651.852
#> 8                                                        13079.287
#> 9                                                        37252.876
#> 10                                                       31378.579
#>    rep.c..TX....NY....CA....10.
#> 1                            TX
#> 2                            NY
#> 3                            CA
#> 4                            TX
#> 5                            NY
#> 6                            CA
#> 7                            TX
#> 8                            NY
#> 9                            CA
#> 10                           TX
#>    round.rlnorm.length.State...meanlog...log.50000...sdlog...1...
#> 1                                                       150553.808
#> 2                                                        31075.817
#> 3                                                        24595.979
#> 4                                                        30288.404
#> 5                                                         9805.364
#> 6                                                        15555.336
#> 7                                                         5651.852
#> 8                                                        13079.287
#> 9                                                        37252.876
#> 10                                                       31378.579
```

`merge` is similar to a **JOIN** operation in a relational database. If you're used to **VLOOKUP** in Excel[1], you'll love `merge`. It's possible to use multiple columns (e.g. state and effective date) when specifying how to join. If no columns are specified, `merge` will use whatever column names the two data frames have in common. Below, we merge a set of rate changes to our premium data.

```
dfRateChange = data.frame(State = c("TX", "CA", "NY"), RateChange = c(.05, -.1, .2))
df = merge(df, dfRateChange)
```

7.1.4 expand.grid

Consider `expand.grid`. This will create a data frame as the cartesian product (i.e. every combination of elements) of one or more vectors. Among the use cases are to check for missing data elements in another data frame.

```
dfStateClass <- expand.grid(State = c("TX", "CA", "NY")
                  , Class = c(1776, 1066, 1492))
```

7.2 Access and Assignment

Access and assignment will feel like a weird combination of matrices and lists, though with an emphasis on the mechanics of a 2D matrix. We'll often use the `[ ]` operator to specify which row and column we'd like. The first arugment will refer to the row and the second will refer to the column. If either argument is left blank, it will refer to every element.

```
df[1, 2]
df[2, ]
df[, 2]
df[2, -1]
```

Note the interesting case when we specify one argument but leave the other blank. I'll wrap it in a `head` function to minimize the output.

```
head(df[2])
#>   round.rlnorm.length.State...meanlog...log.50000...sdlog...1...
#> 1                                                      150553.808
#> 2                                                       31075.817
#> 3                                                       24595.979
#> 4                                                       30288.404
#> 5                                                        9805.364
#> 6                                                       15555.336
```

Once again, if you bear in mind that a data frame is a list, this should makes sense. Without the additional comma, R will assume that we're dealing with a list and will respond accordingly. In this case, that means returning the second element of the list. You can extend this by requesting multiple elements of the list.

```
head(df[1:2])
#>   rep.c..TX....NY....CA....10.
#> 1                           TX
#> 2                           NY
#> 3                           CA
#> 4                           TX
#> 5                           NY
#> 6                           CA
```

[1] If you've never used VLOOKUP, I can't imagine why you're reading this.

```
#>   round.rlnorm.length.State...meanlog...log.50000...sdlog...1...
#> 1                                                      150553.808
#> 2                                                       31075.817
#> 3                                                       24595.979
#> 4                                                       30288.404
#> 5                                                        9805.364
#> 6                                                       15555.336
```

It'll probably be rare that you want this sort of behavior. There are many times that you'll want to return a single vector from a data frame, but in these cases, I'd recommend either using the `$` or the `[[ ]]` operator. It's common to use `$` to return a single column from a data frame. This will force you to use the name of the column and so provides a more expressive way to work with your data.

```
df$EarnedPremium
head(df[[1]])
head(df[["EarnedPremium"]])
```

For multiple columns, we can pass a character vector to get columns by name, rather than by position.

```
head(df[, "EarnedPremium"])
#> [1]  14953.677  65986.637 147889.324   4789.018  76795.627  82936.802
head(df[, c("EarnedPremium", "State")])
#>   EarnedPremium State
#> 1     14953.677    TX
#> 2     65986.637    NY
#> 3    147889.324    CA
#> 4      4789.018    TX
#> 5     76795.627    NY
#> 6     82936.802    CA
```

7.2.1 Subsetting

Remember that we can use the same logical and ordinal access functions which apply to vectors to access a data frame.

```
dfTX = df[df$State == "TX", ]
dfBigPolicies = df[df$EarnedPremium >= 50000, ]

whichState = (df$State == "TX")
dfTX = df[whichState, ]

df10 <- df[1:10, ]
```

If I'm only looking to filter rows, an easier way would be to use the **subset** function.

```
dfTX = subset(df, State == "TX")
dfBigPolicies = subset(df, EarnedPremium >= 50000)
```

Bringing that all together, we can combine row and colum access in crazy ways. Here, we'll get the premium and state for the rows which have losses greater than some value. Let's say we want everything above the 80th percentile, but rounded to the nearest thousand.

```
minLoss <- round(quantile(df$Loss, 0.8), -3)
myDF <- df[df$Loss >= minLoss, c("EarnedPremium", "State")]
```

7.2.2 Altering and adding columns

We can add columns by using the same operators as for access[2], though it's common practice to use `$` and the name of our new column.

```
df$LossRatio = df$Loss / df$EarnedPremium
```

The `transform` will return a new data frame with transformed columns.

```
df <- transform(df, LogLoss = log(Loss), LogPremium = log(EarnedPremium))
```

I find `with` to be a bit easier than `transform`, however it can't (easily) be used to create more than one column at a time. `with` will refer to the columns of a data frame in the statement where it's called. We typically use this to return a single vector which we then assign as a new column to a data frame.

```
df$LogLoss = with(df, log(Loss))
```

7.2.3 Eliminating columns

We can eliminate columns in one of two ways. If we only want to remove one column, we can assign the value NULL to it.

```
df$LossRatio <- NULL
```

If we want to eliminate more than one column, we may construct a new data frame which only includes the columns we'd like to keep.

```
df <- df[, 1:2]
df <- df[, c("State", "EarnedPremium")]
```

If we'd like to remove specific columns, Hadley Wickham [Wic] showed a nice means to do this by using set differences.

```
df <- df[, setdiff(colnames(df), c("RateChange", "Losses"))]
```

7.2.4 Altering column names

```
df$LossRation = with(df, Losses / EarnedPremium)
colnames(df)[4] = "Loss Ratio"
colnames(df)
```

7.2.5 Ordering

For vectors, we can use `sort` to get a sorted vector. For a data frame, we'll need to use `order`. Remember that `order` functions a lot like `which`. It will return indices which may then be used to return specific contents.

```
order(df$EarnedPremium)
#>  [1]  4 26  1 28 12 30 18 10 19 13 25  7  9  8 17 22 11 23 16 29 14 21  2
#> [24]  5 24  6 27 15  3 20
df = df[order(df$EarnedPremium), ]
```

[2] We can also use `cbind` to add columns.

7.2.6 attach

If the name of a data frame is long, typing it to access column elements might start to seem tedious. The `attach` function will alleviate this, by attaching the data frame onto something called the "search path" (which I might have described in the section on packages). What's the search path? Well, all evidence to the contrary, R will look high and low every time you refer to something. As soon as it finds a match, it'll proceed with whatever calculation you've asked it to do. Attaching the data frame to the search path means that the column names of the data frame will be added to the list of places where R will search.

```
attach(dfA)
# do some stuff
attach(dfB)
#> The following objects are masked from dfA:
#>
#>     rep.c..TX....NY....CA....10.,
#>     round.rlnorm.length.State...meanlog...log.50000...sdlog...1...
```

There are many references that suggest using `attach`. I don't. I'll actually advise against it. Why is `attach` a bad idea? If you add a number of similar data frames or packages it can quickly get hard to keep up. The worst case scenario comes when you add two data frames that share column names and you then proceed to carry out analysis.

```
mean(EarnedPremium)
#> [1] 61457.34
```

Which EarnedPremium am I talking about? "Well to tell you the truth in all this excitement I kinda lost track myself." [3] `attach` won't necessarily harm you. But it is a loaded gun. If you get tired of typing the name of a data frame, consider using `with`.

7.3 Summarizing

We can use the standard suite of aggregation functions to summarize a single column of a data frame.

```
sum(df$EarnedPremium)
#> [1] 1843720
max(df$EarnedPremium)
#> [1] 559956
```

For multiple columns, base R comes with the `aggregate` function. Its easiest implementation uses a formula which shows how we want the variable split. In the function below, we're splitting earned premium by state.

```
aggregate(EarnedPremium ~ State, data = df, FUN = sum)
#>   State EarnedPremium
#> 1    CA      673095.7
#> 2    NY      939215.3
#> 3    TX      231409.1
```

That's easy to type, but will only give me one column at a time. If I want multiple columns, I'll ned to do something like the following:

```
aggregate(df[, c("EarnedPremium", "Loss")], by = df["State"], FUN = sum)
#>   State EarnedPremium     Loss
#> 1    CA      673095.7 257519.6
```

[3] Copyright some film company.

```
#> 2    NY      939215.3 423081.7
#> 3    TX      231409.1 228745.6
```

Although `aggregate` gets the job done, I often find it cumbersome for anything complex. Ever since I started using it, I've been hooked on the `dplyr` package. We'll not go into it here, but I'll recommend checking it out. Hover, spend a bit of time playing with `aggregate` first. There are truckloads of packages designed to make manipulation of data frames easier. They're all powerful and they're all a little different. I'd suggest learning the functions in base R first, then moving on to tools like `dplyr` and `data.table`. There's a lot to be gained from understanding the problems those packages were created to solve.

7.4 Reading and writing external data

First, we'll write out some data. Then we'll read it back in. The comma separated value (CSV) format is one of the easiest to work with.

```
write.csv(df, "SomeFile.csv")
```

To read data back in, we use the (wait for it) `read.csv` function.

```
myDF <- read.csv('SomeFile.csv')
```

We can use this same function to read data from the web. If it's accessible and we've provided a proper URL, we can treat it the same as anything on our local file system.

```
URL <- "http://www.casact.org/research/reserve_data/ppauto_pos.csv"
df <- read.csv(URL, stringsAsFactors = FALSE)
```

7.5 Exercises

- Create a data frame with 500 rows. Include a column for policy effective date, policy expiration date and claim count. For claim count, consider using the `rpois` function to simulate from a poisson distribution.
- Save this data to a .CSV file and then reload that file.
- Which policy had the most claims? Which policy year?
- Add a column for

7.5.1 Answer

References

Bibliography

[Mat11] Norman Matloff. *The Art of R Programming*. no starch press, 2011.

[Wic] Hadley Wickham. *Advanced R*.