Tutorial 1 - R Primer & Generative AI

Sander Elliott

Objectives

The objective of today's hands-on activity is to provide a basic overview of R. For each topic covered the first part will be directed – you will follow the prescribed sequence of R commands in order to familiarize yourself with what they do. The second part of each topic will ask you to apply these R commands.

Assignment

For this activity you will turn in a Rmd file that shows your work.

Always check that the Rmd file "knits" before submitting!

All Rmd documents should start with a section that looks something like this:

```
---
title: "My Lab"
author: "My Name"
output: html_document
---
```

Make sure to give your file a title and put ***your name*** in the author field. Notice that the title and author need to be in quotes.

Knitting Rmarkdown documents

To be able to compile Rmd into HTML or any other format you will need to install the "knitr" library.

If you do not already see a button at the top of the editor window that says "Knit" with an icon of a blue ball of yarn next to it, you will need to install this package.

- From the "Packages" tab in the bottom right pane click on "Install"
- Enter knitr as the package name, and then click Install
- From the main menu bar, select Session > Restart R
- The Knit button should now be visible

If you do have the Knit button, now you should click it. This will create a new file, that has the same name as the Rmd file, except it ends in `.html`. Rstudio should automatically preview this file for you. You will see that it contains exactly the same content as the Rmd file, but formatted nicely in to an east to read document. You are free to keep reading the assignment in either the Rmd file or the html file, but **you can**

only edit the Rmd file. When you edit the Rmd file, your changes will not show up in the html file until you select Knit. **Kitting is the best way to check that all your code runs.**

NOTE: this document will not knit until after Question 10, when the variable `met` *is defined*

Code Answers

All your code should be placed into what we call "R chunks."

In the editor pane you can recognize them because:

They start with: ```` ```{r} ```` (the `{r}` is very important because it's what makes it specifically an R chunk, as opposed to a different type of code chunk)

They end with ```` ``` ````

R code chunks also have in the top right hand corner three important icons (from left to right)

1. Optional Settings. Once you get more comfortable using R, they can be useful.
2. Run the code in all chunks up to this one.
3. Run the code in this chunk.

Below is an R chunk. Try hitting the play button. You will see that the output appears below the code chunk.

```
1 + 2
```

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

Written Answers

Any verbal answers should be written in the space between code chunks. You can also include some of your written responses as commented code within the R chunk. Comments start with `#` and are not evaluated by R. Putting comments in code is very useful for remembering what you did when you come back to a file later.

There are multiple ways to format written answers and Rmarkdown is very versatile. What you do comes down to personal preference.

Always check that the Rmd file "knits" before submitting!

Example of a properly answered question

1. **If there are 7 cats and 5 dogs, how many animals are there in total?**

```
cats = 7 # there are 7 cats
dogs = 5 # there are 5 dogs
animals = cats + dogs # the sum of cats and dogs gives us the total animals

animals # this will print out the number of animals
```

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

From the calculations above, we can see that there are 12 animals in total.

You can also use `` `r` `` to reference calculations within your text , which has the advantage that you don't need to update your text if specific numbers change. For example:

We can see that there are 12 animals in total.

R Basics

About R

The R software we are using this semester is open-source statistical software that has gained rapid popularity because of its power and flexibility. In addition, there are a large number of "packages" for R that have been written by users and are freely downloadable from CRAN (Comprehensive R Archive Network). Individual packages do everything from allowing R to interface with supercomputers to solving sudoku puzzles. They contain most every classical statistical test you're likely to come across as well as interfaces that allow R to interact with a large number of other programs and software libraries. Unlike many pieces of software you may be familiar with, R is a scripting language. Usually you will be using R "interactively" which means that the basic mode of operation is to type commands at a command prompt and have it spit back a result, which you'll often want to cut-and-paste elsewhere. R can also be run in "batch" mode, whereby a file containing a list of R commands is run all at once. This mode is particularly useful for large analyses that take a long time to run because batch jobs can be submitted to computer clusters.

What R has to offer

Through your web-browser go to http://www.r-project.org.

- Please briefly look over the "What is R?" section.
- Next, go to the "Manuals" section. This section gives an overview of some of the on-line documentation you may want to use from time to time to gain a greater understanding of how R works and to solve problems you come across. Some of today's activity is borrowed from the "Introduction to R". You are encouraged to read this section later on your own, especially if you have no previous familiarity with R or any other programming language.
- Next, go to the "Search" section and click on the link to "https://search.r-project.org". In the box provided, enter "mantel test." You'll find this gives you a list of different R packages that have different forms of mantel tests (a test of correlation between two matrices). Now open a new tab and go to the generative AI of your choice, like Google Gemini or ChatGPT, and ask it to produce both pseudodata and code to run a mantel test in R. Now turn to your neighbor - did you both get the exact same package and code? If there are differences what are they?

```
I am doing this on my own, but copilot gave me the package "vegan". It created pseudodata that had comu
```

- Next, ask the genAI to explain a mantel test to you. Again, check with your neighbor - do you have the same explanation? Are the differences minor (wording, examples) or major (conceptual disagreements)?
- Back to r-project.org, go to the "CRAN" section, and select one of the sites – it doesn't matter which one since they are identical "mirrors" of each other.
- Click on "Task Views" and then select one of the "tasks" that you find interesting. You will then be presented with a brief overview of major subtopics within that area and the R packages that are useful for those types of problems. These summaries are often an efficient way to familiarize yourself with what R has to offer for a specific type of analysis but are not exhaustive because new packages are constantly being added to R and not all types of analysis have a "Task View".

- Now ask the genAI to explain the topic that you chose. Does the explanation make sense to you? Did it unnecessarily relate your new topic to the mantel test?
- On the left side of your Task View page, click on "Packages" and then "Table of available packages" and you will see a long list of all of the submitted R packages. Look around a bit and then click on one you find interesting. Here you will see basic info on the package including the "Depends" which is the list of packages that you need in order to use this package. Click on the PDF "Reference manual". You'll find that the R documentation for a package describes the inputs and outputs of all the functions in the package and often provides examples of what code calling this function might look like. However, most packages don't describe how a test works, its assumptions, or why you might want to use it – you'll usually want to look up info about a test rather than apply it blindly.

R basics

Lets see how R does some basic arithmetic. Note that in the examples below that a pound sign (#) indicates a comment in the code. Everything after a # is not read by R and is just there for the benefit of the person reading the code (so you don't have to type it all in). Within the Console window (bottom left) type the following at the command prompt symbol '>':

```
3+12            ## addition
5 - pi            ## subtraction
2*8               ## multiplication
14/5                ## division
14 %/% 5          ## integer division
14 %% 5         ## modulus (a.k.a. remainder)
sqrt(25)          ## square root
z = 5               ## assignment to a variable
z                   ## value of a variable
y = sqrt(36)    ## square root
y
```

The window should show

```
3+12            ## addition
```

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

```
5 - pi            ## subtraction
```

```
## [1] 1.858407
```

```
2*8               ## multiplication
```

```
## [1] 16
```

```
14/5                ## division
```

```
## [1] 2.8
```

```
14 %/% 5          ## integer division
```

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

```
14 %% 5         ## modulus (a.k.a. remainder)
```

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

```
sqrt(25)          ## square root
```

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

```
z = 5               ## assignment to a variable
z                   ## value of a variable
```

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

```
y = sqrt(36)    ## square root
y
```

```
## [1] 6
```

The number [1] before the answers just means that this item is the first element of a vector (vectors can be thought of as a collection of related values, such as a column in a data table). From these examples we can see a few things about R. First is that it knows the value of common constants (e.g. /pi) and can use them like numbers. Second, that we can assign both constants and the results of functions to variables and we can see the value of a variable by entering it at the command prompt. This also demonstrates "sqrt", the square root function, and that values are passed to functions within parenthesis. Typically, values passed to function are referred to as the arguments of the functions. Make sure you understand what each operator (+, -, *, /, %%, %/%) does before you proceed. Some of the other common mathematical operators in R that you should try running are:

```
10^2                    ## power
exp(1)                    ## exponential
log(10)                 ## natural logarithm (i.e. ln)
log10(10)                 ## log base 10
log2(10)            ## log base 2
sin(pi/2)                 ## sine
cos(pi)                 ## cosine
tan(pi/4)                 ## tangent
asin(0.5)                 ## arc-sine
acos(0.5)                 ## arc-cosine
atan(1)*180/pi          ## arc-tangent
atan2(-1,-1)*180/pi ## arc-tangent (alternate version)
factorial(10)             ## factorial
```

Note that the atan2 function takes TWO arguments. In R there are many functions that can take more than one argument, and these arguments are always separated by commas. To get information on a function in R you can precede the function name with a ?. So if we wanted to look up what the two arguments of the atan2 function are, we could type

```
?atan2
```

The rest of this activity will often list the names of functions you might find useful, and ? can be used to find out about the syntax of these functions. If on the other hand you are looking for a command (e.g. because you've forgotten its name) you can use help.search. For example:

```
help.search("mantel")
```

This will return a list of relevant functions with the parenthesis after the name indicating what package it is in. help.search will only search packages you have already installed, not all the ones that exist – you'll want to use CRAN to find new packages.

Within RStudio you can also search for help with commands in the **Help** tab in the bottom-right window. This search window combines the functionality of both ? and help.search.

To get new functions in packages that you do not currently have installed, you will have to look up the package you want to use (e.g., on CRAN) and then install it. Packages are installed using the 'Install Packages' button under the "Packages" tab or by using 'install.packages' function on the command line. Packages are loaded by clicking the check box next to a packages name or by using the 'library' function at the command line. Packages only need to be installed once but need to be loaded every time you start R (hint: you'll probably want to list library commands near the top of your script files [described below])

Questions:

There are 26 questions below. Decide now whether you will answer the EVEN or ODD questions. There is no meaningful difference between them in content or difficulty - just choose.

```
evenORodd = "Odd"
```

Now that you have chosen, answer every Odd question on your own, and for the rest use generative AI. As you are going through the questions, be mindful of how you think and feel when you tackle each type of question.

1. Evaluate the following:

a. ln(1)
b. ln(0)
c. ln(e)
d. ln(-5)
e. -ln(5)
f. ln(1/5)

```
log(1)
```

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

```
log(0)
```

```
## [1] -Inf
```

```r
log(exp(1))
```

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

```r
log(-5)
```

```
## Warning in log(-5): NaNs produced
```

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

```r
-log(5)
```

```
## [1] -1.609438
```

```r
log(1/5)
```

```
## [1] -1.609438
```

2. Looking back at the Triginometry section, why are we multiplying the results of the atan and atan2 functions by 180/pi?

All of R's trigonometric functions and their inverse functions use radians, not degrees.

This is one of those "oh right, R uses radians" moments. It's not conceptually hard, but it's easy to forget if you're switching between math, GIS, and ecological modeling. The conversion feels mechanical once you've done it a few times, but the first time you see atan2(-1, -1) return something like -2.356 you immediately realize you need degrees to interpret it intuitively. If you want, we can also walk through what each of those trig functions returns and why they matter in ecological modeling (e.g., bearings, circular statistics, movement paths).

3. What is the difference between log and log10? (hint: use `help()`)

log() takes the natural log while log10() takes log besed 10

4. Given a right triangle with sides x = 5 and y = 13, calculate the length of the hypotenuse (show code and use variables)

```r
x <- 5
y <- 13

hypotenuse <- sqrt(x^2 + y^2)
hypotenuse
```

```
## [1] 13.92839
```

5. Subtract the month you were born from the remainder left after you divide the year you were born by the day you were born (show code)

```
year = 1999 ##Birthday
month = 12
day = 22

year/day ## = 90.86364
```

```
## [1] 90.86364
```

```
remainder = year/day - 90

remainder - month
```

```
## [1] -11.13636
```

ANSWER: -11.13636

R Scripts

Now click on File>New File>R Script to open up a new script window. It is often useful to work on R from a script window (or a new Rmd) because it provides a record of what you did in your analysis and can be reused for similar analyses. It is particularly essential for more complicated analyses. From the File > New File menu you'll also see that R Studio supports a wide range of R file formats (R scripts, Rmd, R notebooks, Quarto) as well as a wide range of other programing languages.

In the script window type

```
x = 1:10
x
#y
```

Unlike at the command prompt nothing happens when you hit return at the end of a line. Highlight the code and hit the Run button. At the command line you should see

```
x = 1:10
x
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
#y
```

Here the result, a vector of ten values from 1 to 10, demonstrates the basic R syntax for creating a sequence of numbers. This example also demonstrates that the comment character '#' also works in scripts. Putting comments in scripts is very useful for remembering what you did when you come back to a file later. For the rest of this activity I'll use boxes to indicate text to be typed in and run, and will use > to indicate that it should be typed on the command prompt and no prompt to indicate that you probably want to type it in a script or Rmd instead. There are a number of other ways of generating useful sequences besides the ':' syntax

```r
seq(1,10,by=0.5)
```

```
##  [1]  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0  5.5  6.0  6.5  7.0  7.5  8.0
## [16]  8.5  9.0  9.5 10.0
```

```r
seq(1,by=0.5,length=10)
```

```
##  [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5
```

```r
rep(1,10)
```

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

```r
x=seq(0,3,by=0.01)
```

In all cases you need to provide the first value in a sequence (the first argument to seq and rep), and after that you need to provide some combination of step size (by), length, and finishing value. Most functions in R can be applied to vectors of data, not just individual data points. Indeed, many only make sense when applied to vectors, such as the following that calculate sums, first differences, and cumulative sums.

```r
sum(1:10)   ## sum up all values in a vector
```

```
## [1] 55
```

```r
diff(1:10)      ## calculate the differences between adjacent values in a vector
```

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

```r
cumsum(1:10)        ## cumulative sum of values in a vector
```

```
##  [1]  1  3  6 10 15 21 28 36 45 55
```

```r
prod(1:10)      ## product of values in a vector
```

```
## [1] 3628800
```

Questions: 6. Describe the difference in output between sum and cumsum.

Both functions operate on vectors, but they return very different types of output sum(x) Returns one single number. That number is the total of all elements in the vector. cumsum(x) Returns a vector of the same length as the input. Each position shows the running total up to that point

7. Generate a sequence of even numbers from -6 to 6

```r
z = -6:6
z
```

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

8. Generate a sequence of values from -4.8 to -3.43 that is length 8 (show code)

```
seq_vals <- seq(from = -4.8, to = -3.43, length.out = 8)
seq_vals
```

```
## [1] -4.800000 -4.604286 -4.408571 -4.212857 -4.017143 -3.821429 -3.625714
## [8] -3.430000
```

- What is the difference between values in this sequence?

```
seq_vals[2] - seq_vals[1]
```

```
## [1] 0.1957143
```

9. What is the sum of the exponential of the sequence in question 8?

```
sum(exp(seq_vals))
```

```
## [1] 0.144133
```

Loading and Saving Data

There are a number of ways to get information into and out of R, but the most simple is in ASCII text formats, such as tab-delimited (.txt) or comma-separated (.csv). It's usually straightforward to export data in one of these formats from most any program (e.g. Excel). Lets begin by opening the "frogs.txt" file in the "data" folder. This data and some of the examples below come from Ben Bolker's handy book "Ecological Models and Data in R".

Note that if you just click on the file from within the **Files** tab, or try to open the file from File > "Open File..." " that this opens the file as a text file, but it it doesn't load the data into R in any way we can use it. Instead, we'll want to run the following command

```
dat = read.table("data/frogs.txt",header=TRUE)
```

The second variable "header=TRUE" informs R that your data file has column headers that should be read rather than treated as just another line of data. For the above command to work **R has to be looking at the correct folder**. You can find out what folder R is currently looking at (its "working directory") using "getwd" and you can change that directory using "setwd" or within the "Files" window tab under **More > Set As Working Directory**.

AUTO-COMPLETE

Be aware that RStudio has the capacity to auto-complete function names, function arguments, and file names. So, for example, if you type 'read.t' and then hit TAB, RStudio will finish typing read.table and it would also show what information you can specify for the read.table function. If you type read.table(and then hit TAB, RStudio will allow you to select the function argument that you want to fill in. If you type read.table(" and then hit TAB, RStudio will show you the files in your current working directory and allow you to select one. If there are a lot of files in the directory, you can start typing the file name you want and then hit TAB again and RStudio will limit what it shows to just those files that match what you've typed so far. The same 'search' functionality applies to file names as well. So, if you type just 're' and then hit TAB, then RStudio will show you all function that begin with re.

For saving ASCII data there is an equivalent command "write.table". Type ?read.table and ?write.table to learn more about these functions. While read.table and write.table can read and write CSV (comma separated value) files by just specifying the 'sep' argument as sep = "," (i.e. a comma in quotes), there are also predefined functions `read.csv` and `write.csv`.

```
write.table(dat,"my_frogs.csv",row.names=FALSE,sep=",")     ## save as CSV
```

R also has a built-in binary data format that is good for saving results for later use in R and can store any number of data types of different shapes and sizes (note just single tables). The function "save" is used to save data

```
save(dat,x,y,z,file="Tutorial1.RData")
```

This command saves any number of data objects, separated by commas, that come before the 'file=' argument, which tells the function the name of the file you want to write to. This data can then be reloaded at a later time, or on a different computer, using "load"

```
load("Tutorial1.RData")
```

There is also a "save.image" function that saves every variable you have defined so far

```
save.image("Tutorial1_all.RData")
```

These commands will be very helpful if you don't finish a activity by the end of the period and want to take your whole R desktop home, for saving work in progress, or archiving results of analyses. When you quit R you will be asked if you want to save your desktop and if you answer 'y', then save.image is called by default and will save to a file simply named ".Rdata" which is automatically loaded the next time you start R. While this is convenient, I actually recommend against it and suggest using save or save.image explicitly instead because otherwise it is very easy to accidentally use variables and data sets defined in previous analyses, or to be unsure which version of an analysis you're working with. You can always see what variables you currently have defined in the **Environment** window or by using the command

```
ls()
```

```
##  [1] "animals"    "cats"       "dat"        "day"        "dogs"
##  [6] "evenORodd"  "hypotenuse" "month"      "remainder"  "seq_vals"
## [11] "x"          "y"          "year"       "z"
```

Within the Environment window, clicking on a variable will show you the contents in a spreadsheet-like format in the script window. Finally, while we won't use these explicitly in this class, there are a large number of other options for getting data in and out of R in specific formats (e.g. GIS data, image data, etc) and ways to connect R to data sources more dynamically (e.g. SQL databases) that can be particularly useful when dealing with large data sets. The **R Data Import/Export** manual on the R website is a place to start to learn more about moving data in and out of R.

Question:

10. Use R commands (e.g., do not click "import dataset" and use the dialog there) to read in the file `met_hourly.csv` and assign it the variable name `met`

```
met <- read.csv("data/met_hourly.csv", header = TRUE)
```

11. Save the vector `x` to a new csv file

```
write.table(x, "data/x.csv",row.names=FALSE,sep=",")
```

Data types and dimensions

One of the first things you'll do with any data set when you first load it up is some basic checks to see what you are dealing with. Typing the variable name will show you its contents, but if you just loaded up something with a million entries then you'll sit for a long time as R lists every number on the screen. The function class will tell you the type of data you've just loaded.

```
class(dat)
```

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

In this case the data is in a "data.frame", which is like a matrix but can also contain non-numeric data. The basic (or atomic) data types in R are integer, numeric (decimal), logical (TRUE/FALSE), factors, and character. Character data in R is usually displayed in double quotes to indicate that it is character data (e.g. the character "1" rather than the number 1). When writing character data in R (e.g. file names in read.data) it is necessary to use double quotes as well so that R can tell the difference between character data and the names of variables and functions. By contrast, R usually reads character data from files correctly even if the data isn't in quotes. In addition to the basic data types in R, there are a wide variety of derived data types built up from these basic types that are used for a wide range of purposes. A common example is the Date type, which can be useful for analyzing and plotting data through time, and which you can learn more about by looking at the help for **as.Date** and **strptime**.

At the most basic level R organizes data into vectors of data of a given data type and each column of a data.frame consists of a vector. R also has a matrix data type, which must contain data of a single basic data type (usually numeric). It is important to be aware of data types because certain operations can only be applied to certain data types (e.g. you can multiply two matrices but not two data frames).

```
str(dat)
```

```
## 'data.frame':    20 obs. of  4 variables:
##  $ frogs   : num  1.1 1.3 1.7 1.8 1.9 2.1 2.3 2.4 2.5 2.8 ...
##  $ tadpoles: num  2.04 2.88 3.06 3.71 3.96 ...
##  $ color   : chr  "red" "red" "red" "red" ...
##  $ spots   : logi  TRUE FALSE TRUE FALSE TRUE FALSE ...
```

will tell you the basic structure of the data. From these we learn that there are four columns of data named "frogs", "tadpoles", "color", "spots" and that there are 20 rows of data, and that the data is numeric for the first two, a factor for the third, and logical for the fourth. If we didn't need all this information

```
names(dat)
```

```
## [1] "frogs"    "tadpoles" "color"    "spots"
```

will tell you the names of the columns in you data frame.

```r
dim(dat)
```

```
## [1] 20  4
```

dim will tell you the dimensions of the data, in this case [1] 20 4 which means we 20 rows and 4 columns. Each of these pieces of information is accessible individually using **nrow** and **ncol**.

```r
nrow(dat)
```

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

```r
ncol(dat)
```

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

Note that dim will not work on a single vector (e.g. dim(x)) but that **length(x)** will tell you the length of a vector.

```r
head(dat)
```

```
##   frogs tadpoles color spots
## 1   1.1 2.036982   red  TRUE
## 2   1.3 2.876231   red FALSE
## 3   1.7 3.062528   red  TRUE
## 4   1.8 3.707180   red FALSE
## 5   1.9 3.955385   red  TRUE
## 6   2.1 4.786983   red FALSE
```

```r
tail(dat)
```

```
##    frogs tadpoles color spots
## 15   3.9 7.681658  blue  TRUE
## 16   4.1 8.103331  blue FALSE
## 17   4.5 8.575123  blue  TRUE
## 18   4.8 9.629233  blue FALSE
## 19   5.1 9.791165  blue  TRUE
## 20   5.3 9.574846  blue FALSE
```

The functions **head** and **tail** show just the first and last few lines of a data set, respectively

```r
summary(dat)
```

```
##      frogs          tadpoles        color              spots
##  Min.   :1.100   Min.   :2.037   Length:20          Mode :logical
##  1st Qu.:2.050   1st Qu.:4.547   Class :character   FALSE:10
##  Median :2.950   Median :5.845   Mode  :character   TRUE :10
##  Mean   :3.065   Mean   :6.081
##  3rd Qu.:3.950   3rd Qu.:7.961
##  Max.   :5.300   Max.   :9.791
```

will give you basic summary statistics on a data set Data within vectors, matrices, and data frames can be accessed using [ ] notation. Subsets of data can also be accessed by specifying just rows, just columns, or ranges within either. These are often referred to as subscripts or indices and the first is the row number while the second is the column.

```
x[5]       # select the 5th element only
```

```
## [1] 0.04
```

```
dat          # select the entire data frame
```

```
##    frogs tadpoles color spots
## 1    1.1 2.036982   red  TRUE
## 2    1.3 2.876231   red FALSE
## 3    1.7 3.062528   red  TRUE
## 4    1.8 3.707180   red FALSE
## 5    1.9 3.955385   red  TRUE
## 6    2.1 4.786983   red FALSE
## 7    2.3 4.909395   red  TRUE
## 8    2.4 4.743633  blue FALSE
## 9    2.5 5.458514   red  TRUE
## 10   2.8 5.488370  blue FALSE
## 11   3.1 6.463224   red  TRUE
## 12   3.3 6.202578  blue FALSE
## 13   3.6 7.913878  blue  TRUE
## 14   3.7 6.666590   red FALSE
## 15   3.9 7.681658  blue  TRUE
## 16   4.1 8.103331  blue FALSE
## 17   4.5 8.575123  blue  TRUE
## 18   4.8 9.629233  blue FALSE
## 19   5.1 9.791165  blue  TRUE
## 20   5.3 9.574846  blue FALSE
```

```
dat[5,1]     # select the entry in the 5th row, 1st column
```

```
## [1] 1.9
```

```
dat[,2]      # select all rows of the second column
```

```
##  [1] 2.036982 2.876231 3.062528 3.707180 3.955385 4.786983 4.909395 4.743633
##  [9] 5.458514 5.488370 6.463224 6.202578 7.913878 6.666590 7.681658 8.103331
## [17] 8.575123 9.629233 9.791165 9.574846
```

```
dat[1:5,]    # select rows 1 through 5, all columns
```

```
##   frogs tadpoles color spots
## 1   1.1 2.036982   red  TRUE
## 2   1.3 2.876231   red FALSE
## 3   1.7 3.062528   red  TRUE
## 4   1.8 3.707180   red FALSE
## 5   1.9 3.955385   red  TRUE
```

```
dat[6:10,2:3]    # select rows 6 through 10, columns 2 and 3
```

```
##    tadpoles color
## 6  4.786983   red
## 7  4.909395   red
## 8  4.743633  blue
## 9  5.458514   red
## 10 5.488370  blue
```

We can also refer to specific columns of data by name using the $ syntax

```
dat$frogs   # show just the 'frogs' column
```

```
##  [1] 1.1 1.3 1.7 1.8 1.9 2.1 2.3 2.4 2.5 2.8 3.1 3.3 3.6 3.7 3.9 4.1 4.5 4.8 5.1
## [20] 5.3
```

```
dat$color[6:10]     # show the 6th though 10th elements of the color column
```

```
## [1] "red"  "red"  "blue" "red"  "blue"
```

In general, it is better to **access data by name**, rather than using the row and column numbers, because this makes your code easier to understand and debug, making the process of coding less error prone. It also makes it much easier to adapt your code to new situations or data sets, where the columns might not come in the same order, or the data might not have the same number of rows or columns. This highlights a more general point, that you should use variables to represent names and numbers, especially if those names and numbers are reused, rather than 'hard coding' numeric values into the code. Finally, there are functions for converting data from one data type to another

```
as.character(dat$color)
```

```
##  [1] "red"  "red"  "red"  "red"  "red"  "red"  "red"  "blue" "red"  "blue"
## [11] "red"  "blue" "blue" "red"  "blue" "blue" "blue" "blue" "blue" "blue"
```

```
as.numeric(dat$spots)
```

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

```
as.logical(0:1)
```

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

```
as.matrix(dat)
```

```
##       frogs tadpoles   color  spots
##  [1,] "1.1" "2.036982" "red"  "TRUE"
##  [2,] "1.3" "2.876231" "red"  "FALSE"
##  [3,] "1.7" "3.062528" "red"  "TRUE"
##  [4,] "1.8" "3.707180" "red"  "FALSE"
```

```
##  [5,] "1.9" "3.955385" "red"  "TRUE"
##  [6,] "2.1" "4.786983" "red"  "FALSE"
##  [7,] "2.3" "4.909395" "red"  "TRUE"
##  [8,] "2.4" "4.743633" "blue" "FALSE"
##  [9,] "2.5" "5.458514" "red"  "TRUE"
## [10,] "2.8" "5.488370" "blue" "FALSE"
## [11,] "3.1" "6.463224" "red"  "TRUE"
## [12,] "3.3" "6.202578" "blue" "FALSE"
## [13,] "3.6" "7.913878" "blue" "TRUE"
## [14,] "3.7" "6.666590" "red"  "FALSE"
## [15,] "3.9" "7.681658" "blue" "TRUE"
## [16,] "4.1" "8.103331" "blue" "FALSE"
## [17,] "4.5" "8.575123" "blue" "TRUE"
## [18,] "4.8" "9.629233" "blue" "FALSE"
## [19,] "5.1" "9.791165" "blue" "TRUE"
## [20,] "5.3" "9.574846" "blue" "FALSE"
```

Questions: 12. Show just the spots data as characters

```
spots_char <- as.character(dat$spots)
spots_char
```

```
##  [1] "TRUE"  "FALSE" "TRUE"  "FALSE" "TRUE"  "FALSE" "TRUE"  "FALSE" "TRUE"
## [10] "FALSE" "TRUE"  "FALSE" "TRUE"  "FALSE" "TRUE"  "FALSE" "TRUE"  "FALSE"
## [19] "TRUE"  "FALSE"
```

13. Show the 3rd through 8th rows of the 1st though 3rd columns

```
q13 <- dat[3:8, 1:3]
q13
```

```
##   frogs tadpoles color
## 3   1.7 3.062528   red
## 4   1.8 3.707180   red
## 5   1.9 3.955385   red
## 6   2.1 4.786983   red
## 7   2.3 4.909395   red
## 8   2.4 4.743633  blue
```

14. Show just the odd numbered rows in the frog data. Write this code for the GENERAL CASE (i.e. don't just type c(1,3,5,...) but use functions that you learned in previous sections to set up the sequence.

```
odd_rows <- dat[ seq(from = 1, to = nrow(dat), by = 2), ]
odd_rows
```

```
##    frogs tadpoles color spots
## 1    1.1 2.036982   red  TRUE
## 3    1.7 3.062528   red  TRUE
## 5    1.9 3.955385   red  TRUE
## 7    2.3 4.909395   red  TRUE
## 9    2.5 5.458514   red  TRUE
```

```
## 11   3.1 6.463224   red  TRUE
## 13   3.6 7.913878  blue  TRUE
## 15   3.9 7.681658  blue  TRUE
## 17   4.5 8.575123  blue  TRUE
## 19   5.1 9.791165  blue  TRUE
```

Combining vectors

There is a simple function **c( )** in R that "combines" vectors or numbers into a single vector. You use it like this:

```
x=c(1,7)
x
```

```
## [1] 1 7
```

```
y=c(10:15,3,9)
y
```

```
## [1] 10 11 12 13 14 15  3  9
```

```
c(x,y)
```

```
##  [1]  1  7 10 11 12 13 14 15  3  9
```

Vectors can also be used for indexing other vectors. For example:

```
y[x]        ## return the 1st and 7th element of y
```

```
## [1] 10  3
```

We can also combine vectors to build up data frames by "binding" them together either are rows or as columns

```
p = 1:10
q = 10:1
cbind(p,q)      # bind as columns
```

```
##        p  q
##  [1,]  1 10
##  [2,]  2  9
##  [3,]  3  8
##  [4,]  4  7
##  [5,]  5  6
##  [6,]  6  5
##  [7,]  7  4
##  [8,]  8  3
##  [9,]  9  2
## [10,] 10  1
```

```
rbind(q,p)      # bind as rows
```

```
##   [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## q   10    9    8    7    6    5    4    3    2     1
## p    1    2    3    4    5    6    7    8    9    10
```

cbind and **rbind** can also be applied to existing data frames, for example to add another column to an existing data frame or to take two data sets with the same columns and bind them together by row to make a larger data set.

Question: 15. Create a character vector that contains the names of 4 people you admire.

```
names <- c("Easter Bunny", "Mr. Potato Head", "George Washington", "Michael Scott")
names
```

```
## [1] "Easter Bunny"      "Mr. Potato Head"   "George Washington"
## [4] "Michael Scott"
```

Logical operators and indexing

R can perform standard logical comparisons, which can be very useful for comparing and selecting data. It's important to know the syntax for the different logical operators, some of which are odd:

```
>   greater than
<   less than
>=  greater than or equal to
<=  less than or equal to
==  equal to (TWO equals signs...you were very close!)
!=  not equal
```

As a simple example you could compare individual numbers:

```
1 > 3
```

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

```
5 < 7
```

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

```
4 >= 4
```

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

```
-11 <= pi
```

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

```
log(1) == 0
```

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

```
exp(0) != 1
```

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

You can also combine multiple logical operators using the symbols for 'and' (&) and 'or' (|)

```
w = 4
w > 0 & w < 10
```

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

```
w < 0 | w > 10
```

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

You can also apply logical operators to vectors and matrices. When you type a "logical" expression like "y > x" in R you get a TRUE/FALSE answer of the same shape as the inputs. e.g.:

```
z = y>13
z
```

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

You will notice that by default logical operations are performed element-by-element. If you want to apply a logical test to a whole vector at a time you can use the function **any** to test if any of the values are true and **all** to test if all values are true

```
any(y>13)
```

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

```
all(y>13)
```

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

You also need to know that logical vectors like "z" above can be used as indices for other vectors of the same length. Commonly, you'll use them as indices to one of the vectors that produced them. e.g.:

```
y[z]
```

```
## [1] 14 15
```

Or, skipping the middleman "z":

```
y[y>13]
```

```
## [1] 14 15
```

These simple comparisons can provide a powerful means for subsetting data. These comparisons can also be used in matrices and data frames the same way we were using sequences of row or column numbers above. For example, if you just wanted the rows where there were 3 or more frogs, you could type

```
dat[dat$frogs >= 3,]
```

```
##    frogs tadpoles color spots
## 11   3.1 6.463224   red  TRUE
## 12   3.3 6.202578  blue FALSE
## 13   3.6 7.913878  blue  TRUE
## 14   3.7 6.666590   red FALSE
## 15   3.9 7.681658  blue  TRUE
## 16   4.1 8.103331  blue FALSE
## 17   4.5 8.575123  blue  TRUE
## 18   4.8 9.629233  blue FALSE
## 19   5.1 9.791165  blue  TRUE
## 20   5.3 9.574846  blue FALSE
```

R also has a built in function **subset** for doing this sort of subsetting that takes the data set as the first argument and the condition used for subsetting as the second argument. So the above could also be rewritten as

```
subset(dat,frogs >= 3)
```

```
##    frogs tadpoles color spots
## 11   3.1 6.463224   red  TRUE
## 12   3.3 6.202578  blue FALSE
## 13   3.6 7.913878  blue  TRUE
## 14   3.7 6.666590   red FALSE
## 15   3.9 7.681658  blue  TRUE
## 16   4.1 8.103331  blue FALSE
## 17   4.5 8.575123  blue  TRUE
## 18   4.8 9.629233  blue FALSE
## 19   5.1 9.791165  blue  TRUE
## 20   5.3 9.574846  blue FALSE
```

subset also has an optional 3rd argument for just returning specific columns. So if you wanted to run the previous subset but only needed the columns tadpoles and spots you could run

```
subset(dat,frogs >= 3,c("tadpoles","spots"))
```

```
##    tadpoles spots
## 11 6.463224  TRUE
## 12 6.202578 FALSE
## 13 7.913878  TRUE
## 14 6.666590 FALSE
```

```
## 15 7.681658  TRUE
## 16 8.103331 FALSE
## 17 8.575123  TRUE
## 18 9.629233 FALSE
## 19 9.791165  TRUE
## 20 9.574846 FALSE
```

Note that when your data is characters you'll need double-quotes in your comparison. e.g.

```
a=c("north","south","east","west")
a == "east"
```

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

Two more things about logical vectors. First, sometimes it's easier or necessary to have a list of the indices to the TRUE values only (e.g. when the list includes NA values). The R function "which" is just for this purpose:

```
which(y==3)
```

```
## [1] 7
```

```
# The seventh element of y equals 3
```

And finally, sometimes TRUE/FALSE behave just like 0/1, which can be very useful. For example, this handy syntax:

```
sum(y>13)
```

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

```
# two elements of y are >13
```

Logicals don't work exactly like 0/1 in some situations, so be careful. You can always convert them explicitly with as.numeric( ) too if need be:

```
as.numeric(y>13)
```

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

dplyr: The filter and select functions

In addition to data manipulation functions provided within the base R libraries, there are a lot of external libraries developed to make data management more efficient. One of the most popular such libraries is `dplyr`.

Here we load the library dplyr. If you haven't installed dplyr yet this line of code will throw an error, which you can correct by either using `install.packages("dplyr")` or the Packages > Install menu in the bottom right window of RStudio

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

dplyr includes a function `filter` that also does the sort of subsetting we saw in the previous tasks. I takes the data set as the first argument and the condition used for subsetting as the second argument. So the previous example could also be rewritten as

```
hotData = dplyr::filter(met, AirTemp > 25)
```

`dplyr` also has an function `select` for just selecting specific columns. So if you wanted to subset just the columns `ShortWave` and `Rain` you could run

```
foo = dplyr::select(met, c("ShortWave","Rain"))
head(foo)
```

```
##   ShortWave  Rain
## 1         0 0.000
## 2         0 0.002
## 3         0 0.003
## 4         0 0.000
## 5         0 0.003
## 6         0 0.006
```

We'll see more dplyr commands below, but a quick "cheat sheet" can be found here

Pipes

When cleaning and organizing data it is common to have to string together multiple functions sequentially. For example, if we had data set `x` we might need to first run x through `function1()` and then run the output through `function2()`. In the examples above we did this either by using temporary variables,

```
y = function1(x)
z = function2(y)
```

or by embedding one function call within another

```
z = function2(function1(x))
```

Many consider the first both options cumbersome, as the first creates a lot of temporary variables we don't need and the latter can be confusing, particularly when multiple functions are involved or the functions have additional arguments, because one ends up reading the functions in the reverse order that they are called.

Pipes, `|>`, aim to address this issue by creating a way to pass the output of one function to another in order without creating a temporary variable.

```
z = x |> function1() |> function2()
```

As another example, if we wanted to both filter the met data by temperature AND select a subset of columns, we could pipe together multiple dplyr commands

```
foo = met |> filter(AirTemp > 25) |> select(c("ShortWave","Rain"))
head(foo)
```

```
##   ShortWave Rain
## 1    808.03    0
## 2    833.36    0
## 3    722.89    0
## 4    639.47    0
## 5    542.09    0
## 6    409.41    0
```

In this instance, when used alone both functions took the dataset as the first arguement and then additional information as the second argument. Here, the first argument is implicit and one only needs to use specify the second. An important thing to remember about pipes is that they will *always* pass in the *first* argument in a function.

```
Question
16. **Using the frog data set and either base or dplyr functions:**
*  a.   display just the rows where frogs have spots
*  b.   display just the rows where frogs are blue
*  c.   create a new object containing just the rows where there are between 3 and 5 tadpoles
*  d.   display just the rows where there are less than 2.5 red frogs
*  e.   display where either frogs do not have spots or there are more than 5 frogs
```

```
# a. rows where frogs have spots
dat_spots <- dat %>%
  filter(spots == TRUE)
dat_spots
```

```
##    frogs tadpoles color spots
## 1    1.1 2.036982   red  TRUE
## 2    1.7 3.062528   red  TRUE
## 3    1.9 3.955385   red  TRUE
## 4    2.3 4.909395   red  TRUE
## 5    2.5 5.458514   red  TRUE
## 6    3.1 6.463224   red  TRUE
## 7    3.6 7.913878  blue  TRUE
## 8    3.9 7.681658  blue  TRUE
## 9    4.5 8.575123  blue  TRUE
## 10   5.1 9.791165  blue  TRUE
```

```r
# b. rows where frogs are blue
dat_blue <- dat %>%
  filter(color == "blue")
dat_blue
```

```
##    frogs tadpoles color spots
## 1    2.4 4.743633  blue FALSE
## 2    2.8 5.488370  blue FALSE
## 3    3.3 6.202578  blue FALSE
## 4    3.6 7.913878  blue  TRUE
## 5    3.9 7.681658  blue  TRUE
## 6    4.1 8.103331  blue FALSE
## 7    4.5 8.575123  blue  TRUE
## 8    4.8 9.629233  blue FALSE
## 9    5.1 9.791165  blue  TRUE
## 10   5.3 9.574846  blue FALSE
```

```r
# c. rows where tadpoles are between 3 and 5
dat_3to5 <- dat %>%
  filter(tadpoles >= 3, tadpoles <= 5)
dat_3to5
```

```
##   frogs tadpoles color spots
## 1   1.7 3.062528   red  TRUE
## 2   1.8 3.707180   red FALSE
## 3   1.9 3.955385   red  TRUE
## 4   2.1 4.786983   red FALSE
## 5   2.3 4.909395   red  TRUE
## 6   2.4 4.743633  blue FALSE
```

```r
# d. rows where there are less than 2.5 red frogs
dat_red_low <- dat %>%
  filter(color == "red", frogs < 2.5)
dat_red_low
```

```
##   frogs tadpoles color spots
## 1   1.1 2.036982   red  TRUE
## 2   1.3 2.876231   red FALSE
## 3   1.7 3.062528   red  TRUE
## 4   1.8 3.707180   red FALSE
## 5   1.9 3.955385   red  TRUE
## 6   2.1 4.786983   red FALSE
## 7   2.3 4.909395   red  TRUE
```

```r
# e. rows where frogs do NOT have spots OR there are more than 5 frogs
dat_spots_or_big <- dat %>%
  filter(!spots | frogs > 5)
dat_spots_or_big
```

```
##    frogs tadpoles color spots
## 1    1.3 2.876231   red FALSE
```

```
## 2    1.8 3.707180   red FALSE
## 3    2.1 4.786983   red FALSE
## 4    2.4 4.743633  blue FALSE
## 5    2.8 5.488370  blue FALSE
## 6    3.3 6.202578  blue FALSE
## 7    3.7 6.666590   red FALSE
## 8    4.1 8.103331  blue FALSE
## 9    4.8 9.629233  blue FALSE
## 10   5.1 9.791165  blue  TRUE
## 11   5.3 9.574846  blue FALSE
```

Plots, tables, and exploratory analysis

Often understanding our data requires more than just being able to subset the raw data, but also the ability to summarize and visualize data. The **table** command can do basic tabulation and cross tabulation of data

```
table(dat$color)
```

```
##
## blue  red
##   10   10
```

```
table(dat$color,dat$spots)
```

```
##
##        FALSE TRUE
##   blue     6    4
##   red      4    6
```

There are also a number of commands for calculating basic statistical measures

```
mean(dat$frogs)
```

```
## [1] 3.065
```

```
median(dat$tadpoles)
```

```
## [1] 5.845474
```

```
var(dat$frogs)              ## variance
```

```
## [1] 1.6245
```

```
sd(dat$frogs)               ## standard deviation
```

```
## [1] 1.274559
```

```
cov(met$AirTemp,met$LongWave)       ## covariance
```

```
## [1] 500.5275
```

```
cor(met$AirTemp,met$LongWave)       ## correllation
```

```
## [1] 0.8425125
```

```
quantile(dat$tadpoles,c(0.05,0.90))     ## 5% and 95% quantiles
```

```
##       5%      90%
## 2.834268 9.580285
```

```
min(dat$frogs)              ## smallest value
```

```
## [1] 1.1
```

```
max(dat$frogs)              ## largest value
```

```
## [1] 5.3
```

Apply

R also has a set of apply functions for applying any function to sets of values within a data structure.

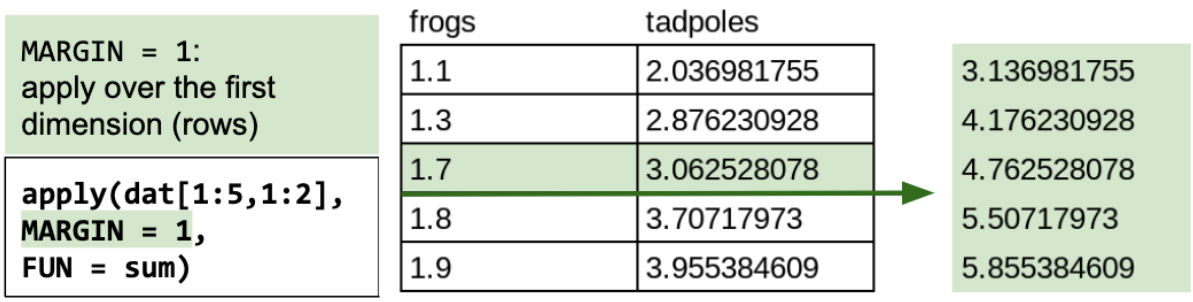
```
apply(dat[,1:2],1,sum)              # calculate sum of frogs & tadpoles by row (1st dimension)
```

```
##  [1]  3.136982  4.176231  4.762528  5.507180  5.855385  6.886983  7.209395
##  [8]  7.143633  7.958514  8.288370  9.563224  9.502578 11.513878 10.366590
## [15] 11.581658 12.203331 13.075123 14.429233 14.891165 14.874846
```

```
apply(dat[,1:2],2,sum)              # calculate sum of frogs & tadpoles by column (2nd dimension)
```

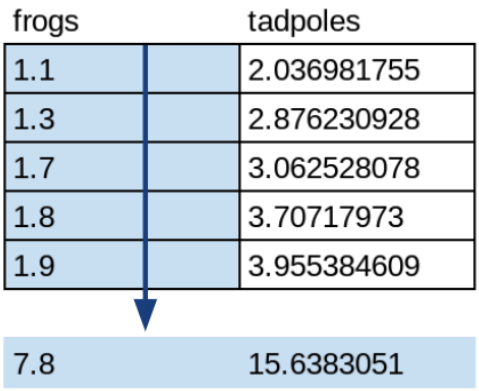
```
##    frogs tadpoles
##  61.3000 121.6268
```

The function *apply* will apply a function to either every row (dimension 1) or every column (dimension 2) of a matrix or data.frame. In this example the commands apply the "sum" function to the first two columns of the data (frogs & tadpoles) first calculated by row (the total number of individuals in each population) and second by column (the total number of frogs and tadpoles)



MARGIN = 2: apply over the second dimension (columns)

```
apply(dat[1:5,1:2], MARGIN = 2, FUN = sum)
```




```
tapply(dat$frogs,dat$color,mean)          # calculate mean of frogs by color
```

```
## blue  red
## 3.98 2.15
```

```
tapply(dat$frogs,dat[,c("color","spots")],mean)  # calculate mean of frogs by color & spots
```

```
##       spots
## color     FALSE  TRUE
##   blue 3.783333 4.275
##   red  2.225000 2.100
```

color	spots	frogs	mean
blue	FALSE	2.4	3.78333
blue	FALSE	2.8	
blue	FALSE	3.3	
blue	FALSE	4.1	
blue	FALSE	4.8	
blue	FALSE	5.3	
blue	TRUE	3.6	4.27500
blue	TRUE	3.9	
blue	TRUE	4.5	
blue	TRUE	5.1	
red	FALSE	1.3	2.22500
red	FALSE	1.8	
red	FALSE	2.1	
red	FALSE	3.7	
red	TRUE	1.1	2.10000
red	TRUE	1.7	
red	TRUE	1.9	
red	TRUE	2.3	
red	TRUE	2.5	
red	TRUE	3.1	

The function **tapply** will apply a function to an R data object, grouping data according to a second variable or set of variables. The first example applies the "mean" function to frogs grouping them by color. The second shows that tapply can be used to apply a function over multiple groups, in this case color X spots.

`dplyr` also provide an alternative to `tapply` using the `group_by` and `summarize` commands. For example, the previous example could be written as

```
dat |> group_by(color,spots) |> summarise(avg = mean(frogs))
```

```
## `summarise()` has grouped output by 'color'. You can override using the
## `.groups` argument.
```

```
## # A tibble: 4 x 3
## # Groups:   color [2]
##   color spots   avg
##   <chr> <lgl> <dbl>
## 1 blue  FALSE  3.78
## 2 blue  TRUE   4.28
## 3 red   FALSE  2.22
## 4 red   TRUE   2.1
```

Note that the `summarize` function creates a new column to store the output of the operation being performed. In this way `summarize` also makes it easy to calculate multiple summary statistics at once for the same groups. For example:

```
dat |> group_by(color,spots) |>
  summarise(avgFrogs = mean(frogs),
            sdFrogs = sd(frogs),
            avgTadpoles = mean(tadpoles),
            sdTadpoles = sd(tadpoles))
```

```
## `summarise()` has grouped output by 'color'. You can override using the
## `.groups` argument.
```

```
## # A tibble: 4 x 6
## # Groups:   color [2]
##   color spots avgFrogs sdFrogs avgTadpoles sdTadpoles
##   <chr> <lgl>    <dbl>   <dbl>       <dbl>      <dbl>
## 1 blue  FALSE     3.78   1.14         7.29      2.11
## 2 blue  TRUE      4.28   0.665        8.49      0.946
## 3 red   FALSE     2.22   1.04         4.51      1.64
## 4 red   TRUE      2.1    0.693        4.31      1.62
```

Another handy dplyr function is `mutate` which provides the ability to add new columns to a dataframe, for example based on computations involving other columns. For example, if we wanted to add a column to `met` that converted the datetime column `time` to just the calendar date we could do this as

```
met = met |>
  mutate(time = as.POSIXct(time,tz="GMT")) |> ## convert from character to datetime
  mutate(date = as.Date(time))                ## convert from datetime to date
head(met)
```

```
##                  time ShortWave LongWave AirTemp RelHum WindSpeed  Rain Snow
## 1 2013-01-30 00:00:00         0   273.53   -1.43  95.24      6.23 0.000    0
## 2 2013-01-30 01:00:00         0   273.53   -1.64  95.53      6.47 0.002    0
## 3 2013-01-30 02:00:00         0   273.53   -1.86  95.89      6.72 0.003    0
## 4 2013-01-30 03:00:00         0   317.29   -2.07  96.18      6.96 0.000    0
## 5 2013-01-30 04:00:00         0   317.29   -1.65  96.29      6.91 0.003    0
## 6 2013-01-30 05:00:00         0   317.30   -1.24  96.38      6.87 0.006    0
##         date
## 1 2013-01-30
## 2 2013-01-30
## 3 2013-01-30
## 4 2013-01-30
## 5 2013-01-30
## 6 2013-01-30
```

Question
17. **Use dplyr to calculate the *daily* mean air temperature**

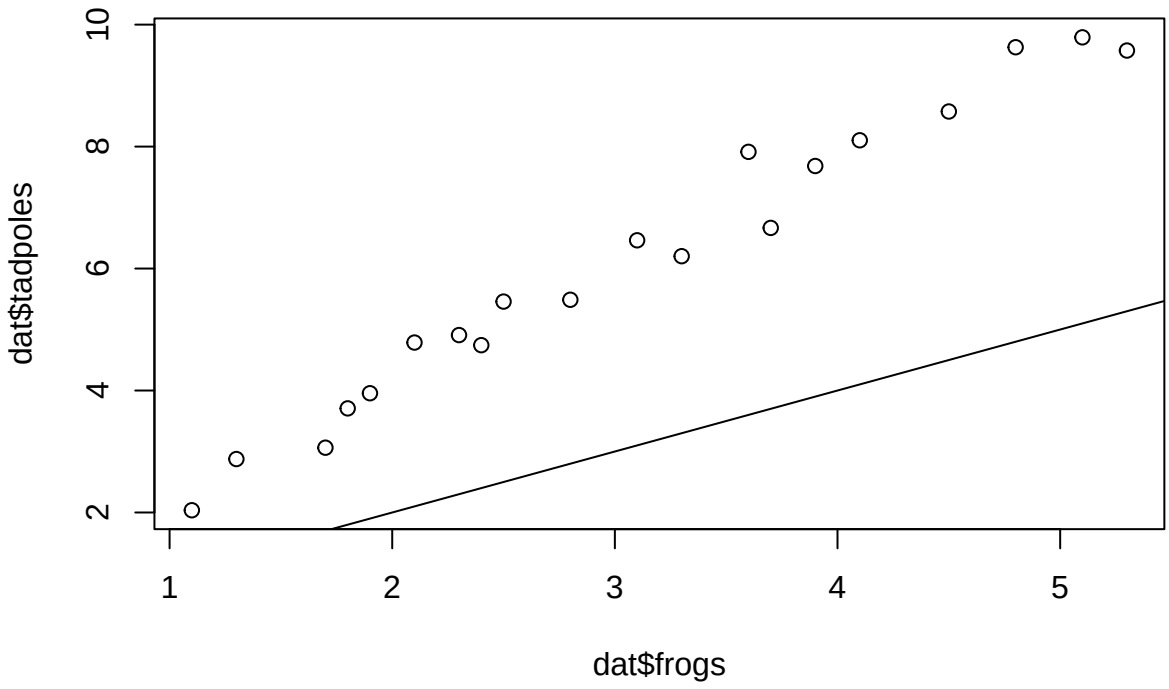
```
mdtemp <- met %>%
  mutate(day = as.Date(time)) %>%
  group_by(day) %>%
  summarise(avgTemp = mean(AirTemp))
head(mdtemp)
```

```
## # A tibble: 6 x 2
##   day        avgTemp
##   <date>       <dbl>
## 1 2013-01-30    1.46
## 2 2013-01-31    4.37
## 3 2013-02-01   -5.30
## 4 2013-02-02   -9.12
## 5 2013-02-03   -6.43
## 6 2013-02-04   -5.89
```

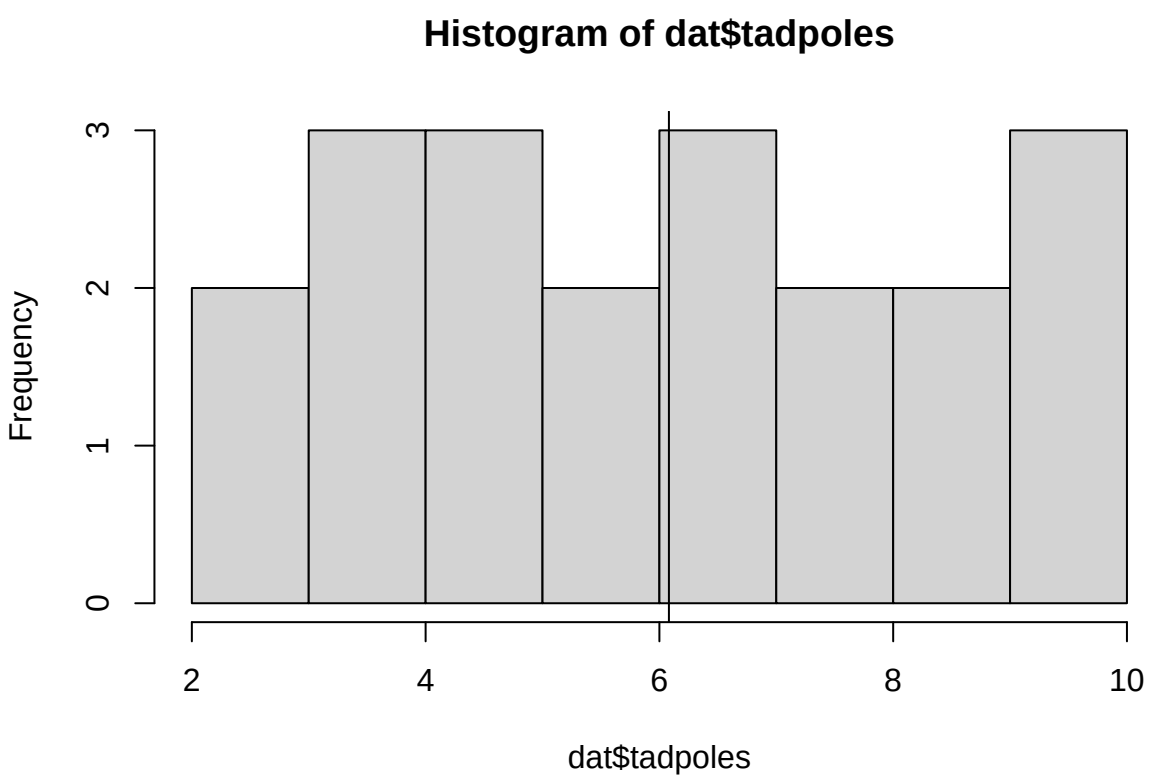
Plotting & visualization

There are a lot of options for plotting data in R. The simplest of these include

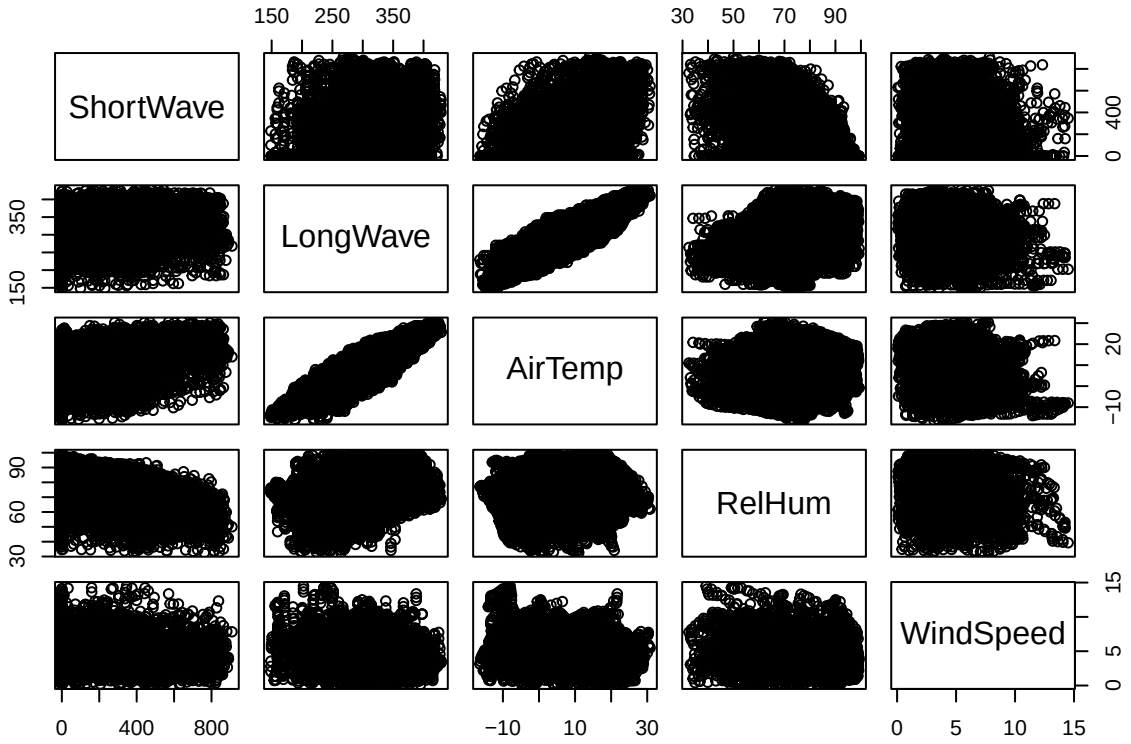
```
plot(dat$frogs,dat$tadpoles)        ## x-y scatter plot
abline(a=0,b=1)                  ## add a 1:1 line (intercept=0, slope=1)
```



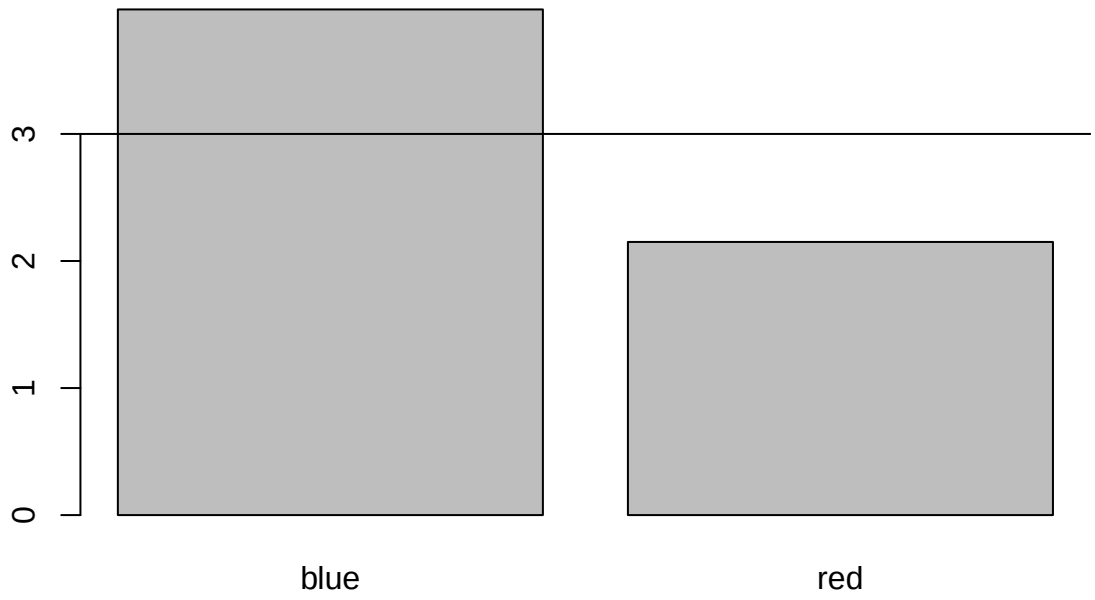

```
hist(dat$tadpoles)                ## histogram
abline(v=mean(dat$tadpoles))              ## add a vertical line at the mean
```




```
pairs(met[,2:6])                    ## pairwise scatter plots for selected columns
```




```
barplot(tapply(dat$frogs,dat$color,mean))       ## barplot of frogs by color
abline(h=3)                  ## add a horizontal line at 3
```



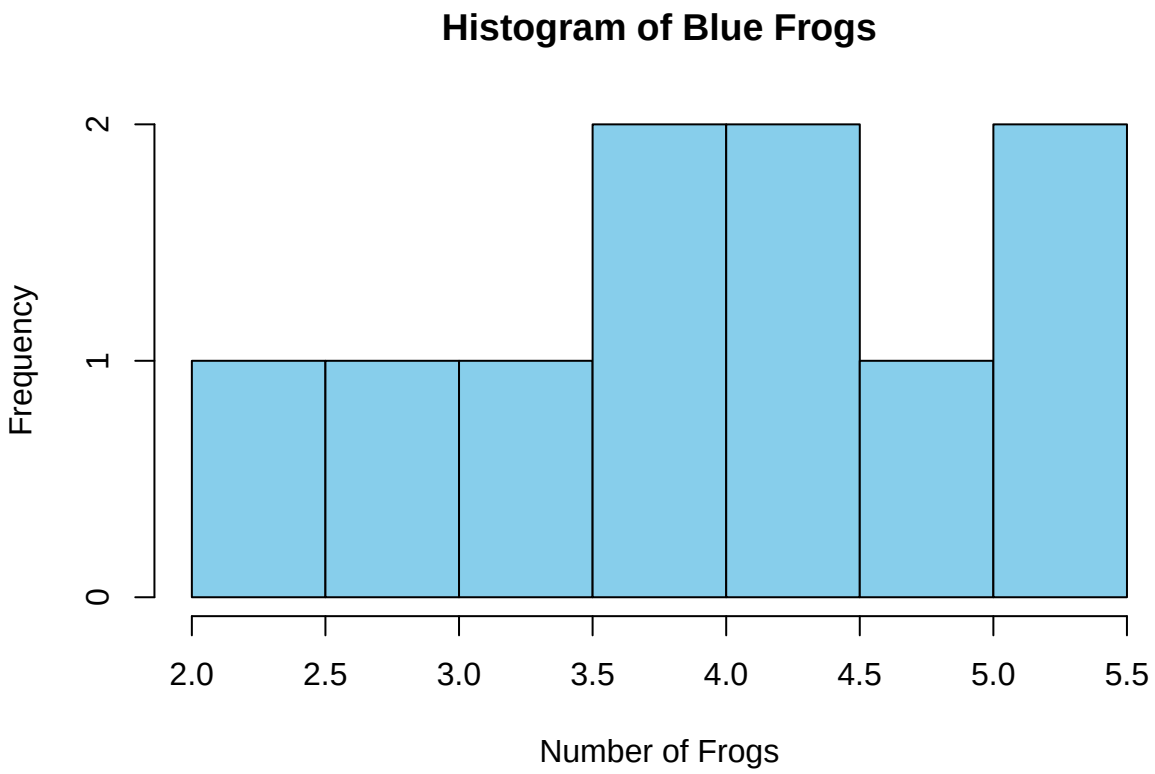
The functions **lines** and **points** are also frequently used to add additional lines and points (respectively) to an existing plot.

Within the Plot window, graphs can be cut-and-pasted into other documents or saved to file fairly simply by using Export. If you want to automate the process of exporting graphics, for example when you generate a whole bunch of figures at once and don't want to Export each one by hand, you'll want to use the graphical functions such as 'postscript', 'pdf', or 'tiff'. For all of these plot functions there are numerous additional (optional) arguments that control the formatting of the plots. The help for par (i.e. ?par) gives a fairly detailed list of these options, some of which you will see in further examples below.

Questions 18. Plot a histogram of blue frogs

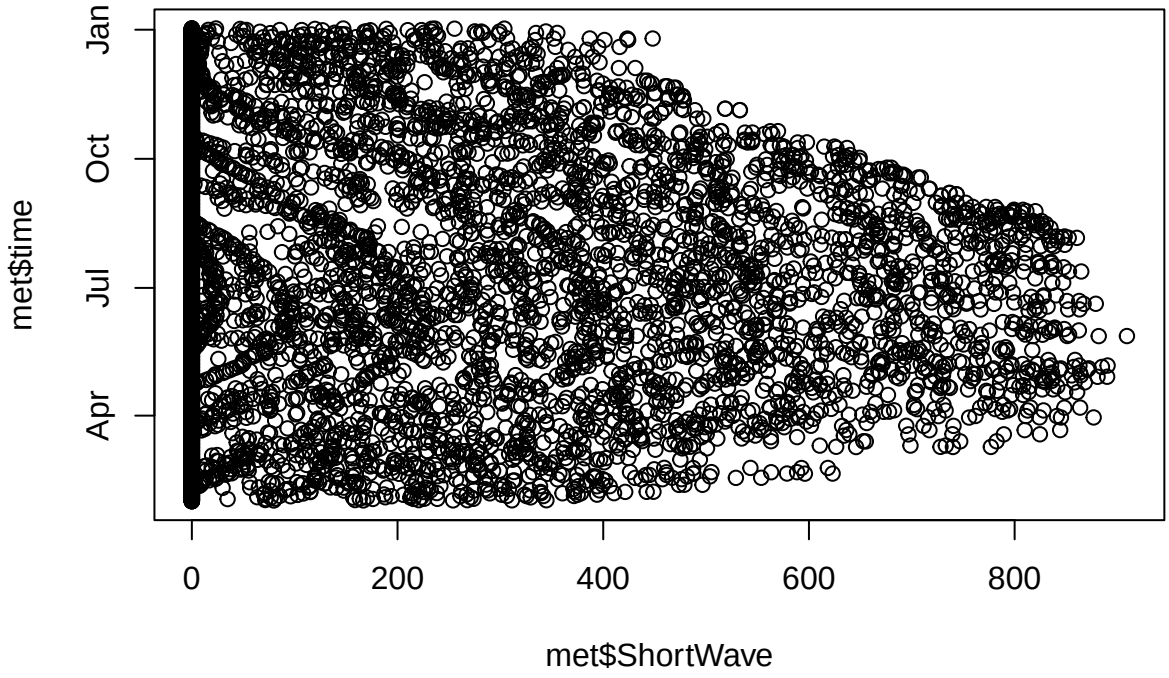
```
blue_frogs <- dat[dat$color == "blue", ]

hist(blue_frogs$frogs,
     main = "Histogram of Blue Frogs",
     xlab = "Number of Frogs",
     col = "skyblue",
     border = "black")
```



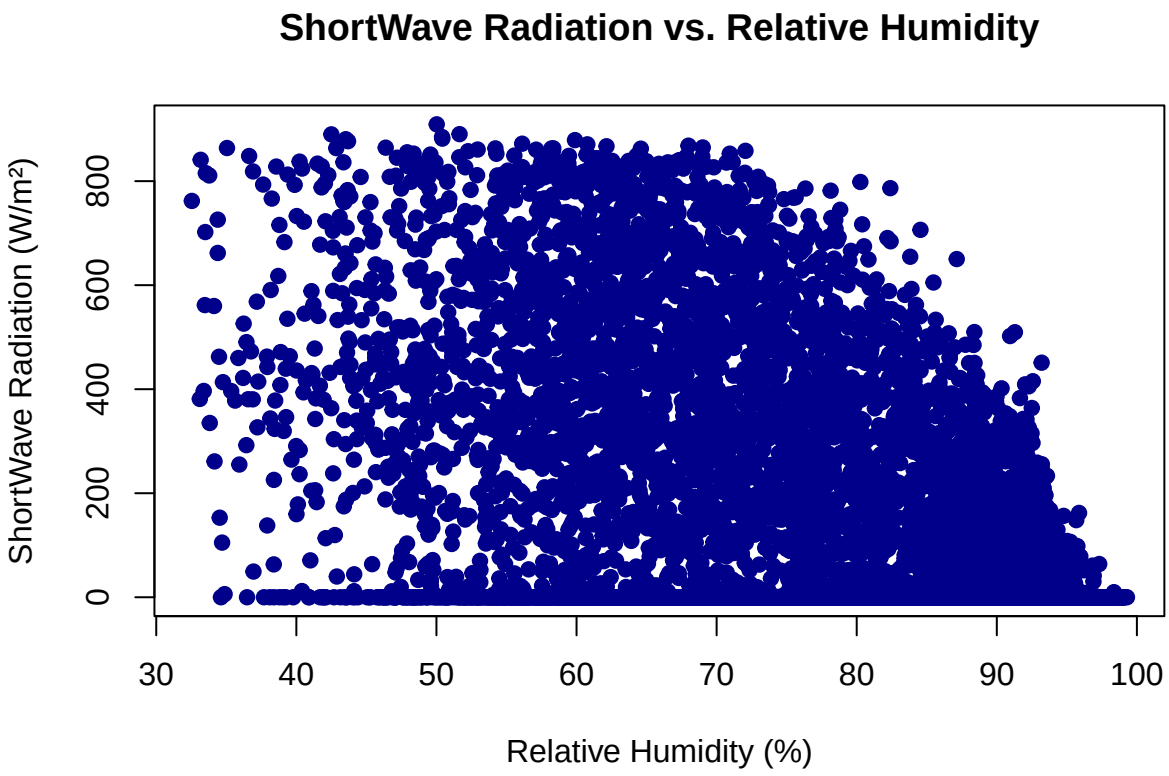
19. Plot shortwave (solar) radiation against time

```
plot(met$ShortWave,met$time)
```



20. Plot shortwave (solar) radiation against relative humidity

```
plot(met$RelHum, met$ShortWave,
     xlab = "Relative Humidity (%)",
     ylab = "ShortWave Radiation (W/m²)",
     main = "ShortWave Radiation vs. Relative Humidity",
     pch = 19, col = "darkblue")
```



Tidyverse

Visualization also brings us to introduce the concept of the `tidyverse`. Over the last decade Hadley Wickham and colleagues have introduces a set of packages for data manipulation and visualization that "share an underlying design philosophy, grammar, and data structures" around what they call "tidy" data. The `dplyr` package, which we saw above, is part of the tidyverse and as we saw provides a number of alternative functions for subsetting and summarizing data (see, for example, the data wrangling chapters in Wickham's R for Data Science). Another popular part of the tidyverse are its data visualization tools, which are anchored around the `ggplot2` package, which numerous other packages build upon (see the Reverse Dependencies listing for the package https://cran.r-project.org/web/packages/ggplot2/index.html)

Classical tests

Since R was initially developed for statistical programming, it can easily perform a wide variety of statistical tests and analyses. In R linear regression is done with the function "lm" (linear models)

```
reg1 = lm(tadpoles ~ frogs,data=dat)   #model syntax: y ~ x
reg1                  # default return from lm
```

```
##
## Call:
## lm(formula = tadpoles ~ frogs, data = dat)
##
```

```
## Coefficients:
## (Intercept)        frogs
##      0.4545       1.8358
```

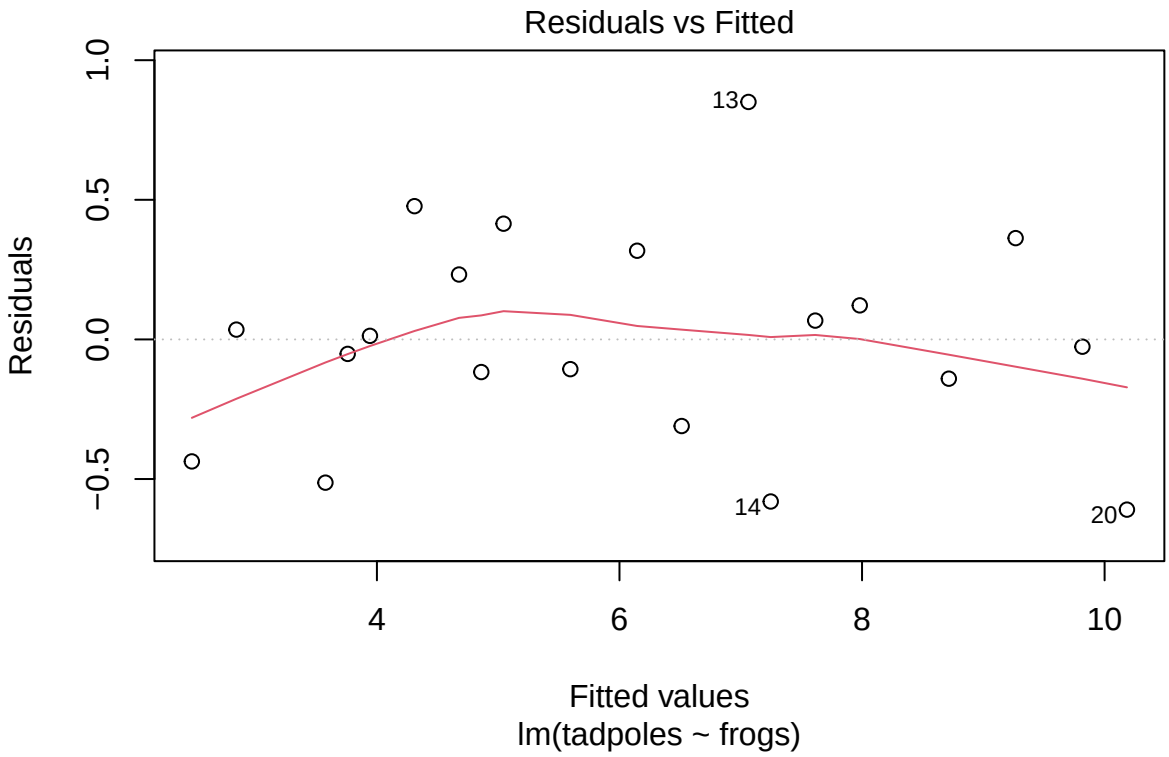
```
summary(reg1)                # detailed summary of results
```

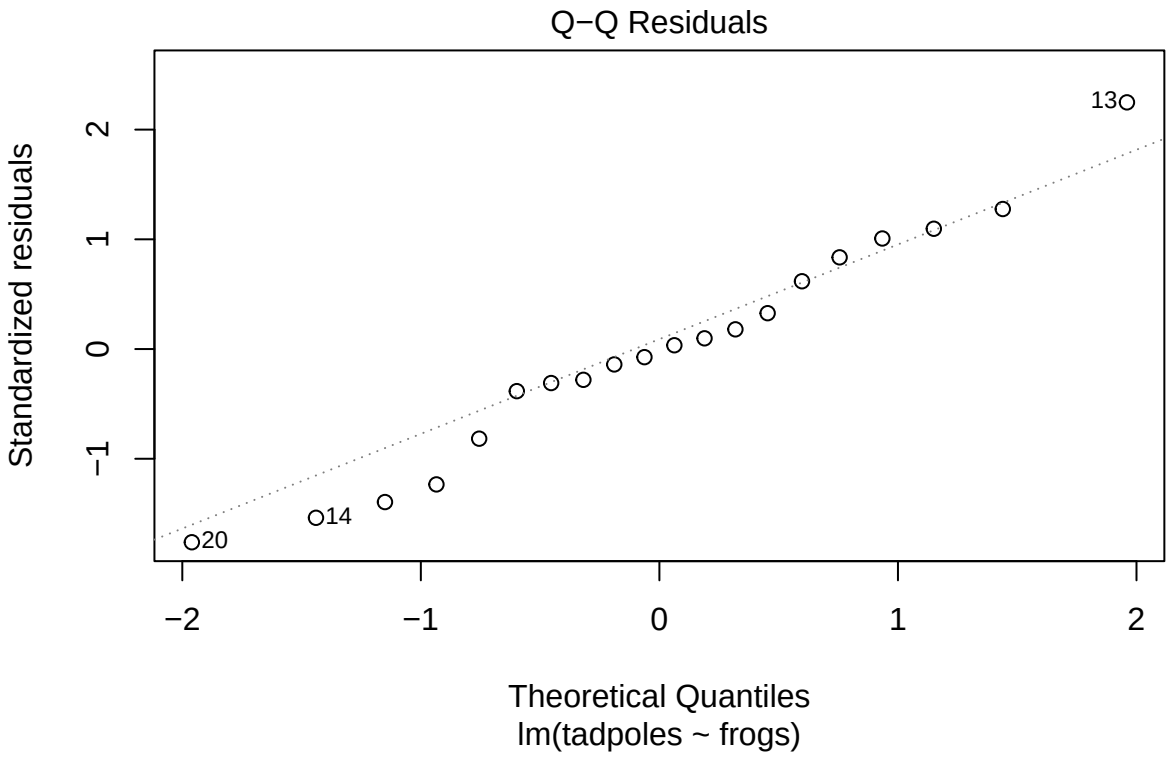
```
##
## Call:
## lm(formula = tadpoles ~ frogs, data = dat)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.60957 -0.18302 -0.00665  0.25375  0.85037
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  0.45453    0.23212   1.958   0.0659 .
## frogs        1.83583    0.07019  26.157 8.97e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.3899 on 18 degrees of freedom
## Multiple R-squared:  0.9744, Adjusted R-squared:  0.9729
## F-statistic: 684.2 on 1 and 18 DF,  p-value: 8.971e-16
```

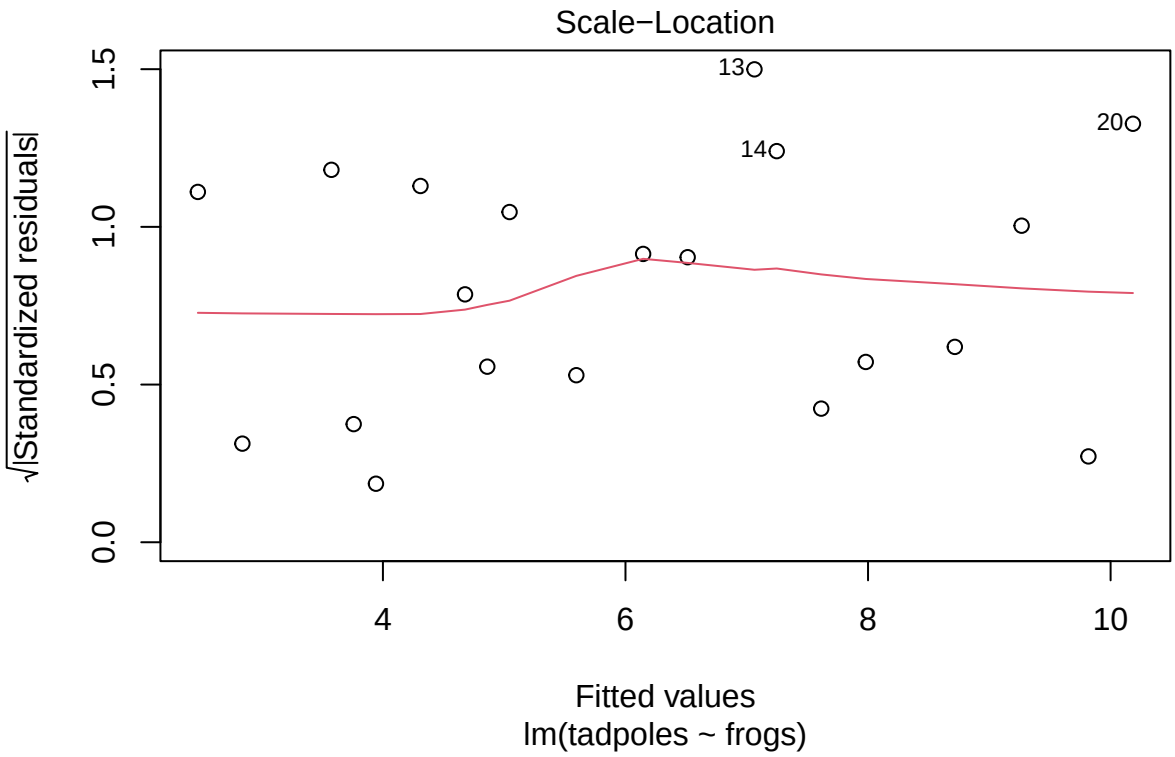
```
anova(reg1)              # ANOVA table of results
```

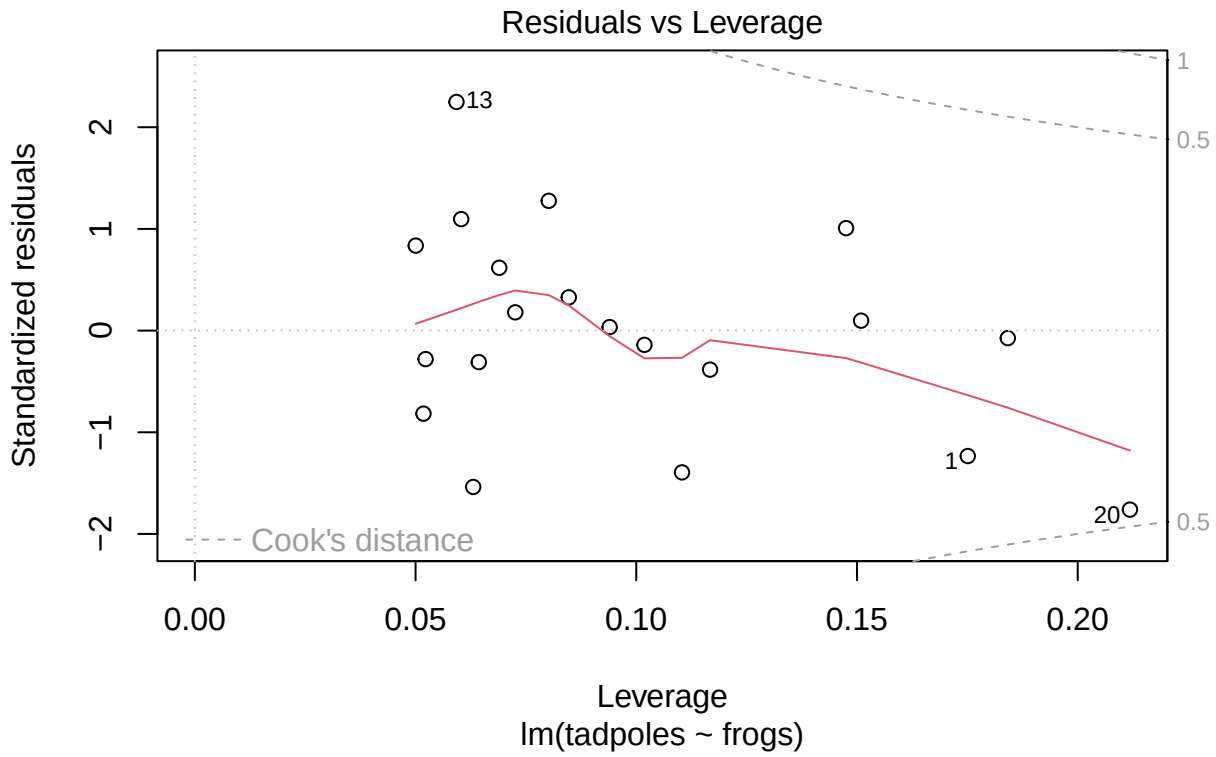
```
## Analysis of Variance Table
##
## Response: tadpoles
##           Df  Sum Sq Mean Sq F value    Pr(>F)
## frogs      1 104.025 104.025  684.17 8.971e-16 ***
## Residuals 18   2.737   0.152
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
plot(reg1)               # diagnostic plots (4 panels)
```

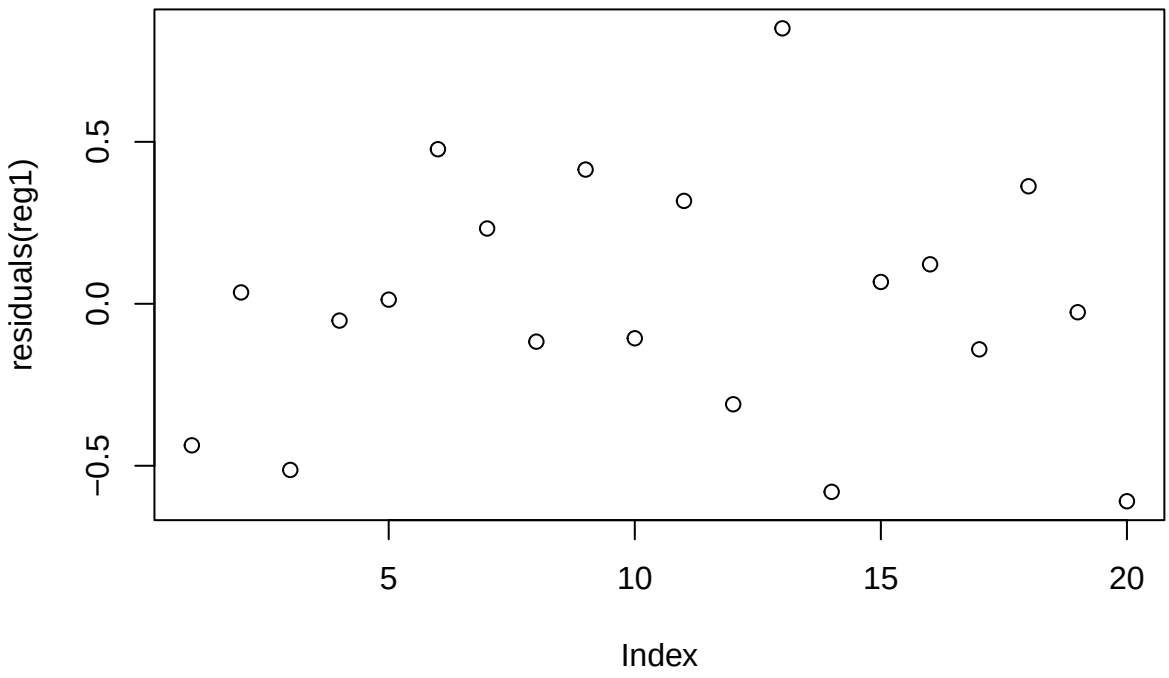

Residuals vs Fitted
Residuals
1.0
0.5
0.0
−0.5
13
14
20
4
6
8
10
Fitted values
lm(tadpoles ~ frogs)


Q−Q Residuals
Standardized residuals
−1
0
1
2
13
14
20
−2
−1
0
1
2
Theoretical Quantiles
lm(tadpoles ~ frogs)


Scale−Location
√|Standardized residuals|
0.0
0.5
1.0
1.5
13
14
20
4
6
8
10
Fitted values
lm(tadpoles ~ frogs)




```
plot(residuals(reg1))       # residuals by row
```



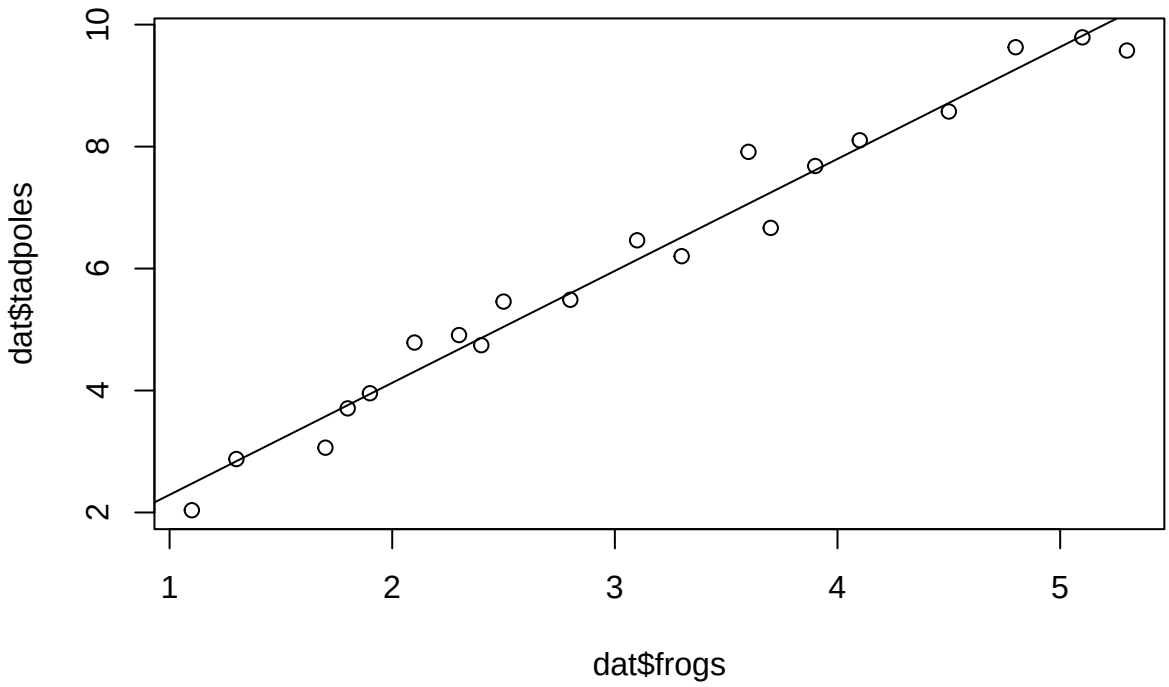

```
coef(reg1)          # parameter coefficients
```

```
## (Intercept)       frogs
##   0.4545345   1.8358260
```

```
vcov(reg1)          # parameter covariance matrix
```

```
##             (Intercept)        frogs
## (Intercept)  0.05387899 -0.015098433
## frogs       -0.01509843  0.004926079
```

```
plot(dat$frogs,dat$tadpoles)
abline(reg1)            # adding regression line to the scatter plot
```



The "equation" syntax for models in R often confuses people because while the order of the data is that you would use for writing down the equation [y = f(x)], it is the opposite order from the scatterplot (x,y). The equation syntax allows one to add additional variables to the regression model (e.g. y ~ x1 + x2). This syntax also makes it easy to specify interaction terms (y ~ x1 + x2 + x1*x2). Note that the linear model is returning an object and that all the other functions are acting on this object. You can use all the functions you used to explore data objects (e.g. class, names, str, summary) to explore the objects returned by functions. Similar to 'lm', ANOVA models are done with 'aov'

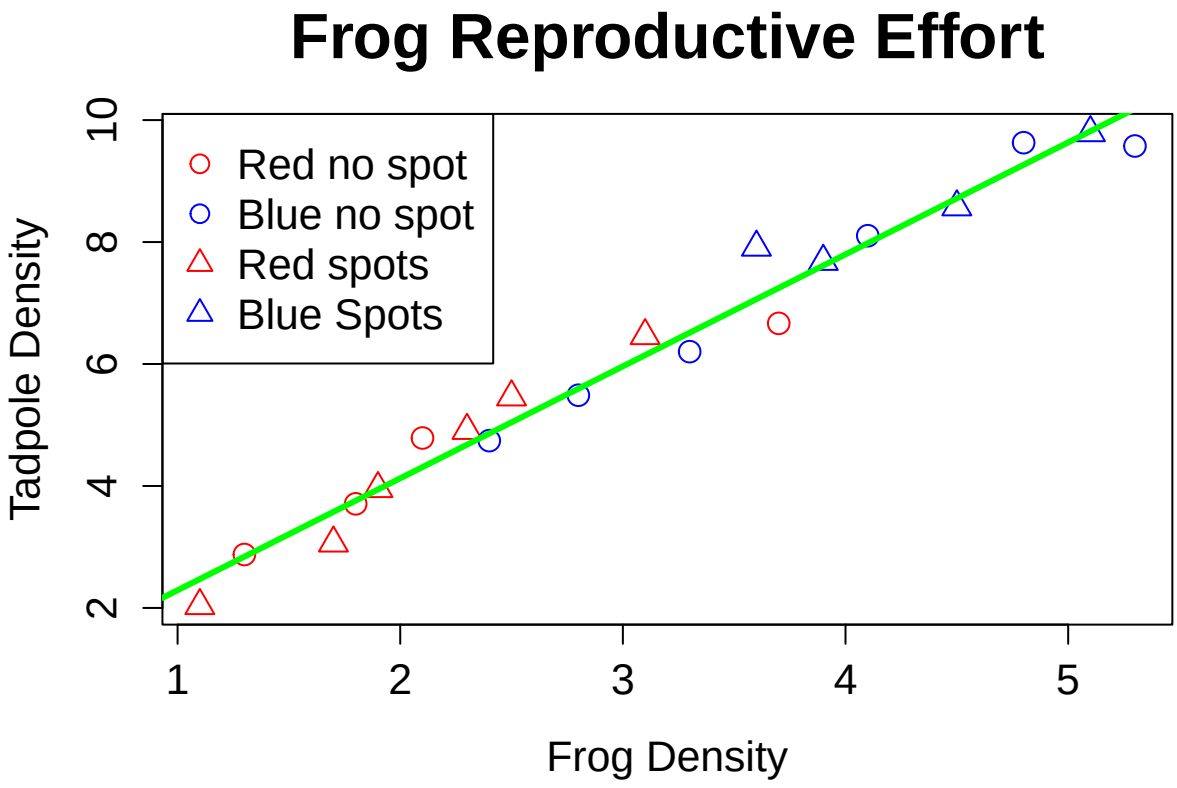
```
anov1 = aov(tadpoles ~ color + spots + color*spots,data=dat)
summary(anov1)
```

```
##             Df Sum Sq Mean Sq F value   Pr(>F)
## color        1  57.06   57.06   19.78 0.000405 ***
## spots        1   1.21    1.21    0.42 0.525970
## color:spots  1   2.34    2.34    0.81 0.381599
## Residuals   16  46.16    2.88
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Finally, we can get a bit more sophisticated in our graphs to display these results. Note that R doesn't care about white space—you can add spaces, tabs, and/or carriage returns wherever you want to make your code more readable.

```
plot(dat$frogs,dat$tadpoles,
     cex=1.5,           # increase the symbol size
```

```
     col=as.character(dat$color),    # change the symbol color by name
     pch=dat$spots+1,            # change the symbol (by number)
     cex.axis=1.3,           # increase the font size on the axis
     xlab="Frog Density",        # label the x axis
     ylab="Tadpole Density",         # label the y axis
     cex.lab=1.3,            # increase the axis label font size
     main="Frog Reproductive Effort", # title
     cex.main=2          # increase title font size
     )
abline(reg1,col="green",         # add the regression line
     ,lwd=3)                 # increase the line width
legend("topleft",
       c("Red no spot","Blue no spot","Red spots","Blue Spots"),
       pch=c(1,1,2,2),
       col=c("red","blue","red","blue"),cex=1.3
       )
```



Question 21. Using the frog data a. Fit a linear model of tadpoles as a function of frogs for just the RED individuals and report the summary data of the fit.

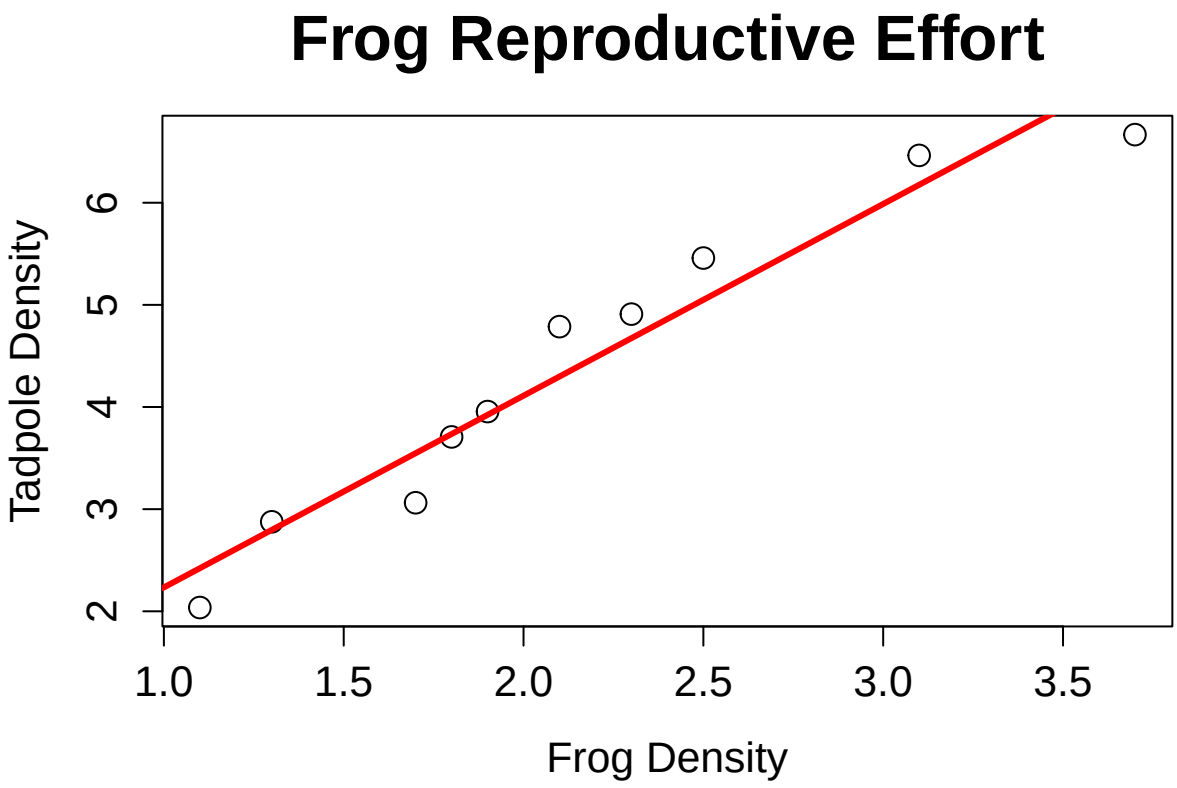
```
dat_red <- dat %>%
  filter(color == "red")
reg2 = lm(tadpoles ~ frogs,data=dat_red)
reg2
```

```
##
```

```
## Call:
## lm(formula = tadpoles ~ frogs, data = dat_red)
##
## Coefficients:
## (Intercept)        frogs
##      0.3569       1.8769
```

b. Make a scatter plot of this data that includes the regression line

```
plot(dat_red$frogs,dat_red$tadpoles,
     cex=1.5,           # increase the symbol size
     cex.axis=1.3,          # increase the font size on the axis
     xlab="Frog Density",       # label the x axis
     ylab="Tadpole Density",        # label the y axis
     cex.lab=1.3,           # increase the axis label font size
     main="Frog Reproductive Effort", # title
     cex.main=2         # increase title font size
     )
abline(reg2,col="red",      # add the regression line
     ,lwd=3)
```



IF statements

Logical operators are not just used for subsetting data, but can be used to control the flow of an analysis and make decisions. The idea is that we want to tell the computer a set of rules, such as "if X happens, then

do Y, otherwise do Z". The syntax for this in R is

```
if(condition){
  ## Do Y
} else {
    ## Do Z
}
```

s The "condition" part of this syntax is always a logical comparison, which does the first part (Y) if the condition is TRUE and the second part if it is FALSE. It should also be noted that the "else{ }" part of the syntax is optional, which would correspond to telling the computer, "if X do Y, otherwise just keep going". For example, if we wanted to do integer division on integers but normal division otherwise we could write

```
if(is.integer(x) & is.integer(y)){
  z = x %/% y   ## Do Integer division
} else {
    z = x/y             ## Do normal division
}
z
```

```
## [1] 0.10000000 0.63636364 0.08333333 0.53846154 0.07142857 0.46666667 0.33333333
## [8] 0.77777778
```

It is also possible to string together multiple if statements sequentially to deal with multiple possible cases and outcomes. For example, we might want the above code to give us a warning if we try to do division on non-numeric data rather than failing with an error

```
if(!is.numeric(x) | !is.numeric(y)){
  warning("Cannot perform division on non-numeric data")
}else if(is.integer(x) & is.integer(y)){
    z = x %/% y     ## Do Integer division
} else {
    z = x/y             ## Do normal division
}
z
```

```
## [1] 0.10000000 0.63636364 0.08333333 0.53846154 0.07142857 0.46666667 0.33333333
## [8] 0.77777778
```

For cases where the outcomes are simple, or when we want to apply an 'if' to every element in a vector, then the ifelse function can be an efficient alternative. ifelse takes three arguments, the condition, what to do if its true, and what to do if its false. For example, the following checks the sign on the frog data before taking a log.

```
ifelse(dat$frogs>0, log(dat$frogs),log(-dat$frogs))
```

```
##  [1] 0.09531018 0.26236426 0.53062825 0.58778666 0.64185389 0.74193734
##  [7] 0.83290912 0.87546874 0.91629073 1.02961942 1.13140211 1.19392247
## [13] 1.28093385 1.30833282 1.36097655 1.41098697 1.50407740 1.56861592
## [19] 1.62924054 1.66770682
```

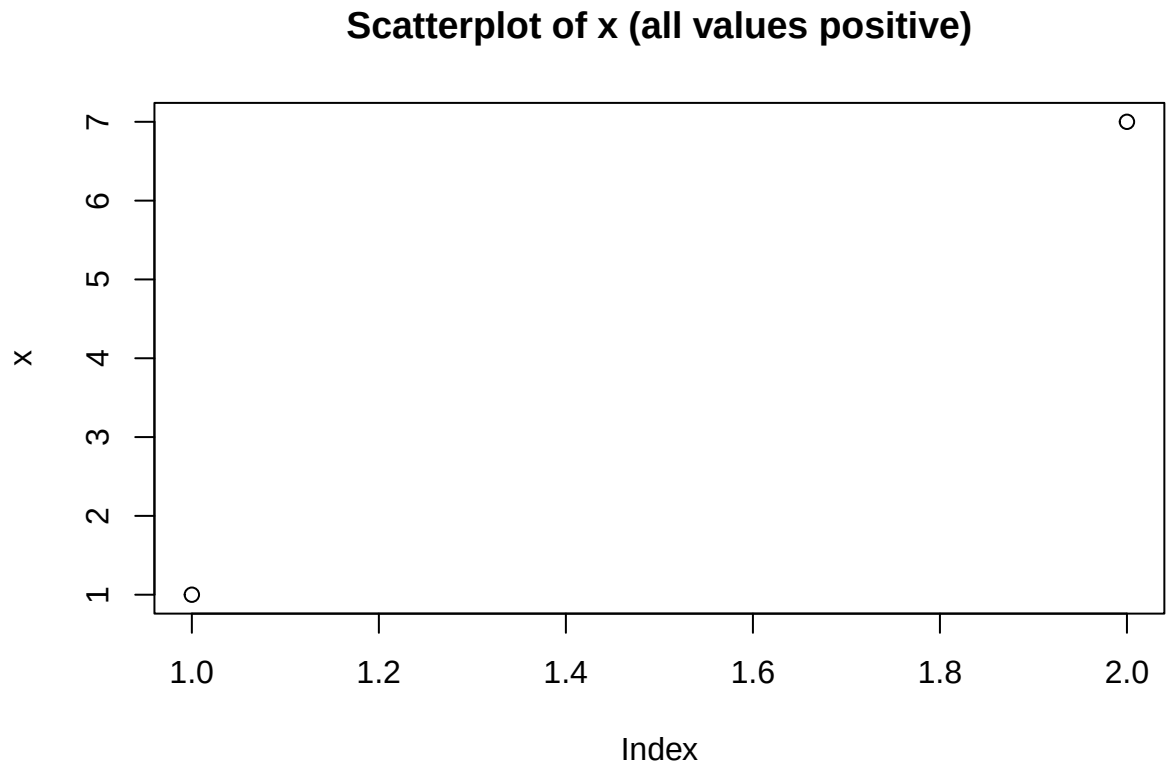
Aside: As is very common in R, as we learn more we often find more efficient ways of solving problems . For example, the above is equivalent to log(abs(dat$frogs))

```
Question
22. Write an if statement that makes a scatter plot of x if all the values are positive, and plots a his
```

```
x
```

```
## [1] 1 7
```

```
if (all(x > 0)) {
  plot(x,
       main = "Scatterplot of x (all values positive)",
       xlab = "Index",
       ylab = "x")
} else {
  hist(x,
       main = "Histogram of x (some values non-positive)",
       xlab = "x")
}
```



Defining custom functions

One of the powerful things about computer languages is that they allow us to encapsulate repetitive tasks into functions, making it easier to abstract a problem. In R you are not limited to the pre-defined functions

but you can define your own functions as well. For example, if we found that we were repeating the previous block of 'if' code multiple places in our code, we might want to convert it to a function so that we could save on retyping the code again and again. Putting the code in one place also means that if we change the code we only need to change it once and it applies everywhere. At the extreme, its often argued that anything you do more than once in a piece of code should be converted to a function. So how do we define a function in R?

```
name = function(arguments){
  # do some calculations
    return(z)
}
```

We need to give it a name, for example we could call the previous if statement 'my.division', and we need to define the arguments to the function. We also need to be explicit in defining what data we want the function to return, since in many cases the outside user doesn't need to know everything that goes on inside the function but is only interested in the result. Putting these together would give us the following

```
my.division = function(x,y){
  if(!is.numeric(x) | !is.numeric(y)){
        warning("Cannot perform division on non-numeric data")
    }else if(is.integer(x) & is.integer(y)){
    z = x %/% y     ## Do Integer division
} else {
    z = x/y             ## Do normal division
}
    return(x)
}

my.division(x,y)
```

```
## [1] 1 7
```

```
my.division(y,x)
```

```
## [1] 10 11 12 13 14 15  3  9
```

```
my.division(x,"5")
```

```
## Warning in my.division(x, "5"): Cannot perform division on non-numeric data
```

```
## [1] 1 7
```

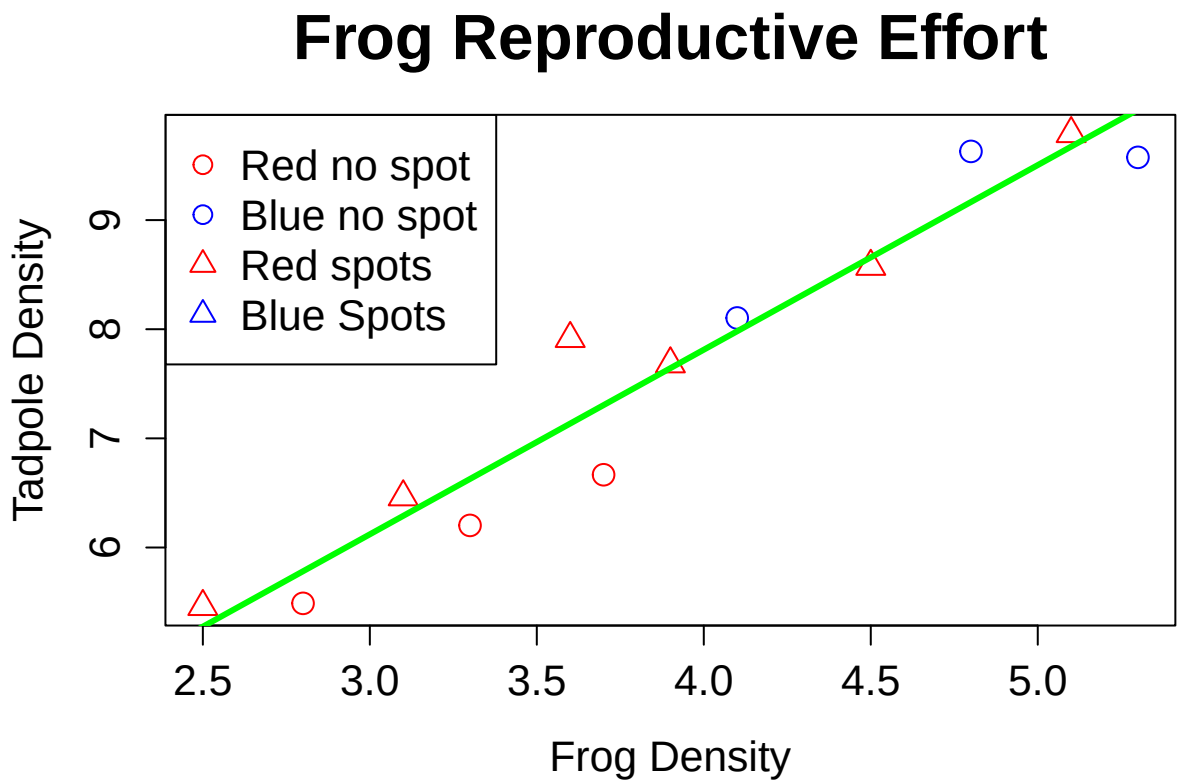
Question
23. Convert the more complicated graphing example at the end of "Classic Tests" into a function that wi

```
classplot = function(frg){
  reg <- lm(tadpoles ~ frogs,data=frg)
  grph <- plot(frg$frogs,frg$tadpoles,
     cex=1.5,           # increase the symbol size
     col=as.character(dat$color),    # change the symbol color by name
```

```
     pch=dat$spots+1,           # change the symbol (by number)
     cex.axis=1.3,          # increase the font size on the axis
     xlab="Frog Density",       # label the x axis
     ylab="Tadpole Density",        # label the y axis
     cex.lab=1.3,           # increase the axis label font size
     main="Frog Reproductive Effort", # title
     cex.main=2         # increase title font size
     )
abline(reg,col="green",     # add the regression line
     ,lwd=3)                # increase the line width
legend("topleft",
       c("Red no spot","Blue no spot","Red spots","Blue Spots"),
       pch=c(1,1,2,2),
       col=c("red","blue","red","blue"),cex=1.3
       )
}

manytad <- dat %>%
  filter(tadpoles >= 5)

classplot(manytad)
```



For loops

Another powerful aspect of computers is there ability to easily repeat the same task time and time again. In fact, one of the major reasons many people learn to code is that they've figured out how to do some analysis once, but they want to apply the same analysis hundreds or thousands of times to different data sets, sites, individuals, pictures, etc. Doing so by clicking through a typical graphical user interface thousands of times if at best mind-numbing, if not outright impossible. Loops allow us to easily repeat an analysis over and over. The most common type of loop we will encounter is the for loop, which will repeat a chunk of code one time for each values specified by some sequence

```
for( variable in sequence){
  ## do something
}
```

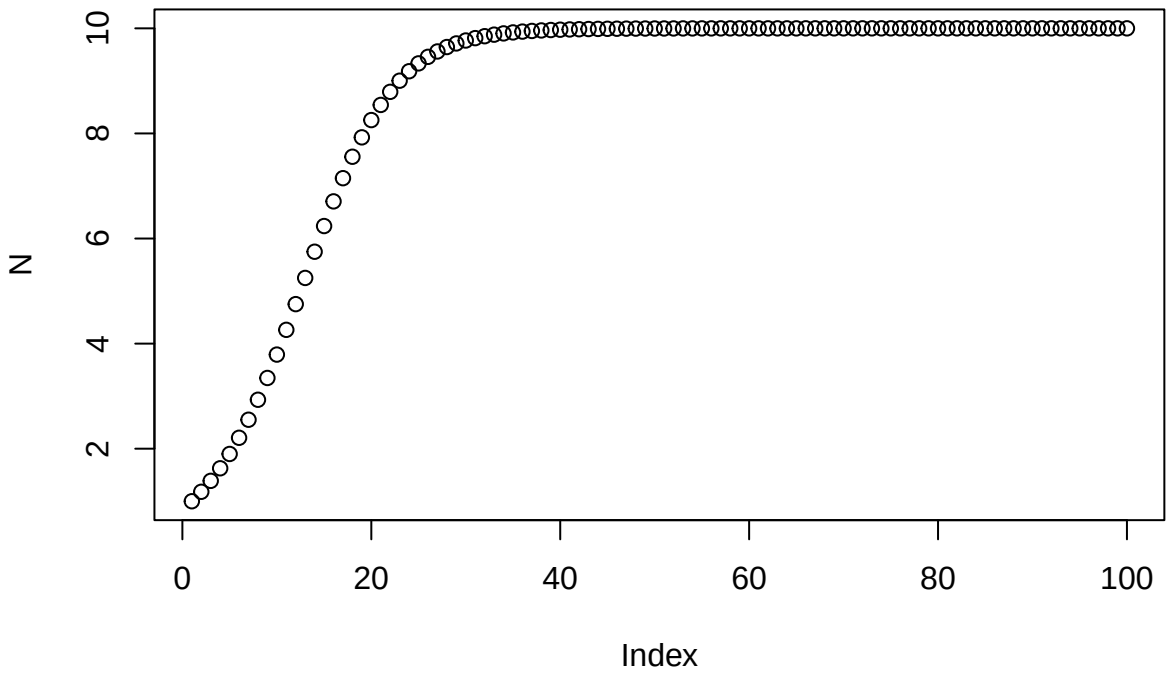
As a very simple example, we might want to print the numbers 1:10

```
for( i in 1:10){
  print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

A more complicated, but common, example might be to loop over all rows in a data set, or to loop over all files in a directory. We also commonly use for loops to do simple simulations. For example, if we want to simulate logistic growth, we might code it as follows:

```
NT = 100        ## number of time steps
N0 = 1              ## initial population size
r = 0.2             ## population growth rate
K = 10              ## carrying capacity
N = rep(N0,NT)
for(t in 2:NT){
    N[t] = N[t-1] + r*N[t-1]*(1-N[t-1]/K)    ## discrete logistic growth
}
plot(N)
```



Question
24. Starting with a vector x = 1:10, write a for loop that adds 5 to each value in the vector. Note tha

```
x <- 1:10

for (i in 1:length(x)) {
  x[i] <- x[i] + 5
}

x
```

```
##  [1]  6  7  8  9 10 11 12 13 14 15
```

Vector and Matrix Math

While loops are a powerful tool, there are many places where a calculation we want to perform can be done so directly on a vector or a matrix as a whole rather than having to loop through each value. For example, in question 26 we had a vector x = 1:10 and we wanted to add 5 to every value. We could have done this more simply as

```
x + 5
```

```
##  [1] 11 12 13 14 15 16 17 18 19 20
```

which adds 5 to each element. We can also multiply, divide, and subtract with vectors, and the operations are applied to each element

```
5*x
```

```
##  [1] 30 35 40 45 50 55 60 65 70 75
```

```
x/5
```

```
##  [1] 1.2 1.4 1.6 1.8 2.0 2.2 2.4 2.6 2.8 3.0
```

```
x-5
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Furthermore, if we have two or more vectors, and we perform mathematical operations on them, the operations are performed element by element

```
y = 10:1
x*y
```

```
##  [1] 60 63 64 63 60 55 48 39 28 15
```

```
x-y
```

```
##  [1] -4 -2  0  2  4  6  8 10 12 14
```

```
x/y
```

```
##  [1]  0.6000000  0.7777778  1.0000000  1.2857143  1.6666667  2.2000000
##  [7]  3.0000000  4.3333333  7.0000000 15.0000000
```

```
dat$frogs + dat$tadpoles
```

```
##  [1]  3.136982  4.176231  4.762528  5.507180  5.855385  6.886983  7.209395
##  [8]  7.143633  7.958514  8.288370  9.563224  9.502578 11.513878 10.366590
## [15] 11.581658 12.203331 13.075123 14.429233 14.891165 14.874846
```

R also allows us to define matrices and perform element-wise math on them as well

```
z = matrix(1:25,5,5)
z
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    6   11   16   21
## [2,]    2    7   12   17   22
## [3,]    3    8   13   18   23
## [4,]    4    9   14   19   24
## [5,]    5   10   15   20   25
```

```
z + 5
```

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

```
z*5
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    5   30   55   80  105
## [2,]   10   35   60   85  110
## [3,]   15   40   65   90  115
## [4,]   20   45   70   95  120
## [5,]   25   50   75  100  125
```

```
z - 1:5
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    0    5   10   15   20
## [2,]    0    5   10   15   20
## [3,]    0    5   10   15   20
## [4,]    0    5   10   15   20
## [5,]    0    5   10   15   20
```

The last example shows that we can also perform element-wise math between a matrix and a vector. Similarly, matrix-to-matrix math is also allowed, though for these cases you need to pay a lot of attention to the dimensions of the matrices and the order that vectors are being applied. In addition to element-wise math, R can also perform standard matrix math. If you have not seen this math before you don't need to worry about what it is doing at this point, any matrix operations will be explained on a case-by-case later in the semester.

```
t(z)        ## transpose
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    6    7    8    9   10
## [3,]   11   12   13   14   15
## [4,]   16   17   18   19   20
## [5,]   21   22   23   24   25
```

```
diag(z)         ## diagonal of an existing matrix
```

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

```
diag(1,5)   ## create a diagonal matrix
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    0    0    0    0
## [2,]    0    1    0    0    0
## [3,]    0    0    1    0    0
## [4,]    0    0    0    1    0
## [5,]    0    0    0    0    1
```

```
w = z + diag(1,5) ## matrix addition
w
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    2    6   11   16   21
## [2,]    2    8   12   17   22
## [3,]    3    8   14   18   23
## [4,]    4    9   14   20   24
## [5,]    5   10   15   20   26
```

```
solve(w)        ## inversion
```

```
##             [,1]        [,2]       [,3]        [,4]       [,5]
## [1,]  0.19021739 -0.51086957 -0.2119565  0.08695652  0.3858696
## [2,] -0.53260870  0.63043478 -0.2065217 -0.04347826  0.1195652
## [3,] -0.25543478 -0.22826087  0.7989130 -0.17391304 -0.1467391
## [4,]  0.02173913 -0.08695652 -0.1956522  0.69565217 -0.4130435
## [5,]  0.29891304  0.05434783 -0.1902174 -0.43478261  0.3206522
```

```
crossprod(z,z)      ## cross product
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]   55  130  205  280  355
## [2,]  130  330  530  730  930
## [3,]  205  530  855 1180 1505
## [4,]  280  730 1180 1630 2080
## [5,]  355  930 1505 2080 2655
```

```
z %*% z         ## matrix multiplication
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]  215  490  765 1040 1315
## [2,]  230  530  830 1130 1430
## [3,]  245  570  895 1220 1545
## [4,]  260  610  960 1310 1660
## [5,]  275  650 1025 1400 1775
```

Synopsis

Part of this document is meant to familiarize you with R, so that when you encounter a problem you need to solve you will have a vague memory of something that might work and can use this as a reference. The other part of this document is to get you to experiment with the capabilities and limitations of generative AI to prepare us for our discussion next class, where we will decide how much to use genAI this semester and *for which tasks*.

If you have never programmed before:

- Focus first on the core concepts that define almost all programming languages
- Mathematical operators
- Logical operators
- Indexing of vectors, matrices, and data frames
- If statements
- For loops
- Functions
- Programming, at its essence, is a process of breaking down a complicated problem into a series of very simple steps, and then translating those steps into a symbolic language (code). This document is mostly about the vocabulary and syntax of one such language (R), the fun and creative process of using this language for problem solving will come through examples and experience.

If you know another language and are picking up R:

- The hardest transition will be SYNTAX

Everyone:

- Save code early and often
- Even better is to commit your code in a version control system like git
- Commit code frequently, typically one new feature per commit
- Keep code well documented
- Use meaningful variable names
- Develop a habit of actively searching and exploring R.
- Read the help documents for a function
- Search for new functions and techniques
- Scour the web when debugging.
- The key to good programming is being able to teach yourself

```
Last Question (answer the question yourself first - then let genAI guess what your answer would be):

25 and 26. Now that you have gone through some simple exercises in R with and without genAI, in 1-3 para
Some guiding questions: Did one set of tasks take more time to complete? Which was more fun? More frust
```

25 *ME* I noticed that genAI sometimes used packages that do not show up earlier in the code. This makes it harder to proof read. I found it saved time to ask it to use a different package over understanding the one it recommended. On the other side, genAI finds packages that I may not find searching CRAN. It is helpful to remember these packages. In the future I may need them and I can search them online based off of the experience of AI recommending them in the past. AI often has helpful suggestions. For example graphing. By searching the commands in the help column I learned how to make better graphs than I would have had I never consulted AI. Finally, genAI helped me find a knitting error around the use of the pi symbol that would have taken me a really long time on my own to find.

26 *Copilot* Working through the exercises with and without generative AI felt noticeably different in terms of pace and mental effort. The questions I answered on my own took more time, especially when I had to recall syntax or look up small details like how seq() handles length.out or how to subset rows in a generalizable way. That slower process was sometimes frustrating, but it also gave me a stronger sense of actually learning the mechanics of R rather than just applying them. When I worked without genAI, I relied on the R help pages, quick Google searches, and my own trial-and-error in the console. Those resources were more fragmented and required more interpretation, whereas genAI provided a single, coherent explanation. Using genAI made many tasks faster and more enjoyable, especially the ones involving syntax patterns I don't use every day. It was particularly helpful for questions that required combining multiple ideas—like filtering with multiple conditions or writing a general-case sequence—because the model could immediately show a clean, idiomatic

solution. At the same time, relying on genAI sometimes made the work feel less like "learning" and more like "applying," since I didn't have to wrestle with the logic myself. The sense of accomplishment was stronger when I solved something manually, even if it took longer. Overall, I noticed that conceptual questions or pattern-recognition tasks were easier to answer on my own, while more mechanical or syntax-heavy tasks were easier with genAI. The contrast made it clear that genAI is a powerful tool for speeding up workflow and checking my understanding, but the deeper learning still comes from working through the code myself.