Forty Days and Forty Nights of Data

Dr. C. E. Brown

2023-07-07

Contents

0.1 Preface, or, What is This Thing?

This thing is a collection of references and exercises for the aspiring data analyst, or scientist, or whatever we are calling them nowadays. 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, 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

Day 1: Introducing Data Analysis and R Syntax

1.1 The First and Last Miles: Visualization

There is an aphorism to the effect that the first and last miles of any data analysis journey is dominated by visualizations. That's true, or close enough to it. The first part of a data analytics project - after the data is available to you! - is to explore the data set through summaries, tables, and lots and lots of visualizations. The last part is to communicate your conclusions and insights to an audience, and this usually involves creating compelling and informative visualizations.

But to be a little more precise, first you have to *get* that data, then analyze it. In this exercise we'll assume you have access to the data as a `.csv` file (a comma-separated value file), which means that our task is simply to read the data into RStudio.

Create a new code block for R in RStudio and execute the following command:

This could have been accomplished with one big line, like

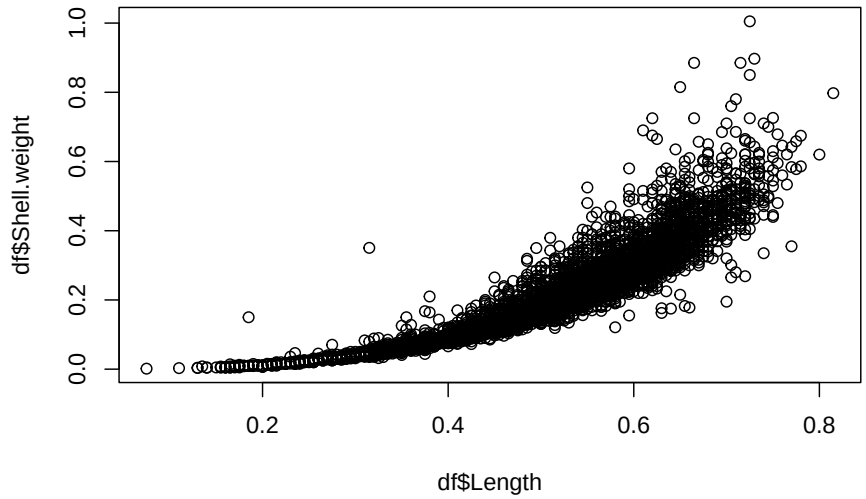
but see what happened in the output? So instead, we worked with chunks of the path to the file and glued them together later. *Most of the time R simply "knows" when a line continues and so you can just insert line breaks as needed for readability and style. However, strings act a little differently. Inserting a line break literally inserts a line break character into the string... which turns out to be bad for a URL.*

You should see that RStudio's Data pane (upper right area of the RStudio app by default) now includes a line with our dataframe name `df` along with some

basic information about it (4177 observations of 7 variables). We can summarize the dataframe:

```
     Sex                Length         Diameter         Height
 Length:4177        Min.   :0.075   Min.   :0.0550   Min.   :0.0000
 Class :character   1st Qu.:0.450   1st Qu.:0.3500   1st Qu.:0.1150
 Mode  :character   Median :0.545   Median :0.4250   Median :0.1400
                    Mean   :0.524   Mean   :0.4079   Mean   :0.1395
                    3rd Qu.:0.615   3rd Qu.:0.4800   3rd Qu.:0.1650
                    Max.   :0.815   Max.   :0.6500   Max.   :1.1300
  Whole.weight     Shell.weight        Rings
 Min.   :0.0020   Min.   :0.0015   Min.   : 1.000
 1st Qu.:0.4415   1st Qu.:0.1300   1st Qu.: 8.000
 Median :0.7995   Median :0.2340   Median : 9.000
 Mean   :0.8287   Mean   :0.2388   Mean   : 9.934
 3rd Qu.:1.1530   3rd Qu.:0.3290   3rd Qu.:11.000
 Max.   :2.8255   Max.   :1.0050   Max.   :29.000
```

And once we have done that we know which variables are numeric and which are not. Let's make a quick scatterplot of `Shell.weight` versus `Length`.



And we've gotten started on the first mile of our data analysis with this data set! You probably have many, many questions about the details in this exercise. We'll flesh those out over the course of several of the early assignments.

Summary: In this exercise you recreated the code above to read the data and create a plot. You may not understand why the commands work the way they do,

and that's okay for now.

1.2 Aside: The Data Analysis Process

Let's break down the data analysis process into six discrete stages: Inquiry, Ecology, Processing, Analysis, Communication, and Action.

- **Inquiry** - Ask questions. Many questions start out vague and broad - "How can we save the world?" and gain specificity during an investigation - "How can we improve the crop yield of this particular grain in this particular region?" As a data analyst, your goal is to make sure that the questions can be addressed measurable data that is accessible. The inquiry process may involve some initial data gathering and some exploratory data analysis as the questions under investigation are made more specific.
- **Ecology** - Data exists everywhere in the wild, but you may have to capture it and store it in a zoo for later use. Understanding what data is out there and how to capture it is a useful skill set. This process is often known as *data ingestion*, but I prefer the term "ecology" to remind you that data exists in the world **in context**, in a certain system, and that can never be ignored. The "zoo" where you store your data may be highly structured (relational databases made up of linked collections of tables) or not (a big folder full of web pages, texts of emails, and images).
- **Processing** - Before use in an investigation, the data must be extracted from the zoo and processed. Data might be problematic in many ways: badly formatted, inconsistent units, stored in different tables, suspiciously out of the range of possibility (a human with a recorded height of 60 feet is probably a data entry error!), or just plain missing. These issues must be addressed before the data is analyzed.
- **Analysis** - Explore, summarize, visualize, and model the data. The goal here is to develop responses to the driving questions that are supported by the available data.
- **Communication** - Report the results of the analysis to the appropriate audience. Communicating results with data is a broad skill set, including: technical skills to create impactful visualizations and animations; design skills to create engaging and aesthetically pleasing supporting materials; critical thinking skills to describe how the conclusions were reached; and communication skills to engage the audience.
- **Action** - Decision makers use the conclusions of the analysis to take action.

We'll also break down data contexts - the real-world situations when we call for data to be analyzed - into three broad categories.

- **One-offs** - There is a particular question and a data set that seems relevant. Analyze! In the one-off, we *don't* need to worry about how often

the analysis must be repeated. The one-off is common in experimental sciences, in which the researcher designs an experiment to collect data, performs it, analyzes the data, and reports. The data is still archived and may be used later, but there is no expectation of that.

- **Updates** - There is a particular question that must be answered, but the answer - and the data that leads to it! - changes over time. For example, a company may be interested in selecting a strategy from among many possible. It gathers data and periodically analyzes that data... and may shift strategies when the data indicates. In this case, *data engineering* is crucial. The new data must be gathered and processed repeatedly, and so we would like an automated system with monitoring tools to do so. Reports must be generated periodically or even updated live on websites, and again we would like an automated system with monitoring tools to do this.

- **Question pools** - The analyst is not focused on one question only, but on a related collection of questions - a question pool. For example, a corporation might have the broad question "How can we increase shareholder value?" That can be sharpened to many more specific questions, and the organization might be interested in answering any or all of them. In this case, the focus is strongly on archiving large data sets and making them accessible to the analysts who need them. This is similar to the **Updates** context in the sense that the focus is on the engineering aspects, but here the focus is strongly on creating robust, flexible computer architectures to store and retrieve data. Computer security is also important; the data must be accessible to the analysts who need it, not accessible to those who don't, editable by those who need to do so and not editable by those who have no need. *You can think about this as you would any shared cloud doc. Who can view it? Who can edit it? Can those permissions change over time and who has the authority to change them?*

Okay, so in this class, what's our focus?

- We're not going to be concerned about data engineering in this course. So we won't worry about how data is stored or accessed, and we won't worry about automating those processes. Data engineering has a great deal of overlap with software engineering and with database design, and I highly recommend taking courses in those areas if you are interested. Having said that...

- We will use tools that allow us to repeat our analyses with updated data sets.

- We will focus strongly on the Analysis and Communication aspects of the data analysis process, and somewhat less heavily on the Processing and Inquiry aspects. Because we focus heavily on Analysis and Communication...

- We will use tools that strongly support Analysis and Communication operations. Thus the choice of the R language and the RStudio platform.

Summary: In this exercise you read about the data analysis process. When you submit your assignment, just indicate that you read this exercise.

1.3 Aside: The Assignment Writeup

In this class I expect you to create a new RMarkdown document for each new assignment. You may want to create a new project for each one (I certainly recommend that). When you write your responses to assignment exercises, you should present code in visible code chunks and you should connect your code chunks with relevant discussion that acknowledges there is a reader who will be viewing your document. You can comment your code with short notes, but that is not a replacement for the connecting language between code chunks.

That is, get in the habit of thinking of each RMarkdown document as a full-featured document, part of which includes code.

Summary: You read about the assignment expectations. In your assignment writeup, indicate that you read and understand this exercise.

1.4 From Math Expressions to R

The syntax for mathematical expressions in R works as in most spreadsheets and calculators.

Create a new RMarkdown document. 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:

```
[1] 20.05768
```

and evaluate to arrive at your answer. Connect your evaluations with some relevant text, like "this is my answer to part (a)". *You may have to search to discover how to write some mathematical 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:

```
[1] 2.294608
```

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

```
[1] -2.158744
```

Summary: In this exercise you evaluated some mathematical expressions using R like a calculator. If you're curious about how I formatted the math expressions to look like math expressions ... well, did you do a web search? Why not? Try it! Get in the habit of formulating good search questions and looking for answers to technical questions.

1.5 Rounding Numbers in R

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

```
[1] 1
```

```
[1] 1.2
```

```
[1] 1.23
```

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

```
[1] 1
```

```
[1] 2
```

```
[1] -2
```

```
[1] -1
```

(c) Explain in plain language what `round`, `floor`, and `ceiling` do.

Summary: You experimented with common rounding functions.

1.6 Scientific Notation in R

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

```
[1] 7000
```

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

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

Summary: You used scientific notation to represent numbers in R.

1.7 Variables in R

While R has some features of general purpose programming languages like Python, it is much more strongly focused on data analysis, and in particular has a hierarchy of *data containers*. In this exercise you'll look at the base-level data containers, *variables*, and in the next assignment you'll look at two more levels of data containers, *vectors* and *lists*.

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

Note: You can also use `4 -> x` and `x = 4` to store the value `4` in the variable `x`. The first reads naturally for someone used to reading left to right, and the second reads naturally for someone used to reading code in other language such as Python. The `<-` is often preferred as a matter of style, to be consistent with other programming languages but to also visually distinguish R code from Python code. In RStudio, you can use both R and Python in the same document, and so distinguishing them can be helpful.

(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?

Summary: You stored values in variables and used the stored values to evaluate expressions.

1.8 The Case for Using Variables

When substituting values into R functions, you can substitute the value or a variable the value has been stored in. Substituting a variable is nearly always better, to avoid user entry error.

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

```
[1] 22.7248
```

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

```
[1] 3.436893
```

Note: Typically, we would not "get" the value for `a` by typing it in, but we would read it from a file. Always try to minimize the amount of direct data entry performed by human agents; each time humans enter data or copy it from

one source to another by manually typing it, errors are potentially introduced. Imagine a system in which a human types a value into a spreadsheet, then a second human reads that value and types it into a Datatel terminal, and a third human reads that value from their datatel screen and types it into their phone, and so on. It becomes a big game of telephone, with the end result often looking nothing like the original. Telephone is a fun party game but a disaster for data analysts.

(c) Your buddy Gordon is a bit lazy (though he prefers the term "efficient"). When working with R, Gordon performs a calculation, sees the 10 digit output on the screen, and then types that ten digit output directly into the exponential function. What are the possible errors that Gordon's choice could cause?

Note: In addition to the possibility of causing errors, direct copying of values from screen into code or functions is not seen by readers of your code. You break the documentation chain when you do this; the reader cannot determine what was done as the value passed out of the screen, into your brain, and then into code in some other part of your analysis. Storing that value in a well-named variable both avoids error and documents your process.

Summary: You read some ideas making the case for using variables as compared to typing values.

1.9 Characters (Strings) in R

Some data - possibly most? - is stored as strings of characters. Let's start working with those.

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

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), and the back tick is used heavily in RMarkdown documents.

(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:

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

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

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

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

Summary: You begin working with strings. You also converted one data type - a number - into another data type - a string.

1.10 Logicals in R

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

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

(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`.

Notes: You should be able to see the results in the environment pane in RStudio. Also note that the logical statement on the right hand side of the assignment operator `<-` is evaluated before the assignment is made. Some people prefer the `->` assignment operator in this situation,

because it reads a bit more clearly left to right. I encourage you to use the assignment operator as presented in the exercise, `<-`, whenever it does not destroy readability. It is more consistent with other languages such as Python and will be more easily readable to software engineers who code in those languages. We will see a natural use for `->` later when we discuss data processing pipelines.

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

```
[1] FALSE
```

```
[1] TRUE
```

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

```
[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 <- 5` as the first line, and then

evaluates whether `x` is less than 10, whether the square of `x` is greater than 21, and whether the cube of `x` is less than 70.

Summary: You began working with logicals.

1.11 Data Types in R: class

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

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

and

```
[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)`?

Summary: You began working with data classes with the `class` function.

1.12 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`.

```
[1] 2
```

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

```
[1] 3
```

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

```
[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.

```
[1] 5
```

```
[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 arrow distinguishes R code from Python code, important in documents that may use both R and Python. Third, the arrows are more flexible allowing the variable to be on one side of the assignment or the other.

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

```
[1] 2
```

```
[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!

```
[1] 2
```

```
[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.

Summary: You learned about different assignment operators in R.

Chapter 2

Day 2: Data Containers - Vectors and Lists

2.1 Vectors in R

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

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

(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, and R will enforce that. Because everything can be written as a string, this can often mean that one string in a vector forces all of the other entries to be converted to strings. Execute the following chunk of code to see this in action.

```
[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.

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

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

Summary: You've seen a few vectors in R, both numerical and character.

2.2 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.

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

```
 [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.

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

```
 [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.

```
[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.

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

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

Summary: You've investigated a few techniques for building vectors from patterns.

2.3 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. So let's raise awareness...

(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`.

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

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

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

Note: This is *recycling* in action. 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. In other programming language that uses vectorized arithmetic, like Python with `numpy` or Matlab, this code would usually lead to an error because the dimensions of the vectors do not match. Just one of the quirks of R.

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

```
[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.

Summary: You've experimented a bit with vector recycling, an unexpected behavior in R.

2.4 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.

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

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

```
 [1] 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.

(b) Extract the levels of `b` by executing the following code chunk.

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

(c) Factors are stored differently than character vectors. Execute the following code chunk:

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

Note: The first entry here tells us that the level recorded in that entry of the vector is the first level in the factor's `level` list. We can find the name of that first level with

```
[1] "A"
```

Storing factors as short integers allows us to store a lot of data with very little computer memory... as long as that data is appropriate for a factor, having only a few levels.

(d) In some cases this clever storage can create some unintended stumbling blocks. Execute the following:

```
[1] "Hello there!" "1"
```

Note that `b[1]` was treated as a `1`, and then cast to a string because of the presence of the other string `"Hello there!"`. If you want to make sure that `b[1]` appears as `"A"` you should convert it to a character first:

```
[1] "Hello there!" "A"
```

Summary: You experimented with factors, a special type of vector in R. Explicitly distinguishing factors is crucial to the operation of many data operations in R, including the creation of meaningful graphics.

2.5 Functions Summarizing 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 <- 1:100`, compute the sum of `x`, the length of `x`, and the mean of `x`. Answer for mean of x:

```
[1] 50.5
```

Summary: You read documentation about three summary functions for vectors. Reading documentation is one of the most important skills you can acquire as a computational data analyst. There's always new software or a new library to learn.

2.6 Extracting Elements From Vectors

(a) Write and evaluate the following R code chunk to build a vector `x`.

(b) Write a code chunk that extracts the first three elements of `x`. There are many ways to do this; here are a few.

```
[1] 3 6 9
```

```
[1] 3 6 9
```

```
[1] 3 6 9
```

Note: Since `1:3` is shorthand for `c(1,2,3)` the first two here are really the same. But `1:3` is easier to type if the number had been larger, like `1:1000`.

(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?

Summary: In this exercise you experimented with extracting elements from vectors using their positions in the vector.

2.7 Extracting Elements With Logicals

Use `x <- 1:20` 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 vector `x` and it should include only values from `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`. Execute the following code chunk to see what the filter is doing:

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

How many of the filter's entries are `TRUE`? And in which positions of the vector do they fall?

(c) Now execute the code chunk

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

(d) And now put it all together in a single line.

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

Note: Sometimes it can be helpful to avoid intermediate variables, like `filter` in part (c). These variables are held in your environment until released, using valuable system resources and visually cluttering up RStudio. On the other hand,

in some cases intermediate variables can be very helpful for readability and code testing. It's a judgement call! You'll figure this out as you get more experience working in R.

(e) Find all of the values in the vector `x` that satisfy the inequality $|x-12| \leq 4$. Absolute value is a function in R, `abs`. Answer:

```
integer(0)
```

Summary: You experimented with extracting elements from a vector with logical vectors, called filters. Filtering is a tremendously important operation. We'll see it come up again in this homework and throughout the course.

2.8 Extracting Elements from Vectors: 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. Although in some database contexts an index is a reference key.*

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. Describe what this returns, and in particular, did it return positions in the vector or values from the vector? 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
```

Summary: You experimented with the `which` *function, which returns the positions of data in a vector meeting a logical condition.*

2.9 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, along with a fictional name,

By *corresponding*, I mean that the first entry in each vector refers to child 1, the second entry in each vector refers to child 2, and so on. So Billy Bob (child 1) is age 10 and 52 inches tall.

Write these vectors in an R code chunk and execute. Then,

(a) A member of your data team 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 years of age. 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 and names of the tallest and shortest children in the data set. Answer, age and ID of tallest:

```
[1] 11
```

```
[1] "Joe Bob"
```

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 or billions 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.

Summary: You learned to extract information from corresponding vectors by

using a filter built from one vector to extract information from others. Each collection of corresponding information is a record. For example, Billy Bob with age 10 and height 52 inches tall is a single record in the data set. We'll soon see how to wrap all of this together in dataframes ... where we will still use the idea of filters built from vectors.

2.10 Re-leveling Factors

Regardless of the method used to extract data from a vector, factor variables pose additional challenges.

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

```
[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.

```
[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.

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

```
[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.

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

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

Summary: You experimented with re-levelign factors in a variety of ways.

2.11 Tables

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

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

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

```
   b
a    d  e  f
  a 10  0  0
  b 10  5 15
  c  0  0  5
```

Summary: You saw a couple of examples of frequency tables. Frequency tables summarize character or factor data in a clearly readable way.

2.12 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 type of container is the *list*.

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

(b) Extract the second element of `a` using `a[[2]]`. Notice the use of the double square brackets to distinguish lists from vectors.

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

Note: Named lists are similar to Python dictionaries in some ways.

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

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

```
[1] 1.2
```

```
[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.

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

```
[1] 1.2
```

Summary: You've experimented a bit with lists and list notation. Pay close attention to the `$` *notation. It will show up a lot in future exercises.*

Chapter 3

Day 3 - Functions and Logical Control With If

3.1 The Structure of an R Function

Functions in R have a well-defined structure: the `function` keyword, followed by `variables`, followed by a code block with a `return` statement. For example,

is a simple function that squares a value. Functions, just like values, can be stored in variables. So we would probably have written our squaring function *and* stored it in a variable:

To use our function, we use the typical `f()` notation as you have seen in your mathematics courses.

```
[1] 4
```

```
[1] 9
```

```
[1] 25
```

For most functions, the code block will have multiple lines and will be much more complex. In those cases, we indicate the beginning and end of the code block using curly braces, `{` and `}`.

Functions may take more than one input.

When calling them you must provide all the inputs.

```
[1] 6
```

```
[1] 25
```

And functions can return any sort of R object you like. Here is an example of a

function that returns a vector.

Testing, we have

```
[1] 5 5 5 5 5 5
```

```
[1] "dog" "dog" "dog"
```

Note that because the inputs have no types specified, the function can be applied to a wide variety of data types. This is good - flexibility! - but it can make debugging your code tricky.

Summary: You've seen the basic structure of R functions. You'll get exercise writing your own in the coming exercises and we'll investigate some of the trickier aspects of writing functions in R. For those of you who have coded in another language like Java or Python, R functions may behave somewhat differently than you expect, so proceed with caution. In your written submission of this assignment just indicate that you have read this exercise.

3.2 Aside: Why Functions?

Why use functions at all? Why not just copy and paste the code block you would have written inside the function everywhere you need it?

Let's first revisit why we use variables. A variable stands for a value and can be used in place of that value in code. In a chunk of code like

```
[1] 16
```

if I want to rerun this code for a different value of `a`, like `5`, I can change the value of `a` in one location only and it updates everywhere else it is needed. No copy-paste at all. And there are dangers in copy paste! Your friend Gordon, trying to be clever, writes the code block above without any variables at all...

```
[1] 16
```

and he arrives at the correct answer. But then he needs to change all of the `a` values to `5`, and so he tries to copy-paste. He uses Find... Replace to find all of the `2`'s and replace them with `5`'s. What happens?

```
[1] 15650
```

Okay, so in trying to be clever, Gordon has replaced the `2`'s with `5`'s... and he replaced all of the `2`'s in the equation formulas as well. That is, the first bit of his formula no longer looks like `a^2+2` but instead like `a^5+5`. This is just one danger when trying to avoid variables; editors cannot distinguish between the numbers you want to replace and other numbers that are part of formulas if they are the same number! *And before you sneer and swear you would never do this, just be aware that one of my students did. On his senior capstone project. With many hundreds of lines of code. Costing him two weeks of debugging time.*

Moral of the story? We use variables because they let us re-use a value many times in our code and they let us update that value by editing one location only.

And that brings us to functions. **Functions are to code what variables are to values**. Functions provide blocks of code that we can re-use many times, and they let us update that code by editing in one location only. This can be incredibly helpful when we need to perform the same operation to every entry in a vector with thousands of pieces of data.

Additionally, we use functions to logically organize computations into understandable "chunks". This makes our code much more readable.

As an example, suppose I want to check every entry of a character vector to see whether it contains the string `data`. I can write a function to do that!

And I can check it on a few examples like

```
[1] TRUE
```

and like

```
[1] FALSE
```

So now I have a function with a meaningful name, `check_for_data`, instead of a function with a less memorable name, `grepl`. And, suppose I encounter some records in my data set like `"THERE IS DATA HERE"`. Checking my function on this string,

```
[1] FALSE
```

I find that it doesn't behave as I would like it to because it distinguishes the case of the word. I can easily return to my definition of my function and alter it:

In practice we would not have done this in a new code chunk, but simply alter the function in place earlier. And now my example behaves as I would like:

```
[1] TRUE
```

Code readability plus updating code in one location only! That's why we use functions.

Summary: You've read a bit about why we use functions in computer science. In you assignment submission you can simply indicate that you have read this exercise.

3.3 Coding Mathematical Functions

Write an R code chunk defining each of the following mathematical functions as a function in R. You should test each of your functions on a few inputs to make sure it works, and demonstrate your tests. For example, given the function $f(x) = |2x - 3|$, we can write the code

```
[1] 7
```

```
[1] 3
```

(a) $f(x) = 3 + 2(x-1) + 7(x-1)^2$

Test on input 2:

```
[1] 12
```

(b) $g(x) = \dfrac{3x^2}{2+x^2}$

(c) $h(x) = 2e^{0.1x}$ *Hint: Use the* `exp()` *function in R.*

(d) $p(t) = 2\ln(t-5)$ *Hint: Use the* `log()` *function in R.*

Summary: You have coded some simple mathematical functions. These can be useful when transforming data stored in vectors.

3.4 Named Inputs to R Functions

Functions in R have one quirk that is shared by few other programming languages. The function inputs are *named*, and using those names overrides the order in which you type in the inputs. Here is a simple example:

In almost any other programming language, we would interpret this function in plain language as "the function that takes two inputs, ignores the second, and returns the first one". And indeed, that is what R seems to do.

```
[1] 5
```

```
[1] "dog"
```

But now let's type in the inputs slightly differently, specifying which is `x` and which is `y`.

```
[1] 1
```

So here, even though 5 is the first input, it is discarded, and the value that has been specified as the `x` value is the one that is returned. So how do we interpret this function's behavior in plain language? This is "the function that takes two inputs `x` and `y` and returns `x`, with a default behavior of assuming the first input is `x`, unless the user specifies which input is `x` with a name." So, that is somewhat more complicated because it assumes a default behavior in the absence of information from the coder.

R will produce an error if multiple inputs are provided for a single variable (try it!)

or if a variable that is used is not specified (try this one too!)

However, if all of the needed variables are present, the function will behave normally:

```
[1] 5
```

But, if more than the needed variables are present, the function will produce an error (try it!).

Summary: Variable inputs to functions carry names and these can be used to reorder the inputs to a function however you like. In your written submission for this assignment, indicate that you have read this exercise.

3.5 Default Input Values

You can specify default values for input variables to functions using `=` in the function's variable list. For example,

sets a default value of `3` for the variable `y`. If the user specifies `y` then `f` returns `x*y`.

```
[1] 20
```

If the user does not specify `y`, then `f` assumes `y` is `3` and computes `x*y` anyway.

```
[1] 15
```

Default input values are tremendously useful when one input will be a certain value most of the time. Defaults are also useful when the function will normally make a decision about the value of a variable based on the other variables, and experienced users can override that decision by setting their own value. And defaults are also useful for setting options, such as whether to print an output to the screen or to a file.

Summary: Default values for input variables are simple and powerful tools for expanding the behavior of functions. For your written submission for this assignment, indicate that you have read this exercise.

3.6 Function Scoping

Can a function use a variable defined outside of the code block of the function? What variables can a function modify? How does the function variable naming affect other variables defined before the function? In this exercise, run the code blocks to see what happens; some produce errors.

(a) Execute the code block and answer the following. We stored `2` in the variable `x` before defining `f`; did that affect `f` in any way, or `x` in any way? *Note*: The `x` used as the input to `f` can be thought of as being bound to `f`. So the `x` in `function(x)` is not the same object as `x`, it's more like `f~x`, and thus is distinct from `x` and treated differently.

```
[1] 2
```

```
[1] 9
```

(b) Can we change the value of an input variable inside the body of a function? Execute the following to find out.

```
[1] 25
```

(c) Can we use a variable defined before a function in the body of the function?

```
[1] 15
```

(d) Can we change a variable defined before a function in the body of the function? In what sense?

```
[1] 25
```

```
[1] 3
```

Note: This behavior can be confusing. When `y` is used inside the body of `f`, we are more or less creating a new copy of `y` bound to `f`, sort of `f~y`. We changed `f~y` inside the body of `f` and used it. But, we didn't change `y`.

(e) But really, can we change a variable defined before a function in the body of the function?

```
[1] 25
```

```
[1] 5
```

Note: This is almost exactly the same code, replacing `<-` by `<<-`. The double arrow head tells R to refer back to the earlier defined `y` and not use `f~y`. This is both a very dangerous and very useful tool. Dangerous because usually functions are limited to producing outputs and not changing earlier defined variables. Useful, because it can change the values of very large data sets in place, without making an entirely new copy `f~y` of a massive data set. And that can both speed up your code and reduce your memory needs.

Summary: You have experimented with scoping of functions. Try some more experiments to make sure you understand how this works. And be very, very cautious when using `<<-`.

3.7 If Statements

The `if` statement in R causes a branching in code. Normally execution would process line 1, then line 2, and so forth. The `if` statement gives a few options for a line or chunk, and it asks for conditions for using one option rather than another.

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

```
[1] 9
```

Note: Here, only one option has been give for the middle line. However, it is wrapped in an `if` statement. The default other option is to do nothing. Try changing the value stored in `a` to 7 and re-running the code chunk to see what happens.

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

```
[1] 1.732051
```

Note: The placement of the `else` statement is important in R. If you place the `else` on the beginning of the next line down you will encounter an error.

(c) You may want several branches. Execute the following to see what happens.

```
[1] 1
```

Summary: You've learned about the `if` statement in R, a valuable code flow control tool.

3.8 Functions With If

(a) `if` control statements pair nicely with functions. Execute the following code chunk to see what happens.

```
[1] 0
```

```
[1] 1
```

Note: There are several ways to write this function. I gave a default value of 0 and stored it in `r`, and then asked if a certain exception held and if it did, then change `r`. The return is `r` at the end.

(b) Write an R code chunk defining the following function and test it on the inputs -2, 0.5, and 4. Use the `if` control statement.

$$f(x) = \begin{cases} 0 & x < 0 \\ 1 - x^2 & 0 \leq x \leq 1 \\ x - 1 & x > 1 \end{cases}$$

Output of tests:

```
[1] 0
```

```
[1] 0.75
```

```
[1] 3
```

Summary: You've seen examples of using `if` statements inside function bodies. This is very common when processing data for use.

3.9 Vectorizing Functions

One reason to write functions is to easily reuse the code performed by the function. And when we reuse that code, we would like it to be the case that we only have to alter the code once, and it will update in all locations where it is used. Functions do that!

When we reuse code, we often want to apply it repeatedly to all of the elements of a vector. Functions with this behavior are called *vectorized*. There is a logical problem with this, though. When `x` is a vector and `f` is a function, should writing `f(x)` apply the function to the vector `x` or apply `f` repeatedly to each individual element in `x`? As an example, when we compute `length(1:4)`, should the computation return `4` (the length of the vector) or `c(1,1,1,1)` (the length of each of the elements)? R uses the default behavior that the function applies to the vector. We can change that default behavior in two ways: define `f` using functions that are already vectorized, or explicitly tell R that `f` should be vectorized.

(a) Execute the followigng code chunk to see what happens. Is `f` vectorized?

```
 [1]   1   4   9  16  25  36  49  64  81 100
```

(b) Execute the following code chunk to see what happens (it should produce an error). Is `f` vectorized?

(c) Modify the previous code chunk as follows and execute it. Is `f` vectorized?

Note: Note the capitalization of the `Vectorize` command.

(d) Modify the code chunk in (b) in the following way and execute it. Does it produce output as if `f` is vectorized?

Summary: Vectorizing functions is useful when processing data stored in vectors.

Chapter 4

Day 4 - Applying Functions to Data

4.1 Review - Vectorized Functions

Remember that any function that returns simple data types such as numerics, logicals, or strings can be vectorized. Simply write the function, then use the `Vectorize` operation in R. The code will look something like

Summary: This is a brief review of the structure for vectorizing functions. In your written submission, indicate that you have reviewed this concept.

4.2 Clipping

In data analysis, we may want to clip the values of a vector of data so that all values below the stated minimum become the minimum and all values above the stated maximum become that maximum.

(a) Define a function with one input. Your function should return the input value if the input is between 0 and 100, 0 if the input is below zero, and and 100 if the input is greater than 100.

(b) Vectorize your function from (a) using `Vectorize`.

(c) Create a vector using `x <- 200*runif(100)-50`. Apply your vectorized function from (b) to the vector `x` and store the resulting vector in `y`.

(d) How many entries in `y` are 0 and how many are 100? *Hint*: Don't count by visual inspection. Use a filter and the `length` function.

Summary - You have defined a simple function that clips data values to a range.

4.3 Standardizing a Vector

Suppose that we have some numerical data, and we wish to "compress" it so that all values have been transformed to lie in the interval $0 \leq x \leq 1$. This operation is called **standardizing** the data. We can use the function

$$f(x) = \frac{x - m}{M - m}$$

where m and M are the minimum and maximum of the data, respectively.

Define a function that takes a vector as input and normalizes it. Test your function on the data `x <- (1:10)^2`.

Summary: You have written a simple function that standardizes data to the range $[0, 1]$*. In statistics, "standardize" is sometimes used to mean rewriting the data in units equaling the standard deviation and shifting the mean to 0. This is not that standardize!*

4.4 Transforming Data With a Vectorized Function

(a) Use `temps <- rnorm(100,20,4)` to simulate some temperature data. These temperatures are in degrees Celsius.

(b) Write a vectorized function that converts a temperature in degrees Celsius to a temperature in degrees Fahrenheit. *If you don't happen to know this off the top of your head, feel free to Google it. Or, just remember that it is a linear function and we know the corresponding freezing and boiling temperatures of water in both temperature systems.*

(c) Apply your temperature system conversion function from (b) to your vector `temps`.

Summary: You have written a simple unit conversion function.

4.5 Functions to Tweak R Commands

In homework 1, we encountered the `which.max` function. It has an important limitation: applied to numeric vectors, it only returns the index of the *first* maximum. For example,

```
[1] 3
```

returns 3 and not the index of the second maximum. But what if we would like to find all the indices of the maximum value in a vector?

(a) For the vector `x` above, write a piece of code that finds the maximum *value* in the vector and store it in the variable `M`.

(b) Write a logical filter that looks only for the maximum value in `x`. Something `==M`, perhaps? Once you have this filter, combine it with the `which` function to find all of the indices of the vector `x` holding the maximum value.

(c) Combine your work from (a) and (b) to write a function, `which.max.all`. Your function should take a numeric vector `x` as input and should return a vector of all of the indices of `x` containing the maximum value. For the vector `x` written above, your function should return `c(3,5)`. To get you started, your function should look something like...

Summary: You have altered an R function to better serve a particular application.

4.6 A Function Defining "Pretty Close"

In some data it is natural to allow for small measurement errors. So, we may not want to find only the indices of the maximum value of a vector, but indices containing values that are "pretty close" to the maximum value in the vector.

(a) Imagine that you have been given a numeric vector `x` and a positive (small) value `epsilon` (we'll use `epsilon` in part (b)). Write an R code chunk that stores the maximum value from `x` in the variable `M`. *Hint:* You may want to write your own vector `x` and number `epsilon` for testing purposes.

(b) What does the R expression `abs(x-M)<epsilon` return, and what does it do?

(c) Now, modify your work from exercise 5. You should produce a function that takes a numeric vector `x` and a positive value `epsilon` as inputs, and returns a vector of the indices in `x` containing values that are within `epsilon` units of the maximum value in the vector. Hint:* Your function here should look similar to your function from exercise 5. Major difference, from a coding point of view... there is an additional input.

Summary: You have written a function to find approximate maximizers.

4.7 A Function Defining "Within n Standard Deviations of the Mean"

Adapt your code from the previous exercise to write a function that returns the indices of the values in a vector that are within a specified number `n` standard deviations of the mean of the data in that vector.

4.8 A Function to Extract Information from Strings

In some cases, data that should be numeric is input as character data. This can happen when unwisely using any number of data collection methods, especially electronic entry forms.

For this exercise, let's suppose that users have entered their heights as a number, then a space, then a unit. Here is a sample from the data vector:

(a) Evaluate the code `strsplit("23 skedoo"," ")[[1]]`. What data type does it return, and what does it do?
(b) Evaluate the code `as.numeric("23")`. What does it do?
(c) Write a function that takes as input a string of the form `"23 skedoo"`: a number, then a single space, and then some characters. Your function should output the numeric part of the string as a numeric data value.
(d) Repeat (c), but build this second function to return the non-numeric part of the string.
(e) Now we'll address the actual formatting problem. Write a vectorized function that takes as input a string formatted as seen in the `heights` vector and as described in (c) and (d). Your function should output the number of inches; if the units in the data are in meters, your function will need to convert the numeric value to inches before returning it. Once you are done, apply your function to `heights` as a test.

Note: Differently formatted data is a very common problem to face as a data analyst. This can result from poor data collection technique, but more often results from combining data sets acquired from different sources. Learning how to manipulate strings is an important data analytics skill, and is useful in the data cleaning process. Google "regular expressions" for more information than you will want to know; this rabbit hole goes deep.

Summary: You have written a function to deconstruct a string and process its information based on its contents. This is a useful process in data analysis.

4.9 Coding Text Fields With Logical Flags (One-Hot Encoding)

Data might be in the form of text instead of numbers. For example, a common data field in ecology and in medicine is a *notes* section, where the observer makes some quick notes on whatever they are seeing.

We might want to *code* text data with a logical flag. A common action is to create a logical vector that corresponds to our character vector where `TRUE` indicates the presence of a certain word and `FALSE`indicates its absence.

Define the test vector `test <- c("doggos","cats","Bears","Doggies")`. We

want to write a function that flags each entry as `TRUE` whenever `dog` is present and `FALSE` otherwise.

(a) Evaluate the code `grepl("alpha",c("alphabet","al"))`. What does it do?

(b) Evaluate the code `grepl("al",c("alphabet","al"))`. What does it do? Compare to (a).

(c) Evaluate the code `grepl("Al",c("alphabet","al"))`. What does it do? Read the docs for the `grepl` function; if the case of the letters in the pattern does not matter to you, is there an easy way to fix that, i.e. to ignore the case?

(d) Write a vectorized function that takes as input a vector of strings (like `test`). Your function should return a logical vector where `TRUE` indicates the presence of the string `dog`, and case should not matter.

(e) This is an incredibly useful tool. Write a short description of what this exercise is getting at. This will help set it in your memory; it will almost certainly come up again, and seems to come up often in the projects students select in the course.

Summary: You have coded text fields to indicate the presence or absence of a small string. This is a common operation in data analysis.