Data Science Exercises

Dr. Brown

2022-11-13

Contents

0.1 Preface, or, What is This Thing?

This thing is a collection of references and exercises for the less traditional aspiring data scientist. I hesitate to call it a book, as that hints at aspirations far beyond what you read.

Who are you?

- You're a student. No, not someone enrolled at an institution of higher education, though you might be. I mean that you're interested in studying.
- You might be in business, but you might not.
- You're on a budget. You have limited financial resources and your computing resources are limited to your laptop. All of the exercises in this thing can be run on a common laptop with an internet connection and using freely available tools.
- You like to figure things out and make connections. When you run into a "Huh, that's weird!" situation, you look at it as an opportunity.
- You're aware that there is data all over the place nowadays, and you want to see what it can tell you.

Where did this thing come from? I've been teaching an undergraduate course in data analysis for over a decade. Most of these exercises were developed in one of two ways:

- I worked with a messy data set, completed my analysis, and then broke down the steps into sequences of exercises. I scattered these throughout the homework sets so that each set builds on the earlier ones.
- I worked with a student with a messy understanding. I created some data that isolated and clarified some ideas for that student, then found that those clarifications resonated with others. This is where most of the synthetic data exercises came from.

How is this thing organized? The first dozen chapters are the twelve homework sets for the class I teach, expanded somewhat, and in the order I present them. Students have roughly a week to work through each homework set, and there are a few additional weeks in the semester given over to work on course projects. The next few chapters are expansions on what I feel are both important tools for the data scientist and trouble spots for the learning practitioner. The next few chapters after that are some longer case studies, taken step by step. And the last two chapters include references to sources of data and to reference works.

Who am I? I'm a mathematician by academic training and I have an inordinate fondness for differential equations and stochastic processes, but don't let that scare you. I'm a data scientist in practice. I'm still at heart a punk kid and D&D player from the 80's and 90's, with all of the DIY attitude that entails. I've been coding since 1982, but I'm not nostalgic for the old days and I'm happy with modern fast computers and slick, easier-to-use languages.

0.2 How Do I Use This Thing in my Class?

I assign one homework per week. Thirteen homeworks plus half a dozen class periods for time to work on the two course projects fills up the fifteen week semester.

Other notes:

- Depending on the class and their familiarity or lack thereof with coding, I might expand the time spent on the first homework set to two weeks and skip the thirteenth homework.
- Homework 13 is entirely optional. While the subject material is fascinating and raises interesting questions both mathematical and philosophical, unsupervised learning is quite different from the earlier course material.
- Hypothesis testing does not comprise a large component of the course; this was not intended to be an introductory statistics course. I don't recommend omitting the material in homework 8 entirely, as many metrics for model performance are couched in the language of p-values. However, the exercises on power estimation and ANOVA can be omitted if time is short.
- I record video captures of myself working through various R/RStudio tasks. This is difficult to capture in a static book form and so easy as a video!

Chapter 1

The First Homework: An Introduction to R

1.1 From Math Expressions to R

Generally, the syntax for mathematical expressions in R works as in most spreadsheets.

Evaluate each of the following mathematical expressions with a chunk of R code. For example, if the expression is $3e^{2*1.45-1}$ you would write the chunk of R code:

```
3*exp(2*1.45-1)
```

```
[1] 20.05768
```

and evaluate to arrive at your answer. *You may have to search to discover how to write some functions in R. This is an opportunity to practice searching for technical details about your coding language, an invaluable skill.*

(a) $3 \cdot 5^2 - 2.4 \cdot 10^{-1}$ Answer:

```
[1] 74.76
```

(b) $\sqrt{14 - 10 \cdot 0.321}$ Answer:

```
[1] 3.284814
```

(c) $\sqrt{\frac{14 + 5.2}{2.1 - 0.9}}$ Answer:

```
[1] 4
```

(d) $0.2 \cdot e^{2.4+0.2^2}$ Answer:

```
0.2*exp(2.4+0.2^2)
```

```
[1] 2.294608
```

(e) $\ln(0.2) - \ln(\sqrt{3})$ Answer:

```
[1] -2.158744
```

1.2 Rounding Numbers in R

(a) Execute the following code chunk to see what happens.

```
round(1.234)
```

```
[1] 1
```

```
round(1.234,1)
```

```
[1] 1.2
```

```
round(1.234,2)
```

```
[1] 1.23
```

(b) Execute the following code chunk to see what happens.

```
floor(1.234)
```

```
[1] 1
```

```
ceiling(1.234)
```

```
[1] 2
```

```
floor(-1.234)
```

```
[1] -2
```

```
ceiling(-1.234)
```

```
[1] -1
```

1.3 Scientific Notation in R

(a) Execute the following code chunk to see what happens.

```
7e3
```

```
[1] 7000
```

(b) Multiply `2.1 x 10^8` and `3.4 x 10^7`. Answer:

```
[1] 7.14e+15
```

1.4 Variables in R

(a) Use an R code chunk to store 4 in the variable x and -2 in the variable y.

(b) Now use an R code chunk to evaluate $2x^2 + 3xy - 5y^2$. Answer:

```
[1] -12
```

(c) Now, write a single code chunk that does (a), then (b). Evaluate to make sure you have a correct working chunk. Then in your code chunk, change the 4 to a 0 and the -2 to a 3 and re-evaluate. Answer:

```
[1] -45
```

(d) Based on your work in (c), what is the benefit of storing data in variables rather than typing data in by hand? ## Characters (Strings) in R

(e) Execute the following code block to define strings and store them in `a` and `b`. Notice the extra space at the end of the second string.

```
a <- "apple"
b <- 'banana '
```

Note: You may use double or single quote to surround a string in R. Double quotes are my own preference, because single quotes are easily mistaken for the back tick (next to the 1 key on most keyboards).

(b) Use the `nchar` function to count the letters in `a`. Answer:

```
[1] 5
```

(c) Use the `trimws` function to trim the extra whitespace (spaces, new lines, etc.) from `b` and store the result in `b`, then use `nchar` to count the characters in the new `b`. Answer:

```
[1] 6
```

(d) Join the two strings `a` and `b` with the `paste` function, using `,` as a separator with `sep`. Answer:

```
paste(a,b,sep=",")
```

```
[1] "apple,banana"
```

(e) Convert the number `1204.73` to a string and store it in `x` by executing the following code chunk.

```
x <- as.character(1204.73)
```

(f) Create a string expressing a distance with units, meters, by pasting together the string `x` from (e) with "meters". Answer:

```
[1] "1204.73 meters"
```

1.5 Logicals in R

A logical data type is usually the result of evaluating an expression as true or false. In R, these are written `TRUE` and `FALSE`.

(a) Execute the following code chunk to evaluate the truth or falsity of the inequality.

```
sqrt(1234) > 36
```

(b) The results of evaluating logical statements can be stored in variables. Execute the following code chunk to store logical results in `a` and `b`.

```
a <- 3.2^2 < 10
b <- 7.8*15 > 10^2
```

Note: You should be able to see the results in the environment pane in RStudio.

(c) Logical statements can be connected by the "and", `&`, and "or", `|`, logical connectives. Evaluate the following code chunk to see what happens.

```
a <- 3 < 5
b <- 7 > 10
a & b
```

```
[1] FALSE
```

```
a | b
```

```
[1] TRUE
```

(d) The unary "not" operator is denoted `!` in R. Execute the following code chunk to see what happens.

```
!(3<5)
```

```
[1] FALSE
```

(e) We may encounter situations in which we are given a numeric value `x` and we wish to evaluate whether it meets every one of a set of given constraints. Write a code chunk with `x <-` a value at the top, and that evaluates whether `x` is less than 10, whether the square of `x` is greater than 3.1, and whether the cube of `x` is less than 70.

1.6 Data Types in R

State the data type of each of the following expressions in R. You can use the `class` function, as in

```
class(1.2)
```

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

and

```
class(2.3>0)
```

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

Note: If you need finer-grained understanding of the internal representation of data, you can use the `typeof` function instead of class. Generally, `class` will tell us more about data-oriented things and `typeof` will tell us how the data is represented in our computer's memory.

(a) `7.4^2`

(b) `"December 2, 1921"`

(c) `"greed^2"`

(d) `32.1-10 > 4.0`

(e) The class of an expression can be unexpected, until you understand an important fact about R's behavior: when a logical is used where a numeric *should* be used, the logical is often cast to 1 when `True` and 0 when `False`. Here's an experiment to see this in action.

Compute `exp(32.1-10 > 4.0)` in an R code chunk and see what happens! Compare to e^1; did you get the same result? Now try to evaluate `exp(3 > 5)`. What happens? And in particular, what is the class of `exp(3 > 5)`?

1.7 Different Assignment Operators

There are usually a number of different ways to "say" something in R, as in most languages. Instead of asking "how do I say this?", ask "what are the different ways I can say this, and which is best for this context?"

There are a number of different notations for assigning a value to a variable. In this exercise we'll look at several.

(a) Execute the code chunk and make sure that `2` is printed. This verifies that `2` is stored in `a`.

```
a <- 2
a
```

```
     [1] 2
```

(b) Now let's store `3` in `a` using a different notation. Execute the following code chunk!

```
a = 3
a
```

```
     [1] 3
```

(c) Next, store `4` in `a`. Execute the following code chunk.

```
4 -> a
a
```

```
     [1] 4
```

(d) This is getting a bit ridiculous, but ... let's store `5` in `a`, and then `6` in `a`. Execute the following code chunk.

```
a <<- 5
a
```

```
     [1] 5
```

```
6 ->> a
a
```

```
     [1] 6
```

(e) While the `=` sign is most familiar as an assignment operator to coders coming from other languages, it is not the best assignment operator to use in R. Why not? First, it is easy to confuse the `=` assignment operator with the `==` logical test. Second, the arrows are a lot more flexible.

But why do we have single-headed arrows and double-headed arrows? Execute the following two code chunks. We'll discuss functions in more detail later.

```
a <- 1
f <- function() {
  a <- 2
  print(a)
}
f()
```

```
     [1] 2
```

```
a
```

```
     [1] 1
```

Note: calling the function `f` assigns `2` to `a` inside the function's space, or *scope*, and then prints whatever is in `a`. But this does not affect the values stored in

variables outside of the scope of the function. So, when we execute `a` after `f()`, you should see `1` printed, because the value of `a` outside of `f` did not change.

Same code chunk except with a double arrow head inside the function scope!

```
a <- 1
f <- function() {
  a <<- 2
  print(a)
}
f()
```

```
     [1] 2
```

```
a
```

```
     [1] 2
```

Note: This time, executing the function `f` does result in a change to the variable `a` outside of the function's scope; that's the effect of the double arrow head!

Generally speaking, you should not change the value of a variable outside the scope of the function without an exceptionally good and well-articulated reason.

1.8 Vectors - Not the Physics Kind

(a) Use an R code chunk to write the vector `c(3,2,5,0)` and store it in the variable `x`.

(b) Evaluate `x^2` and `exp(x)`. What happens? Numerical answer for x^2:

```
     [1]  9  4 25  0
```

(c) Use an R code chunk to write the vector `c(-2,-1,-4,1)` and store it in the variable `y`. Then evaluate `x+y`. What happens? Numerical answer:

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

(d) Take a moment and write in plain language how R seems to handle arithmetic with vectors.

(e) Vectors may be non-numerical as well. Use an R code chunk to store the vector `c("apple","banana","cat")` in the variable `L`.

(f) Elements of vectors must have the same type of data throughout. Execute the following chunk of code and see whether you understand what R is doing to make sure that all the data in the vector `a` is the same class.

```
a <- c(1.2,"cat",TRUE)
a
```

```
     [1] "1.2"  "cat"  "TRUE"
```

(g) You may use `c()` to concatenate vectors into a single vector. Execute the following code chunk to see what happens.

```
a <- c(1,3,5)
b <- c(2,4,6)
together <- c(a,b)
together
```

```
     [1] 1 3 5 2 4 6
```

What would have happened if we replaced `c(a,b)` by `c(b,a)`?

1.9 R Commands for Building Vectors from Patterns

(a) You can create a *range* of values using `:`. Execute the following chunk of code to see what happens.

```
a <- 1:5
a
```

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

```
b <- -4:7
b
```

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

(b) You can create a vector of a pattern of values using the `seq` command. Execute the following code chunk to see what happens.

```
a <- seq(1,13,by=2)
a
```

```
     [1]  1  3  5  7  9 11 13
```

```
b <- seq(-1,1,length.out=20)
b
```

```
      [1] -1.00000000 -0.89473684 -0.78947368 -0.68421053 -0.57894737 -0.47368421
      [7] -0.36842105 -0.26315789 -0.15789474 -0.05263158  0.05263158  0.15789474
     [13]  0.26315789  0.36842105  0.47368421  0.57894737  0.68421053  0.78947368
     [19]  0.89473684  1.00000000
```

(c) In many cases you can use vectorized arithmetic to create sequences quickly. Execute the following code chunk to see what happens.

```
a <- seq(3,24,by=3)
b <- 3*(1:8)
a
```

```
 [1]  3  6  9 12 15 18 21 24
```

```
b
```

```
 [1]  3  6  9 12 15 18 21 24
```

(d) You can create a vector with one element repeated many times using `rep`. Execute the following code chunk to see what happens.

```
a <- rep("red",10)
a
```

```
 [1] "red" "red" "red" "red" "red" "red" "red" "red" "red" "red"
```

Note: the `rep` command can be useful for labeling data in data frames.

1.10 Recycling Vectors

R exhibits a peculiar behavior with respect to arithmetic of vectors, called *recycling*. It is occasionally useful in practice. More often, it causes code to execute in ways that seem strange unless you are aware of this behavior.

(a) Use an R code chunk to store the vector `1:4` in `a` and `5:8` in `b`.

(b) Now execute the following code chunk to see what happens when we multiply `a` and `b`.

```
a * b
```

```
 [1]  5 12 21 32
```

(c) Next, store the vector `c(-1,1)` in `b` and compute `a * b`. Answer:

```
b <- c(-1,1)
a * b
```

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

Note: The behavior we are seeing here is called *recycling* in R. When executing `a * b`, the first two values are multiplied, then the second two, and so on. If one vector "runs out" of values, then R simply returns to the beginning of that vector for the next value and cycles through again. This continues until all the values in the longer vector have been used.

(d) Execute the following code chunk to see what happens.

```
1:5 * c(-1,1)
```

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

Note: R provides a warning when the shorter vector is not a multiple of the longer vector, but it uses the recycling behavior anyway.

1.11 Factors

A special - and incredibly important! - type of vector is a *factor*, also known as a categorical variable. A factor is a vector whose entries represent categories to which a data entry belongs. The categories in a factor are known as its *levels*. A great way to think about factors is as responses to multiple choice questions.

Execute the following code chunk to see what happens.

```
a <- c(rep("A",10),rep("B",5),"C","C")
b <- factor(a)
a
```

```
 [1] "A" "A" "A" "A" "A" "A" "A" "A" "A" "A" "B" "B" "B" "B" "B" "C" "C"
```

```
b
```

```
 [1] A A A A A A A A A A B B B B B C C
Levels: A B C
```

Note: The factor `b` prints the levels as well as the entries.

1.12 Functions Measuring Properties of Vectors

Some functions in R take vectors as inputs and produce single numbers as outputs. In this exercise I'd like you to look up the documentation for the `sum` and `length` functions in R. Then, for the vector `x` from Exercise 4, use R code chunks to compute the sum of `x`, the length of `x`, and the mean of `x`. Answer for mean of x:

```
[1] 2.5
```

1.13 Extracting Elements From Vectors

(a) Write and evaluate the following R code chunk. Make sure you understand what it is doing!

```
x <- 3*(1:10)
```

(b) Write a code chunk that extracts the first three elements of `x`.

(c) Write a code chunk that extracts the last three elements of `x`. *Hint:* ***`tail()`*** *function.*

(d) Write a code chunk that extracts the 1st, 5th, and 7th elements of `x` in one vector. Answer:

```
[1]  3 15 21
```

(e) Write a code chunk that attempts to extract the 2nd, 4th, and 13th elements of `x` in one vector. What goes wrong, and why?

1.14 Extracting Elements With Logicals

Use `x <- 1:1000` to create a vector and store it in `x`. Then,

(a) Evaluate `x[x^2<10]` in a code chunk. The output should be much shorter than the original vector `x`.

(b) To figure out what the code in (a) is doing, let's break it down. The code splits naturally into two pieces. The *filter* is the logical vector $x^2 < 10$ and the data vector is x. How many of the filter's entries are true? Does this help explain why `x[x^2<10]` has so few entries?

(c) Find all of the values in the vector `x` that satisfy the inequality $|x-300| \leq 25$. Absolute value is a function in R, `abs`. Answer:

```
 [1] 275 276 277 278 279 280 281 282 283 284 285 286 287 288 289 290 291 292 293
[20] 294 295 296 297 298 299 300 301 302 303 304 305 306 307 308 309 310 311 312
[39] 313 314 315 316 317 318 319 320 321 322 323 324 325
```

Note: Filtering with logical vectors is a tremendously important operation. We'll see it come up again in this homework and throughout the course.

1.15 Positions or Values?

In some cases, we don't need to know the *values* in a vector that satisfy some filter, but we need to know their *positions* in the vector. Note that the position of a value in a vector is often called its *index*.

Use `x <- 2*(1:10)` to create a vector and store it in `x`. Make sure you know what the vector `x` contains! Then,

(a) Evaluate `which(x^2<75)` in an R code chunk. Does this return what you think it ought to? Answer:

```
[1] 1 2 3 4
```

(b) Which indices in `x` have data satisfying the inequality $|x-4| < 3$? Use an R code chunk and the `which` function to determine your answer. *Remember, the indices are the positions in the vector, not the values. So your answer here should look like "the third and fifth entries" or something along those lines.* Answer:

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

1.16 Extracting With Conditions From Corresponding Vectors

Suppose we have collected data about some children. For each child, we have measured their age in years and their height in inches. We have stored this data in two corresponding vectors,

```
age <- c(10,8,5,11,10,6,5)
height <- c(52,48,41,60,54,43,43)
```

By *corresponding*, I mean that the first entry in each vector refers to child 1's measurements, the second entry in each vector refers to child 2's measurements, and so on.

Write these vectors in an R code chunk. Then,

(a) Your teammate executes the expression `age[height>62]`. What question are they trying to answer? How many entries does the result have?

(b) Write an R code chunk to find the ages of all children taller than 50 inches. Answer:

```
[1] 10 11 10
```

(c) Write an R code chunk to find the heights of all children under 10. Answer:

```
[1] 48 41 43 43
```

(d) Let's return to your teammate's code, `age[height>62]`. How does the code `age[which(height>62)]` compare? Does it produce the same result?

(e) Hmm. That last exercise raises a question of why we need the `which` function in R. Let's compare two ways to filter for children with heights greater than 62 inches. Evaluate the expressions `height>62` and `which(height>62)`. Clearly one is shorter to type, but what are the other differences? In particular, would one be easier to store in a file than the other?

(f) The `which` function has some related functions, `which.max` and `which.min`. Read the R documentation for these functions, and then use them to find the indices of the tallest and shortest children in the data. Answer for tallest:

```
[1] 4
```

(g) Use your results from part (f) to find the ages of the tallest and shortest children in the data set. Answer, age of tallest:

```
[1] 11
```

Note: We could have answered any of these questions by a simple inspection of the data set, because it has very few records. In order to learn the software, make sure that you are always performing your work as if the data has millions of records. Do your results rely at any point on a visual inspection? Can you work around that? These are questions we will ask ourselves throughout the course.

1.17 Re-leveling Factors

Regardless of the method used to subset, factor variables sometimes pose a problem.

(a) Execute the following code chunk to see what happens.

```
a <- factor(c("A","A","B","B","C"))
b <- a[1:4]
b
```

```
[1] A A B B
Levels: A B C
```

(b) In part (a), what are the distinct values in `b`? What does R state as the levels of `b`?

(c) In some applications we will need to retain the "memory" of the levels from which the subset was taken. That is, we might need to know that `b` was drawn from a factor with levels `levels(b)`. This is the reason that R preserves this information by default.

But, in many applications, we simply need to know the levels of `b`. We can re-level `b` as in the following code chunk.

```
b <- factor(b)
b
```

```
[1] A A B B
Levels: A B
```

(d) In some cases we might need to specify an order on the levels of the factors. Execute the following code chunk to see what happens.

```
a <- factor(c("A","A","B","B","C"),
            levels=c("A","B","C"),
            ordered=TRUE)
a
```

```
[1] A A B B C
Levels: A < B < C
```

```
a <- factor(c("A","A","B","B","C"),
            levels=c("B","A","C"),
            ordered=TRUE)
a
```

```
     [1] A A B B C
     Levels: B < A < C
```

Note: Ordering levels in a factor can be useful in a variety of situations, such as ordering the months in a year or the days in a week. Ordering levels helps other functions like plotters create structures that are aware of the intended order.

(e) In some cases we may want to rename the levels of a factor. Execute the following code chunk to see what happens.

```
a <- factor(c("Level A1","Level B1","Level C1"))
a
```

```
     [1] Level A1 Level B1 Level C1
     Levels: Level A1 Level B1 Level C1
```

```
levels(a) <- c("A","B","C")
a
```

```
     [1] A B C
     Levels: A B C
```

1.18 Tables

(a) Execute the following code chunk to see what happens.

```
a <- c(rep("a",10),rep("b",30),rep("c",5))
table(a)
```

```
     a
      a  b  c
     10 30  5
```

(b) Execute the following code chunk to see what happens.

```
a <- c(rep("a",10),rep("b",30),rep("c",5))
b <- c(rep("a",20),rep("b",5),rep("c",20))
table(a,b)
```

```
        b
     a    a  b  c
       a 10  0  0
       b 10  5 15
       c  0  0  5
```

1.19 Lists

Vectors in R must contain data that is all of the same class. Knowing that all of the data in a container is the same class is useful for coders, who can then write functions assuming the data is all numerical, or all character, or all logical. But, sometimes we will need containers that are able to hold data of different classes. In R, this is the *list*.

(a) Execute the following code chunk to build a list `a`.

```
a <- list("apple",1204.73,FALSE)
```

(b) Extract the second element of `a` using `a[[2]]`.

(c) Execute the following code chunk to build a named list `b`.

```
b <- list(dog="Fido",size=1.2,color="reddish")
```

(d) Execute the following code chunk to see what happens.

```
b[[1]]
```

```
     [1] "Fido"
```

```
b[["size"]]
```

```
     [1] 1.2
```

```
b$color
```

```
     [1] "reddish"
```

Note: This gives a few different ways of accessing the elements of a list. The last method using `$` will be used often when we move to *data frames*, a special kind of list.

(e) Execute the following code chunk to see what happens. The object returned is a list.

```
b[1:2]
```

(f) Execute the following code chunk to see what happens.

```
b[2:3][[1]]
```

```
     [1] 1.2
```