# Open Source Open Science Workshop – R Intro

## by: Danielle Walkup (and Google!)

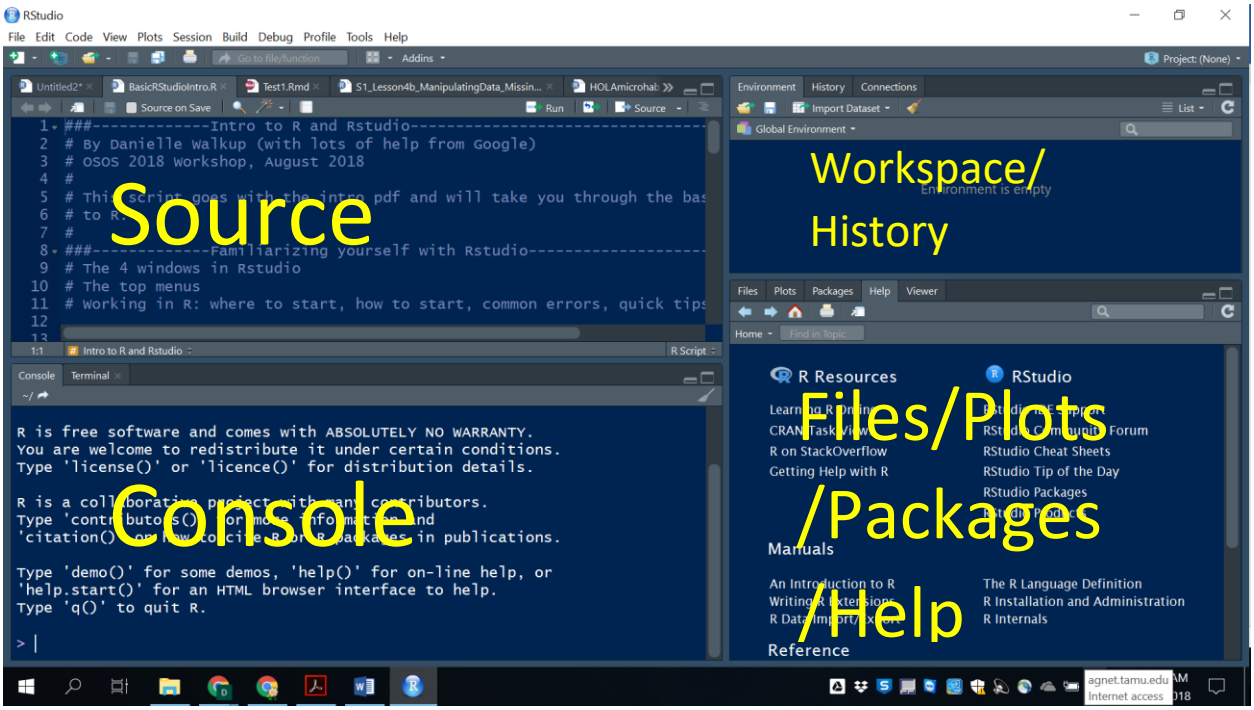
## 31 August 2018

## Part 1. An intro to R and RStudio

For this section, we will be using "1_BasicRStudioIntro.R"

### *Familiarizing yourself with RStudio:*

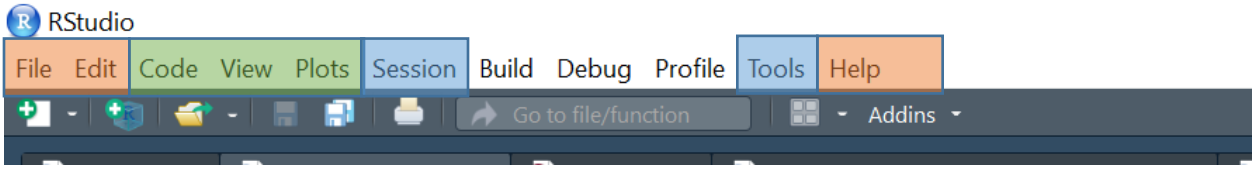
**The default 4 windows:**




1. Source (upper left) – this is where you will write your code, in scripts (and any other file type you want). The scripts are a reusable record of your code that you can edit, check, and re-run.
2. Console (bottom left) – runs the code and displays the results. This should be used to test code, run examples, and see results – your whole analysis shouldn't be run through here. Eventually you run out of display room!
3. Workspace/History (upper left) – shows the objects you have created as well as the list of commands run.
4. Files/Plots/Packages/Help (bottom left):

a. Files – shows all the files in your working directory
b. Plots – shows the plots you have created during the session. Can also export and save them through this window
c. Packages – lists all the packages that come with R, as well as any packages you have installed
d. Help – lets you search R information

**The Menus:**




1. File, Edit, Help – Generally the default menus we see in most programs (e.g. open, close, save, undo, redo, more help).
2. Code, View, Plots – Generally helpful tools that you will probably end up learning the keyboard shortcuts for or doing within the code.
3. Session, Tools – Some helpful options for overall program stuff (e.g. set working directory, global options).

**(Some) Good Coding Practices:**

1. Organizing Scripts: Consistent organization of your scripts makes it easier for people to read and follow (even if you are the only one using it!). Some general tips are:
    a. Add general purpose and authors at the top of the script, as well as any notes you think will be helpful
    b. Install and load packages at the top of the script so we know what we need upfront
    c. Check/Set the working directory at the top of the script
2. Comment, comment, and comment your code!
    a. In R, anything preceded by a '#' will not be evaluated
    b. The scripts are for people, so use comments to help people follow through the script, future you will thank you!
    c. Explain WHY you are doing things and making the choices you are.
3. Use a consistent style within your code
    a. Naming conventions: .df for data.frames, .mat for matrices, etc.
    b. Use meaningful names (e.g. lizard.df, not mydata)
    c. Wrap long lines (recommended length seems to be around 80 characters. **NOTE**: you can add a margin to help with this through tools -> global options -> code -> display -> show margin (margin column 80).
    d. Use spaces around operators and after commas (e.g. col = 'green', pch = 4)
4. Possibly controversial tips:
    a. Use <- for assignments, and = in functions or arguments (TIP: alt + - is the shortcut for <-).

   b. Avoid assigning new data to objects using reserved names: mean <- 6.3, pi <- 3.14
5. These are suggestions, not hard and fast rules. Do what works for you!

## *Working in RStudio*

R as a calculator: **TIP**: In BasicRStudioIntro.R use ctrl-enter to run each line of code

```
## The obligatory R can be used as a calculator tutorial:
## You can do basic arithmetic with R
3+5+8
## It will follow order of operations rules
5*9+14
5*(9+14)
## Other operators:
12^2  #can do exponents
sqrt(100)  #and square roots
pi*3  #there are some built in constants
log(5)  #log in R equals the natural log (i.e. ln)
log10(5)  #this will give you log base 10
```

Functions to get you started:

```
## Handy Functions:
## "<-" is the assignment operator: (some example data:)
ex.data <- c(4,6,3,6,7,3,4,6,7,2)
# The c() means to concatenate, where all the numbers are joined together in a
# vector. We use c() quite a lot.

# We can check to make sure it's there
ex.data

# and check out some of its properties:
mean(ex.data)
min(ex.data)
max(ex.data)
median(ex.data)
sd(ex.data)
```

Starting to understand and examine your data:

These are some useful tools that you will use over and over again throughout your scripts to help you make sure that things are loading ok, that your code is working, that objects are being put together correctly, etc. **TIP**: As you go back through and use your code over and over you may find that you had these in here to troubleshoot, but don't need them any longer. Just comment them out instead of erasing them, so you have them handy if things change. This also lets anyone else who may be using your scripts troubleshoot as needed.

```
## Some basic tools that help you understand and check your objects:
summary(ex.data)
str(ex.data)
dim(ex.data)  #NULL here, better for dataframes or other objects
length(ex.data)  #since dim() didn't work, lets use length() instead.
head(ex.data)  #by default returns the 1st 6 things in the object
head(ex.data, 2)  #but we can tell it how many we actually want (> or < 6)
tail(ex.data, 3)
#
```

# Part 2: Loading Data and Understanding Data Types

For this section, we will be working with "2_LoadData_DataTypes.R"

## *Packages*

Many programs include all packages with the initial program installation by default, but R does not. Instead, when you first install R, the program itself is installed along with a set of "base" packages. These packages are simply text files with the ".R" extension that contain functions to perform specific tasks. Every time R is opened, these "base" packages are loaded ("sourced") and the contents of the scripts are submitted to your current R session.

There are thousands of additional packages available on the CRAN website to do all kinds of specialized analyses (phylogenetic analyses, ecological analyses, special plotting functions, etc.). These packages are not automagically downloaded with R, but you can download them yourself. To see what's available, check http://cran.r-project.org under Packages.

To install a package:

install.packages("package-name")  #the quotation marks are necessary

IF you haven't already, R will prompt you to choose a CRAN mirror site; select one that's nearby.

To source a package (i.e. load it into the R session so you can use it):

library("package-name")  #Now all the functions in the package are available for your use

An example from our Rscript:

```
#
## One of the earlier parts of your script should include a list of packages that
## will be used in the script. I generally set it up as a commented out install.packages()
## command (because I typically have them already loaded), along with a set of
## library(), that loads each package
#
# For example, some packages you might like:
#install.packages(c("ggplot2","gplots","dplyr","plyr"))  #plotting and data manipulation
#install.packages(c("sp","rgdal","rgeos","ggmap","raster"))  #some handy spatial packages
# and one we will use in this script
install.packages("openxlsx")  #note the quotes here
```

```
library(openxlsx)  #don't have to have quotes here
#
```

## *The Working Directory*

This is the location on your computer that R is working from – where it reads files and where it writes files.

```
# To check your current directory:
getwd()  #the default directory when you open RStudio is the Documents folder
dir()  #shows what is currently in the working directory

## Remember you can also see what is in the working directory by clicking on
## "files" in the bottom left window.

## TIP: If you open an R script from somewhere else in your computer, the
## working directory defaults to the folder where your R script was stored.
## Note that even that folder may not be where you actually want to work from

# We can change the working directory:
setwd("")  #the easiest way to do this is to open the folder and copy and paste
           #the file name into the quotes.
dir()

## TIP: You can see what is is sub-folders in the directory
dir("")  #choose one of the folder names and type it between the quotes to see
         #what's in it.

## NOTE:
# 1) We double-checked to make sure that setwd() worked by using dir() again - always be
#    troubleshooting your code!
# 2) On pc's you need to use / in the path, \ is used as an escape character in
#    R, so if you copy-paste, make sure you change them to the forward slash
# 3) If it just isn't working and you need to get on with your life, you can use
#    Session -> Set Working Directory -> Choose Directory OR ctrl-shift-H
```

*Organizing your projects:*

Many times you are not working with just one script, one data set, and one output. So organizing your project can help keep things tidy and easily accessible. There are many ways to do organize your projects, so experiment and find the ways that work for you. One suggestion is to have one folder for the project, with sub-folders for input data, R scripts, and output data. That way, your overall folder can be your working directory, and you can direct R to read and write things from and into the sub-folders. I generally use different scripts for reading and cleaning data (you really want to try to avoid changing the original data set!), running preliminary data exploration (checking assumptions, plots, etc.), and then final analysis and writing graphs or tables.

You can also use the Projects within Rstudio to help organize these. I haven't used this function of Rstudio much, but more information can be found at the links below:

https://support.rstudio.com/hc/en-us/articles/200526207-Using-Projects

https://swcarpentry.github.io/r-novice-gapminder/02-project-intro/

## *Types of Data*

Use class() to check your data.

The basic object classes are (from Rebecca Clark, Intro to R):

| R class (Atomic Vectors) | Other terms | Examples |
|---|---|---|
| Numeric | Real, Continuous, Quantitative | 4.69; 3.5; 3,405,285 |
| Integer | Count data | 1, 2, 3 |
| Factor | Ordinal, Categorical, Nominal, Discrete | Low, Medium, High;<br>Site1, Site2, Site3;<br>Lizard, Bird, Mammal; |
| Logical | | TRUE, FALSE |
| Date | | 8/31/2018 |
| Character | | "gravid"; "Dropped lizard" |

**NOTE**: Time and date classes can get complicated when formatting. See the references at the end of this document for more information.
**NOTE**: Ordering of factors can also be an issue, many times the order defaults to alphabetical

```
## Some examples:
?class()  #we can learn a little something about the classes in R
#
# The classes can vary depending on the type of data:
liz.wt.num <- c(1.4, 5.6, 8.3, 5.6, 5.0)
class(liz.wt.num)  #returns numbers

liz.wt.ch <- c("1.4", "5.6", "8.3", "5.6", "5.0")
class(liz.wt.ch)  #returns characters

# if you load a data set and need to change the class, you can:
liz.wt.change <- as.numeric(liz.wt.ch)
class(liz.wt.change)

liz.wt.change2 <- as.character(liz.wt.num)
class(liz.wt.change2)
```

```
# Notice what happens when we change them to integers:
liz.wt.int <- as.integer(liz.wt.ch)
class(liz.wt.int)
liz.wt.int

## Factors
# Ordering of factors defaults to alphabetical:
fruit <- factor(c("apple","pear","banana","grape"))
fruit
# But we can change that if we want to:
fruit2 <- factor(c("apple","pear","banana","grape")
                 , levels = c("apple","pear","banana","grape"))
fruit2
#
# We can also create an ordinal variable:
fruit.ord <- factor(c("apple",'pear',"banana","grape"), ordered = TRUE)
fruit.ord  #note the < between the levels now AND our levels changed back to alphabetical
fruit.ord2 <- factor(c("apple",'pear',"banana","grape")
                     , levels = c("apple","pear","banana","grape")
                     , ordered = TRUE)
fruit.ord2  #Much better - if you like grapes...
#
# Notice there are some constraints on classes:
fruit.ch <- as.character(fruit)
class(fruit.ch)
fruit.ch

fruit.num <- as.numeric(fruit.ch)  #It'll it, but it'll do it badly.
class(fruit.num)
fruit.num
```

The atomic vectors can be combined in different ways to create different data structure. Some common ones are in the table below:

| Data Structures | | Examples |
|---|---|---|
| Vectors | A collection of elements (typically of character, logical, integer, or numeric) | a <- c(1, 2, 3, 4, 5)<br>b <- c("apple", "pear", "banana", "grape") |
| Lists | A special type of vector, each element can be a different type, it can even be a list of lists. | my.list <- list(1, "a", TRUE, 1+4i)<br>big.list <- list(a = "Testing", Again b = c(1,3,5,6,7), flowers = head(iris)) |
| Matrices | An extension of numeric or character vectors that has dimensions: rows and columns. | my.mat <- matrix(1:10, nrow = 2)<br>my.mat2 <- matrix(1:10, nrow = 2, byrow = TRUE) |

| | | |
|---|---|---|
| Data frames | A special type of list where every element has the same length. This is generally the most commonly used data structure for tabular data and what we typically use for statistics | my.df <- data.frame(id = rep(c("a","b","c"),4), height = 1:12, width = 12:1) |

Some examples:

```
# Vectors - collection of elements of the same type
a <- c(1, 2, 3, 4, 5)
b <- c("apple", "pear", "banana", "grape")
# We can see elements in the vectors:
b[3] #the single bracket lets us see the element in the 3rd place in the vector
b[8] #there's nothing in here - because we only have the 4 fruits!
# We can see all sorts of things about the vectors:
class(a)  #notice this returns the atomic vector type
length(b)
is.na(a)  #a handy function to check for missing values
sum(a)  #we can run functions specific to the given class
sum(b)  #no numbers to add!
str(a)
#
# Lists
my.list <- list(1, "a", TRUE, 1+4i)
class(my.list)  #now we see this is a list, with its special list properties!
# We can still see the different elements and characteristics
my.list[2]
length(my.list)
str(my.list)
# You can also name the elements in each list
big.list <- list(a = "Testing", b = 1:10, flowers = head(iris))
#
# now if we want to see what's in the list:
names(big.list)  #we can also see the names
big.list[2]  #and what's in the list
big.list$b  #alternatively, if we have elements in the list named, we can use the
            # '$' to choose the different names and show what's in them
length(big.list)  #but we still only have 3 things in the list
#
# Matrices
?matrix
my.mat <- matrix(1:10, nrow = 2)
my.mat
my.mat2 <- matrix(1:10, nrow = 2, byrow = TRUE)
#TIP: instead of writing out 1,2,3,4,5,6,7,8,9,10, we use 1:10 to give us the
#    sequence of numbers
my.mat2
#
# and we can see characteristics:
```

```
class(my.mat)
str(my.mat)
length(my.mat)
dim(my.mat)  #Now we can see the dimensions!
my.mat[2,1]  #now that we have 2 dimensions, we can use [row,column] to id an element
my.mat[,2]  #or see just a column
my.mat[2,]  #or see just a row
sum(my.mat)  #can still do math on a numeric matrix -
t(my.mat)  #can also transpose a matrix
#
# Data frames
# here we will create a data frame with 3 columns: id, height, and width
my.df <- data.frame(id = rep(c("a","b","c"),4), height = 1:12, width = 12:1)
my.df  #lets make sure it did what we expected
#
# and check out some of its characteristics:
class(my.df)
str(my.df)
summary(my.df)
length(my.df)  #we can still get a length, but it may not quite be the info we want
dim(my.df)
t(my.df)  #we can still transpose the data frame
#
# and we can see the different parts:
my.df$id  #note that id has levels, so it is a factor
class(my.df$id)  #which we can double check here
class(my.df$width)  #and it has defaulted width to integer, which we could change:
as.numeric(my.df$width)
class(my.df$width)
# We could even do math within the data frame:
sum(my.df$width)
mean(my.df$width)
```

## *Loading in data from a file:*

Although we can create data frames, like we did in the previous examples, for the most part, we are probably going to be importing our data from something like an excel sheet. Here's an example to get us started:

```
## read.csv or read.table are the typical functions we will use to retrieve data
?read.csv  #lets check out the parts of the function
#
# First, lets make sure the file is where we think it is!
dir("data")  #looks good

# I'll set a file path here (makes it more easily accessible)
liz.file <- "data/LizardData.csv"  #NOTE: by using data/ i'm telling R to look in
                                   #that folder for the file

## If the file is saved as a .csv we can simply read it in:
```

```
liz.data <- read.csv(liz.file)
head(liz.file)  #double check that it worked
#
## If we have a text file, we can still read it in using read.csv
liz.file.txt <- "data/LizardData.txt"
liz.data <- read.csv(liz.file.txt, sep = "\t")  #sep = "\t" tells R its a tab-
                                                #delimited text file
head(liz.data)  #and it looks the same as above!
#
## read.csv has a ton of options to read in your data set, check out the different arguments:
# header  - tell it whether or not there is a head row
# col.names - can change the names of the columns
# row.names - or add row names
# colClasses - can specify the column classes
# strip.white - removes leading or trailing white spaces
# blank.lines.skip - skips empty rows
# na.strings - what values should be considered NA
#
# and we can check out our new data frame:
class(liz.data)
head(liz.data)  #look it automatically reads our headings!
str(liz.data)  #tells us the column classes and info about the data.frame
dim(liz.data)
summary(liz.data)  #gives us a brief overview of each column, already, I can see
                   #some indicators of a messy data set: check out the Sex, Regen,
                   #and the Recap column summaries!
#
## Finally, we can also load the excel file using our package openxlsx:
liz.file.xl <- "data/LizardData.xlsx"
liz.data.xl <- read.xlsx(liz.file.xl)
head(liz.data.xl)
# but note the differences in how it default loads the data:
str(liz.data.xl)
str(liz.data)
str(liz.data.txt)
```

**TIPS**: Things to check before converting your Excel file to text:

1. You can only have one row of headers
2. Use short descriptive headers without spaces
    a. Use "_" or caps between words (i.e. first_name or FirstName)
    b. Headers are type over, and over, and over, so keep them short
    c. Don't start a header with a number; R will accept it, but will put an X in front of it
3. Remove all summary data (e.g. Average, sum, max, min, etc.)
4. Check that all values in a column are of the same type (e.g. number, date, character, etc.)
5. Remove commas and # symbols throughout
    a. Commas will screw up comma-delimited files
    b. # indicates a comment in R
6. Missing data are okay, but keep an eye on it.

# Part 3: Working with our data set

For this section, we will use the "3_DataCleanExplore.R" script

Most of the field data that we work with is unlikely to be clean and organized. If you can process and clean your data in a script, then our process is more transparent and repeatable. If you comment your script as you go, then you can also keep track of your thought processes as you are going through and cleaning the data. Finally we can output clean versions of our data to work with. For this section we will load, clean, subset, and do some exploratory data manipulations.

There are many basic R functions to help clean and manipulate your data. There are also a number of packages available. One package that has been designed for manipulating data is the tidyverse, which calls multiple packages to manipulate, reform, and plot data. We will just be hitting the surface of the available functions below, so definitely check out their website if you are interested in more information: https://www.tidyverse.org/

We will be working with a real data set: LizardData (Fig XX). This is data collected from 2012-2015 from a 6 x 6 grid of grids using pitfall traps. Every lizard captured (i.e. each row in the database) in these traps has a species code **ID** using the first two letters of genus and first two letters of species (e.g. *Uta stansburiana* – UTST); a numeric individual **Mark** that is unique within species, but not across species; location data in the form of a **Grid** (36 alphabetical code, all should start with E) and **Trap** number (there are 9 traps within each Grid); **Sex** should be male, female or unknown (for some species it is hard to tell the sex of juveniles); 4 body measurements: **SVL** – snout-vent length (mm; a typical lizard measurement), **Tail** – total tail length (mm) from vent to tip of tail, **Regen** – if there is any regeneration of the tail, this is the length of that regeneration (mm), and **Mass** (g); **Notes**; **Recap** – should be a yes/no variable; and a **Date** of capture. The same data is available as a comma delimited file, a text file (tab-delimited), and an excel file in the OSOS_IntroR -> data folder. You've already practiced loading the three different types of files, so here we are just going to work with the csv file. The data has been run through the checklist at the end of Part 2, the only thing we really need to keep an eye on is the missing data.

## *Packages and Directory*

Since this is a new script – First things, first:

```
## Lets be good and set up any packages we are going to use in this script:
#install.packages(c("tidyverse",""))
install.packages("tidyverse")
library(tidyverse)
#
```

```
## And double check our working directory:
getwd()  #we are still in the OSOS_IntroR folder where we want to be!
#setwd()  #but we could change it if we needed to
#
```

## *Now we can load our practice data set*

```
## We've already done this at the end of our last script, but let's make sure
## we have the correct data set loaded and ready to go
dir("data")  #I can never remember the exact file name, so lets check the directory
#
liz.file <- "data/LizardData.csv"
liz.data <- read.csv(liz.file)
# and check that it loaded ok:
head(liz.data)
tail(liz.data)  #I like to check the end too to make sure there're no blanks
#
```

## *Cleaning our Data Set*


"This is not what I meant when I said 'we need better data cleansing!'"

www.iwaysoftware.com/go/dataquality

Let's run through the data set and see the parts and pieces. Keep an eye out for any obvious issues.

```
# Let's check the summary again, cause we had spotted some issues previously
summary(liz.data)
# Some issues:
# Mark - we have 101 before a blank - that could indicate some issues, possibly
# Sex - we have some duplicate categories (M vs. Male) and an age class (J)
# SVL - looks ok generally
# Regen - We have No and numbers, that's not good, this should just be numbers
# Recap - again, with the duplicate categories! (N vs no vs No)
#
```

```
# Let's double check the structure and make sure everything is the way we want it:
str(liz.data)
# ID = species code, should be a factor, let's double check the levels
levels(liz.data$ID)  #and we see an issue immediately, with the white space!
#
# let's reload the data and strip the white spaces
## NOTE: normally, I'd just edit the read.csv(liz.file) above, but we'll just do
## it here instead so we can keep track of the work flow
#
liz.data <- read.csv(liz.file, strip.white = T)
# and double check
levels(liz.data$ID)  #these look ok now
str(liz.data) #lets keep looking
#
# Mark is a factor, with 95 levels. This looks good for now. Even though the Marks
# are numbers, they identify individual lizards so we want to leave them as as factor
#
# Grid is also a factor, with 35 levels which is fine, because I know there were
# 36 available grids, except there are some blank ones (the ""), and some with
# only 2 letters, when I know it should be a 3 letter code where all start with E.
#
# Trap is an integer (should range from 1 - 9), we can check that in a minute
#
# Sex is a factor with 5 levels. Really should either be F, M, or blank in this
# context.
#
# SVL, Tail, Regen, and Mass should actually be numeric, these are measurements
# that we are going to want to work with.
#
# Notes is a factor. If I had assigned colClasses I probably would have made it
# a character, but we won't be using it here, so it shouldn't matter too much.
#
# Recap is a factor with 6 levels. Should be either Y or N.
#
# Date is a factor here, but we actually want a date.
#
## So now that we've looked at everything, we could go back through and reassign
## the individual columns like so:
liz.data$SVL <- as.numeric(liz.data$SVL)  #Note that we assign this column with
                                          #the changes so they are "saved"
class(liz.data$SVL)
#
## OR since this data set isn't very big, we can just assign colClasses
head(liz.data,2)  ##I'm going to look at the data frame so I know the column order
liz.data <- read.csv(liz.file, strip.white = T
                     , colClasses = c("factor", "factor", "character", "integer"
                                      , "factor", "numeric", "numeric", "numeric"
                                      , "numeric", "character", "factor", "Date"))
#
## BUT we run into an error: expected 'a real', got 'Yes' - From prior experience,
## I suspect that the No in the Regen column is probably the issue - R expected
## to see a real number, but got a word instead, so let's change the numeric to
## a character for now:
```

```
liz.data <- read.csv(liz.file, strip.white = T
                     , colClasses = c("factor", "factor", "character", "integer"
                                      , "factor", "numeric", "numeric", "character"
                                      , "numeric", "character", "factor", "Date"))
#
## OH and another error. So charToDate(x) is telling me that it is having trouble
## reading my dates as dates. This may be because people were entering the dates
## in different formats in excel, so lets just leave that as a character for now
liz.data <- read.csv(liz.file, strip.white = T
                     , colClasses = c("factor", "factor", "character", "integer"
                                      , "factor", "numeric", "numeric", "character"
                                      , "numeric","character", "factor", "character"))
#
## Woohoo, it worked this time!...But let's double check
head(liz.data)  #Mmmmhmmm look at that Date column...there's our issue!
str(liz.data)  #lets double check that columns were properly assigned
```

OK, so we have our columns how we want them, let's get some of our issues cleaned up:

```
## Now we can tackle some of the pesky issues:
# let me introduce you to a girl's best friend - the tidyverse!
# A collection of R packages for data science: https://www.tidyverse.org/
# R for Data Science by Hadley Wickham and Garrett Grolemund: http://r4ds.had.co.nz/
#
# First lets get rid of those no's (and probable yes's) in the Regen
# I'm just going to run a table to see what we have going on
table(liz.data$Regen)  #so we have 82 blanks, 3 No's, 1 Yes, and the weird 38/6
#
# let's check out that weird 38/6, remember the bracket's ([row,col]) we used in
# the last script to pull out elements of the matrices and data.frames? We can
# use them here to find the row that matches the one weird record:
liz.data[liz.data$Regen == "38/6", ]  #so here I'm telling R that I want the one row
                                      #from the Regen column that matches the
                                      #character "38/6", then I use the ", ]" which
                                      #says I want all the columns
# Other ways to subset:
liz.data[liz.data$Regen %in% "38/6", ]
which(liz.data$Regen == "38/6")  #tells us the row:
liz.data[47,]  #you can use that actual row OR
liz.data[which(liz.data$Regen == "38/6"), ]
?subset
subset(liz.data, Regen == "38/6")
#
# so the notes tells me that there were two areas of tail regeneration, so I'm just
# going to change this to 38
# using dplyr::recode
liz.data$Regen <- recode(liz.data$Regen, "38/6" = "38")
# and we can double check:
table(liz.data$Regen) #and our "38/6" is gone, and we have an 3rd 38 - that's good
#
# and get rid of the No and Yes
liz.data$Regen <- recode(liz.data$Regen, "No" = "")  #we just want blanks here
liz.data[liz.data$Regen == "Yes", ]  #lets see if they measured the regen...
```

```
# nope they sure didn't. Since the total tail length includes the regen and we
# won't be doing much with the regen, I'm just going to leave it blank. If
# you were using this variable, you may have to account for this in another way.
#
liz.data$Regen <- recode(liz.data$Regen, "Yes" = "")
# and one last table to double check
table(liz.data$Regen)  #looks good
#
# so lets make that numeric now.
liz.data$Regen <- as.numeric(liz.data$Regen)
#
## Let's clean up the sex column, because we do want to use that:
table(liz.data$Sex)
# lets change the Male to M - capitilization matters!
liz.data$Sex <- recode(liz.data$Sex, "Male" = "M")  #TIP: This is a nice function
# because the recode changes the factor levels as well. Sometimes when you are
# working with factors, the levels can get complicated, so this is convenient
#
# and the J to blank, since it was small it was probably too hard to determine sex
liz.data[liz.data$Sex == "J",]  #Juv ASMA are too small to check sex
liz.data$Sex <- recode(liz.data$Sex, "J" = "")
class(liz.data$Sex)  #this is still a factor, and...
levels(liz.data$Sex)  #now we are down to the three levels we wanted
#
## So now that we have those basic columns cleaned up, lets subset the data frame
## down to just the columns we want to use
head(liz.data,2)
liz.sub <- liz.data[, c(1:2, 5:9)]  #here we are going to keep just some of the cols
head(liz.sub)
# Alternative ways to subset:
#liz.sub1 <- liz.data[, c("ID", "Mark", "Sex", "SVL", "Tail", "Regen", "Mass")]
#liz.sub2 <- subset(liz.data, select = c("ID", "Mark", "Sex", "SVL", "Tail", "Regen", "Mass")
#
## So many NA's - what if we wanted to create a subset of our data frame with
## no NA's?
liz.subna <- liz.sub[!is.na(liz.sub$Sex), ]
is.na(liz.subna$Sex)  #that's a lot of falses, lets summarize that
sum(is.na(liz.subna$Sex))  #sum counts FALSE as 0 and TRUE as 1, letting us get
                    # a count of na's
#
# BUT! because this was a factor that had levels assigned, just because it got
# subsetted doesn't mean those factors disappear! One way to get around this is
# to change the column to numeric or character, subset, then change back to factor
levels(liz.subna$Sex)
#
```

Now that we have a clean, subsetted version of our data, we can export a copy to use later

```
## Finally I'm going to export my clean data frame so that I can use it in other
## Places as needed
dir.create("outputs/data")  #first I'm going to add a folder to the directory
write.csv(liz.sub, "outputs/data/LizardData_Clean.csv")
#
```

## *Descriptive Statistics*

A review of the summary functions:

```
# since we are only working with this one data frame, I'm going to attach it here
# so I can just type the column names instead of dataframe$columnname
attach(liz.sub)  # just don't forget to detach at the end!
#
mean(Regen)  #Wait, we changed regen to numeric, why isn't it working???
mean(Regen, na.rm = T)  #R can't process it with the na's so we need to remove them
# so many decimal points!
round(mean(Regen, na.rm = T), 2)  #you can nest functions
#
#
sd(Regen, na.rm = T)
max(Regen, na.rm = T)
min(Regen, na.rm = T)
median(Regen, na.rm = T)
range(Regen, na.rm = T)
quantile(Regen, na.rm = T)
```

Apply functions:

The apply() family pertains to the R base package and is populated with functions to manipulate slices of data from matrices, arrays, lists and dataframes in a repetitive way. These functions allow crossing the data in a number of ways and avoid explicit use of loop constructs. They act on an input list, matrix or array and apply a named function with one or several optional arguments.

```
# Apply functions
apply(liz.sub[, 4:7], 2, mean, na.rm = T)

apply(liz.sub[, 4:7], 2, function(x) mean(x, na.rm = T))

tapply(SVL, ID, mean, na.rm = T)

detach(liz.sub)
#
```

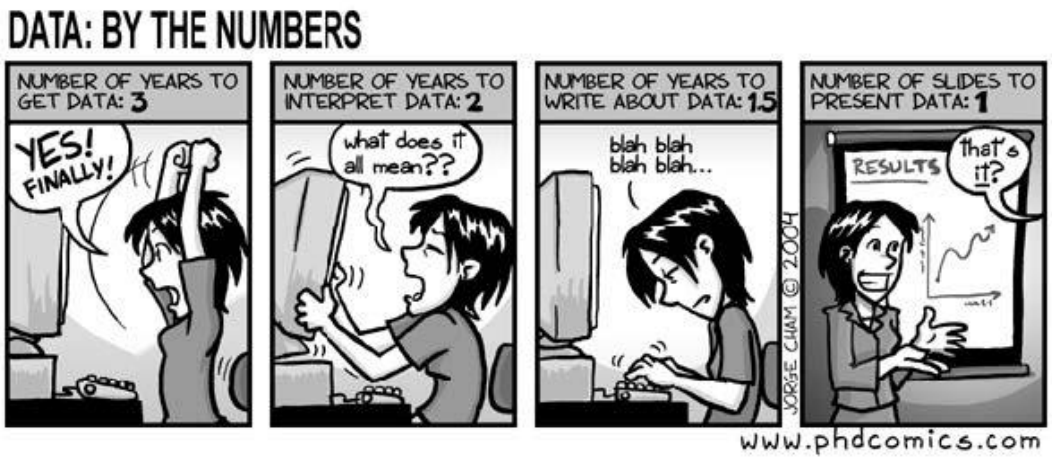
Summarising Data:

```
## What if you just want a table of summary information???
liz.summ <- liz.sub %>% group_by(ID) %>%
            summarise(avgSVL = mean(SVL, na.rm = T), sdSVL = sd(SVL, na.rm = T)
                      , maxSVL = max(SVL, na.rm = T), minSVL = min(SVL, na.rm = T)
                      , avgTail = mean(Tail, na.rm=T), sdTail=sd(Tail, na.rm = T)
                      , avgMass = mean(Mass, na.rm=T), sdMass=sd(Mass, na.rm=T)
                      , Nliz = n()
                      )

## Whoa, whoa, whoa, where did those "%>%" come from?! The package magrittr (which
```

```
## is called automatically through tidyverse) uses %>% essentially as a pipeline,
## breaking what we would normally nest in parentheses and telling R to take the
## value of that which is to the left and pass it to the right as an argument.
## So to translate what we just told R above: Here's the data frame (liz.sub),
## we want to evaluate for each species (group_by(ID)), and what we want to evaluate
## is everything in the summarise function.
#
# As a more digestible example, consider:
liz.sub$SVL %>% mean(na.rm = T)
liz.sub %>% with(SVL) %>% mean(na.rm = T)
liz.sub %>% .$SVL %>% mean(na.rm = T)  #note the ".$" which sends R back to the prior argument
# VS:
mean(liz.sub$SVL, na.rm = T)
#
# so we've essentially just rearranged the order we type, but R can still evaluate
# it fine. This takes practice, and isn't always better, but it definitely can
# make writing some more complex or nested code clearer. There are almost always
# multiple ways you can write code in R, so it's just becoming comfortable with
# what you know, but also working on writing clearer and more concise code.
#
## One other thing to be aware of, when working in the tidyverse: the tibble.
class(liz.summ)
head(liz.summ, 2)
# so now liz.summ has multiple classese associated with it, and now when you look
# at the data frame, we see it is called a tibble and you have more info about
# each column. Tibbles are data frames but they do much less: it never changes
# the type of the inputs (e.g. it never converts strings to factors!), it never
# changes the names of variables, and it never creates row names., but they tweak
# some older behaviours to make life a little easier. But we don't want to jump
# into those now. But for more info: http://r4ds.had.co.nz/tibbles.html
#
#
# because we want to keep this as a data frame, we are going to add one argument
# at the end of the code to tell R to turn it back to a data frame once the
# summarising is done:
liz.summ <- liz.sub %>% group_by(ID) %>%
                summarise(avgSVL = mean(SVL, na.rm = T), sdSVL = sd(SVL, na.rm = T)
                                , maxSVL = max(SVL, na.rm = T), minSVL = min(SVL, na.rm = T)
                                , avgTail = mean(Tail, na.rm=T), sdTail=sd(Tail, na.rm = T)
                                , avgMass = mean(Mass, na.rm=T), sdMass=sd(Mass, na.rm=T)
                                , Nliz = n()) %>%
                 as.data.frame()
#
# and again we can save that summary output:
write.csv(liz.summ, "outputs/data/LizardSummaryData.csv")
#
```

## *Graphing the Data*




<u>And finally, a quick intro to plotting using the R base options</u>

```
## R has a number of basic plotting functionalities that I'll show examples of
## here. There is also ggplot2, lattice, and many other packages, which I won't cover here.
#
## NOTE: I'm going to attach liz.sub here again to make our lives easier:
attach(liz.sub)  # TIP: when you write attach(), go down a couple lines and add
                 # detach() that way you don't forget to close it out!
# Basic plot types:
?plot  #so we can see the plot format of (x,y,...), where "..." is a myriad of arguments
#
plot(SVL, Tail)  #for a basic scatterplot
plot(Tail ~ SVL)  #we can also use the "~", here the format is y ~ x though
hist(SVL)  #we have different types of plots
boxplot(SVL ~ ID)  #more plots
plot(SVL ~ ID)  #but R can be smart about formatting the plots based on the data
#
# we can change different aspects:
plot(Tail ~ SVL)
plot(Tail ~ SVL, pch = 15)  #pch controls the shape
plot(Tail ~ SVL, cex = 2)  #cex give a relative size control
plot(Tail ~ SVL, col = "purple")  #you can change the color
plot(Tail ~ SVL, col = ID)  #you can plot different colors for different factors
plot(Tail ~ SVL, col = Sex)
plot(Tail ~ SVL, col = ID, xlab = "Snout-vent Length (mm)", ylab = "Tail Length (mm)")
# we can add a legend:
?legend
legend(15, 225, legend = levels(ID), fill = 1:6, cex = 0.5)

# Let's look at some of the other arguments:
boxplot(SVL ~ ID)
points(SVL ~ ID)  #we can actually add the points
points(SVL ~ jitter(as.numeric(ID)), col = "green")  #NOTE: we can't use jitter
```

```
                                                    # on a factor, so we adjust it
# we can add a fit line to the plot
plot(Tail ~ SVL)
abline(lm(Tail ~ SVL))  #lm() fits a linear regression
#
```

Saving our plots to files:

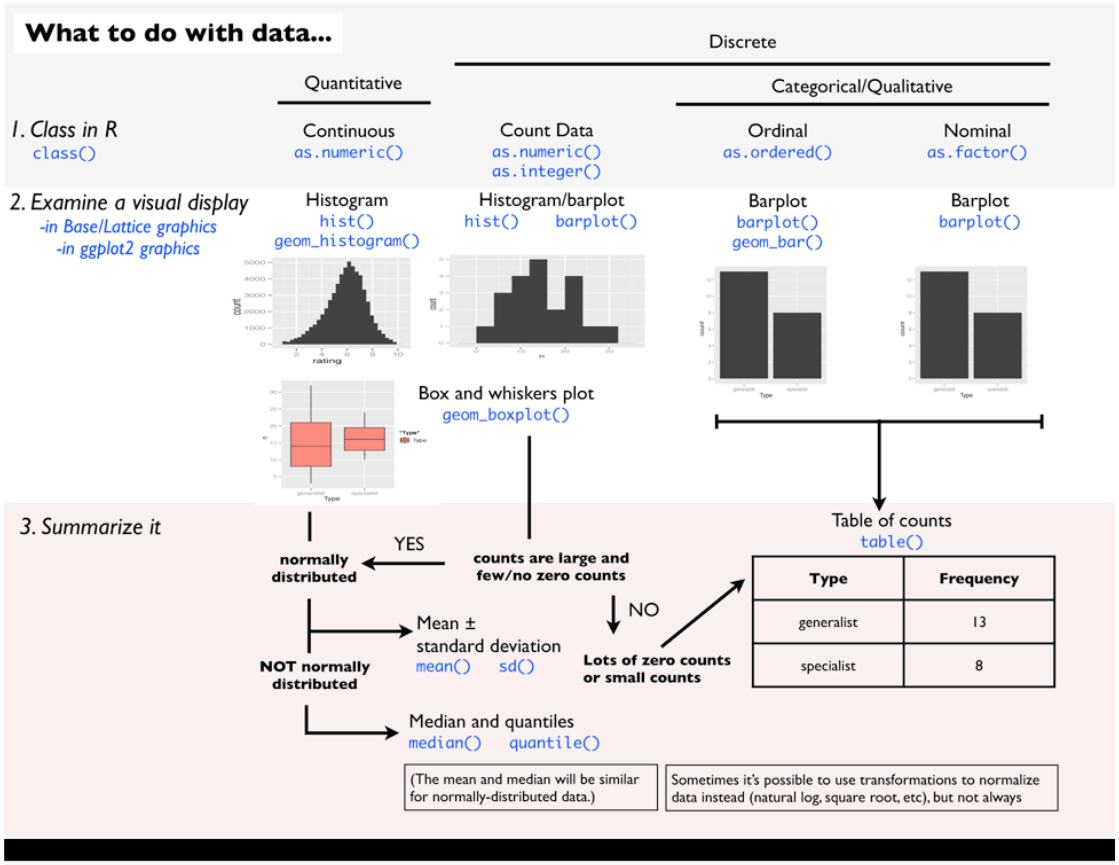
```
## Before we go let's export a plot:
?jpeg
dir.create("outputs/figures")
jpeg(filename = "outputs/figures/SVL_Tail_Scatter.jpg", width = 1200, height = 1200
       , units = "px", res = 300)
plot(Tail ~ SVL, col = ID, xlab = "Snout-vent Length (mm)"
        , ylab = "Tail Length (mm)")
legend(15, 225, legend = levels(ID), fill = 1:6, cex = 0.5)
dev.off()
#
# we can do other file types too!
png(filename = "outputs/figures/SVL_Tail_Scatter.png", width = 1200, height = 1200
      , units = "px", res = 300, bg = "transparent")
plot(Tail ~ SVL, col = ID, xlab = "Snout-vent Length (mm)"
      , ylab = "Tail Length (mm)")
legend(15, 225, legend = levels(ID), fill = 1:6, cex = 0.5)
dev.off()
#
detach(liz.sub)
#
```

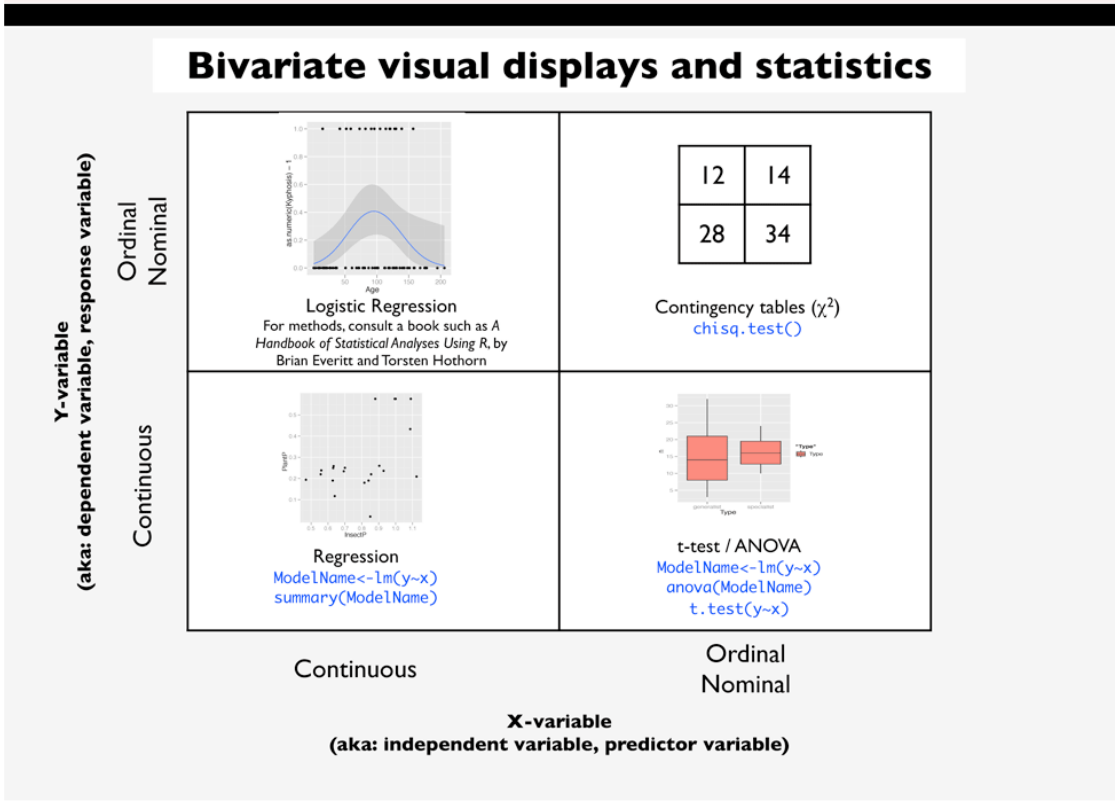
For inspiration: https://www.r-graph-gallery.com/

# Part 4: Basic Data Analyses

Here we will go over some very basic data analyses and trouble-shoot some issues. For this section, we will be running t-tests, anovas, and logistic regressions on our cleaned lizard data. We will be using the “4_BasicDataAnalyses.R” script.

Brief Overview (from Rebecca Clark, OSOS 2017)



## *Working Directory and Packages*

First things first:

```
# Set up our packages to install and load:
#install.packages(c("car","PerformanceAnalytics"))
library(car)
library(PerformanceAnalytics)
#
# Set up our working directory:
getwd()
#setwd()  #change the directory as needed
#
```

## *Get Our Data Frame*

```
# Lets load our cleaned data set:
dir("outputs/data")  #I can never remember the file names
#
liz.file <- "outputs/data/LizardData_Clean.csv"
liz.df <- read.csv(liz.file, strip.white = T)
#
head(liz.df)  #I always double check my loaded files!
str(liz.df)  #let's reload these with numeric
liz.df <- read.csv(liz.file, colClasses = c("character","factor","factor","factor"
                                            , "numeric", "numeric", "numeric"
                                            , "numeric")
                   , strip.white = T)
str(liz.df)  #looks good
#
```

## *Data Analysis Tools*

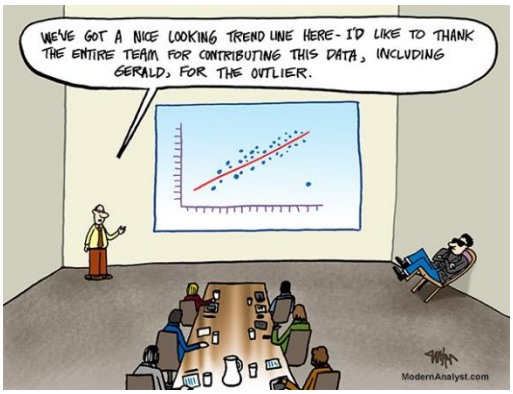
First – the t-test:

```
#the t-test:
?t.test
t.test(liz.df$SVL ~ liz.df$Sex)  #error because we have some too many levels
#
# let's double check our factor:
levels(liz.df$Sex)#
# let's try to remove them this way:
liz.df.tt <- liz.df[!is.na(liz.df$Sex), ]
sum(is.na(liz.df.tt$Sex))  #we should have no NA's
t.test(liz.df.tt$SVL ~ liz.df.tt$Sex)  #doesn't work again because of the factor levels
levels(liz.df.tt$Sex)  #so we still have the blank level showing, remember just
#because we subset out a factor doesn't mean it disappears
#
#let's try a workaround
liz.df.tt$Sex <- as.character(liz.df.tt$Sex)
table(liz.df.tt$Sex)  #Let's check what we have - blanks not NA's here
# so let's remove the blanks:
```

```
liz.df.tt <- liz.df.tt[liz.df.tt$Sex != "", ]
table(liz.df.tt$Sex)  #finally!!!
#
# and now we can turn it back to a factor to use it later
liz.df.tt$Sex <- as.factor(liz.df.tt$Sex)
levels(liz.df.tt$Sex)  #and we are down to two factors, lets try the ttest
#
t.test(liz.df.tt$SVL ~ liz.df.tt$Sex)  #it worked! Sadly this is likely confounded by species!
#
```

## Second, some ANOVAs

```
# let's try an anova:
# One-Way ANOVA:
liz.aov <- aov(liz.df.tt$SVL ~ liz.df.tt$ID)
summary(liz.aov)
plot(residuals(liz.aov))
TukeyHSD(liz.aov)  #check the pairwise differences
#
# Two-way ANOVA:
liz.2aov <- aov(SVL ~ ID + Sex + ID:Sex, data = liz.df.tt)
liz.2aov2 <- aov(SVL ~ ID*Sex, data = liz.df.tt)
summary(liz.2aov)
TukeyHSD(liz.2aov)
#
# But are the assumptions met? We'll look at the one-way ANOVA
liz.res <- residuals(liz.aov)
shapiro.test(liz.res)  #haha, nope
leveneTest(SVL ~ ID, data = liz.df.tt)  #thats ok
hist(liz.df.tt$SVL)
plot(liz.aov)  #can see the residuals
#
# What if we transformed the data? We can add a column to the data frame
liz.df.tt$logSVL <- log10(liz.df.tt$SVL)
shapiro.test(liz.df.tt$logSVL)  #better, but still not normal
#
# we could also just run the test but not save the data
shapiro.test(log10(liz.df.tt$SVL))
hist(log10(liz.df.tt$SVL))
#
```



```
# Still not normal, what about non-parametric tests to relax the normality assumption
# Kruskal-Wallace test (non-parametric ANOVA)
?kruskal.test
kruskal.test(SVL ~ ID, data = liz.df.tt)
# Wilcox Rank Sum test (non-parametric t-test)
?wilcox.test
wilcox.test(SVL ~ Sex, data = liz.df.tt)
#
```

## Finally, a little bit of Logistic Regression

```
# Linear models basics:
liz.lm <- lm(Mass ~ SVL + ID + Sex + ID:Sex, data = liz.df.tt)
summary(liz.lm)
plot(liz.lm)
#
# But as we know, correlated variables can affect the accuracy
# So lets check the correlation between SVL, Tail, and Mass
liz.df.cor <- liz.df.tt[,c(5,6,8)]
liz.df.cc <- liz.df.cor[complete.cases(liz.df.cor),]
#
# we can use the default
cor(liz.df.cc)
#
# I also like this function in the PerformanceAnalytics Package
chart.Correlation(liz.df.cc)
dev.off()  #i've found after this chart.Correlation, you want to clear the plot
        #area, otherwise future plot margins are funky
#
```

# Part 5: Random Tips

1. To update R: In R (not RStudio!):
    a. install.packages("installr")
    b. library(installr)
    c. updateR()  #if there is a new version, R will update

2. Some handy shortcuts:
    a. “Alt + -“ is the shortcut for <-
    b. “Ctrl + enter” to run a line of code
    c. “Ctrl + shift + p” to re-run the code you just ran

3. ?mean(), ??mean(), help(mean), example(mean) are all different ways you can look up info on the mean() function.

4. citation() will give you the current R citation; citation(ggplot2) will give you a package citation

# Resources and References

*The R Book, 2nd Edition*, by Michael Crawley. Wiley, 2012. **Definitive introduction to R for programming, statistics, and graphics.**

**Data Manipulation and Statistical Analysis**
*Data Manipulation With R*, by Phil Spector. **Invaluable reference no matter what you do**. Walks through all the basic R data classes and functions needed to manipulate them.

*A Handbook of Statistical Analyses Using R*, by Brian Everitt and Torsten Hothorn. A clear, thorough, up-to-date introduction with plenty of examples. Most useful if you know what kind of statistic you want to use; less useful if you're still learning statistics.

*Discovering Statistics Using R*, by Andy Field, Jeremy Miles, and Zoe Field. I have not used this, but a friend recommends it and it looks comprehensive for a new graduate student wanting to learn statistics and R at the same time.

*Mixed-Effects Models in S and S-Plus*, by Jose Pinheiro and Douglas Bates. Mixed-effects models are useful for repeated-measures designs where measurements are repeated on the same individuals or units. Only acquire this book if the subject is relevant to you.

**Tom Short's R Refcard**: http://cran.r-project.org/doc/contrib/Short-refcard.pdf This is a pretty comprehensive RefCard that's fairly well organized, from way back in 2004. The original site no longer works, but this copy comes from the CRAN.

**Quick-R**: http://www.statmethods.net/ Set up for people who know statistics, and want to know how to do stats with R. A great reference menu for the syntax needed for different types of statistical analyses. Also see their book suggestion page. And note that the site has also been transformed into a **Book** (link on the website).

# Functions and Commands Demonstrated

***Functions***

- aov fits an analysis of variance model
- apply is used to evaluate a function separately on the rows (second argument to apply is 1) or the columns (second argument to apply is 2) of a matrix or data frame
- as.character/character creates or coerces objects to type character
- as.data.frame/data.frame creates or coerces objects to type data.frame
- as.factor/factor creates or coerces objects to type factor
- as.integer/integer creates or coerces objects to type integer
- as.list/list creates or coerces objects to type list
- as.matrix/matrix creates or coerces objects to type matrix
- as.numeric/numeric creates or coerces objects to type numeric
- attach adds a data frame to the default search path so that variables can be specified without reference to the data frame in which they reside
- boxplot produces a box plot or side-by-side box plots of a specified variable
- c the catenation function that turns the elements making up its arguments into a single vector
- PerformanceAnalytics::chart.Correlation returns a plot matrix with correlations, histograms, and bivariate scatterplots with a fitted line
- class returns the class of an R object
- colnames returns the column names of a data frame. Can also be used to assign column names to a data frame
- complete.cases returns a TRUE/FALSE variable for whether a row has data in all columns
- cor returns a correlation matrix
- data.frame defines a data frame from its arguments which should be a set of vectors all of the same length
- detach undoes attach, removes a data frame from the search path
- dev.off shurts down the current device (usually a plot), we used it to close out our jpeg/png files
- dim returns the number of rows and columns of a data frame
- dir list the files in the working directory
- dimnames returns both the row names and the column names of a data frame in a list format
- getwd returns the absolute file path of the current working directory
- dplyr::group_by allows operations to be performed on one or more groups (tidyverse)
- head returns the first 6 elements of an R object
- hist plots a histogram from continuous data
- install.packages download and install packages from repositories or local files
- is.na is a logical function that returns TRUE if a value is missing (NA) and FALSE otherwise
- jitter randomly adds a small value to each element of its argument
- jpeg/png/tiff/bmp/pdf outputs a plot to a file with the specified type
- kruskal.test runs the Kruskal-Wallace test (non-parametric ANOVA)

- length returns the number of elements of a vector counting both non-missing and missing values
- levels returns the level attributes of a factor variable
- cars::leveneTest calculates the Levene test for homogeneity of variances
- library load and attach add-on packages
- lm runs the logistic regression on a formula
- log returns the natural logarithm of the value
- log10 returns the common (base 10) logarithm of the value
- max returns the maximum value of a vector
- mean calculates the mean of individual column entries of a data frame
- median calculates the median of vector or data frame column
- min returns the minimum value of a vector
- names displays names of variables in a data frame or list
- plot plots the R objects – type depends on the data provided
- points adds individual points to the currently active plot
- read.csv reads in comma separated data from an external file
- openxlsx::read.xlsx reads in excel files or worksheets from an external file
- range returns the minimum and maximum values of a vector or data column
- rep is used to create patterned vectors of repeated units
- dplyr::recode replaces numeric or character values in a vector (tidyverse)
- residuals extracts model residuals from objects
- round rounds its argument to the number of decimals specified.
- sd calculates the standard deviation of the individual column entries of a data frame
- setwd used to set the working directory
- shapiro.test runs the Shapiro-Wilks test for normality
- sqrt is the square root function in R
- str displays the internal structure of an R object
- sum calculates the sum of all entries of a vector or matrix
- dplyr::summarise reduces multiple values down to a single value (tidyverse)
- summary returns summary information for vectors, lists, data frames, and the results of other functions. Data returned depends on the class of the input.
- t transposed a matrix or data frame
- t-test performs one and two sample t-tests on vectors of data
- tail returns the last 6 elements of an R object
- tapply stands for table apply. It applies a function (3rd argument) to a variable (1st argument) separately for each group specified by the second argument
- TukeyHSD calculates the Tukey's Honest Significant Differences between factors from an anova object
- write.csv outputs a matrix or data frame as a csv file to the working directory

***Symbols***

- # indicates a given line of code is a comment and should be ignored
- <- the assignment operator in R, a less than symbol followed by a dash, that is supposed to symbolize an arrow. The arrow points in the direction of assignment.

- [ ] used for specifying elements of vectors or portions of data frames and matrices
- [[ ]] denotes an element of a list
- $ list notation symbol that can be used to reference columns of a data frame
- ! is the logical not operator in R
- ^ denotes exponentiation
- ? followed by a function name brings up a help window on that function
- ?? does a broad search for the term
- ~ symbol used in defining expressions in R for model fitting. We used it in the boxplot function
- %>% pipeline symbol, allows R to read functions from left to right, instead of nested parentheses

***Common arguments***

- cex = (argument to many graphics functions) specifies the character expansion for plotting symbols when used with the points function
- col = (argument to many graphics functions) specifies the color to use in plotting points and/or line segments
- header = (argument to read.csv) takes on values TRUE or FALSE, indicates whether the first line of a text file contains the variables names (TRUE) or not (FALSE)
- legend = (argument to many graphics functions) can add a legend to a plot
- pch = stands for print character and is used to designate the plotting symbol for use in various plotting functions: plot, points, etc.
- na.rm = (argument to mean, sum, and sd) take on values TRUE or FALSE, indicates whether missing values should be removed (TRUE) before performing calculations. If set to FALSE and there are missing value the function returns NA as its value.
- outline = (argument to boxplot) when set to FALSE it turns off the display of outliers in a box plot
- sep = (argument to read.table) specifies the character that was used to separate fields in the text file to be read into R. For example, sep=',' indicates that the entries are separated by commas while sep='\t' indicates that the entries are separated by tabs.
- xlab = (argument of boxplot) a user-specified value to be used as the label for the x-axis, e.g., xlab="WSSTA"
- ylab = (argument of boxplot) a user-specified value to be used as the label for the y-axis, e.g., ylab="Disease Prevalence "