

Lab 1: Introduction to R

Introduction to R, Summary Statistics of Data

Cori Carver

1/21/2021

General information

This lab is due Wednesday, January 27th by 11:59 pm and is worth 5 points. You must upload your .rmd file and your knitted PDF to the assignment folder on Canvas.

You are welcome and encouraged to talk with classmates and ask for help. However, each student should turn in their own lab assignment and all answers, including all code, needs to be solely your own.

How to use the pdf and R markdown document

Download the .pdf and .rmd documents for each week's lab and keep in the same folder, along with any data. Then, double click to open the .Rmd document in R studio. Enter your name and date above where it says "Your Name Here". The .rmd file is intended for you to edit and input your answers. The .pdf contains the same exact information but rendered in an easier-to-read format. I suggest reading through the pdf document, and then switching to .rmd when you are asked to input code or give an answer.

Objective

The goal of this lab is to familiarize yourself with using R, learn the basics of data manipulation in R, and to explore descriptive statistics in R.

Basics of R

R is a versatile computer language that operates as both an excellent statistical package and a programming language. In actuality, the lines between these will blur for us, as we will do our own programming but also will explore built-in statistical packages once we have the conceptual frameworks mastered. R has become the 'computer language for ecology,' meaning that knowing R will help you reproduce and modify others' results (as R code is commonly published online with papers) as well as in future labs.

At a very basic level, R can operate as a calculator. When you open RStudio, your screen will likely be split into four panels. At the bottom is the *console* and at the top, you have your R Markdown or R Scripts open. In the console, you will see a command prompt, `>`. Go ahead and type in an equation, say $2 + 2$, and hit ENTER. You see that R prints the answer, just like using a calculator. Now, try typing in some other basic arithmetic commands, like $2 - 2$, $2 * 4$, $4/2$, and 4^2 .

For reproducibility, it is best to type all non-exploratory code in an R Script or R Markdown file. That way, you can go back and follow each step. For labs, all final code must be written in the R Markdown document. Otherwise, it will be impossible to know what you did. To do so, we can type code directly into the R Markdown document. To separate code from text, we use the following syntax: ``{r}`` with curly brackets around the `r` begins a part of code and ``` denotes the end of code. A section of code delimited in

this way in a .Rmd is called a code chunk, or more simply a “chunk”. Chunks are visually demarcated from text within a .Rmd file by appearing in a different color background. When you export (or “knit”) the .rmd to a PDF, the code is evaluated and output is displayed.

Insert a Chunk of Code

To insert chunks, you can type the ```r{}` by hand, or go up to the menu and select Code/Insert Chunk.

To write comments on your code you can use the hashtag or pound symbol. Whatever is typed after the hashtag will be ignored by R’s computer. This is helpful for describing what each line of your code does.

```
# code goes here. We'll use the 2 + 2 example  
2 + 2
```

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

```
4/2
```

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

```
5^7
```

```
## [1] 78125
```

```
3+4-9
```

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

Note that the answer to this math is included as output in the Rmd file (below the chunk of R code) and in the console below.

Note also that the ```{r}` and “” do not render (print) in the pdf, but rather tell R that this is a code section.

I know this is a bit confusing, but the pound sign means something different depending on whether it is inside or outside a chunk of R code in a .Rmd file. Inside a code chunk, the pound symbol indicates that you are writing comments that should not be ignored as a command. This helps you (or others) follow your logic. Outside of a code chunk, a `#` preceding a line of text inserts a section break within the .Rmd and will output the line of text as a header (appears as bigger, bold text) for that section in the knitted document. You will see this above with the General Information header, etc. By changing the number of pound signs that precede text you can change the type of header that is displayed in the knitted document (first level header vs. subsequent level subheaders). See how the *Basics of R* header and *Inserting a Chunk of Code* sub-header are coded above and how they appear in the knitted pdf file.

Running a Chunk of Code

In a .Rmd file you have text outside of chunks that is just evaluated as text and code within chunks that is designed to run (i.e. do something). So how do you get R to run (i.e. evaluate) the code in one of your chunks. In RStudio, the Console is the business end that does all of the running of code. It is in the window below. If you want to run some code from your .Rmd file you can copy and paste it in the Console and hit enter. This will run this code. Alternatively, you can type directly into the Console. In RStudio, however, there are some shortcuts that make this easier. For example, you can place the cursor in a line of code or highlight multiple lines of code and then select ‘Run’ and ‘Run Selected Lines’ to run those lines of code. Alternatively, you can place your cursor in the line of code you want to evaluate or highlight multiple lines of code and then press Ctrl+Enter (Windows) or Command+Return (Mac). This will run this line of code. You can see what was done in the Console window below. You do not need to “knit” your R markdown into PDF each time you evaluate code.

Practice writing some simple arithmetic equations in the above code chunk and evaluating them by running the code.

Using the assignment operator

Instead of just typing in an equation, say $4 * 8$, you will often want to save the output value to use later on. You do so using the *assignment operator*, `<-`. For example,

```
x <- 4 * 8
```

The solution to the equation $4 * 8$ is saved as a variable called `x`. You could substitute “`x`” for any other letter or variable name you like. When you assign variables using `<-`, the answer is not automatically printed, but is rather stored as an *object*. In the top right window in RStudio there is a tab called ‘Environment’. If you click on this tab you should be able to see your new object called “`x`” as well as its value. This ‘Environment’ is called your workspace. You can easily see the answer by just typing in `x` to the console or on your R Markdown sheet.

Sometimes you will see the `=` sign used instead of `<-`. They are equivalent in most cases, though most R users agree that `<-` should be used for assigning variables.

If you want to see the value of a variable you just assigned, just type the name of the variable and hit enter. This is also called “printing” the variable. You will often be asked to print/display certain values in your assignments, meaning you should put the name of that value in your code chunk.

For instance, a question might ask you to multiply 8 and 4 and save that value as the variable “`x`”. Then, I might ask you to print the value of `x` (i.e., the answer).

```
x <- 4 * 8
x # prints the value now saved as x
```

```
## [1] 32
```

```
a = 4 * 8
a # prints the value now saved as a
```

```
## [1] 32
```

Q1: You can use previously assigned variables in future operations. For example try multiplying `x` by 5 and then adding 12 to this value. Save this answer as a new variable (call it something new!). Now take this new variable and divide it by 2.2, again saving your answer as a new variable. Print out the value of this new variable.

```
r <- x * 5 + 12
h <- r / 2.2
h
```

```
## [1] 78.18182
```

Note that R is case-sensitive. Try typing `X` on the console. What does R return?

We don’t want to store all the variables we create forever. Best practice is to export important datasets into `.csv` or `.txt` files (or save the R script that created them) rather than rely on stored objects in R. It is usually a good idea to start your R coding session with a “clean slate” by removing variables stored from previous sessions. You can remove an object from the workspace using the `rm` function.

```
rm(x)
```

Note that after running this line of code the “`x`” object is now removed from our “environment” or workspace. Now, try to print the variable `x` you assigned above, by typing it into your console and hitting return. What happens?

We can also remove all of the objects in our workspace by clicking on the ‘broom’ symbol in the Environment panel in the top right. You can also use the following code to remove all of the objects in your workspace:

```
rm(list=ls())
```

BEWARE: This code will remove all objects in your current workspace. It is generally not a good idea to have bits of code like this in your files because this code could accidentally remove all of the objects in your workspace.

Instead it is preferred to use the ‘broom’ icon in the Environment window.

Getting help in R

R has wonderful help and example code. If you click on *Help* on the right hand side of RStudio above the lower right panel, you can see manuals and resources both for R and RStudio.

Additionally, if you type `?` followed by an R command in the console, R will open documentation on the command. [Note: “commands” are functions in R that perform some type of task, such as `mean()`, `sum()`, or `plot()`]. For example:

```
?mean
```

This is helpful when you can’t remember all of the arguments for a command you are using (much more on this later). Additionally, typing `example(command)`, provides example code on how to use the given function.

Since R is open-source, there are great resources and question forums online as well. If there is a function you think R should be able to do, you are probably right! Google is a great resource for R. Also, stack overflow (stackoverflow.com) has answers to thousands of questions from R users. Likely someone has asked a similar question that is already answered on stack overflow.

Q2: One of the many built-in commands in R is the square root function, `sqrt()`. Enter code for opening the R Help documentation on the square root function.

```
?sqrt()
```

Q3: Now, use the square root command to find the square root of 9.999, saving the answer as variable, `sqEx`. Print the answer as well.

```
sqEx <- sqrt(9.999)
sqEx
```

```
## [1] 3.16212
```

Vectors and Dataframes

Vectors

Vectors, matrices, and data frames are all ways to hold multiple data elements, called *elements*, of the same type. Elements can be numbers or character strings (i.e. words). Vectors can be created in multiple ways. The most common are to use the `c()`, `seq()`, or `rep()` functions.

The `c()` function combines multiple elements together, while `seq()` automatically lists a sequence of values, allowing you to specify the starting value, ending value, the amount you count by, or the number of elements you want to include in a given vector. Type `?seq()` for more information. Finally, `rep()` allows you to repeat an element a certain amount of times. Let’s look at some examples. Run and explore the following code:

```
ex1 <- c(1, 5, 8, 9)
ex2 <- c(1, 10:15) # What does ex2 look like? How many elements?
# remember, to see the value of something you just created, you should
# type the name and run it to print the answer below the code chunk:
ex2
```

```
## [1] 1 10 11 12 13 14 15
```

```
# Multiple functions can be called at the same time.
```

```
ex3 <- c(1, 5, seq(from = 1, to = 10, by = 2.2)) # What is this code doing?
```

```
ex4 <- c(rep(5, 3), rep(3, 5), seq(from = -5, to = 0))
```

Make sure you understand the difference between `c()`, `seq()`, or `rep()`!

To extract elements from a vector, you use square brackets, `[]`. For example, if you want the third element from `ex1`:

```
ex1[3]
```

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

Another handy function for vectors is `length()`, which gives you the number of elements in a vector.

```
length(ex4)
```

```
## [1] 14
```

```
# Or, to find the value of the last element in ex4,  
ex4[length(ex4)]
```

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

Note that this finds the last element in this vector because `length(ex4)` is equal to 14 and `ex4[14]` just returns the 14th element in the vector. All we have done is to place one command within another. This is a helpful feature of coding that we will use a lot.

We don't have to just use numbers when creating vectors or assigning variables in R. We can also use characters.

```
birds <- c("Mourning Dove", "Downy Woodpecker", "American Robin")
```

```
weights <- c(120, 28, 80)
```

```
# Note how R denotes number vs. characters
```

```
class(birds)
```

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

```
class(weights)
```

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

Q4: Now, add a fourth bird to our vector of birds. If you don't know, try googling first! Hint: what element number should it be? Print the vector to make sure it worked.

```
birds[4] = "Black-capped Chickadee"
```

```
birds[4]
```

```
## [1] "Black-capped Chickadee"
```

Data Frames

Data frames are commonly used in statistical analyses. A dataframe is analogous to a typical Excel table that you would use to organize your data, with a header row and each column containing a different type of data. Different columns can be of different classes. Therefore, in a data frame, you can mix numbers, characters, etc. across columns, which, you can imagine, is quite useful. For example, say you have sampled sites across an elevational gradient and recorded abundances of different species at each site. A nice structure for your data would then be a column of site names (characters), a column with the elevation for each site, followed by multiple columns of abundances of different species.

Let's go ahead and look at one of R's example datasets, called CO2. You can print the whole dataframe if you type CO2 in your console (make sure you pay attention to capitalization!).

Q5: Describe what is included in each column in the CO2 dataframe. Note: refer to the R help '?' to do this.

CO2

##	Plant	Type	Treatment	conc	uptake
## 1	Qn1	Quebec	nonchilled	95	16.0
## 2	Qn1	Quebec	nonchilled	175	30.4
## 3	Qn1	Quebec	nonchilled	250	34.8
## 4	Qn1	Quebec	nonchilled	350	37.2
## 5	Qn1	Quebec	nonchilled	500	35.3
## 6	Qn1	Quebec	nonchilled	675	39.2
## 7	Qn1	Quebec	nonchilled	1000	39.7
## 8	Qn2	Quebec	nonchilled	95	13.6
## 9	Qn2	Quebec	nonchilled	175	27.3
## 10	Qn2	Quebec	nonchilled	250	37.1
## 11	Qn2	Quebec	nonchilled	350	41.8
## 12	Qn2	Quebec	nonchilled	500	40.6
## 13	Qn2	Quebec	nonchilled	675	41.4
## 14	Qn2	Quebec	nonchilled	1000	44.3
## 15	Qn3	Quebec	nonchilled	95	16.2
## 16	Qn3	Quebec	nonchilled	175	32.4
## 17	Qn3	Quebec	nonchilled	250	40.3
## 18	Qn3	Quebec	nonchilled	350	42.1
## 19	Qn3	Quebec	nonchilled	500	42.9
## 20	Qn3	Quebec	nonchilled	675	43.9
## 21	Qn3	Quebec	nonchilled	1000	45.5
## 22	Qc1	Quebec	chilled	95	14.2
## 23	Qc1	Quebec	chilled	175	24.1
## 24	Qc1	Quebec	chilled	250	30.3
## 25	Qc1	Quebec	chilled	350	34.6
## 26	Qc1	Quebec	chilled	500	32.5
## 27	Qc1	Quebec	chilled	675	35.4
## 28	Qc1	Quebec	chilled	1000	38.7
## 29	Qc2	Quebec	chilled	95	9.3
## 30	Qc2	Quebec	chilled	175	27.3
## 31	Qc2	Quebec	chilled	250	35.0
## 32	Qc2	Quebec	chilled	350	38.8
## 33	Qc2	Quebec	chilled	500	38.6
## 34	Qc2	Quebec	chilled	675	37.5
## 35	Qc2	Quebec	chilled	1000	42.4
## 36	Qc3	Quebec	chilled	95	15.1
## 37	Qc3	Quebec	chilled	175	21.0
## 38	Qc3	Quebec	chilled	250	38.1
## 39	Qc3	Quebec	chilled	350	34.0
## 40	Qc3	Quebec	chilled	500	38.9
## 41	Qc3	Quebec	chilled	675	39.6
## 42	Qc3	Quebec	chilled	1000	41.4
## 43	Mn1	Mississippi	nonchilled	95	10.6
## 44	Mn1	Mississippi	nonchilled	175	19.2
## 45	Mn1	Mississippi	nonchilled	250	26.2
## 46	Mn1	Mississippi	nonchilled	350	30.0
## 47	Mn1	Mississippi	nonchilled	500	30.9

```
## 48 Mn1 Mississippi nonchilled 675 32.4
## 49 Mn1 Mississippi nonchilled 1000 35.5
## 50 Mn2 Mississippi nonchilled 95 12.0
## 51 Mn2 Mississippi nonchilled 175 22.0
## 52 Mn2 Mississippi nonchilled 250 30.6
## 53 Mn2 Mississippi nonchilled 350 31.8
## 54 Mn2 Mississippi nonchilled 500 32.4
## 55 Mn2 Mississippi nonchilled 675 31.1
## 56 Mn2 Mississippi nonchilled 1000 31.5
## 57 Mn3 Mississippi nonchilled 95 11.3
## 58 Mn3 Mississippi nonchilled 175 19.4
## 59 Mn3 Mississippi nonchilled 250 25.8
## 60 Mn3 Mississippi nonchilled 350 27.9
## 61 Mn3 Mississippi nonchilled 500 28.5
## 62 Mn3 Mississippi nonchilled 675 28.1
## 63 Mn3 Mississippi nonchilled 1000 27.8
## 64 Mc1 Mississippi chilled 95 10.5
## 65 Mc1 Mississippi chilled 175 14.9
## 66 Mc1 Mississippi chilled 250 18.1
## 67 Mc1 Mississippi chilled 350 18.9
## 68 Mc1 Mississippi chilled 500 19.5
## 69 Mc1 Mississippi chilled 675 22.2
## 70 Mc1 Mississippi chilled 1000 21.9
## 71 Mc2 Mississippi chilled 95 7.7
## 72 Mc2 Mississippi chilled 175 11.4
## 73 Mc2 Mississippi chilled 250 12.3
## 74 Mc2 Mississippi chilled 350 13.0
## 75 Mc2 Mississippi chilled 500 12.5
## 76 Mc2 Mississippi chilled 675 13.7
## 77 Mc2 Mississippi chilled 1000 14.4
## 78 Mc3 Mississippi chilled 95 10.6
## 79 Mc3 Mississippi chilled 175 18.0
## 80 Mc3 Mississippi chilled 250 17.9
## 81 Mc3 Mississippi chilled 350 17.9
## 82 Mc3 Mississippi chilled 500 17.9
## 83 Mc3 Mississippi chilled 675 18.9
## 84 Mc3 Mississippi chilled 1000 19.9
```

The first column is plant, then type, then treatment, then CO2 concentration and finally CO2 uptake.

To refer to specific rows or columns of a dataframe, you will use brackets [,]. The first number in a bracket is the row number, and the second number in a bracket is the column number. Your bracket will always contain two slots if you are referring to a 2-dimensional dataset (e.g. a dataframe). Since vectors are one dimensional, we use a single number to pull out specific records (as demonstrated above).

For example, we can ask R to return the element in the 1st row and the 4th column.

```
CO2[1, 4]
```

```
## [1] 95
```

Though you can enter a row value and a column value, if you leave one of the slots blank, it will display all rows or all columns.

For instance, we can print just the third column of CO2. Leaving the first argument blank means that we will display all of the rows:

```
CO2[ ,3]
```

```
## [1] nonchilled nonchilled nonchilled nonchilled nonchilled nonchilled
## [7] nonchilled nonchilled nonchilled nonchilled nonchilled nonchilled
## [13] nonchilled nonchilled nonchilled nonchilled nonchilled nonchilled
## [19] nonchilled nonchilled nonchilled chilled chilled chilled
## [25] chilled chilled chilled chilled chilled chilled
## [31] chilled chilled chilled chilled chilled chilled
## [37] chilled chilled chilled chilled chilled chilled
## [43] nonchilled nonchilled nonchilled nonchilled nonchilled nonchilled
## [49] nonchilled nonchilled nonchilled nonchilled nonchilled nonchilled
## [55] nonchilled nonchilled nonchilled nonchilled nonchilled nonchilled
## [61] nonchilled nonchilled nonchilled chilled chilled chilled
## [67] chilled chilled chilled chilled chilled chilled
## [73] chilled chilled chilled chilled chilled chilled
## [79] chilled chilled chilled chilled chilled chilled
## Levels: nonchilled chilled
```

Or, we could look at the 40th row of CO2. In this case, we have left the second argument blank, meaning that we will display all of the columns:

```
CO2[40, ]
```

```
##   Plant   Type Treatment conc uptake
## 40   Qc3 Quebec   chilled  500   38.9
```

Some other functions you will often use with dataframes:

- Print the names of each column in a data frame using the function `names()`.
- Access a single column in a dataframe using the dollar symbol to give the column name (instead of using the column number).
- Preview the first few rows of a data frame using the function `head()`.
- See the structure of the dataset using the `str()` command. This command is a useful first step to explore datasets!
- Summarize a dataset or vector using `summary()`.
- Perform calculations on certain columns, such as `mean()`, `min()`, `max()`, `sd()`.
- Count the number of records in a column using `length()`.

For example:

```
# First, let's rename CO2, since we'll want to manipulate it ourselves.
CO2ex <- CO2
```

```
# Print each of the column names
names(CO2ex)
```

```
## [1] "Plant"      "Type"       "Treatment"  "conc"       "uptake"
```

```
# Print just one of the columns, the Treatment column
CO2ex$Treatment
```

```
## [1] nonchilled nonchilled nonchilled nonchilled nonchilled nonchilled
## [7] nonchilled nonchilled nonchilled nonchilled nonchilled nonchilled
## [13] nonchilled nonchilled nonchilled nonchilled nonchilled nonchilled
## [19] nonchilled nonchilled nonchilled chilled chilled chilled
## [25] chilled chilled chilled chilled chilled chilled
```



```
## [31] chilled      chilled      chilled      chilled      chilled      chilled
## [37] chilled      chilled      chilled      chilled      chilled      chilled
## [43] nonchilled    nonchilled    nonchilled    nonchilled    nonchilled    nonchilled
## [49] nonchilled    nonchilled    nonchilled    nonchilled    nonchilled    nonchilled
## [55] nonchilled    nonchilled    nonchilled    nonchilled    nonchilled    nonchilled
## [61] nonchilled    nonchilled    nonchilled    chilled       chilled       chilled
## [67] chilled       chilled       chilled       chilled       chilled       chilled
## [73] chilled       chilled       chilled       chilled       chilled       chilled
## [79] chilled       chilled       chilled       chilled       chilled       chilled
## Levels: nonchilled chilled
```

```
# Print the 77th value in the uptake column
CO2ex$uptake[77]
```

```
## [1] 14.4
```

```
# Display the first few rows of the dataframe
head(CO2ex)
```

```
##   Plant   Type Treatment conc uptake
## 1   Qn1 Quebec nonchilled   95  16.0
## 2   Qn1 Quebec nonchilled  175  30.4
## 3   Qn1 Quebec nonchilled  250  34.8
## 4   Qn1 Quebec nonchilled  350  37.2
## 5   Qn1 Quebec nonchilled  500  35.3
## 6   Qn1 Quebec nonchilled  675  39.2
```

```
# Display the value in row 10, column 5
CO2ex[10,5]
```

```
## [1] 37.1
```

```
# Find the mean uptake rate
mean(CO2ex$uptake)
```

```
## [1] 27.2131
```

```
# An equivalent approach, since uptake is the 5th column:
mean(CO2ex[, 5])
```

```
## [1] 27.2131
```

```
# See what type of data are stored in each column of the dataframe
str(CO2ex)
```

```
## Classes 'nfnGroupedData', 'nfGroupedData', 'groupedData' and 'data.frame': 84 obs. of 5 variables
## $ Plant : Ord.factor w/ 12 levels "Qn1"<"Qn2"<"Qn3"<...: 1 1 1 1 1 1 1 2 2 2 ...
## $ Type : Factor w/ 2 levels "Quebec","Mississippi": 1 1 1 1 1 1 1 1 1 1 ...
## $ Treatment: Factor w/ 2 levels "nonchilled","chilled": 1 1 1 1 1 1 1 1 1 1 ...
## $ conc : num 95 175 250 350 500 675 1000 95 175 250 ...
## $ uptake : num 16 30.4 34.8 37.2 35.3 39.2 39.7 13.6 27.3 37.1 ...
## - attr(*, "formula")=Class 'formula' language uptake ~ conc | Plant
## .. ..- attr(*, ".Environment")=<environment: R_EmptyEnv>
## - attr(*, "outer")=Class 'formula' language ~Treatment * Type
## .. ..- attr(*, ".Environment")=<environment: R_EmptyEnv>
## - attr(*, "labels")=List of 2
## ..$ x: chr "Ambient carbon dioxide concentration"
## ..$ y: chr "CO2 uptake rate"
## - attr(*, "units")=List of 2
```

```
## ..$ x: chr "(uL/L)"
## ..$ y: chr "(umol/m^2 s)"
```

Q6: Print out the entry in the 10th row and 3rd column of the subsetted Quebec data

```
CO2ex[10,3]
```

```
## [1] nonchilled
## Levels: nonchilled chilled
```

We might also want to filter out particular records of interest. For instance, if we only want the entries in the dataframe that came from Quebec, we can pull these records out and save them as a new dataframe, using the `subset` function. In the `subset` function, the first argument is the name of the original dataframe (CO2ex), and the second argument is the column you want to use to filter the data (Type), followed by which entries you want to include. When the filtering is done on non-numeric data, you'll want to use quotations.

```
quebecData <- subset(CO2ex, Type == "Quebec")
# print out to make sure the subset worked
quebecData
```

```
##   Plant   Type Treatment conc uptake
## 1   Qn1 Quebec nonchilled    95   16.0
## 2   Qn1 Quebec nonchilled   175   30.4
## 3   Qn1 Quebec nonchilled   250   34.8
## 4   Qn1 Quebec nonchilled   350   37.2
## 5   Qn1 Quebec nonchilled   500   35.3
## 6   Qn1 Quebec nonchilled   675   39.2
## 7   Qn1 Quebec nonchilled  1000   39.7
## 8   Qn2 Quebec nonchilled    95   13.6
## 9   Qn2 Quebec nonchilled   175   27.3
## 10  Qn2 Quebec nonchilled   250   37.1
## 11  Qn2 Quebec nonchilled   350   41.8
## 12  Qn2 Quebec nonchilled   500   40.6
## 13  Qn2 Quebec nonchilled   675   41.4
## 14  Qn2 Quebec nonchilled  1000   44.3
## 15  Qn3 Quebec nonchilled    95   16.2
## 16  Qn3 Quebec nonchilled   175   32.4
## 17  Qn3 Quebec nonchilled   250   40.3
## 18  Qn3 Quebec nonchilled   350   42.1
## 19  Qn3 Quebec nonchilled   500   42.9
## 20  Qn3 Quebec nonchilled   675   43.9
## 21  Qn3 Quebec nonchilled  1000   45.5
## 22  Qc1 Quebec   chilled    95   14.2
## 23  Qc1 Quebec   chilled   175   24.1
## 24  Qc1 Quebec   chilled   250   30.3
## 25  Qc1 Quebec   chilled   350   34.6
## 26  Qc1 Quebec   chilled   500   32.5
## 27  Qc1 Quebec   chilled   675   35.4
## 28  Qc1 Quebec   chilled  1000   38.7
## 29  Qc2 Quebec   chilled    95    9.3
## 30  Qc2 Quebec   chilled   175   27.3
## 31  Qc2 Quebec   chilled   250   35.0
## 32  Qc2 Quebec   chilled   350   38.8
## 33  Qc2 Quebec   chilled   500   38.6
## 34  Qc2 Quebec   chilled   675   37.5
## 35  Qc2 Quebec   chilled  1000   42.4
```

```
## 36 Qc3 Quebec chilled 95 15.1
## 37 Qc3 Quebec chilled 175 21.0
## 38 Qc3 Quebec chilled 250 38.1
## 39 Qc3 Quebec chilled 350 34.0
## 40 Qc3 Quebec chilled 500 38.9
## 41 Qc3 Quebec chilled 675 39.6
## 42 Qc3 Quebec chilled 1000 41.4
```

Note that in the above code we used a new notation `==`. If you remember before we used `<-` or equivalently `=` to define a value for an object. For example:

```
x = 54
```

This defines `x` to be equal to 54. In the notation above, we instead used `==`. Instead of defining a value, this notation (`==`) ask a question: “Is it equal to?”.

So we could instead ask:

```
x ==54
```

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

You can see here, that R answers that “yes” in fact `x` is equal to 54. Alternatively...

```
x == 10
```

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

So inside the `subset()` command we are asking for only those records where `type == "Quebec"` is true.

Another example, filtering by data that were in Mississippi and have an uptake rate less than 12

```
Miss.LowUptake <- subset(CO2ex, Type == "Mississippi" & uptake < 12)
Miss.LowUptake
```

```
##      Plant      Type Treatment conc uptake
## 43 Mn1 Mississippi nonchilled 95 10.6
## 57 Mn3 Mississippi nonchilled 95 11.3
## 64 Mc1 Mississippi chilled 95 10.5
## 71 Mc2 Mississippi chilled 95 7.7
## 72 Mc2 Mississippi chilled 175 11.4
## 78 Mc3 Mississippi chilled 95 10.6
```

Q7: Filter out data from entries that were in the "chilled" treatment and had an ambient concentration of more than 500. Save this it as a new dataframe. Use 'str' or 'nrow' to see how many records there are in this new dataframe.

```
chilled.highConc <- subset(CO2ex, Treatment == "chilled" & conc > 500)
chilled.highConc
```

```
##      Plant      Type Treatment conc uptake
## 27 Qc1 Quebec chilled 675 35.4
## 28 Qc1 Quebec chilled 1000 38.7
## 34 Qc2 Quebec chilled 675 37.5
## 35 Qc2 Quebec chilled 1000 42.4
## 41 Qc3 Quebec chilled 675 39.6
## 42 Qc3 Quebec chilled 1000 41.4
## 69 Mc1 Mississippi chilled 675 22.2
## 70 Mc1 Mississippi chilled 1000 21.9
## 76 Mc2 Mississippi chilled 675 13.7
## 77 Mc2 Mississippi chilled 1000 14.4
```

```
## 83   Mc3 Mississippi   chilled  675   18.9
## 84   Mc3 Mississippi   chilled 1000   19.9
```

```
nrow(chilled.highConc)
```

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

There were 12 records chilled in ambient conditions greater than 500

Syntax is extremely important in R and in the beginning, you will likely be producing some frustrating error messages. The use of parentheses, capital letters, commas, etc. makes a big difference in how code is evaluated. Unfortunately the error messages in R don't always help diagnose the problem. It's good to get practice interpreting these errors and de-bugging your code! The lack of an error code doesn't mean your code ran correctly, either. It is good practice to view data frames, vectors, plots, etc. after you perform some type of command to make sure it worked the way you intended it to.

Q8: Debug the following lines of code so that they perform the requested function. To do this, you must first uncomment the lines of code (not the comments themselves) by deleting the pound sign. Then run the line of code, correct the errors, and re-run to make sure it worked.

```
# 1. View the first few lines of code
```

```
head(CO2ex)
```

```
##   Plant   Type Treatment conc uptake
## 1   Qn1 Quebec nonchilled   95   16.0
## 2   Qn1 Quebec nonchilled  175   30.4
## 3   Qn1 Quebec nonchilled  250   34.8
## 4   Qn1 Quebec nonchilled  350   37.2
## 5   Qn1 Quebec nonchilled  500   35.3
## 6   Qn1 Quebec nonchilled  675   39.2
```

```
# 2. Subset data from Mississippi
```

```
# Hint: print out the subsetted data frame. Did it work?
```

```
missData <- subset(CO2ex, Type=="Mississippi")
```

```
missData
```

```
##   Plant      Type Treatment conc uptake
## 43   Mn1 Mississippi nonchilled   95   10.6
## 44   Mn1 Mississippi nonchilled  175   19.2
## 45   Mn1 Mississippi nonchilled  250   26.2
## 46   Mn1 Mississippi nonchilled  350   30.0
## 47   Mn1 Mississippi nonchilled  500   30.9
## 48   Mn1 Mississippi nonchilled  675   32.4
## 49   Mn1 Mississippi nonchilled 1000   35.5
## 50   Mn2 Mississippi nonchilled   95   12.0
## 51   Mn2 Mississippi nonchilled  175   22.0
## 52   Mn2 Mississippi nonchilled  250   30.6
## 53   Mn2 Mississippi nonchilled  350   31.8
## 54   Mn2 Mississippi nonchilled  500   32.4
## 55   Mn2 Mississippi nonchilled  675   31.1
## 56   Mn2 Mississippi nonchilled 1000   31.5
## 57   Mn3 Mississippi nonchilled   95   11.3
## 58   Mn3 Mississippi nonchilled  175   19.4
## 59   Mn3 Mississippi nonchilled  250   25.8
## 60   Mn3 Mississippi nonchilled  350   27.9
## 61   Mn3 Mississippi nonchilled  500   28.5
## 62   Mn3 Mississippi nonchilled  675   28.1
## 63   Mn3 Mississippi nonchilled 1000   27.8
```

```
## 64 Mc1 Mississippi chilled 95 10.5
## 65 Mc1 Mississippi chilled 175 14.9
## 66 Mc1 Mississippi chilled 250 18.1
## 67 Mc1 Mississippi chilled 350 18.9
## 68 Mc1 Mississippi chilled 500 19.5
## 69 Mc1 Mississippi chilled 675 22.2
## 70 Mc1 Mississippi chilled 1000 21.9
## 71 Mc2 Mississippi chilled 95 7.7
## 72 Mc2 Mississippi chilled 175 11.4
## 73 Mc2 Mississippi chilled 250 12.3
## 74 Mc2 Mississippi chilled 350 13.0
## 75 Mc2 Mississippi chilled 500 12.5
## 76 Mc2 Mississippi chilled 675 13.7
## 77 Mc2 Mississippi chilled 1000 14.4
## 78 Mc3 Mississippi chilled 95 10.6
## 79 Mc3 Mississippi chilled 175 18.0
## 80 Mc3 Mississippi chilled 250 17.9
## 81 Mc3 Mississippi chilled 350 17.9
## 82 Mc3 Mississippi chilled 500 17.9
## 83 Mc3 Mississippi chilled 675 18.9
## 84 Mc3 Mississippi chilled 1000 19.9
```

```
# 3. Extract the 50th row of the CO2ex dataframe and save it as a new vector
Fifty <- CO2ex[50,]
Fifty
```

```
## Plant Type Treatment conc uptake
## 50 Mn2 Mississippi nonchilled 95 12
```

Loading data

Often you will want to load your own data into R. R can read .csv and .txt files using base R commands but not .xls files (there are bundled functions called “R packages” written by other R users we can install and use to do things like read in .xls or .xlsx files directly, but for now we are focusing on commands that come with R, i.e. “base R” code). We will read in the file *Pelts.csv*, which can be found on the course Canvas page. This file contains a list of the number of recorded pelts collected by the Hudson Bay Company from 1752 to 1819.

To input the data into R, we will use the function *read.csv()*. To read a csv file, it first has to be saved into your working directory, which is likely the same directory (or folder) as this R Markdown file (unless you have changed it). To see what R is using as your working directory, run the command *getwd()*:

```
getwd()
```

```
## [1] "/Users/coricarver/Desktop/R_Files/R_EBIO_Stats/lab1"
```

Navigate to this folder on your computer and make sure *Pelts.csv* is in there. After you have made sure the dataset is in the same folder, you can load the csv file using *read.csv()*. You want to assign the loaded data to an object in R so you can work with it. For example, you could type *pelts <- read.csv("Pelts.csv")*.

Do not use *file.choose()* with Rmarkdown documents as it will impair knitting. Get in the habit of organizing your files and using *read.csv()*!

A good shortcut to see the files currently in your working directory, is to go to the files tab in the panel to the right. You can hit the **More** menu and select “go to working directory” and view the files currently in that folder.

Read in *Pelts.csv* and save it as an object.

```
pelts <- read.csv("Pelts.csv")
```

Examine what type of data Pelts contains by using the `class()` or `str()` functions. The function `summary()` also gives a nice overview of your data. Try all three out.

```
class(pelts)
```

```
## [1] "data.frame"
```

```
str(pelts)
```

```
## 'data.frame': 68 obs. of 2 variables:
## $ date : int 1752 1753 1754 1755 1756 1757 1758 1759 1760 1761 ...
## $ no.pelts: int 4009 7179 4198 1444 838 631 917 1881 3842 5338 ...
```

```
summary(pelts)
```

```
##      date      no.pelts
## Min.   :1752   Min.   : 116.0
## 1st Qu.:1769   1st Qu.: 986.8
## Median :1786   Median :1551.5
## Mean   :1786   Mean    :2137.3
## 3rd Qu.:1802   3rd Qu.:2858.8
## Max.   :1819   Max.    :7179.0
```

Q9: What type of object is pelts? What other information do you get from using the `str()` function?

Pelts is a data frame with 68 observations of two variables (date and number of pelts).

Q10: Calculate the mean number of pelts harvested between 1752 and 1761. Use any of the functions we've gone over already (like the subset, length, mean, or sum functions). Store your answer as an object and print it. Comment your code so that a reader could follow your logic.

```
#Create a subset of the pelts dataframe extracting the number of pelts harvested between 1752 and 1761
numPelts <- subset(pelts, date > 1751 & date < 1762)
```

```
#Find the mean number of pelts harvested
avgNumPelts <- mean(numPelts$no.pelts)
```

```
#Print the mean number of pelts harvested
avgNumPelts
```

```
## [1] 3027.7
```

I have also placed a copy of this file on GitHub. You can also load data from a csv file at a specific URL using `read.csv()`

```
pelts_web<-read.csv("https://raw.githubusercontent.com/StatsTree/Datasets/master/Pelts.csv")
```

Making your own data frame

Often you will load in previously compiled data (say, from an Excel workbook converted to .csv). Sometimes you will want to create your own data frames from scratch in R. We will start by creating a few different vectors and then combine them into a dataframe. Here is an example of a dataframe with 3 columns:

```
fish_type<- c("lake trout", "brown trout", "lake trout", "rainbow trout", "brown trout")
parasite<- c("yes", "no", "yes", "yes", "no")
length_fish<- c(40, 63, 48, 51, 69)
```

```
df<-data.frame(fish_type,parasite,length_fish)
str(df)

## 'data.frame':    5 obs. of  3 variables:
## $ fish_type : chr  "lake trout" "brown trout" "lake trout" "rainbow trout" ...
## $ parasite  : chr  "yes" "no" "yes" "yes" ...
## $ length_fish: num  40 63 48 51 69
```

```
head(df)
```

	fish_type	parasite	length_fish
## 1	lake trout	yes	40
## 2	brown trout	no	63
## 3	lake trout	yes	48
## 4	rainbow trout	yes	51
## 5	brown trout	no	69

Knitting your Rmd document

You will turn in your biometry assignments in as both an .Rmd document (like the one you are currently working in) and a PDF. The PDF version is a nicely formatted document which will display results and graphics. The process of making the PDF version is called “knitting”. Try knitting the current document by hitting the icon that says “knit” in the menu bar above, or choosing “knit document” from the File menu.

If there are errors in your code, the knitting process does not work. If you get an error message, navigate to the line listed and de-bug the code.

Still not working? If you are at the end of your rope and cannot fix broken code (and you’ve already tried meeting with your TA!), you may “comment out” the line that does not work using a # before the line of code, and then knit the PDF.