A practical introduction to R and RStudio for applied epidemiologists

Kelly Shaw, EIS 2017

0) A note about philosophy.

The first rule of R is probably to always talk about R. It is the most versatile analysis tool available, it's actually really simple (this guide will hopefully show you), and it's free!! I mean, come on!

Other than that, as you are learning R I would highly recommend this rule: Google early. Google often. In fact, if anyone wants to make R club t-shirts, that (in Latin) should probably be the motto.

Seriously. I have been programming heavily in R for 5+ years and still have to google every day to accomplish tasks. No matter what your problem or question, thousands of other people have probably had the same issue and posted it somewhere you can go and ~~steal~~ learn from the solution. Learning *what* to google is a skill unto itself, so I would highly encourage practicing it as you go along. Even if you think no one would have ever thought of it, you might be surprised. Ask "how do I do [thing I want] in R". Bookmark useful answers but most importantly, ALWAYS SAVE YOUR WORKING CODE. Cannibalize it *ruthlessly* and adapt it as you face similar problems in the future.

Lastly I will note that the following code is just one person's way of doing R. There are multiple ways of doing the same tasks, so don't feel constrained by what I have written. In fact, there are probably much more efficient ways of coding than I have presented here. My background is *not* in computer science, it is in basic science. The main target audience for this guide is folks who are completely new to programming like I was when I started, or those who might be switching to R from another language.

This guide assumes that you already have R and RStudio installed and working, because I am 100% unqualified to help with your Fort-Knox-level government computer security. You're going to have to rely on IT for those kind of issues.

All that being said, we're going to launch right into:

1) Getting to know RStudio

The first thing to do is open Rstudio and get oriented to the different panes.

- First on the left is the "console". This is where the code that you type gets executed to produce output.
- On the right at the top is the "environment" pane. Whenever you create data objects in R, they are listed here.
  - you can also access the history of commands you've run in your R session (the first time I ever looked at this tab was preparing this document)
- In the right lower corner, I usually keep this on the "plots" tab, so this is where any graphics that you create are shown.
  - but you can also see the contents of the folder you're working in under the "files" tab
  - you can see the different packages that are running in the "packages" tab (packages are what you can install to get more useful R functions [will talk more about later])
  - there is also a "help" tab where you can see R documentation for functions (I would recommend just going to google to find a more understandable resource - to me the documentation seems geared toward either computer scientists or mathematicians and rarely helps me)
  - I'll be honest, I have no idea what the "viewer" tab is for. I really keep it on "plots" all the time

2) Working with scripts

Now, the very first thing when you're going to be doing work in RStudio is to open a script (hit the white document button with the green plus underneath "file" then click "R Script"). This is basically like keeping a lab notebook, or having a protocol for your analysis. You type your code in here, then hit "Run" (I always just use Ctrl + R on my Windows computer; I think newer RStudio versions use Ctrl + Enter) and the code will execute in the console and you can see the output. Save all the data manipulations and statistics you do with your given data set and you should have a completely reproducible analysis. Just hit the save button in the script pane, or the save button under the menu bar, or click "File" then "Save" and your script will save as a ".R" file. Any person could then take your starting data and this script containing your code and get the same result you got.

3) Leaving yourself a trail of breadcrumbs

Let's start out by making a note to ourselves about what we're going to do first. Using the number sign (or "hash mark") will tell R not to run this line of code. In programming this is called a comment. So again to improve the shareability of our code, it's good to describe what different sections of our code do.

```
##### set working directory FIRST THING EVERY TIME
```

4) Setting a working directory

As I wrote in that comment, the first thing you should do is set a working directory. This is going to be your base of operations for your R session. Whenever I'm starting a new project, I create a folder on my desktop for all my data files. I set that folder as my working directory to easily read in data from that folder and also write data or graphs back out to that folder without losing them.

We can use the function getwd() to see our current working directory. An R function is just a command that performs a pre-programmed action. To "call" a function you simply type the name of the function, and then parentheses. Often we have to put something in parentheses for the function to act on, but not in this case.

```
getwd()
```

```
## [1] "C:/Users/ajz36853/Desktop/R_intro"
```

We can use our current working directory as a guide and take advantage of RStudio's really useful autocomplete function to choose a better folder. Type setwd("C://") and then hit the "Tab" key on your keyboard. Use your arrow keys to select "User", then your ID, and then navigate through your Desktop or Documents to the folder you want. If you're working on a government computer, it might give you some error messages because of random permissions issues; just re-run getwd() to ensure your command was successful.

```
setwd("C://Users/ajz36853/Desktop/R_intro/")
```

5) R at its most boring

Now we can start playing around in R! One of the simplest things you can do in R is just treat it like a calculator. Add, subtract, multiply, raise something to a power. R observes the usual order of operations.

```
5 * 10 + 30 / 2
```

```
## [1] 65
```

```
(8 - 4) ^ 3
```

```
## [1] 64
```

6) Simple functions

However some mathematical operations require functions. The function will be performed on whatever you type into the parentheses.

```
sqrt(64)
```

```
## [1] 8
```

```
sqrt(6 * 4 + 80 / 20 - 12)
```

```
## [1] 4
```

```
log10(100)                          # log() will give the natural log
```

```
## [1] 2
```

```
exp(2)
```

```
## [1] 7.389056
```

```r
abs(-48)
```

```
## [1] 48
```

7) Creating variables

On to where the real magic happens! Variables and objects! In R we can store values or entire data sets as “objects” in R. We just use the “<-” which means “assign what’s to the right of this arrow to the name on the left of this arrow.” As you assign values you will see them show up in the top right pane. You can view them by just typing their name again.

```r
a <- 10
a
```

```
## [1] 10
```

```r
b <- 5
b
```

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

Numbers are stored as “numeric” type variables. (Note: using a single “=” will behave in the same manner as “<-”, but most people only use “=” for its special purpose of changing options within functions; for clarity it is best just to use “<-” for assignment.)

You can perform any mathematical function on a numeric variable just like the number itself.

```r
a + b
```

```
## [1] 15
```

You can even store the results in another variable if you want.

```r
c <- a + b
c
```

```
## [1] 15
```

Logical variables are “TRUE” and “FALSE”, and are another feature of R. These variables are especially useful for subsetting data (which we will get to later).

```r
a > b
```

```
## [1] TRUE
```

```r
a < b
```

```
## [1] FALSE
```

```r
a == b              # equal to (if you only use one = you will assign the value of b to a!)
```

```
## [1] FALSE
```

```r
a != b              # not equal to
```

```
## [1] TRUE
```

```r
d <- a >= b * 2     # store the answer to whether a is greater than or equal to b times 2
d
```

```
## [1] TRUE
```

You can also create character type variables by using quotation marks around text or numbers.

```r
e <- "yes!"
e
```

```
## [1] "yes!"
```

Characters without quotation marks are interpreted as objects by R, and it will generally politely inform you that it can’t find that object.

```r
e <- yes
```

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

Whenever you create variables, you can use str() to see what type and what values of data are contained in the object.

```r
str(a)
```

```
##  num 10
```

```r
str(d)
```

```
##  logi TRUE
```

```r
str(e)
```

```
##  chr "yes!"
```

You can remove objects by using rm().

```r
rm(b)
```

But be aware that you can easily rewrite any object–R won't warn you when you re-use a name!

```r
e
```

```
## [1] "yes!"
```

```r
e <- "no!"
e
```

```
## [1] "no!"
```

For that reason, it can be good to make object names in your analysis meaningful. You can really name your data objects anything… (just use your name instead of mine and R can help you with your daily affirmations!)

```r
kelly <- "awesome"
```

Just don't ever use spaces in object names or column names because R won't know what to do with them. Luckily R will admit this.

```r
kelly is <- "awesome"
```

```
## Error: <text>:1:7: unexpected symbol
## 1: kelly is
##           ^
```

You can use underscores, periods, or capitalization instead of spaces.

```r
kellyIs <- "awesome"
```

Text containing spaces can be stored within a data object though.

```r
kelly <- "is awesome"
```

8) Vectors

Now let's create a more complicated object. Let's make a "vector". A vector stores multiple values. You can think of it as like a column or a row of data. To vectorize some values, use the c() function, which tells R to "combine those values".

```r
f <- c(5, 10, 20, 40)
f
```

```
## [1]  5 10 20 40
```

Now you can perform operations on the entire vector, such as multiplying everything by a constant.

```r
f * 2
```

```
## [1] 10 20 40 80
```

Or you can see which variables in your vector meet a given condition.

```r
f > 15
```

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

Or you can take your vector and perform mathematical operations on it using a vector of the same length. The length just needs to match! You can always check the length of a vector with the function length() first.

```r
length(f)
```

```
## [1] 4
```

```r
f * c(1000, 2, 1, 2)
```

```
## [1] 5000   20   20   80
```

If the lengths do not match R should helpfully yell at you. However, the operation will still execute! This can extremely dangerous, because R doesn't just leave the last number alone, it cycles back through the vector to apply the first value again.

```r
f * c(1000, 2, 1)
```

```
## Warning in f * c(1000, 2, 1): longer object length is not a multiple of
## shorter object length
```

```
## [1]  5000    20    20 40000
```

It's also important to be able to pull individual numbers back out of your vector, and you do this using brackets "[ ]" touching the object name. What you put in the brackets is the "index" of the value you want to pull out. In R, vector indices are in numeric order starting at 1. So the first value in the vector is 1, the second is 2, the third is 3, and so on.

```r
f[1]          # pulls the first value
```

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

```r
f[4]          # pulls the fourth value
```

```
## [1] 40
```

```r
f[7]          # pulls no value, because your vector is only 4 values long!
```

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

To pull more than one value, you can submit a vector containing the indices for the values you want using "c()"…

```r
f[c(1,3)]
```

```
## [1]  5 20
```

…or you can use ":" if the values are sequential numbers.

```r
f[1:2]
```

```
## [1]  5 10
```

The colon symbol ":" is a very useful tool in R. It is interpreted as a vector of the entire range of numbers starting at the first value and ending with the last value – this saves a lot of typing!

```r
1:100
```

```
##   [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
##  [18]  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34
##  [35]  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51
##  [52]  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68
##  [69]  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85
##  [86]  86  87  88  89  90  91  92  93  94  95  96  97  98  99 100
```

You can also combine these ranges with other indices using c() to combine them into a single vector.

```r
f[c(1:2,4)]
```

```
## [1]  5 10 40
```

Though in this case it is probably easier to use the minus sign "-" to simply exclude the third value.

```
f[-3]
```

```
## [1]  5 10 40
```

You can also store characters in a vector…

```
poss_answers <- c("yes", "no", "unsure")
```

…and access them the same way.

```
poss_answers[2:3]
```

```
## [1] "no"     "unsure"
```

But all variable types in a vector have to match! If you store numbers and characters in a vector, everything will be treated as a character. (Even though 20 is a number, using quotation marks tells R to treat it as a string)

```
g <- c(5, 10, "20", 40)
str(g)
```

```
##  chr [1:4] "5" "10" "20" "40"
```

9) Matrices

Okay now let's get two-dimensional with our data. We can do this by combining vectors…

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

side-by-side ("column bind")…

```
d <- cbind(a, b)
d
```

```
##      a b
## [1,] 1 4
## [2,] 2 5
## [3,] 3 6
```

or above-and-below ("row bind") each other.

```
e <- rbind(a, b)
e
```

```
##   [,1] [,2] [,3]
## a    1    2    3
## b    4    5    6
```

You can use the View() function to look at these objects in the script pane. These 2-D data objects are called "matrices", and like vectors, they can only be one type of data – all numeric, all character, or all logical. Like with vectors, if you combine types, R will not treat your data like you might want it to. Everything in the following example becomes a character.

```
a <- c(1, 2, 3)
b <- c("yes", "no", "no")
c <- c("5","3","1")
d <- cbind(a, b, c)
d
```

```
##      a   b     c
## [1,] "1" "yes" "5"
## [2,] "2" "no"  "3"
## [3,] "3" "no"  "1"
```

Let's think about this as an epi data set. Perhaps the vector "a" represents 3 individuals whose IDs are 1, 2, and 3. And suppose the vector "b" is their response to whether they were exposed to something, and "c" is the number of illness symptoms they experienced.

First, we can change the column names to something more valuable by supplying a vector to replace the existing colnames() vector.

```
colnames(d)
```

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

```
colnames(d) <- c("ID", "exposure", "symptoms")
```

And we have the beginnings of a real line list!

10) Data frames

But with all the different types of variables that we might want to have in a data set, it's not useful to force them to be only one type. Luckily there's a more sophisticated way of storing data. Data frames, like matrices, are two-dimensional, but each column can store a different type of variable. Let's turn our matrix into a data frame (in online examples you will most commonly see "df" used as the generic object name, but like with other values or vectors you can use any name).

```
df <- data.frame(d)
```

Now you can see in the top-right pane that df shows up with 3 observations (rows) of 3 variables (columns). We can also obtain this information using functions nrow() and ncol()

```
nrow(df)
```

```
## [1] 3
```

```
ncol(df)
```

```
## [1] 3
```

Clicking the little arrow to the left of df's name shows the names of the 3 columns. Maybe now we don't like the name of the ID column, so let's change it to something else for kicks.

```
colnames(df)[1] <- "person_ID"
```

Let's take a minute to learn about the different ways you can select data from a data frame. Because a vector is one-dimensional, you only need to supply one index to pull out a value (e.g., poss_answers[2]). For a data frame, there are spaces for two indices, the ROW, then COLUMN, always in that order (e.g., df[1,1]). If you want an entire row or column, you can just include the required comma but leave an index on either side blank.

```
# remember [ROW, COLUMN]
df[1,]
```

```
##   person_ID exposure symptoms
## 1         1      yes        5
```

```
df[,1]
```

```
## [1] 1 2 3
## Levels: 1 2 3
```

If you aren't sure of the column number but know the name, you can use that to call the column.

```
df[,"exposure"]
```

```
## [1] yes no  no
## Levels: no yes
```

The easiest way to call a single column is to use the data frame name followed by a dollar sign followed by the column name. This way, autocomplete can help you out!

```
df$exposure
```

```
## [1] yes no  no
## Levels: no yes
```

Just like in a vector, you can also call multiple rows or columns at once by supplying a vector of numbers as indices.

```r
df[1:2, c(1,3)]
```

```
##   person_ID symptoms
## 1         1        5
## 2         2        3
```

If you assign this to a new data frame, you have just taken a subset of your data (more on this later).

```r
df_sub <- df[1:2, c(1,3)]
```

I mentioned that columns in data frames can store multiple types of variables. Let's look at our data set to see what types of variables we have right now. We can see this information by expanding the blue arrow to the left of the df object name or by using the function str().

```r
str(df)
```

```
## 'data.frame':    3 obs. of  3 variables:
##  $ person_ID: Factor w/ 3 levels "1","2","3": 1 2 3
##  $ exposure : Factor w/ 2 levels "no","yes": 2 1 1
##  $ symptoms : Factor w/ 3 levels "1","3","5": 3 2 1
```

Because d was a character matrix with limited values in each column, all of our columns were converted to "factors". Factors can also be called categorical variables. Order can either be important (ordinal variables like age groups) or unimportant (yes/no or eye color, for example). We can use the summary() function to quickly see counts of each category.

```r
summary(df$exposure)
```

```
##  no yes
##   2   1
```

Whether order is important to you or not, it is important to note that order exists for factor variables. We can see this order using levels().

```r
levels(df$exposure)
```

```
## [1] "no"  "yes"
```

Factor levels are automatically assigned in alphabetical order, so for exposure the levels are "no", then "yes". As a factor, "no"s are therefore coded as "1"s (because R starts counting at 1) and "yes"s are "2"s. We can change the order if we would like (more on that later).

We are fine with exposure as a factor and also person_ID as a factor because it also is categorical.

However, we might want to do something mathematical with the symptoms column like find the average number of symptoms. We can't do that with a factor.

```r
summary(df$symptoms)
```

```
## 1 3 5
## 1 1 1
```

So let's convert it to a number. We just have to be sure to first convert the symptoms column to a character variable first; otherwise as.numeric() will provide the factor code (for example with the exposure column a "no" would become a "1" and "yes" would become "2") rather than the value itself.

```r
df$symptoms_num <- as.numeric(as.character(df$symptoms))
summary(df$symptoms_num)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##       1       2       3       3       4       5
```

We can also replace the original column in the data set, but BE CAREFUL with this. Luckily if you make a mistake you can just re-read your data set into R, but it's always important to be on the look-out for your mistakes! Double- and triple-check your output is what you expect!

```r
df$symptoms <- as.numeric(as.character(df$symptoms))
```

11) Working with external data sets

Now, as epis we don't want to (and really shouldn't even if we wanted to) hand-type our data sets into R. Thankfully R makes it easy to import our external data sets.

First I'd like to briefly describe some of the common types of files, and why I exclusively use .txt and .csv files. There are three different versions of the same practice data set in the R_intro folder. When you open the .txt file using Notepad you will see that this file is tab-delimited, which just means that each column is separated in space by a tab character, and each row is on a new line. When you open the .csv file using Notepad you will see almost the exact same thing, except instead of tabs each column is separated by commas. Now we'll open the Excel Workbook with Notepad and you should see it looks… interesting. These files encode not only data, but all of the formatting you put in the file like fonts, cell colors, borders, etc. These files are not as straightforward to interpret, and I'm always worried that the formatting will throw off my data somehow. Therefore even if I'm working on a data set in Excel, I will always save the file as either .csv or .txt.

I'll therefore focus on how to import these two types of data using read.csv() and read.txt().

We've focused on simple functions where we didn't need to change any options up until now, but now lets look at read.csv() and read.table(), which have additional "arguments" we might want to supply to the functions to change their behavior. We can simply type a question mark followed by the function name to bring up the help page for these functions.

```
?read.csv()
```

The help documentation is often very overwhelming but is also occasionally helpful. Under the "usage" section you can see the functions along with almost every argument that can be passed to the function with its default value. The "arguments" section gives more detail about what different options are available. You *only* need to use arguments if you want to change the default.

For example, if we want to read in a .csv file, we can generally just accept all of the default settings, because our .csv file doesn't have any weird extra formatting. The only thing we have to supply is the file name. This is another great place where auto-complete can help.

```
df <- read.csv(file="practice_data.csv")
```

Now we can View() or just click on the object name in the top right pane to eyeball our data and make sure it was read in correctly. The read.csv() option knows the correct symbol that separates columns by default (the default argument is sep=",") and also uses the first line of the data set as the column names by default (default argument header=TRUE).

Reading in a text file requires us to specify those as additional options. We want to change the default separator we see in the documentation from "" to "\t" for a tab-delimited file. Additionally we want to tell R that it is TRUE our .txt file has a header so the first line in the file will be used as the column names.

```
df <- read.table(file="practice_data.txt", sep="\t", header=TRUE)
```

It's worth noting that while looking at the help file creating this guide I also learned about the read.delim() function, which already has these separator and header defaults, so "df <- read.delim(file="practice_data.txt")" should do the exact same thing in an easier fashion.

Often in epi work your data set may have some kind of free text field containing commas, quotation marks, or other characters that could confuse R when it's trying to read in your file. Usually you'll get an error that a certain line has too many columns. You can use the extra arguments passed to this next function to tell R not to interpret the confusing characters as delimiting (new-column) characters.

```
df <- read.table(file="practice_data.txt", sep="\t", header=T, quote="", comment="")
```

Now that we have our data in R, let's revisit some of the ways we can look at a data frame.

```
nrow(df)
```

```
## [1] 87
```

```
ncol(df)
```

```
## [1] 12
```

```
# remember [ROW, COLUMN]
# view first row
df[1,]
```

```
##   completed age    state attend_event classification illness_onset
## 1         1  52 Virginia            1           Case     10/1/2017
##   sum_symptoms fever hcp hospitalized food_A food_B
## 1            7     1   1            1      1      2
```

```
# view first column
df[,1]
```

```
##  [1] 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0
## [36] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [71] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

```
colnames(df)
```

```
##  [1] "completed"      "age"            "state"          "attend_event"
##  [5] "classification" "illness_onset"  "sum_symptoms"   "fever"
##  [9] "hcp"            "hospitalized"   "food_A"         "food_B"
```

```
colnames(df)[1]
```

```
## [1] "completed"
```

```
# view first column using column name rather than number
df[,"completed"]
```

```
##  [1] 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0
## [36] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [71] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

```
df$completed
```

```
##  [1] 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0
## [36] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [71] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

```
summary(df$completed)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.000   1.000   1.000   0.954   1.000   1.000
```

```
# the field is not meaningful as numeric so let's look at it as a factor
summary(factor(df$completed))
```

```
##  0  1
##  4 83
```

It appears we have 4 records that are not actually complete! Let's get rid of these. We will use the which() function, which returns a vector containing the indices of your data frame for which your specified condition is TRUE.

```
which(df$completed==1)
```

```
##  [1]  1  2  3  4  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
## [24] 27 28 29 30 31 32 33 34 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
## [47] 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73
## [70] 74 75 76 77 78 79 80 81 82 83 84 85 86 87
```

If we use length() to see how long this vector is, we see it is the same length as the number of "1"s ("yes"s) we saw in the variable summary.

```
length(which(df$completed==1))
```

```
## [1] 83
```

Now we can easily supply the vector of these numbers to subset our data set to only have completed surveys.

```
df_sub <- df[which(df$completed==1),]
```

You can use any logical condition to subset your data. In this case, saying you want rows that aren't equal to 0 is the same as saying you want the rows equal to 1. Just use quotation marks for character or factor variables.

```r
df_sub <- df[which(df$completed!=0),]
df_sub <- df[which(df$classification=="Not ill"),]
```

You can also subset columns in the same step or use multiple steps.

```r
df_sub <- df[which(df$completed==1), c(1:3, 5, 8, 10:ncol(df))]
```

You can easily do descriptive statistics in R. For example, with a continuous variable:

```r
summary(df_sub$age)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's 
##    1.00   39.00   53.00   49.96   63.00   92.00       2
```

While the summary function automatically separates out NAs, some of these other functions require na.rm=T to be passed as an argument to remove NAs and work properly.

```r
max(df_sub$age, na.rm=T)
```

```
## [1] 92
```

```r
min(df_sub$age, na.rm=T)
```

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

```r
mean(df_sub$age, na.rm=T)
```

```
## [1] 49.96296
```

```r
sd(df_sub$age, na.rm=T)
```

```
## [1] 18.72595
```

```r
fivenum(df_sub$age)
```

```
## [1]  1 39 53 63 92
```

```r
stem(df_sub$age)
```

```
## 
##   The decimal point is 1 digit(s) to the right of the |
## 
##   0 | 12467
##   1 | 6
##   2 | 3478
##   3 | 12335567779
##   4 | 0023346699
##   5 | 0001112223345555677889999
##   6 | 012333445566667899
##   7 | 002245
##   8 | 8
##   9 | 2
```

And here is an example using a factor variable.

```r
summary(df_sub$state)
```

```
##       Delaware       Maryland     New Jersey North Carolina   Pennsylvania 
##              8             22              3              1              9 
##       Virginia  West Virginia 
##             39              1
```

```r
summary(df_sub$state)/nrow(df_sub)
```

```
##       Delaware       Maryland     New Jersey North Carolina   Pennsylvania 
##     0.09638554     0.26506024     0.03614458     0.01204819     0.10843373 
##       Virginia  West Virginia 
##     0.46987952     0.01204819
```

You can even make this appear as a rounded percent.

```
round(summary(df_sub$state)/nrow(df_sub) * 100, digits=1)
```

```
##       Delaware       Maryland     New Jersey North Carolina   Pennsylvania 
##            9.6           26.5            3.6            1.2           10.8 
##       Virginia  West Virginia 
##           47.0            1.2
```

If you want to look separately at the ill and not ill, that is fairly easy too; just run the function on each subset.

```
summary(df_sub[which(df$classification=="Not ill"),"state"])
```

```
##       Delaware       Maryland     New Jersey North Carolina   Pennsylvania 
##              8             13              2              1              6 
##       Virginia  West Virginia           NA's 
##             19              0              4
```

```
summary(df_sub[which(df$classification=="Case"),"state"])
```

```
##       Delaware       Maryland     New Jersey North Carolina   Pennsylvania 
##              0              9              1              0              3 
##       Virginia  West Virginia 
##             20              1
```

Just make sure to change your denominator accordingly.

```
summary(df_sub[which(df$classification=="Not ill"),"state"]) / nrow(df_sub[which(df$classification=="Not ill")
```

```
##       Delaware       Maryland     New Jersey North Carolina   Pennsylvania 
##     0.15094340     0.24528302     0.03773585     0.01886792     0.11320755 
##       Virginia  West Virginia           NA's 
##     0.35849057     0.00000000     0.07547170
```

One of the most common things you may find yourself wanting to do is create new variables in R – maybe to make a continuous variable binary, or combining information from multiple columns to calculate a new value or classify a case of disease.

Let's turn age into age groups as an example of making a new variable. I generally start by creating a column for the variable that is a full column of "NA"s. (New columns are simply added at the end of your data frame.)

```
df_sub$age_group <- NA
```

Now we are going to select the subset of the data frame who is younger than 18, and then for the column we are going to direct R to the new variable we created. We are going to assign the value "child" to everyone under 18.

```
df_sub[which(df_sub$age<18), "age_group"] <- "child"
```

We repeat this for the "middle age" group, but now we require two conditions – that they are older than one age and younger than another age.

```
df_sub[which(df_sub$age>=18 & df_sub$age<80), "age_group"] <- "adult"
```

Lastly we assign the final age group we've decided on.

```
df_sub[which(df_sub$age>=80), "age_group"] <- "exceedingly wise"
```

There is one important exception to how we call data, and that is for "NA"s. NAs, often excluded automatically from many functions, should be called with is.na(). This function returns a logical vector where NAs are marked as TRUE and all other values are marked with FALSE.

```
is.na(df_sub$age)
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [23] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
## [34] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [45] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [56] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [67] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [78] FALSE FALSE FALSE FALSE FALSE  TRUE
```

```
df_sub[is.na(df_sub$age),]
```

```
##    completed age    state classification fever hospitalized food_A food_B
## 34         1  NA Virginia           Case     0            0      3      2
## 87         1  NA Maryland        Not ill    NA            0      3      2
##    age_group
## 34      <NA>
## 87      <NA>
```

Now we can see a breakdown of the age groups we were interested in.

```
summary(factor(df_sub$age_group))
```

```
##            adult            child exceedingly wise             NA's
##               73                6                2                2
```

If we were going to do anything with this variable we would likely want to fix the factor ordering to be from youngest to oldest instead of alphabetical.

Let's recode a different variable to optimize it for statistics later. This data set was exported from REDCap, where exposure answers were coded as 1=Yes, 2=No, and 3=Unsure. We likely want to change this coding so that it is ("Yes", "No", NA) or (1, 0, NA). Changing the "Unsure"s to NAs will ensure that the values for this group are not considered in our statistical testing.

Here is one way to directly change the values:

```
summary(factor(df_sub$food_A))
```

```
##  1  2  3
## 34 44  5
```

```
df_sub[which(df_sub$food_A==1),"food_A"] <- "Yes"
df_sub[which(df_sub$food_A==2),"food_A"] <- "No"
df_sub[which(df_sub$food_A==3),"food_A"] <- NA
summary(factor(df_sub$food_A))
```

```
##   No  Yes NA's
##   44   34    5
```

Another way to approach the problem is once we've changed the NAs...

```
summary(factor(df_sub$food_B))
```

```
##  1  2  3
##  3 78  2
```

```
df_sub[which(df_sub$food_B==3),"food_B"] <- NA
```

Look at what the factor levels are.

```
levels(factor(df_sub$food_B))
```

```
## [1] "1" "2"
```

The factor() function will convert this variable to a function, and we can specify new labels for the two levels we just saw. It is EXTREMELY important that you supply the labels in the correct order, otherwise you will mislabel your values!

```
df_sub$food_B <- factor(df_sub$food_B, labels=c("Yes","No"))
```

Again, check your work to make sure the result is what you expect it to be.

```
summary(df_sub$food_B)
```

```
##  Yes   No NA's
##    3   78    2
```

12) Exporting your data back out of R

Now that we have gotten the relevant subset and cleaned up our variables, we can write our dataset out to .csv, .txt, or other possible file types. Again, I use the first two file types exclusively. I generally make sure to export my cleaned data set so that I don't have to go back through all of the data manipulation steps every time I mess up my statistics steps :)

```
write.csv(df_sub, "mydata_cleaned.csv", quote=F, row.names=F)
write.table(df_sub, "mydata_cleaned.txt", sep="\t", quote=F, row.names=F)
```

13) Simple statistics in R

We can now look at doing some simple statistical tests in R.

First let's compare the means of two categories. For both a t-test and linear model, use the usual regression equation format y~x. In this example we are interested in the effect of case status on age.

```
##### T-test
t.test(df_sub$age~df_sub$classification)
```

```
##
##  Welch Two Sample t-test
##
## data:  df_sub$age by df_sub$classification
## t = 1.7813, df = 62.216, p-value = 0.07974
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.9186085 15.9656673
## sample estimates:
##    mean in group Case mean in group Not ill
##              54.70000              47.17647
```

Whenever you are running a model, be sure that your factors are organized how you want them. The comparison group will be whichever group is the first factor level; remember that by default the levels are set in alphabetical order.

```
##### Simple linear model
reg <- lm(df_sub$age~df_sub$classification)
summary(reg)
```

```
##
## Call:
## lm(formula = df_sub$age ~ df_sub$classification)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -50.700 -10.176   2.824  12.824  37.300
##
## Coefficients:
##                              Estimate Std. Error t value Pr(>|t|)
## (Intercept)                    54.700      3.374  16.211   <2e-16 ***
## df_sub$classificationNot ill   -7.524      4.252  -1.769   0.0807 .
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 18.48 on 79 degrees of freedom
##   (2 observations deleted due to missingness)
## Multiple R-squared:  0.03811,    Adjusted R-squared:  0.02594
## F-statistic:  3.13 on 1 and 79 DF,  p-value: 0.08071
```

```
levels(df_sub$classification)
```

```
## [1] "Case"    "Not ill"
```

We want the not ill to be our reference group, not those with illness! So let's change the order of our levels. Whereas previously we used the labels() argument to change just the appearance of our factor, in this instance we will use levels() and provide the order we want for our factor.

```
df_sub$classification <- factor(df_sub$classification, levels=c("Not ill", "Case"))
# and re-run our model
reg <- lm(df_sub$age~df_sub$classification)
summary(reg)
```

```
##
## Call:
## lm(formula = df_sub$age ~ df_sub$classification)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -50.700 -10.176   2.824  12.824  37.300
##
## Coefficients:
##                             Estimate Std. Error t value Pr(>|t|)
## (Intercept)                   47.176      2.588  18.229   <2e-16 ***
## df_sub$classificationCase    7.524      4.252   1.769   0.0807 .
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 18.48 on 79 degrees of freedom
##   (2 observations deleted due to missingness)
## Multiple R-squared:  0.03811,    Adjusted R-squared:  0.02594
## F-statistic:  3.13 on 1 and 79 DF,  p-value: 0.08071
```

Now we have the comparison group we prefer!

What if we want to test whether having a fever seems to be independent from being hospitalized? We can perform a chi-sqared test using chisq.test() with "outcome, exposure" specified in the parentheses.

```
chisq.test(df_sub$hospitalized, df_sub$fever)
```

```
## Warning in chisq.test(df_sub$hospitalized, df_sub$fever): Chi-squared
## approximation may be incorrect
```

```
##
##  Pearson's Chi-squared test with Yates' continuity correction
##
## data:  df_sub$hospitalized and df_sub$fever
## X-squared = 2.4967, df = 1, p-value = 0.1141
```

That's an interesting warning that R just threw. "Chi-squared approximation may be incorrect"? Let's look at the data to see why that may be the case. You can create a crosstab of the data by using the table() function, again with "outcome, exposure" in parentheses. The outcome is represented in rows and exposure in columns.

```
table(df_sub$hospitalized, df_sub$fever)
```

```
##
##      0  1
##   0 21  6
##   1  1  3
```

Oh, it appears that some of our cells have counts less than 5! So let's try a Fisher exact test instead.

```
fisher.test(df_sub$hospitalized, df_sub$fever)
```

```
##
##  Fisher's Exact Test for Count Data
##
## data:  df_sub$hospitalized and df_sub$fever
## p-value = 0.06274
## alternative hypothesis: true odds ratio is not equal to 1
## 95 percent confidence interval:
##    0.6381726 574.0700737
## sample estimates:
## odds ratio
##   9.538306
```

Okay, surely by now we've made the statisticians cry with a bunch of fast and loose statistical testing. I'll finally clarify that these data are from a cohort study, so we would rather use statistics that compare risk ratios rather than odds ratios in exposed and unexposed groups. Our basic R installation lacks the function required to test for association between exposure and case classification using risk ratios. We will therefore need to install a package that has this functionality. Literally thousands of packages are available with specialized functions. These are usually created by R users who face specific programming problems and want to share their solutions so other people don't have to re-invent the wheel! To install one of these packages, just use install.packages() with the package name (epitools in this instance) in quotes. You only need to install the package once, so you can just run this code in your console, or comment it out of your script (remember, this can be done with "#") so it doesn't download every time you use your script.

```
install.packages("epitools")
```

Once a package is installed to your R folder, you will need to turn on the package to use it. This code *does* need to be repeated each time you start a new R session. You can just put it at the top of your script along with setwd().

```
library(epitools)
```

Once we have epitools on and installed, we can run our test exactly like a chi-square or Fisher exact test. For the epitab() function you just need to specify the method, which in our case is "riskratio". Other options can be supplied for odds ratios or other epi tests.

```
# test for association between food B and illness
twobytwoB <- table(df_sub$food_B, df_sub$classification)
twobytwoB
```

```
##
##       Not ill Case
##   Yes       2    1
##   No       50   28
```

```
epitab(twobytwoB, method="riskratio")
```

```
## Warning in chisq.test(xx, correct = correction): Chi-squared approximation
## may be incorrect
```

```
## $tab
##
##       Not ill        p0 Case        p1 riskratio     lower    upper
##   Yes       2 0.6666667    1 0.3333333  1.000000        NA       NA
##   No       50 0.6410256   28 0.3589744  1.076923 0.2115187 5.483029
##
##       p.value
##   Yes      NA
##   No        1
##
## $measure
## [1] "wald"
##
## $conf.level
## [1] 0.95
##
## $pvalue
## [1] "fisher.exact"
```

Notice that the way we created the food_B variable resulted in "Yes" being used as the reference group. Let's change the levels and try the test again.

```r
df_sub$food_B <- factor(df_sub$food_B, levels=c("No","Yes"))
epitab(df_sub$food_B, df_sub$classification, method="riskratio")
```

```
## Warning in chisq.test(xx, correct = correction): Chi-squared approximation
## may be incorrect
```

```
## $tab
##          Outcome
## Predictor Not ill        p0 Case        p1 riskratio     lower    upper
##       No       50 0.6410256   28 0.3589744 1.0000000        NA       NA
##       Yes       2 0.6666667    1 0.3333333 0.9285714 0.1823809 4.727714
##          Outcome
## Predictor p.value
##       No       NA
##       Yes       1
##
## $measure
## [1] "wald"
##
## $conf.level
## [1] 0.95
##
## $pvalue
## [1] "fisher.exact"
```

Now let's test the association between food A and illness.

```r
# test for association between food A and illness
twobytwoA <- table(df_sub$food_A, df_sub$classification)
twobytwoA
```

```
##
##       Not ill Case
##   No       41    3
##   Yes       9   25
```

```r
epitab(twobytwoA, method="riskratio")
```

```
## $tab
##
##       Not ill        p0 Case         p1 riskratio   lower    upper
##   No       41 0.9318182    3 0.06818182   1.00000      NA       NA
##   Yes       9 0.2647059   25 0.73529412  10.78431 3.55126 32.74934
##
##           p.value
##   No           NA
##   Yes 9.243137e-10
##
## $measure
## [1] "wald"
##
## $conf.level
## [1] 0.95
##
## $pvalue
## [1] "fisher.exact"
```

14) Conclusion

I hope that this guide has helped to show not only how powerful R is as a statistical tool, but also how capable you are of using it! This document is definitely a work in progress; if you have any comments, questions, or suggestions for improvement, please feel free to reach out to me at nrb7@cdc.gov.

My current plans for future updates include:

- Come up with catchier name like "R-n't you glad I didn't say SAS" or something like that
- Add information on pulling duplicate rows or unique values
- Data visualization with ggplot
- Alternative ways of working with data objects using dplyr and tidyr
- Connecting to external databases (for example in Virginia our NEDSS-based system, VEDSS) to query data without having to use SQL
    - Note to self: do **not** give everyone code to access VEDSS

Reference Table 1) Types of expressions R can evaluate that are covered by this guide

Type	**Subtype**	**Description**
Function		A function performs a programmed action on whatever is in parentheses; arguments are supplied to functions to change default behavior.
Literal	character	Strings of letters of any length
	numeric	Numbers
	logical	TRUE or FALSE
Mathematical operation		Anything you would type into a calculator
Assignment		This is how you store information into an object (using "<-")
Data object	vector	A one-dimensional data object that is all the same literal type (all character or all numeric, for example). It is easiest to think of vectors as a single column or row of data.
	matrix	A two-dimensional data object (multiple vectors joined together) where all elements are the same literal type.
	data frame	A two-dimensional data object where all elements do *not* have to be the same type (columns can be any combination of character and numeric and other). This is the most familiar data setup for epis; each row can be interpreted as a record and each column as a field.

Reference Table 2) Useful functions covered by the intro

Function	Action
abs()	take the absolute value of
as.character()	convert to character
as.numeric()	convert to numeric
c()	combine into a vector
cbind()	bind vectors together side-by-side
chisq.test()	compares whether two categorical variables are independent
colnames()	see the column names of an object
data.frame()	convert another object (like a vector or matrix) to a data frame
epitab()	can perform hypothesis testing to compare risk ratios, odds ratios, can do stratified analysis; requires the "epitools" package in R
exp()	exponentiate (raise e to some power)
factor()	treat what's in parentheses as a factor variable; the argument labels=c() designates the names of the variables in their current order (e.g., from c(0,1) to c("No","Yes")); the argument levels=c() changes the order of the variables using their current labels (e.g., from c(0,1) to c(1,0))
fisher.test()	compares the odds ratios of groups
fivenum()	gives the 1st, 2nd, 3rd, 4th, and 5th quintile values for a numeric variable
getwd()	see your working directory
install.packages()	installs a given package (required only once)
is.na()	condition that returns TRUE for NAs or FALSE for other values
length()	view the length of a vector
levels()	view levels of a factor
library()	turns on a package for your R session (required after you exit and restart R)
lm()	compares the means of groups via simple linear regression
log()	take the natural log
log10()	take the log10
max()	find maximum value
mean()	find the average of
min	find minimum value
ncol()	see number of rows in an object
nrow()	see number of columns in an object
rbind()	bind vectors together above-and-below
read.csv()	pull a .csv file into your R session
read.table()	pull a table (usually .txt) file into your R session
rm()	remove a data object
rownames()	see the row names of an object
sd()	find the standard deviation of
setwd()	set a new working directory
sqrt()	take a square root
stem()	shows a nifty stem and leaf plot for a numeric variable
str()	see what type of data is in an object
summary()	see a summary of the object; for factors you will get counts of each category, for numbers you will get the range, mean, quintiles, etc.
t.test()	compares the means of groups via t-test
table()	creates a table of values
View()	view the data object in the script panel
which()	returns a vector of numbers that meet a given criterion
write.csv()	writes an object to a .csv file
write.table()	writes an object to a .txt file