QuEST Coding Workshop

### Intro (45 min)

Hello everyone, I hope you’re doing well! My name’s George Ni, I’m a 2nd year Phd Student in the Biology Department working with Nicholas Gotelli, and I’m going to be going over some of the basics of RStudio and R today. This workshop was created because one of the required courses for QuEST trainees, Foundations of Quantitative Reasoning, is unavailable this year. Instead, another course called Modeling Complex systems led by Professor Laurent will replace it, which is a little more advanced. Because of that, we want to make sure that you all have the tools necessary to succeed in that course and give you a crash course of sorts for what you would’ve covered in that original class. We’ve designed this workshop to last 3 days, with this first day covering the foundations of RStudio to make sure you’re familiar with it, then the 2nd day will be taking an in-depth look at some interesting packages called the Tidyverse, and then the last day will be covering the mathematical foundations needed for Laurent’s class.

So, to get started, we’re going to go ahead and get RStudio and R downloaded-so who went and downloaded these? You should’ve gotten a link from Brendan in an email previously. Here are the links: <http://lib.stat.cmu.edu/R/CRAN/> and <https://rstudio.com/products/rstudio/download/#download> . Then go ahead and install RMarkdown by opening up RStudio, hitting File -> New File -> RMarkdown. Alternatively, you can type install.packages(“rmarkdown”).

We also want to install something called Latex, which is used to prepare documents for publication. Luckily, RStudio has access to a package that makes installation very simple. In your console, go ahead and type “install.packages(”tinytex“)” and then press enter.

Finally, depending on time, we may do a little bit of work with GitHub in developing a repository. Here’s the link to the download: git-scm.com

So, now that we’ve gone over the installations, let’s go over some of the basics of RStudio. The first thing to note is the difference between R and RStudio. R is the programming language that we can use for statistical computing, while RStudio is how we can use the R language to develop statistical programs. Let’s go ahead and open up RStudio and take a look at what we’re dealing with. So when you open up RStudio, you should be able to see something similar to what’s on my screen. There are basically 4 panels that we’ll be working with, with some options to customize if you’d like.

Starting with the top left panel, This panel is where you will create script and document files. These files consist of lines of computer code. These are instructions to R to carry out calculations and report the results. Think of this window as an old-fashioned typewriter. You will enter plain text here, save the files to disk, and then execute them to carry out your R code. As you will see later, you can also use the Source window as a very powerful word-processor (much easier to use than Word).

For the lower left panel, we have the console. Here’s where you’ll basically communicate directly with R. You can write commands and send them, and it’ll complete them immediately. The > (chevron) at the console is a prompt and means that R is waiting for instructions from you to do something. Working in the console, you have to enter commands one line at a time, sort of like a calculator. When R generates output from its calculations, the output also appears in the console window. And, when you have made a mistake in your code, R will tell you about the problem in this location.

At the upper right panel, we have our Workspace. This is where we’ll have the environment, history,and various other tabs. The environment tab will let us know what sort of variables and objects we’ve created and exist. The history tab will give us some records of the lines of code that have actually been employed/run. These all have important information but won’t necessarily be useful at this stage. In order to clear your environment (which you’ll often have to do), you can either hit the broom icon at the top right of your panel, or type in “rm(list=ls())” to your console.

Finally we have the Plot panel, where the files, plots, packages, and help tabs are. The Files options will have a condensed look at your file locations and what readable files you have, and is also a good place to set and find your working directory (which we’ll talk about in just a little bit). Plots will be where your graphics and plots show up, and whenever you’re confused/curious about a particular function or package you can use the help function to take a closer look at the documentation for it.

So now, I just want to go over a few customization options that are available to you that are just quality-of-life changes. The first being going to Tools->Global options->Code->Soft-wrap text. That way your lines won’t just keep travelling to the right and still be readable for yourself. You can also hit “Appearance”, and edit your font, text size, etc. But, ultimately, these are aesthetic, and you can do what you want.

For now, let’s kind of go through the motions of starting a new assignment, or project, or whatever it may be. One of the most important things to keep in mind while coding is making sure that you’re generating reproducible results and work. That means that you have things neat and organized, and that a random researcher/editor could take a look at your code quickly without needing you to explain anything. The first thing you’d want to do then is making sure you have an organized working directory. This means that your workspaces are compartmentalized and distinct from each other.

So, let’s say you have a particular project with multiple files and documents related to it. The smart thing to do would be to have all of those things in the same spot so you can access them at once. That’s what setting your work directory is going to do. So the commands to do this are “getwd()” and “setwd()”. getwd() is going to tell you what your current working directory is, and the exact file path that it’s using. Setwd() is how you actually change it and set it manually. Alternatively, you can use the Files tab in the bottom right panel to root around in your files and try to find the best working directory.

So, for example, let’s look through my files for this project I’ve been working on about Emerald Ash Borer. So I have all of my related files here in this folder, so I’ve made this my working directory, since I can just pull all of those files easily from here, and then I’ve also made my R Project and placed it here just to make it easier to find.

So what’s a project and R script? An R Script file is basically a record or paper trail of the commands that we want issued to the console as well as the results that they’ll produce. But first, we will tell RStudio to set up a “project” for us. The project is just a file folder and associated documents that RStudio creates for organizing your work. Once you have created your project, R knows where to find everything, and you can pick up your work exactly where you left off, without having to reopen all sorts of files. In the RStudio “File” menu select “New project. . .” and follow the prompts. Name the new directory something like “myFirstProject” and use the “Browse” button to direct RStudio to create the project on your desktop (or wherever you want to store it). Now go ahead and quit RStudio from the “File” menu. Find the new directory on your desk top and inside you will find a file called myFirstProject.Rproj. If you double-click this file, it will open up RStudio and automatically set everything up where you last left off. Remember you only have to create a new project at the start, and then you can re-open it and it will keep all of your work organized

Now that we have everything organized, we have an RProject, as well as a good working directory, we can get started. So we open up a fresh script file, and we can get started. So when you’re working in a script file, there are a few things to note. First, this is unlike a normal word processing document, in that every line is treated as a line of code that is supposed to be sent to R and create some output. Because of that, everything should have a structure and follow the conventional coding rules.

If you want to actually include some lines of text, whether that’s for you or for another reader, you can include comments by placing a # before whatever you want to be a comment. So the building blocks of data types we’ll be working with are numeric, character, integer, logical, and factor variables. It’s important to be aware of what kind of data type you’re working with at any one point, especially if you want your variables to interact somehow with other variables, in which case they often need to be of compatible types, or you’ll run into errors. So we’ll start working in our script file here and assign some variables. This is how we’ll create objects in the environment that we can actually use. That’s using the chevron (>) character in the script file, as seen here.

### Vectors (45 Min)

#Vectors are a basic data structure in RStudio, and are just structures with data of the same type. In order to create a vector you just need to assign a variable with:  
a <- 0  
a = 0 #But we don't really use the = sign, because it has multiple contextual uses  
a <- "Character" #We can also assign it to be any of the data types we had previously determined  
a <- 8 > 9 #Boolean/Logical Operator  
class(a) #At any point you can determine the class of your variable by typing in "class(a)"

## [1] "logical"

a <- as.integer(0) #Numeric data types mean that there are trailing zeros (0.000). If you require just integers, or any other data types, you can coerce it by using the command as.integer(x). Or, you can just add L to your integer value  
class(a)

## [1] "integer"

#Of course, this is just looking at vectors with single elements. If you want to create a vector with multiple elements we can use the c() function to do this. c() stands for concatenate, but you can also think of it as combine, as it's doing something similar to that.  
b <- c(1, 4, 5, 6, 1, 22)  
#You can also concatenate strings themselves using the paste() function  
a <- c(2, 3, 4, 4, 6, 1)  
paste(a, "then", b)

## [1] "2 then 1" "3 then 4" "4 then 5" "4 then 6" "6 then 1" "1 then 22"

#you'll want an easier way to create a vector than just manually writing them up, so you can either use 1:N, or seq() functions.  
x <- 1:30  
b <- seq(1, 20)  
b <- seq(0, 20, by = 2)

### Data Structures (45 min)

Alright, so now let’s talk about data structures. These are ways in which those earlier data types that were in vectors can be oriented into multi-dimensional data structures. These will take the form of matrices, dataframes, and lists. Let’s start with a matrix. So a vector is a single line of data that is of the exact same type. You can think of it as a 1 dimensional array. A matrix is a two-dimensional array as it holds rows and columns. We can make a matrix using the matrix() function.

matrix(ncol = 4, nrow = 4) #So we've created a matrix with 4 rows and 4 columns, but of course we haven't given it any other information, so of course everything inside is empty, defaulting to NA's. If we want to give it information, then we need to specify it, like so.

## [,1] [,2] [,3] [,4]  
## [1,] NA NA NA NA  
## [2,] NA NA NA NA  
## [3,] NA NA NA NA  
## [4,] NA NA NA NA

matrix(ncol = 4,  
 nrow = 4,  
 data = 1:16) #By default, the data goes by the columns and not rows. You can switch that by using the byrow=TRUE argument.

## [,1] [,2] [,3] [,4]  
## [1,] 1 5 9 13  
## [2,] 2 6 10 14  
## [3,] 3 7 11 15  
## [4,] 4 8 12 16

matrix(ncol = 4,  
 nrow = 4,  
 1:16,  
 byrow = TRUE) #You can also create a matrix by binding vectors. The commands for this are cbind() and rbind()

## [,1] [,2] [,3] [,4]  
## [1,] 1 2 3 4  
## [2,] 5 6 7 8  
## [3,] 9 10 11 12  
## [4,] 13 14 15 16

a <- c(1, 2, 3)  
b <- c(1, 2)  
m <-  
 cbind(a, b) #Warning, the number of rows is not a multiple of the vector. If you want it to match perfectly you can fill out the entire vector b, or you can just extend the a vector to match the number in b.

## Warning in cbind(a, b): number of rows of result is not a multiple of vector  
## length (arg 2)

m

## a b  
## [1,] 1 1  
## [2,] 2 2  
## [3,] 3 1

#To rename matrix column names you should use colnames()  
  
#LIST  
#A list is a container for data, often used because of its flexibility. Elements within a list can be any type of data, so it can be a vector, dataframe, etc.  
  
x <- list("here's a list item", c(1, 2, 3, 4, 5))  
x

## [[1]]  
## [1] "here's a list item"  
##   
## [[2]]  
## [1] 1 2 3 4 5

#Data.Frames  
#Data frames are similar to how we think of spreadsheets-with rows and columns, and column variables able to have different data types. We can see an easy example of a data.frame by using the "read.csv" function on any particular excel spreadsheet we might have.  
  
data("iris")  
x <- iris  
x <- read.csv("Test\_data.csv")  
class(x)

## [1] "data.frame"

### Indexing(45 Min)

So now that we’ve gone over the most common data structures, we can talk about how exactly we interact with these things. Often we won’t actually work with the full dataset,and we’ll instead often work with subsets of the data in R. There’re many different ways to actually subset data, so we’ll go through some of them now.

#Let's first start with a vector to work with.  
x <- 5:20  
#[] is a function that means "retrieve the nth element from this vector. This can be referred to as "slicing"  
x[4]

## [1] 8

x[1:4]#You can ask for a sequence

## [1] 5 6 7 8

x[c(1, 2, 1)] #or the same thing multiple times. Remember, when indexing, the first element of a vector has an index of 1.

## [1] 5 6 5

#We can remove elements by using the - sign  
x[-2]

## [1] 5 7 8 9 10 11 12 13 14 15 16 17 18 19 20

x[c(-1, -5)]

## [1] 6 7 8 10 11 12 13 14 15 16 17 18 19 20

x[-c(1, 5)]

## [1] 6 7 8 10 11 12 13 14 15 16 17 18 19 20

#We can subset by name  
x <- c(a = 1, b = 2, c = 3)  
x[c("a", "b")]

## a b   
## 1 2

#Or by logical operators to turn elements "on" or "off"  
x[c(TRUE, FALSE, FALSE)]

## a   
## 1

#More conditionals  
# & represents the "and" operator  
# | represents the "or" operator  
# ! represents the "not" operator  
  
1:10 %in% c(1, 3, 5, 9)

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

### Subsetting Matrices

# The [] function also applies to matrices. You must specify both the rows and columns, in that order.   
vec <- 1:12  
m <- matrix(data = vec, nrow = 4)  
m

## [,1] [,2] [,3]  
## [1,] 1 5 9  
## [2,] 2 6 10  
## [3,] 3 7 11  
## [4,] 4 8 12

m[2, 3]

## [1] 10

#Getting multiple values  
m[2:3, 2:3]

## [,1] [,2]  
## [1,] 6 10  
## [2,] 7 11

m[2, ] #Leaving it blank means you want the entire row or entire column

## [1] 2 6 10

m[, 2, drop = FALSE] #don't include drop=FALSE if you want it to be a vector

## [,1]  
## [1,] 5  
## [2,] 6  
## [3,] 7  
## [4,] 8

### Subsetting Dataframes

varX <- 1:12  
varY <- rep(c("Low", "Med", "High"), each = 4)  
varZ <- runif(12)  
df <- data.frame(varX, varY, varZ, stringsAsFactors = FALSE)  
str(df)

## 'data.frame': 12 obs. of 3 variables:  
## $ varX: int 1 2 3 4 5 6 7 8 9 10 ...  
## $ varY: chr "Low" "Low" "Low" "Low" ...  
## $ varZ: num 0.0785 0.4276 0.9873 0.8452 0.5937 ...

#Let's say we have another df that we're concerned with and want to append a row  
rand <- list(varX = 22, varY = "Low", varZ = 0.223)  
print(rand)

## $varX  
## [1] 22  
##   
## $varY  
## [1] "Low"  
##   
## $varZ  
## [1] 0.223

newframe <- rbind(df, rand)  
  
#Now we can also add a new column  
varA <- runif(13) #Make sure that you have enough elements  
newframe <- cbind(newframe, varA) #rbind for "row", cbind for "column"

### Subsetting Lists

newlist <-  
 list(1:10, matrix(nrow = 4, 1:8, byrow = TRUE), letters[1:3], pi)  
print(newlist)

## [[1]]  
## [1] 1 2 3 4 5 6 7 8 9 10  
##   
## [[2]]  
## [,1] [,2]  
## [1,] 1 2  
## [2,] 3 4  
## [3,] 5 6  
## [4,] 7 8  
##   
## [[3]]  
## [1] "a" "b" "c"  
##   
## [[4]]  
## [1] 3.141593

newlist[2]#Single brackets subsets refer only to the element, of type list.

## [[1]]  
## [,1] [,2]  
## [1,] 1 2  
## [2,] 3 4  
## [3,] 5 6  
## [4,] 7 8

newlist[c(1, 2, 3)]

## [[1]]  
## [1] 1 2 3 4 5 6 7 8 9 10  
##   
## [[2]]  
## [,1] [,2]  
## [1,] 1 2  
## [2,] 3 4  
## [3,] 5 6  
## [4,] 7 8  
##   
## [[3]]  
## [1] "a" "b" "c"

newlist[[2]] - 3#Double brackets will actually be the contents within it, and we can apply things to it. Once you have called the double brackets, you can then access individual elements within that.

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

newlist[[2]][3, 2] #Right now, we're referring to these based on positions within this list. But, it'll be a lot easier for us if we can just name these and refer to them that way.

## [1] 6

newlist <-  
 list(  
 val = 1:10,  
 mat = matrix(nrow = 4, 1:8, byrow = TRUE),  
 alph = letters[1:3],  
 pi = pi  
 )  
newlist$mat[3, 2]

## [1] 6

newlist$mat

## [,1] [,2]  
## [1,] 1 2  
## [2,] 3 4  
## [3,] 5 6  
## [4,] 7 8

newlist$mat[2, ]

## [1] 3 4

newlist$mat[2] #Gives us 3(byrow=TRUE, which is why 2 as it's counted normally is actually 3 in this case)

## [1] 3

#If you were to want to unlist these things, unlist() does the trick  
unlisted <- unlist(newlist)  
unlisted

## val1 val2 val3 val4   
## "1" "2" "3" "4"   
## val5 val6 val7 val8   
## "5" "6" "7" "8"   
## val9 val10 mat1 mat2   
## "9" "10" "1" "3"   
## mat3 mat4 mat5 mat6   
## "5" "7" "2" "4"   
## mat7 mat8 alph1 alph2   
## "6" "8" "a" "b"   
## alph3 pi   
## "c" "3.14159265358979"

Quick exercise!

1. Create a vector object of length 12 that has all the month names as elements in order.
2. Use the bracket operator [] to display the 3rd through the 5th month.
3. Display this vector object in reverse order (when you put a vector into the bracket operator, it will display the output in the same order as the vector)
4. Create a boolean vector that displays whether a given element of the month vector is equal to May.
5. Use help command on the rnorm function to learn how to generate a vector of 10 random numbers from a random variable with a normal distribution with mean 6 and sd, σ, of 3.
6. Use the functions mean, sd and sqrt to calculate the mean and standard error of the mean for the random variable from question 5. Recall that the formula for standard error of the mean is

#1  
month <- c("January","February","March","April","May","June",  
 "July","August","September","October","November",  
 "December")  
season <- c(rep("Winter",2),rep("Spring",3),rep("Summer",3),  
 rep("Fall",3),"Winter")  
#2  
months2<-month[3:5]  
  
#3  
months3<-month[5:3]  
  
#4  
bool1<-month=="May"  
bool1

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

#5  
rand1<-rnorm(10,mean=6, sd=3)  
  
#6  
mean1<-mean(rand1)  
standerr<-sd(rand1)/sqrt(10)

### Cleaning Data

So now let’s talk about how to clean data. You’ve got your dataset, you know how to do some basic manipulation, interact with it, take some elements out, things like that. Well, let’s say you were looking through your dataset and noticed that some of your observations were missing-rather, there was missing data. You choose not to leave it in there, but how do you take it out? Well, thankfully R has a few options to make this a little easier. First, we can use the complete.cases() function so R can look over our dataset for us. So complete.cases is going to return a logical vector, so true/false, based on whether the cases are complete and have no missing values.

DF <- data.frame(x = c(1, 2, 3), y = c(0, 10, NA))  
complete <- complete.cases(DF$y)  
clean <- DF[complete.cases(DF$y), ]  
which(!complete.cases(DF$y))

## [1] 3

#If your spreadsheet had different codes for NA's, such as NaN, or 9999, etc, then you can recode them to be NA's.  
DF$x[DF$y == 10] <- NA  
#You can also use the na.omit function to clean out the NA's.  
na.omit(DF)

## x y  
## 1 1 0

### Control Structures- if, For and While Loops (45)

So, often times we don’t want to manually type out code and complete basic tasks repeatedly-that’s the whole purpose of using a machine, right? So to make RStudio a little bit more practical, we have some really handy functions that can streamline processes for us. So I’m pretty sure most of you should be familiar with ifelse statements, correct? If you aren’t, then it’s one of the most basic ways of giving commands to a computer-given a certain condition, if it is true, then a command/function must be completed, else something else happens, usually the code stopping. if statements can also be standalone, with the structure of if(condition){action}. Otherwise, the full statement is if(condition){action}else{some other action}. So let’s give that a try with a basic example

#Logical operators  
5 > 3

## [1] TRUE

5 < 3

## [1] FALSE

5 >= 5

## [1] TRUE

5 <= 5

## [1] TRUE

5 == 3

## [1] FALSE

5 != 3

## [1] TRUE

# use & for AND  
FALSE & FALSE

## [1] FALSE

FALSE & TRUE

## [1] FALSE

TRUE & TRUE

## [1] TRUE

5 > 3 & 1 != 2

## [1] TRUE

1 == 2 & 1 != 2

## [1] FALSE

lions <- 20  
tigers <- 15  
  
if (lions > tigers) {  
 print("Lions win!")  
}

## [1] "Lions win!"

lions <- sample(1:100, 1)  
tigers <- sample(1:100, 1)  
if (lions > tigers) {  
 print("Lions win!")  
} else{  
 print("Tigers win!")  
} #You can also chain together multiple ifelse statements depending on what you're working with. The ifelse statement can also be built-in using ifelse(), which has a similar function but in one single line.

## [1] "Lions win!"

#Iteration  
#for() loops are your friends  
#The structure is for(x in y){action}  
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

#Nested for loops  
for (i in 1:10) {  
 for (j in letters[1:5]) {  
 print(paste(i, j))  
 }  
}

## [1] "1 a"  
## [1] "1 b"  
## [1] "1 c"  
## [1] "1 d"  
## [1] "1 e"  
## [1] "2 a"  
## [1] "2 b"  
## [1] "2 c"  
## [1] "2 d"  
## [1] "2 e"  
## [1] "3 a"  
## [1] "3 b"  
## [1] "3 c"  
## [1] "3 d"  
## [1] "3 e"  
## [1] "4 a"  
## [1] "4 b"  
## [1] "4 c"  
## [1] "4 d"  
## [1] "4 e"  
## [1] "5 a"  
## [1] "5 b"  
## [1] "5 c"  
## [1] "5 d"  
## [1] "5 e"  
## [1] "6 a"  
## [1] "6 b"  
## [1] "6 c"  
## [1] "6 d"  
## [1] "6 e"  
## [1] "7 a"  
## [1] "7 b"  
## [1] "7 c"  
## [1] "7 d"  
## [1] "7 e"  
## [1] "8 a"  
## [1] "8 b"  
## [1] "8 c"  
## [1] "8 d"  
## [1] "8 e"  
## [1] "9 a"  
## [1] "9 b"  
## [1] "9 c"  
## [1] "9 d"  
## [1] "9 e"  
## [1] "10 a"  
## [1] "10 b"  
## [1] "10 c"  
## [1] "10 d"  
## [1] "10 e"

#While() loops  
#While(conditional){action}  
#While loops aren't the most useful, simply because you have to give it a conditional that will actually end-otherwise it'll just be stuck in an infinite loop and you'll have to stop the program yourself.  
z <- 1  
while (z > 0.1) {  
 z <- runif(1)  
 cat(z, "\n")  
}

## 0.9836062   
## 0.07269428

### Functions (45 Min)

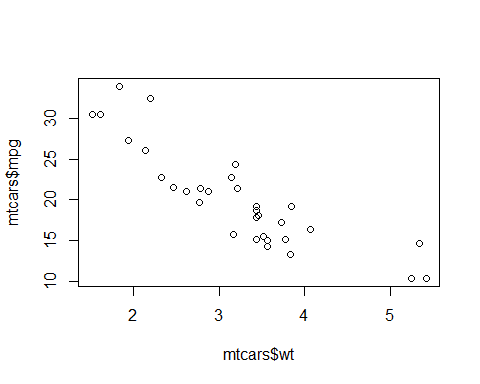
airTemp <- c(32, 34, 35, 37, 39, 40, 32, 36, 41)  
eabAbundance <- c(100, 104, 98, 132, 138, 150, 103, 92, 170)  
df <- as.data.frame(cbind(airTemp, eabAbundance))  
  
grab\_stats <- function(df) {  
 model <- lm(df$airTemp ~ df$eabAbundance)  
 summary(model)  
 u <- unlist(summary(model))  
 slope <- u$coefficients2  
 pval <- u$coefficients8  
 print(paste("slope=", slope))  
 print(paste("p-value=", pval))  
}  
  
grab\_stats(df)

## [1] "slope= 0.104754937820044"  
## [1] "p-value= 0.0021396056185668"

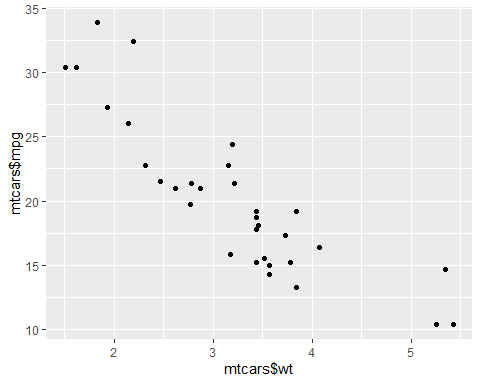
#It's also a good idea to have these functions as a list running in a separate script file. That way, you can source the files whenever you'd like. You can use the command source(file path) to do this.

### Plotting Fundamentals (45 min)

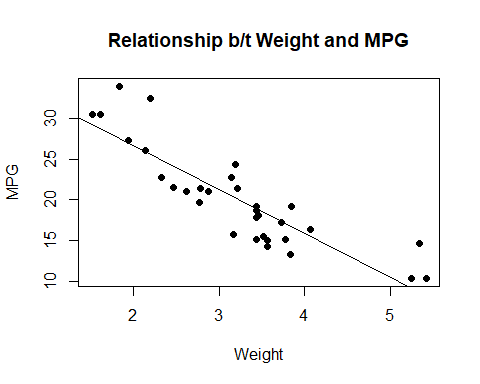
data(mtcars)  
#scatter plot  
plot(mtcars$wt, mtcars$mpg) #basic plotting function



qplot(mtcars$wt, mtcars$mpg) #qplot, or "quick" plot is a convenient way of throwing up a plot, but still doesn't have all of the necessary information we might be looking for.

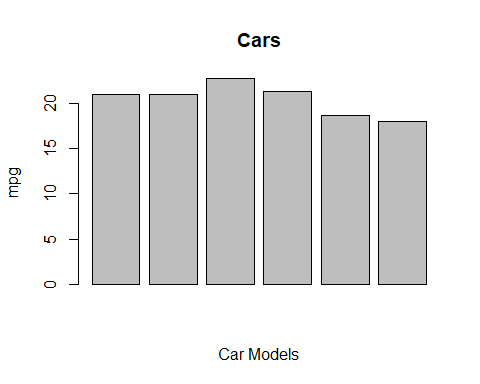


plot(  
 mtcars$wt,  
 mtcars$mpg,  
 main = "Relationship b/t Weight and MPG",  
 xlab = "Weight",  
 ylab = "MPG",  
 pch = 16  
) + abline(lm(mtcars$mpg ~ mtcars$wt)) #pch=point type, abline includes a line, we used lm(y~x) to tell it how to actually draw the line

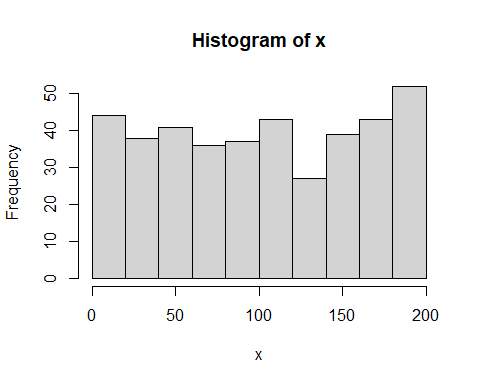


## integer(0)

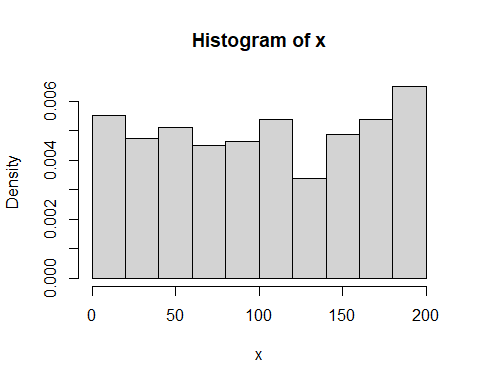
#bar graph  
cars <- mtcars[1:6, 1:5]  
barplot(cars$mpg,  
 main = "Cars",  
 xlab = "Car Models",  
 ylab = "mpg")



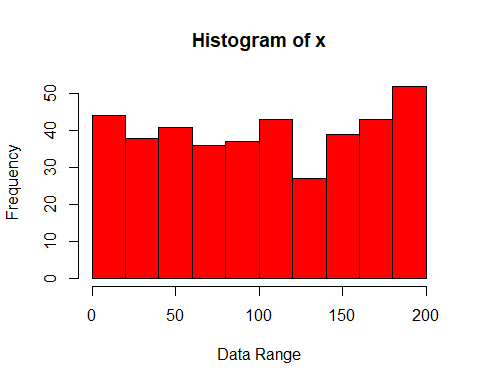
#histogram  
x <- runif(400, 1, 200)  
hist(x) #plotting the frequency



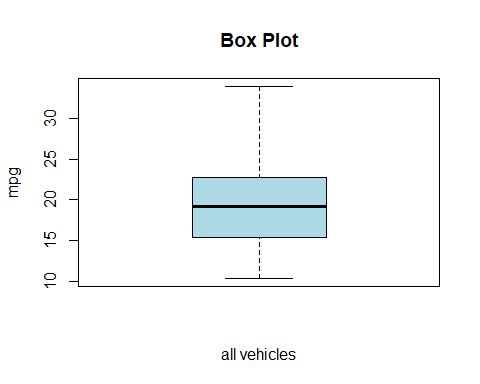
hist(x, freq = FALSE) #plot the proportion. Also, there are similar levels of customization available to you as with the scatter plot and bar plot.



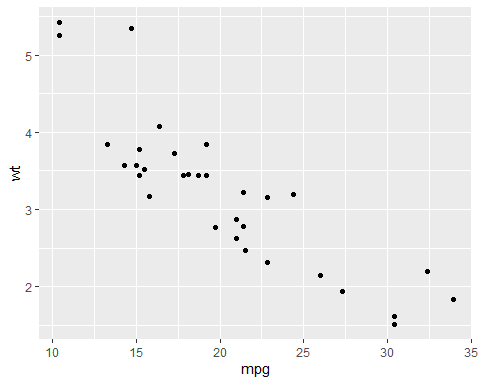
hist(x, col = "red" , xlab = "Data Range", ylab = "Frequency")



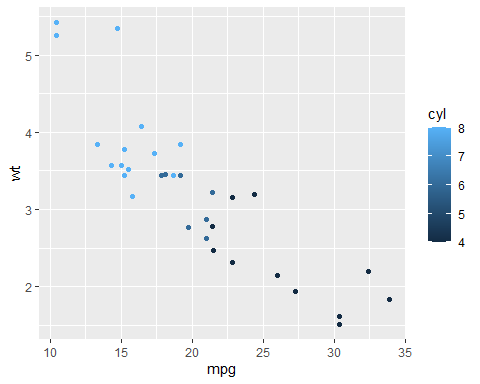
#box plot  
boxplot(  
 mtcars$mpg,  
 col = "lightblue",  
 ylab = "mpg",  
 xlab = "all vehicles",  
 main = "Box Plot"  
)



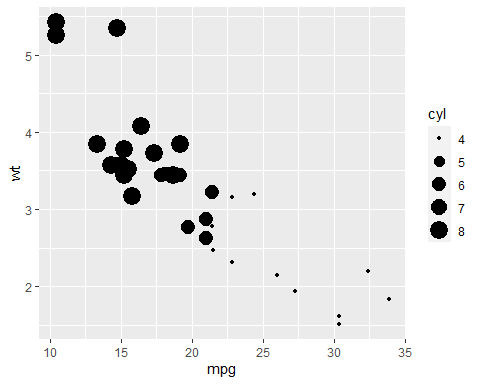
#qplot  
qplot(mpg, wt, data = mtcars)



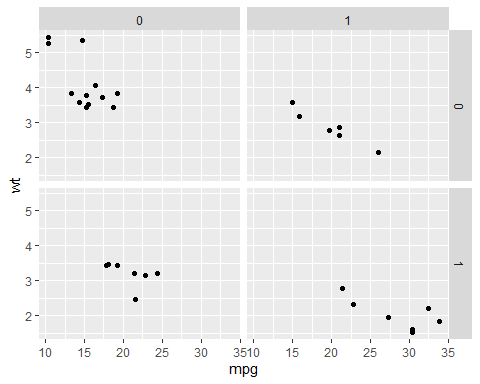
qplot(mpg, wt, data = mtcars, colour = cyl)



qplot(mpg, wt, data = mtcars, size = cyl)



qplot(mpg, wt, data = mtcars, facets = vs ~ am)



### Fitting Models (45 min)

#Regression Analysis  
n = 50 # number of observations (rows)  
varA <- runif(n) # random uniform values (independent)  
varB <- runif(n) # a second random column (dependent)  
varC <- 5.5 + varA \* 10 # a noisy linear relationship with varA  
ID <- seq\_len(n) # creates a sequence from 1:n (if n > 0!)  
regData <- data.frame(ID, varA, varB, varC)  
head(regData)

## ID varA varB varC  
## 1 1 0.127911003 0.86152156 6.779110  
## 2 2 0.903967579 0.84943310 14.539676  
## 3 3 0.317250568 0.83220452 8.672506  
## 4 4 0.239077536 0.03507954 7.890775  
## 5 5 0.264866428 0.62322771 8.148664  
## 6 6 0.002061983 0.09131141 5.520620

str(regData)

## 'data.frame': 50 obs. of 4 variables:  
## $ ID : int 1 2 3 4 5 6 7 8 9 10 ...  
## $ varA: num 0.128 0.904 0.317 0.239 0.265 ...  
## $ varB: num 0.8615 0.8494 0.8322 0.0351 0.6232 ...  
## $ varC: num 6.78 14.54 8.67 7.89 8.15 ...

# model  
regModel <- lm(varB ~ varA, data = regData)  
  
# model output  
regModel # printed output is sparse

##   
## Call:  
## lm(formula = varB ~ varA, data = regData)  
##   
## Coefficients:  
## (Intercept) varA   
## 0.3121 0.3213

#str(regModel) # complicated, but has "coefficients"  
#head(regModel$residuals) # contains residuals  
  
# 'summary' of model has elements  
#summary(regModel) #  
#summary(regModel)$coefficients  
#str(summary(regModel))  
  
# best to examine entire matrix of coefficients:  
summary(regModel)$coefficients[] #shows all

## Estimate Std. Error t value Pr(>|t|)  
## (Intercept) 0.3121136 0.08062042 3.871396 0.000326592  
## varA 0.3213480 0.14077431 2.282718 0.026917694

# can pull results from this, but a little wordy  
summary(regModel)$coefficients[1, 4] #p value for intercept

## [1] 0.000326592

summary(regModel)$coefficients["(Intercept)", "Pr(>|t|)"]

## [1] 0.000326592

# alternatively unfurl this into a 1D atomic vector with names  
z <- unlist(summary(regModel))  
#str(z)  
#z  
z$coefficients7

## [1] 0.000326592

# grab what we need and put into a tidy list  
  
regSum <- list(  
 intercept = z$coefficients1,  
 slope = z$coefficients2,  
 interceptP = z$coefficients7,  
 slopeP = z$coefficients8,  
 r2 = z$r.squared  
)  
  
# much easier to query and use  
print(regSum)

## $intercept  
## [1] 0.3121136  
##   
## $slope  
## [1] 0.321348  
##   
## $interceptP  
## [1] 0.000326592  
##   
## $slopeP  
## [1] 0.02691769  
##   
## $r2  
## [1] 0.09792751

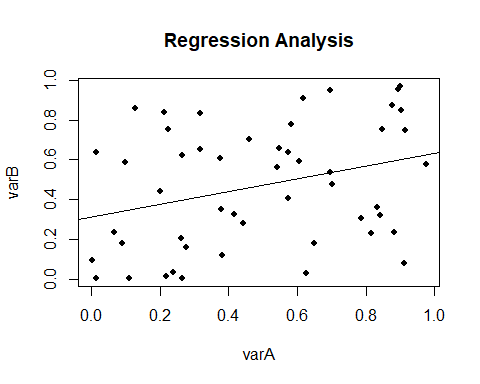
regSum$r2

## [1] 0.09792751

regSum[[5]]

## [1] 0.09792751

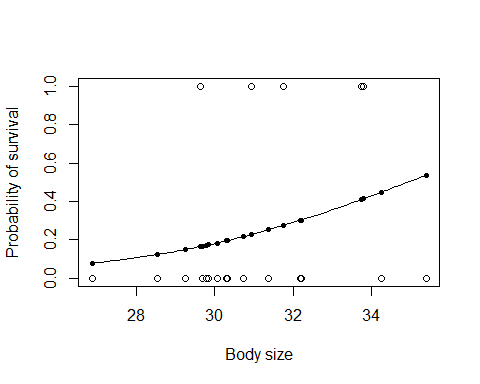
plot <-  
 plot(  
 regData$varA,  
 regData$varB,  
 pch = 16,  
 cex = 0.8,  
 main = "Regression Analysis",  
 xlab = "varA",  
 ylab = "varB"  
 ) + abline(regModel$coefficients[1], regModel$coefficients[2])



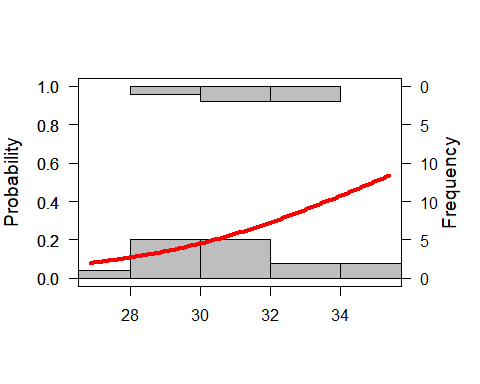
#Logistic Regression Analysis  
bodysize <- rnorm(20, 30, 2) # generates 20 values, with mean of 30 & s.d.=2  
bodysize <- sort(bodysize) # sorts these values in ascending order.  
survive <- c(0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1) # assign 'survival' to these 20 individuals non-randomly... most mortality occurs at smaller body size  
  
n<-20  
survive<-sample(c(0,1), replace=TRUE, size=n)  
dat = as.data.frame(cbind(bodysize, survive)) # saves dataframe with two columns: body size & survival  
dat # j

## bodysize survive  
## 1 26.85471 0  
## 2 28.53240 0  
## 3 29.24530 0  
## 4 29.63758 1  
## 5 29.68843 0  
## 6 29.79478 0  
## 7 29.83582 0  
## 8 30.05468 0  
## 9 30.28966 0  
## 10 30.31571 0  
## 11 30.72361 0  
## 12 30.92542 1  
## 13 31.37594 0  
## 14 31.76631 1  
## 15 32.19725 0  
## 16 32.20621 0  
## 17 33.75326 1  
## 18 33.79746 1  
## 19 34.27233 0  
## 20 35.40961 0

plot(bodysize, survive, xlab = "Body size", ylab = "Probability of survival")# plot with body size on x-axis and survival (0 or 1) on y-axis  
  
g <- glm(survive ~ bodysize, family = binomial, dat) # run a logistic regression model (in this case, generalized linear model with logit link). see ?glm  
  
curve(predict(g, data.frame(bodysize = x), type = "resp"), add = TRUE) # draws a curve based on prediction from logistic regression model  
  
points(bodysize, fitted(g), pch = 20) # optional: you could skip this draws an invisible set of points of body size survival based on a 'fit' to glm model. pch= changes type of dots.



#Alternatively  
  
library(popbio)  
logi.hist.plot(bodysize,  
 survive,  
 boxp = FALSE,  
 type = "hist",  
 col = "gray")



### RMarkdown (45 min)

So now let’s talk about RMarkdown. This is an efficient way to integrate both your code and word processing to create robust documents with your code. RMarkdown files behave differently from normal r script files, which we’ll see in just a little bit. Files are .Rmd’s. So let’s go ahead and open up an RMarkdown file by going to file->new file->RMarkdown. So RMarkdown will always give you this default page with code already written inside so that you have a sense of how it’s structured. The grey’ed out lines are code chunks, which are essentially just mini script files that we’ve placed inside here. So within these code chunks, everything that we’ve learned about coding applies, the assigning conventions, commands, things like that. What determines a code chunk are these backticks that you see here (hover over that) as well as including the curly brackets. These curly brackets basically determine what steps you want Markdown to initialize with-meaning, you have a lot of options to adjust this code chunk-like whether you want to display the code, or just the results of them, or the sizes of the plots/graphics that you created. There’ll be a list of options you can set in the cheat sheet I’ll send to you.

Outside of these code chunks though, as you’ll notice, you can write normally like how you would in Word. However, there are still formatting commands in this section that you should be aware of. So you’ll be writing your report in plain text, but the specific markdown syntax will actually format your text.

Plain Text

# Header 1

## Header 2

### Header 3

Italic \*\*

bold-face \*\*\*\*

subscript ~~

superscript ^^

strikethrough ~~~~

quotations ><text with no closing mark

“Hello”

Hello!

## Equations

* in-line uses $
* centered uses $$
* basic math and text is handled by Latex

# Subscripts

H\_0 = Z\_{a + b}

# Superscripts

S = cA^z

S=cA^z\_1 + z\_{2 + x}

# Fractions and Greek

=

# Summation

z = \_{i=1}^X{K}

### Cheat Sheets