Open Source Open Science Workshop – R Intro

Danielle Walkup (and Google!)

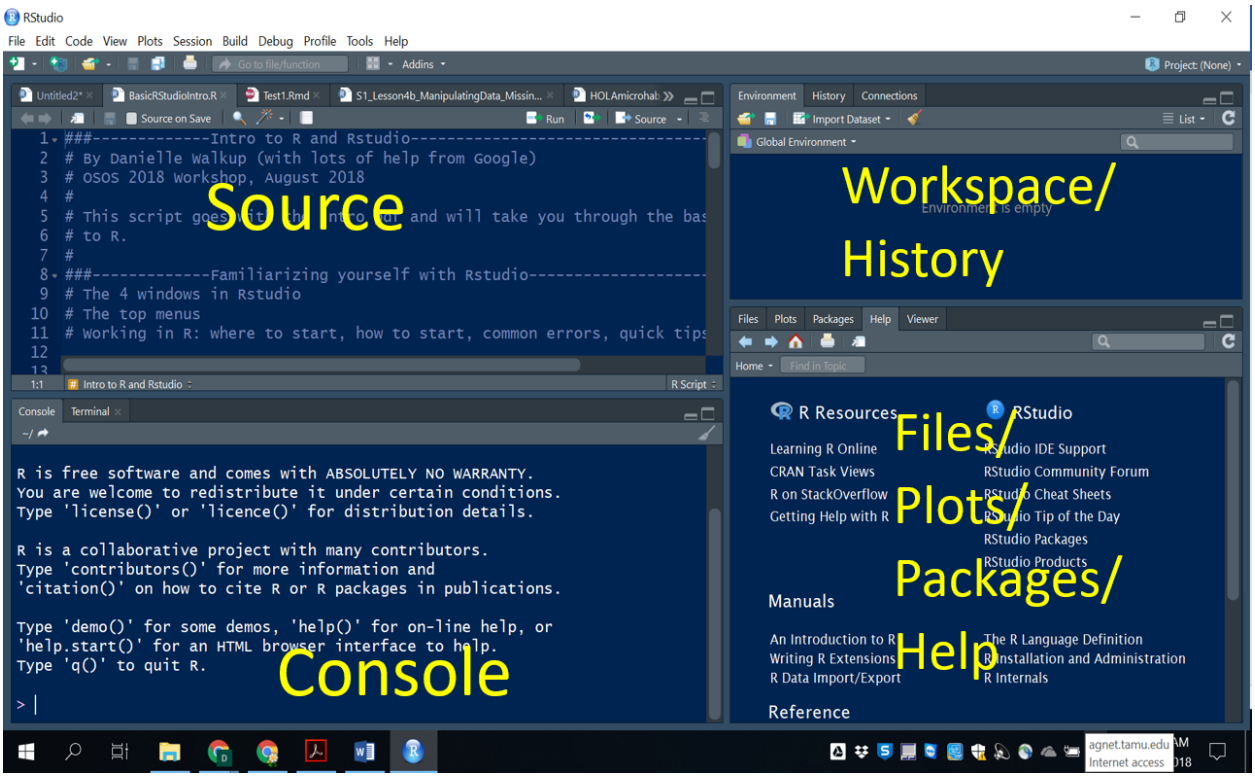
8/30/2019

1 An Intro to R and RStudio

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

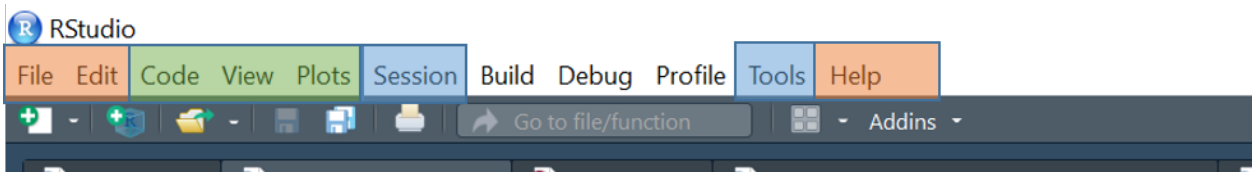
1.1 Familiarizing yourself with RStudio

1.1.1 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 right) – shows the objects you have created as well as the list of commands run.
4. Files/Plots/Packages/Help (bottom right):
   - Files – shows all the files in your working directory
   - Plots – shows the plots you have created during the session. Can also export and save them through this window
   - Packages – lists all the packages that come with R, as well as any packages you have installed
   - Help – lets you search R information

1.1.2 The RStudio 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).

1.2 The Working Directory

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

```
## The default directory when you open RStudio is the Documents folder. But to
## check your current directory you use:
getwd()

## If you want to see the files currently in the working directory:
dir()

## If you need to change the working directory:
setwd("Path-To-The-Directory")
# the easiest way to do this is to open the folder and copy and paste the file
# path into the quotes.

## Note that on a pc, you need to use `/` in the path, `\` is used as an escape
## character in R, so if you copy-paste, make sure you change `\` to `/`.

## Let's double check it worked:
getwd()
dir()
```

TIP: If you open an R script from somewhere else in your computer (without R/RStudio being open), the working directory defaults to the folder where your R script was stored.

TIP: If setwd() just isn't working and you need to get on with your life, you can use the RStudio menu: Session -> Set Working Directory -> Choose Directory (OR ctrl-shift-H). Yay RStudio!

1.3 Working in RStudio

The obligatory R can be used as a calculator tutorial:
TIP: In any script, you can use ctrl-enter to run each line of code

```
## You can do basic arithmetic with R
3 + 5 + 8
```

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

```
## It will follow order of operations rules
5 * 9 + 14
```

```
## [1] 59
```

```
5 * (9 + 14)
```

```
## [1] 115
```

```
## Other operators:
12^2  #can do exponents
```

```
## [1] 144
```

```
sqrt(100)  #and square roots
```

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

```
pi * 3  #there are some built in constants
```

```
## [1] 9.424778
```

```
log(5)  #log in R equals the natural log (i.e. ln)
```

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

```
log10(5)  #this will give you log base 10
```

```
## [1] 0.69897
```

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.

NOTE: <- is the assignment operator, we will discuss this more later.

```
## 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
```

```
##  [1] 4 6 3 6 7 3 4 6 7 2
```

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

```
## [1] 4.8
```

```
min(ex.data)
```

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

```
max(ex.data)
```

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

```
median(ex.data)
```

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

```
sd(ex.data)
```

```
## [1] 1.813529
```

```
# Or you can see these in
summary(ex.data)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    2.00    3.25    5.00    4.80    6.00    7.00
```

```
## Some basic tools to help you understand and check your objects
str(ex.data)
```

```
##  num [1:10] 4 6 3 6 7 3 4 6 7 2
```

```
dim(ex.data)  #NULL here, better for dataframes or other objects
```

```
## NULL
```

```
length(ex.data)  #since dim() didn't work, lets use length() instead.
```

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

```
head(ex.data)  #by default returns the 1st 6 things in the object
```

```
## [1] 4 6 3 6 7 3
```

```
head(ex.data, 2)  #but we can tell it how many we actually want (> or < 6)
```

```
## [1] 4 6
```

```
tail(ex.data, 3)
```

```
## [1] 6 7 2
```

1.4 Packages - the workhorses of R

Many programs include all the functionality 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 automatically downloaded with R, but you can download them yourself. To see what's available, check **cran.r-project.org** under Packages. The packages included in the CRAN repository are subject to review and have to conform to specific policies. Often, developers may host packages on platforms like Github as they develop or update packages, and there are ways to load packages from github or other sources. However, those packages may not be held to the same standards as those on the R CRAN repository.

1.4.1 Installing and Loading Packages

To install a package, you can use the function `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 (there should be a Texas one). You'll only have to choose a mirror the first time you set up a new R version.

NOTE: Once you install a package, you don't have to re-install it each time you want to use it.

NOTE: To actually use a package in a script, you do have to load the package each time. To load a package, you can use the function `library(package-name)`. Then all the functions in the package are available for your use. Some packages have dependencies - other packages that they use functions or data object types from. These other packages are typically installed alongside the main package, but sometimes you may get an error and have to download those packages directly.

NOTE: If you load multiple packages, some may have functions with the same name. In that case, the most recently loaded package will mask the function from a previously loaded package. To call a function from a specific package you can use package-name::function-name.

```
## 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 - you'll use this set tomorrow (and a
# little later on today) for data manipulation, exploration, and plotting
install.packages("tidyverse")

# Some handy spatial packages
install.packages(c("sp", "rgdal", "rgeos", "ggmap", "raster"))

# and one we will use later on
install.packages("openxlsx")  #note the quotes here
library(openxlsx)  #don't have to have quotes here
```

One of the potentially convenient features of working in RStudio is that you can search, install, update, and load packages from the bottom right corner window. If you scroll through you can see there are a couple of base packages that are uploaded each time. If there is a package you want to use, you can just check the box on the far left side to load the package into your RStudio environment. If you try it, you can see that `library(package-name)` shows up in the console and loads that package and any dependencies. However, because I generally work using R scripts, I typically just include all the packages I use at the top of the script, because I will be using that specific set everytime I use that script.

1.4.2 Functions in R

Packages in R are made up of functions - specific chunks of code that take an object and arguments and return a specific output. Much of the code we have run so far has been made up of functions: `install.packages()`, `dir()`, `mean()` are all functions. To find out how to use a specific function, you can search help for more information on a function. Typically, help on a function will return a brief Description; Usage of a function showing the arguments and their defaults; more in-depth explanation of Arguments; a description of the output or Value; any relevent References; and finally, Examples.

```
`?`(lm)
```

```
## starting httpd help server ... done
```

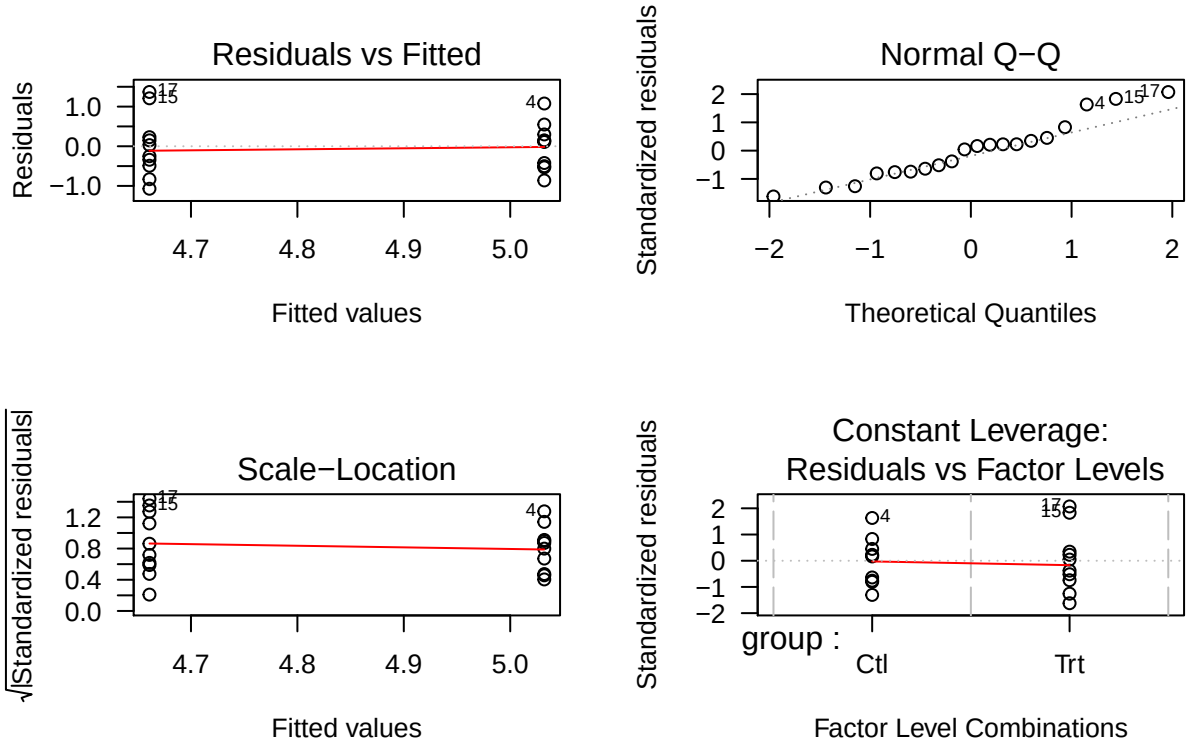
```
args(lm)
```

```
## function (formula, data, subset, weights, na.action, method = "qr",
##     model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE,
##     contrasts = NULL, offset, ...)
```

```
## NULL
```

```
example(lm)
```

```
##
## lm> require(graphics)
##
## lm> ## Annette Dobson (1990) "An Introduction to Generalized Linear Models".
## lm> ## Page 9: Plant Weight Data.
## lm> ctl <- c(4.17,5.58,5.18,6.11,4.50,4.61,5.17,4.53,5.33,5.14)
##
## lm> trt <- c(4.81,4.17,4.41,3.59,5.87,3.83,6.03,4.89,4.32,4.69)
##
## lm> group <- gl(2, 10, 20, labels = c("Ctl","Trt"))
##
## lm> weight <- c(ctl, trt)
##
## lm> lm.D9 <- lm(weight ~ group)
##
## lm> lm.D90 <- lm(weight ~ group - 1) # omitting intercept
##
## lm> ## No test:
## lm> ##D anova(lm.D9)
## lm> ##D summary(lm.D90)
## lm> ## End(No test)
## lm> opar <- par(mfrow = c(2,2), oma = c(0, 0, 1.1, 0))
##
## lm> plot(lm.D9, las = 1)      # Residuals, Fitted, ...
```




```
##
## lm> par(opar)
##
## lm> ## Don't show:
## lm> ## model frame :
## lm> stopifnot(identical(lm(weight ~ group, method = "model.frame"),
## lm+                     model.frame(lm.D9)))
##
## lm> ## End(Don't show)
## lm> ### less simple examples in "See Also" above
## lm>
## lm>
## lm>
```

You can also write your own functions in R, following the same basic layout of:

```
my_function <- function(arg1, arg2, ...) {
  function body - lines of statements of what the function does
}
```

```
## A basic function example
test_function <- function(x, y, z) {
    math <- x + y + (x * y) - z
    print(math)
}

## To call the function, you use the function name and feed it some arguments
```

```r
## Arguments can be called in order:
test_function(3, 5, 11)
```

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

```r
# Or by name:
test_function(x = 5, y = 11, z = 3)
```

```
## [1] 68
```

Writing your own functions can be helpful for reducing the amount of time you spend doing repetitive tasks, making your code shorter and easier to read and interpret, and reducing the errors that might come from copying and pasting repetitive tasks. "Good" functions typically are short, do only one thing, and have a descriptive name. Functions can also be written in a different R script and called in the script you are working on using `source(file.R)`.

1.4.3 For Loops, If-Else, and apply statements

In addition to functions, there are some other typical programming features that you can use. These include for or while loops, If-Else statments, and R's apply functions. Generally, since R is not as fast as other programming languages, it's recommended to use apply or write a new function instead of using for loops.

For loops take the structure of:

```r
for (value in sequence) {
  do something(s)
}
```

```r
## An example of a short for loop:
for (i in 1:10) {
    print(sqrt(i))
}
```

```
## [1] 1
## [1] 1.414214
## [1] 1.732051
## [1] 2
## [1] 2.236068
## [1] 2.44949
## [1] 2.645751
## [1] 2.828427
## [1] 3
## [1] 3.162278
```

While loops are similar to for loops, however the loop will continue to run as long as the specified condition is true. This means you can have a while loop that will run forever!
Tip: If you ever get stuck with a function that is running and running and running, you can click the red stop sign that will appear at the top of the console.

```r
while(condition) {
  do something(s)
}
```

```r
## A brief example
x <- 4
while (x < 500) {
    print(x)
```

```
    x <- x^2
}
```

```
## [1] 4
## [1] 16
## [1] 256
```

```
# Notice that because we were assigning the x-squared value to x, the final x
# value is 256 squared.
x
```

```
## [1] 65536
```

If-Else statements take the structure of:

```
## If alone:
if(condition *is true*) {
  do something(s)
}

## If-else:
if(condition *is true*) {
  do something(S)
} else *condition is false* {
  do something(s)
}
```

```
## A brief example:

if (x > 0) {
    print("x is positive")
} else {
    print("x is negative")
}
```

```
## [1] "x is positive"
```

R also has apply functions: The apply() functions are part of the R base package and are designed 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 family functions
apply()  # works over arrays or matrices
lapply()  # works over vectors, data frames, or lists; returns a list
sapply()  # works like lapply; returns the simplest data structure - so may return a vector
# instead of a list
mapply()  # a multivariate version of sapply. It will apply the specified function to the
# first element of each argument first, followed by the second element, and so
# on.
tapply()  # applies a function to numeric data distributed across various categories
```

```
## Some brief examples
ex_mat <- matrix(c(1:5, 13:17), nrow = 2, ncol = 5)
ex_mat
```

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

```
## [1,]    1    3    5   14   16
## [2,]    2    4   13   15   17
```

```
# apply(X = data, MARGIN = direction the function is applied, FUN = function to
# apply)
apply(ex_mat, 1, sum)  # the 1 tells apply to sum by row
```

```
## [1] 39 51
```

```
apply(ex_mat, 2, sum)  # the 2 tells apply to sum by column
```

```
## [1]  3  7 18 29 33
```

```
# lapply works similarly, but the input needs to be a list. Note that lapply
# doesn't have the MARGIN argument, because the function is applied to each
# element of the list
ex_list <- list(x = 1:5, y = 45:50, z = 15:20)
lapply(ex_list, sum)
```

```
## $x
## [1] 15
##
## $y
## [1] 285
##
## $z
## [1] 105
```

```
# sapply works like lapply, but returns a different class of object
sapply(ex_list, sum)
```

```
##   x   y   z
##  15 285 105
```

```
class(lapply(ex_list, sum))  # returns a list
```

```
## [1] "list"
```

```
class(sapply(ex_list, sum))  # returns a vector of integers
```

```
## [1] "integer"
```

```
# however, we can force sapply to return a list and not simplify by adding an
# additional argument
class(sapply(ex_list, sum, simplify = F))
```

```
## [1] "list"
```

```
# mapply is the multivariate version of sapply
mapply(sum, 1:5, 1:5)  # applies the function sum to the first element of each vector
```

```
## [1]  2  4  6  8 10
```

```
# (1 + 1), then the second element (2 + 2), and so on
mapply(sum, 1:5, 5:1)
```

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

```
# tapply allows you to evaluate based on a categorical factor
ex_df <- data.frame(Type = rep(c("red", "blue"), 5), x = rnorm(10))
tapply(ex_df$x, ex_df$Type, mean)  # we take the average of x for each type, red or blue
```

```
##       blue        red
##  1.1932376 -0.6178804
```

2 Organization and Good Coding Practices

The current recommended R style guide can be found here: **Tidyverse Style Guide**

2.1 Organization

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. File names for scripts should be meaningful (and don't forget to name them with `.R`). If you have multiple files that need to be run in a specific order, you can prefix them with numbers.

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.
RStudio Projects
Software Carpentry Projects Tutorial

2.2 (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:
   - Add general purpose and authors at the top of the script, as well as any notes you think will be helpful
   - Install and load packages at the top of the script so we know what we need upfront
   - Check/Set the working directory at the top of the script
2. Comment, comment, and comment your code!
   - In R, anything preceded by a '#' will not be evaluated
   - The scripts are for people, so use comments to help people follow through the script, future you will thank you!
   - Explain WHY you are doing things and making the choices you are.
3. Use a consistent style within your code
   - Use "# Section Name —-" to create sections in your script that break it up into smaller chunks
   - Use meaningful names (e.g. lizard_df, not mydata)
   - Wrap long lines (recommended length seems to be around 80 characters. This includes both comments and function calls.
   - Use spaces around operators and after commas (e.g. col = 'green', pch = 4)
4. Possibly controversial tips:
   - Use <- for assignments, and = in functions or arguments (alt + - is the shortcut for <-).
   - 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!

TIP: you can add a margin to help with this through tools -> global options -> code -> display -> show margin (margin column 80)

3 R Objects

3.1 Data Types

For this section, we will be using "2_BasicRStudioIntro.R"

R has a number of basic object types that we use.

R data type	Other terms	Examples
Numeric	Real, Continuous, Quantitative	4.69; 3.5, 3, 405, 285
Integer	Counts	1, 2, 3
Factor	Ordinal, Categorical, Nominal, Discrete	Low, Medium, High; Site1, Site2, Site3
Logical		TRUE, FALSE
Character		gravid, lizard, bird

Some examples of the data types:

```
# The classes can vary depending on the type of data:
liz.wt.num <- c(1.4, 5.6, 8.3, 5.6, 5)
class(liz.wt.num)  #returns numbers
```

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

```
typeof(liz.wt.num)
```

```
## [1] "double"
```

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

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

```
typeof(liz.wt.ch)
```

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

```
# if you need to change the class, you can:
liz.wt.change <- as.numeric(liz.wt.ch)
class(liz.wt.change)
```

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

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

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

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

```
## [1] "integer"
```

```
liz.wt.int
```

```
## [1] 1 5 8 5 5
```

```r
## Factors Ordering of factors defaults to alphabetical:
fruit <- factor(c("apple", "pear", "banana", "grape"))
fruit
```

```
## [1] apple  pear   banana grape
## Levels: apple banana grape pear
```

```r
# But we can change that if we want to:
fruit2 <- factor(c("apple", "pear", "banana", "grape"), levels = c("apple", "pear",
    "banana", "grape"))
fruit2
```

```
## [1] apple  pear   banana grape
## Levels: apple pear banana grape
```

```r
# 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
```

```
## [1] apple  pear   banana grape
## Levels: apple < banana < grape < pear
```

```r
fruit.ord2 <- factor(c("apple", "pear", "banana", "grape"), levels = c("apple", "pear",
    "banana", "grape"), ordered = TRUE)
fruit.ord2  #Much better - if you like grapes...
```

```
## [1] apple  pear   banana grape
## Levels: apple < pear < banana < grape
```

```r
# Notice there are some constraints on classes:
fruit.ch <- as.character(fruit)
class(fruit.ch)
```

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

```r
fruit.ch
```

```
## [1] "apple"  "pear"   "banana" "grape"
```

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

```
## Warning: NAs introduced by coercion
```

```r
class(fruit.num)
```

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

```r
fruit.num
```

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

```r
# BUT, be careful, because we can coerce factors to numbers or integers:
fruit.num2 <- as.numeric(fruit)  #using the alphabetically ordered fruit
class(fruit.num2)
```

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

```r
fruit.num2  #Note that it keeps the ordering
```

```
## [1] 1 4 2 3
```

```r
fruit.int <- as.integer(fruit2)  #using our fruit orders
class(fruit.int)
```

```
## [1] "integer"
```

```
fruit.int  #Again, note that it keeps the changed ordering
```

```
## [1] 1 2 3 4
```

These data types can be combined in different ways to create different data objects.

Data.Structures	Description	Examples
Vectors	A collection of elements of one type (typically of character, logical, integer, or numeric).	a <- c(1, 2, 3, 4, 5) 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.	ex_list <- list(1, 'a', TRUE, 1+4i) ex_biglist <- list(a = 'Testing', b = c(1,3,5,6,7), flowers = head(iris))
Matrices	An extension of numeric or character vectors that has dimensions: rows and columns.	ex_mat <- matrix(1:10, nrow = 2) ex_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.	ex_df <- data.frame(id = rep(c('a','b','c'),4), height = 1:12, width = 12:1)

Vectors - collection of elements of the same type. (These are generally what we've been working with up to this point). They can be vectors of any type, but all elements within them are forced to the same type.

```
# Set up a couple of example vectors:
a <- c(1, 2, 3, 4, 5)
b <- c("apple", "pear", "banana", "grape")
c <- c("blue", 3, 0+5i, "lizard")

# We can see elements in the vectors:
b[3]  #the single bracket lets us see the element in the 3rd place in the vector
```

```
## [1] "banana"
```

```
b[8]  #there's nothing in here - because we only have the 4 fruits!
```

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

```
# We can get all sorts of information about the vectors:
class(a)  #notice this returns the atomic vector type
```

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

```
class(c)
```

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

```
length(b)
```

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

```
is.na(a)  #a handy function to check for missing values
```

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

```
sum(a)  #we can run functions specific to the given class
```

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

```
# sum(b) #no numbers to add!
str(a)
```

```
##  num [1:5] 1 2 3 4 5
```

List Examples:

```
# set up a list
ex_list <- list(1, "a", TRUE, 1 + (0+4i))
class(ex_list)  #now we see this is a list, with its special list properties!
```

```
## [1] "list"
```

```
# We can still see the different elements and characteristics
ex_list[2]
```

```
## [[1]]
## [1] "a"
```

```
length(ex_list)
```

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

```
str(ex_list)
```

```
## List of 4
##  $ : num 1
##  $ : chr "a"
##  $ : logi TRUE
##  $ : cplx 1+4i
```

```
# You can also name the elements in each list
ex_biglist <- list(a = "Testing", b = 1:10, flowers = head(iris))

# now if we want to see what's in the list:
names(ex_biglist)  #we can also see the names
```

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

```
ex_biglist[2]  #and what's in the list
```

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

```
ex_biglist$b  #alternatively, if we have elements in the list named, we can use the '$' to choose the d
```

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

```
length(ex_biglist)  #but we still only have 3 things in the list
```

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

Matrix Examples:

```
# let's set up another matrix
ex_mat <- matrix(1:10, nrow = 2)
ex_mat
```

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

```
ex_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
ex_mat2
```

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

```
# and we can see characteristics:
class(ex_mat)
```

```
## [1] "matrix"
```

```
str(ex_mat)
```

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

```
length(ex_mat)
```

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

```
dim(ex_mat)  #Now we can see the dimensions!
```

```
## [1] 2 5
```

```
ex_mat[2, 1]  #now that we have 2 dimensions, we can use [row,column] to id an element
```

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

```
ex_mat[, 2]  #or see just a column
```

```
## [1] 3 4
```

```
ex_mat[2, ]  #or see just a row
```

```
## [1]  2  4  6  8 10
```

```
sum(ex_mat)  #can still do math on a numeric matrix -
```

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

```
t(ex_mat)  #can also transpose a matrix
```

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

Data Frame Examples:

```r
# here we will create a data frame with 3 columns: id, height, and width
ex_df <- data.frame(id = rep(c("a", "b", "c"), 4), height = 1:12, width = 12:1)
ex_df  #lets make sure it did what we expected
```

```
##    id height width
## 1   a      1    12
## 2   b      2    11
## 3   c      3    10
## 4   a      4     9
## 5   b      5     8
## 6   c      6     7
## 7   a      7     6
## 8   b      8     5
## 9   c      9     4
## 10  a     10     3
## 11  b     11     2
## 12  c     12     1
```

```r
# and check out some of its characteristics:
class(ex_df)
```

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

```r
str(ex_df)
```

```
## 'data.frame':    12 obs. of  3 variables:
##  $ id    : Factor w/ 3 levels "a","b","c": 1 2 3 1 2 3 1 2 3 1 ...
##  $ height: int  1 2 3 4 5 6 7 8 9 10 ...
##  $ width : int  12 11 10 9 8 7 6 5 4 3 ...
```

```r
summary(ex_df)
```

```
##  id        height          width
##  a:4   Min.   : 1.00   Min.   : 1.00
##  b:4   1st Qu.: 3.75   1st Qu.: 3.75
##  c:4   Median : 6.50   Median : 6.50
##        Mean   : 6.50   Mean   : 6.50
##        3rd Qu.: 9.25   3rd Qu.: 9.25
##        Max.   :12.00   Max.   :12.00
```

```r
length(ex_df)  #we can still get a length, but it may not quite be the info we want
```

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

```r
dim(ex_df)
```

```
## [1] 12  3
```

```r
t(ex_df)  #we can still transpose the data frame
```

```
##        [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
## id     "a"  "b"  "c"  "a"  "b"  "c"  "a"  "b"  "c"  "a"   "b"   "c"
## height " 1" " 2" " 3" " 4" " 5" " 6" " 7" " 8" " 9" "10"  "11"  "12"
## width  "12" "11" "10" " 9" " 8" " 7" " 6" " 5" " 4" " 3"  " 2"  " 1"
```

```r
# and we can see the different parts:
ex_df$id  #note that id has levels, so it is a factor
```

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

```
## Levels: a b c
```

```
class(ex_df$id)  #which we can double check here
```

```
## [1] "factor"
```

```
class(ex_df$width)
```

```
## [1] "integer"
```

```
# and it has defaulted width to integer, which we could change:
as.numeric(ex_df$width)
```

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

```
class(ex_df$width)
```

```
## [1] "integer"
```

```
# We could even do math within the data frame:
sum(ex_df$width)
```

```
## [1] 78
```

```
mean(ex_df$width)
```

```
## [1] 6.5
```

3.2 Reading and Writing Data

For this section, we'll use "3_DataLoadCleanExport.R".

3.2.1 Importing Data

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 upload data

## read.csv has a ton of options to read in your data set, check out the different
## arguments:
args(read.csv)
```

```
## function (file, header = TRUE, sep = ",", quote = "\"", dec = ".",
##     fill = TRUE, comment.char = "", ...)
## NULL
```

```
help(read.csv)
# 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

# First, lets make sure the file is where we think it is!
dir("data")  #looks good
```

```
## [1] "LizardData.csv"  "LizardData.txt"  "LizardData.xlsx"
```

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

## If the file is saved as a .csv we can simply read it in:
liz_df <- read.csv(liz_csv)
head(liz_df, 2)  #double check that it worked
```

```
##     ID Mark Grid Trap Sex SVL Tail Regen Mass Notes Recap      Date
## 1 ASMA  101  EFB    4   M  94  221         27           N 5/26/2012
## 2 ASMA  201  EFB    7   M  92  213    86   25           N 26-May-12
```

```
# and we can check out our new data frame:
class(liz_df)
```

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

```
head(liz_df)  #look it automatically reads our headings!
```

```
##      ID Mark Grid Trap Sex SVL Tail Regen Mass Notes Recap      Date
## 1  ASMA  101  EFB    4   M  94  221         27           N 5/26/2012
## 2  ASMA  201  EFB    7   M  92  213    86   25           N 26-May-12
## 3  ASMA  202  EFB    7   M  84  199    83   18           N 26-May-12
## 4  ASMA  203  EFC    8   M  92  228         23           N 26-May-12
## 5  ASMA  301  EBA    7   M  91  120         20          no 5/26/2012
## 6 ASMA   302  ECE    1   M 105  235         24          no 5/26/2012
```

```
str(liz_df)  #tells us the column classes and info about the data.frame
```

```
## 'data.frame':    107 obs. of  12 variables:
##  $ ID   : Factor w/ 7 levels "ASMA","ASMA ",..: 1 1 1 1 1 2 1 1 1 1 ...
##  $ Mark : Factor w/ 95 levels "","10","101",..: 3 39 40 41 51 52 56 57 62 63 ...
##  $ Grid : Factor w/ 35 levels "","BE","CB","DE",..: 31 31 31 32 10 20 21 27 2 3 ...
##  $ Trap : int  4 7 7 8 7 1 6 1 8 8 ...
##  $ Sex  : Factor w/ 5 levels "","F","J","M",..: 4 4 4 4 4 4 5 5 4 4 ...
##  $ SVL  : int  94 92 84 92 91 105 89 82 93 74 ...
##  $ Tail : int  221 213 199 228 120 235 186 124 200 177 ...
##  $ Regen: Factor w/ 21 levels "","1","121","13",..: 1 19 18 1 1 1 21 20 5 1 ...
##  $ Mass : num  27 25 18 23 20 24 22 19 27 15.5 ...
##  $ Notes: Factor w/ 36 levels "","2 REGENS ONE AT 38 AND ONE AT 6",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ Recap: Factor w/ 6 levels "?","N","no","No",..: 2 2 2 2 3 3 4 4 3 3 ...
##  $ Date : Factor w/ 9 levels "2/7/2013","26-May-12",..: 7 2 2 2 7 7 7 7 7 7 ...
```

```
dim(liz_df)
```

```
## [1] 107  12
```

```
summary(liz_df)  #gives us a brief overview of each column, already, I can see some indicators of a mes
```

```
##      ID          Mark         Grid         Trap         Sex
##  ASMA :17   101    : 3   EAA    :10   Min.   :1.000       : 1
##  ASMA : 1           : 2   ECA    : 8   1st Qu.:3.000   F   :35
##  ASSE : 1   126     : 2   EBA    : 7   Median :5.000   J   : 1
##  PHCO : 2   201     : 2   EBB    : 6   Mean   :5.038   M   :63
##  SCAR :28   202     : 2   EEA    : 6   3rd Qu.:7.000   Male: 4
##  SCCO : 3   301     : 2   EFB    : 6   Max.   :9.000   NA's: 3
##  UTST :55   (Other):94   (Other):64   NA's   :1
##       SVL              Tail            Regen         Mass
```

```
##  Min.   : 13.00   Min.   : 18.00          :82   Min.   : 1.40
##  1st Qu.: 44.00   1st Qu.: 60.25   No     : 3   1st Qu.: 2.90
##  Median : 48.00   Median : 74.00   13     : 2   Median : 3.90
##  Mean   : 52.96   Mean   : 85.95   35     : 2   Mean   : 6.46
##  3rd Qu.: 57.25   3rd Qu.: 88.75   38     : 2   3rd Qu.: 5.70
##  Max.   :105.00   Max.   :235.00   1      : 1   Max.   :27.00
##  NA's   :9        NA's   :9        (Other):15   NA's   :10
##                                Notes    Recap          Date
##                                   :69   ? : 1   2/7/2013 :26
##  GRAVID                           : 2   N :38   5/10/2013:18
##  RTB                              : 2   no: 7   5/26/2012:17
##  RTW                              : 2   No: 4   27-Feb-14:15
##  2 REGENS ONE AT 38 AND ONE AT 6: 1   Y :56   4/1/2014 :13
##  2,3,5 MISSING                    : 1   Y?: 1   5/9/2013 : 6
##  (Other)                          :30           (Other)  :12
```

```
## If we have a text file, we can still read it in using read.csv
liz_txt <- "data/LizardData.txt"
liz_df_txt <- read.csv(liz_txt, sep = "\t")  #sep = '\t' tells R its a tab-delimited text file
head(liz_df_txt, 2)  #and it looks the same as above!
```

```
##     ID Mark Grid Trap Sex SVL Tail Regen Mass Notes Recap      Date
## 1 ASMA  101  EFB    4   M  94  221         27           N 5/26/2012
## 2 ASMA  201  EFB    7   M  92  213    86   25           N 26-May-12
```

```
## Finally, we can also load the excel file using our package openxlsx:
liz_xl <- "data/LizardData.xlsx"
liz_df_xl <- read.xlsx(liz_xl)
head(liz_df_xl, 2)
```

```
##     ID Mark Grid Trap Sex SVL Tail Regen Mass Notes Recap  Date
## 1 ASMA  101  EFB    4   M  94  221  <NA>   27  <NA>     N 41055
## 2 ASMA  201  EFB    7   M  92  213    86   25  <NA>     N 41055
```

```
liz_df_xl2 <- read_excel(liz_xl)
head(liz_df_xl2)
```

```
## # A tibble: 6 x 12
##   ID    Mark  Grid   Trap Sex     SVL  Tail Regen  Mass Notes Recap
##   <chr> <chr> <chr> <dbl> <chr> <dbl> <dbl> <chr> <dbl> <chr> <chr>
## 1 ASMA  101   EFB       4 M        94   221 <NA>     27 <NA>  N
## 2 ASMA  201   EFB       7 M        92   213 86       25 <NA>  N
## 3 ASMA  202   EFB       7 M        84   199 83       18 <NA>  N
## 4 ASMA  203   EFC       8 M        92   228 <NA>     23 <NA>  N
## 5 ASMA  301   EBA       7 M        91   120 <NA>     20 <NA>  no
## 6 ASMA  302   ECE       1 M       105   235 <NA>     24 <NA>  no
## # ... with 1 more variable: Date <dttm>
```

```
# but note the differences in how read.xlsx default loads the data:
str(liz_df)
```

```
## 'data.frame':    107 obs. of  12 variables:
##  $ ID   : Factor w/ 7 levels "ASMA","ASMA ",..: 1 1 1 1 1 2 1 1 1 1 ...
##  $ Mark : Factor w/ 95 levels "","10","101",..: 3 39 40 41 51 52 56 57 62 63 ...
##  $ Grid : Factor w/ 35 levels "","BE","CB","DE",..: 31 31 31 32 10 20 21 27 2 3 ...
##  $ Trap : int  4 7 7 8 7 1 6 1 8 8 ...
##  $ Sex  : Factor w/ 5 levels "","F","J","M",..: 4 4 4 4 4 4 5 5 4 4 ...
```

```
##  $ SVL  : int  94 92 84 92 91 105 89 82 93 74 ...
##  $ Tail : int  221 213 199 228 120 235 186 124 200 177 ...
##  $ Regen: Factor w/ 21 levels "","1","121","13",..: 1 19 18 1 1 1 21 20 5 1 ...
##  $ Mass : num  27 25 18 23 20 24 22 19 27 15.5 ...
##  $ Notes: Factor w/ 36 levels "","2 REGENS ONE AT 38 AND ONE AT 6",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ Recap: Factor w/ 6 levels "?","N","no","No",..: 2 2 2 2 3 3 4 4 3 3 ...
##  $ Date : Factor w/ 9 levels "2/7/2013","26-May-12",..: 7 2 2 2 7 7 7 7 7 7 ...
```

```
str(liz_df_txt)
```

```
## 'data.frame':    107 obs. of  12 variables:
##  $ ID   : Factor w/ 6 levels "ASMA","ASSE",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ Mark : Factor w/ 95 levels "","10","101",..: 3 39 40 41 51 52 56 57 62 63 ...
##  $ Grid : Factor w/ 35 levels "","BE","CB","DE",..: 31 31 31 32 10 20 21 27 2 3 ...
##  $ Trap : int  4 7 7 8 7 1 6 1 8 8 ...
##  $ Sex  : Factor w/ 5 levels "","F","J","M",..: 4 4 4 4 4 4 5 5 4 4 ...
##  $ SVL  : int  94 92 84 92 91 105 89 82 93 74 ...
##  $ Tail : int  221 213 199 228 120 235 186 124 200 177 ...
##  $ Regen: Factor w/ 21 levels "","1","121","13",..: 1 19 18 1 1 1 21 20 5 1 ...
##  $ Mass : num  27 25 18 23 20 24 22 19 27 15.5 ...
##  $ Notes: Factor w/ 36 levels "","2 REGENS ONE AT 38 AND ONE AT 6",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ Recap: Factor w/ 6 levels "?","N","no","No",..: 2 2 2 2 3 3 4 4 3 3 ...
##  $ Date : Factor w/ 9 levels "2/7/2013","26-May-12",..: 7 2 2 2 7 7 7 7 7 7 ...
```

```
str(liz_df_xl)
```

```
## 'data.frame':    107 obs. of  12 variables:
##  $ ID   : chr  "ASMA" "ASMA" "ASMA" "ASMA" ...
##  $ Mark : chr  "101" "201" "202" "203" ...
##  $ Grid : chr  "EFB" "EFB" "EFB" "EFC" ...
##  $ Trap : num  4 7 7 8 7 1 6 1 8 8 ...
##  $ Sex  : chr  "M" "M" "M" "M" ...
##  $ SVL  : num  94 92 84 92 91 105 89 82 93 74 ...
##  $ Tail : num  221 213 199 228 120 235 186 124 200 177 ...
##  $ Regen: chr  NA "86" "83" NA ...
##  $ Mass : num  27 25 18 23 20 24 22 19 27 15.5 ...
##  $ Notes: chr  NA NA NA NA ...
##  $ Recap: chr  "N" "N" "N" "N" ...
##  $ Date : num  41055 41055 41055 41055 41055 ...
```

```
str(liz_df_xl2)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    107 obs. of  12 variables:
##  $ ID   : chr  "ASMA" "ASMA" "ASMA" "ASMA" ...
##  $ Mark : chr  "101" "201" "202" "203" ...
##  $ Grid : chr  "EFB" "EFB" "EFB" "EFC" ...
##  $ Trap : num  4 7 7 8 7 1 6 1 8 8 ...
##  $ Sex  : chr  "M" "M" "M" "M" ...
##  $ SVL  : num  94 92 84 92 91 105 89 82 93 74 ...
##  $ Tail : num  221 213 199 228 120 235 186 124 200 177 ...
##  $ Regen: chr  NA "86" "83" NA ...
##  $ Mass : num  27 25 18 23 20 24 22 19 27 15.5 ...
##  $ Notes: chr  NA NA NA NA ...
##  $ Recap: chr  "N" "N" "N" "N" ...
##  $ Date : POSIXct, format: "2012-05-26" "2012-05-26" ...
```

TIP: 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 - Use "_" or caps between words (i.e. first_name or FirstName) - Headers are typed over, and over, and over, so keep them short - 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 - Commas will mess up comma-delimited files - # indicates a comment in R 6. Missing data are okay, but keep an eye on it.

TIP: We can also import data directly from repositories and sources on the internet

```
# install.packages(c('dismo', 'rinat', 'raster'))

# pull data from gbif (https://www.gbif.org/) library(dismo)
dismo::gbif()

# pull data from iNaturalist (https://www.inaturalist.org/) library(rinat)
rinat::get_inat_obs()

# pull bioclim data (https://www.worldclim.org/bioclim) library(raster)
raster::getData()
```

3.2.2 Exporting Data

Working with the liz_data data set, we will do a little cleaning (you'll learn more about cleaning and manipulating data sets tomorrow), and output some summary data as a csv, table, and figures. When setting up data to work with analyses you typically end up rearranging, subsetting, and adding data columns to get a clean and tidy data set. (What does **tidy data** look like?). Here we are just going to run through a few examples to clean and summarize data in order to export a cleaned version of our data. You will learn more about data manipulation tomorrow.

```
## let's actually load our data again. This time we are going to feed read.csv
## some additional arguments to help get ahead of some of the issues with this
## data set

# I typically like to include 'strip.white = T' in any data that I'm loading in,
# this removes any leading or trailing white space, because R considers 'F', '
# F', and 'F ' three different factors.  Since this data set isn't very big, we
# can just assign colClasses that way each column starts with the correct class,
# although I will assign date and regen as charactors because of some data entry
# issues
liz_csv <- "data/LizardData.csv"

liz_df <- read.csv(liz_csv, strip.white = T, colClasses = c("factor", "factor", "character",
    "integer", "factor", "numeric", "numeric", "character", "numeric", "character",
    "factor", "character"))
# and check that it loaded ok:
head(liz_df)
```

```
##     ID Mark Grid Trap Sex SVL Tail Regen Mass Notes Recap      Date
## 1 ASMA  101  EFB    4   M  94  221         27           N 5/26/2012
## 2 ASMA  201  EFB    7   M  92  213    86   25           N 26-May-12
## 3 ASMA  202  EFB    7   M  84  199    83   18           N 26-May-12
## 4 ASMA  203  EFC    8   M  92  228         23           N 26-May-12
## 5 ASMA  301  EBA    7   M  91  120         20          no 5/26/2012
## 6 ASMA  302  ECE    1   M 105  235         24          no 5/26/2012
```

```
tail(liz_df)
```

```
##       ID Mark Grid Trap Sex SVL Tail Regen Mass
## 102 SCAR  829  EBC    1   F  47   65        3.3
## 103 SCAR  839  EAA    7   M  50   82        5.4
## 104 SCCO  818  ECB    2   F  53   44    14  5.4
## 105 UTST   10  EFC    7   M  13   70    23  4.5
## 106 UTST  126  EBE    7   M  49   83    13  5.0
## 107 UTST  144  EBE    2   F  45   77        3.5
##                             Notes Recap      Date
## 102                                    Y 5/10/2013
## 103                 SCAR ON BACK     Y 5/10/2013
## 104                       GRAVID     N 5/10/2013
## 105                2,3,5 MISSING    Y? 5/10/2013
## 106                     SHEDDING     Y 5/10/2013
## 107 KINK IN TAIL AT 29MM; GRAVID     Y 5/10/2013
```

```
# Let's check the summary again, cause we had spotted some issues previously
summary(liz_df)
```

```
##     ID          Mark        Grid                Trap         Sex
##  ASMA:18   101    : 3   Length:107         Min.   :1.000       : 1
##  ASSE: 1           : 2   Class :character   1st Qu.:3.000   F   :35
##  PHCO: 2   126    : 2   Mode  :character   Median :5.000   J   : 1
##  SCAR:28   201    : 2                      Mean   :5.038   M   :63
##  SCCO: 3   202    : 2                      3rd Qu.:7.000   Male: 4
##  UTST:55   301    : 2                      Max.   :9.000   NA's: 3
##            (Other):94                      NA's   :1
##       SVL              Tail           Regen               Mass
##  Min.   : 13.00   Min.   : 18.00   Length:107         Min.   : 1.40
##  1st Qu.: 44.00   1st Qu.: 60.25   Class :character   1st Qu.: 2.90
##  Median : 48.00   Median : 74.00   Mode  :character   Median : 3.90
##  Mean   : 52.96   Mean   : 85.95                      Mean   : 6.46
##  3rd Qu.: 57.25   3rd Qu.: 88.75                      3rd Qu.: 5.70
##  Max.   :105.00   Max.   :235.00                      Max.   :27.00
##  NA's   :9        NA's   :9                           NA's   :10
##     Notes            Recap       Date
##  Length:107         ? : 1   Length:107
##  Class :character   N :38   Class :character
##  Mode  :character   no: 7   Mode  :character
##                     No: 4
##                     Y :56
##                     Y?: 1
##
```

```
# Some known issues: Sex - we have some duplicate categories (M vs. Male) and an
# age class (J) 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) Date
# is a character here, but we actually want a date. But because people entered
# the data in multiple formats, it's going to take some work

## 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_df$Regen)  #so we have 82 blanks, 3 No's, 1 Yes, and the weird 38/6
```

```
##
##         1  121   13  130   14   19   20   21   23   28   32   34   35   36
##   82    1    1    2    1    1    1    1    1    1    1    1    2    1
##   38 38/6   83   86   No  Yes
##    2    1    1    1    3    1
```

```
# 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_df[liz_df$Regen == "38/6", ]
```

```
##      ID Mark Grid Trap Sex SVL Tail Regen Mass
## 47 UTST  626  EAC    3   M  48   47  38/6  3.8
##                             Notes Recap     Date
## 47 2 REGENS ONE AT 38 AND ONE AT 6     Y 2/7/2013
```

```
## Other ways to subset:
liz_df[liz_df$Regen %in% "38/6", ]
```

```
##      ID Mark Grid Trap Sex SVL Tail Regen Mass
## 47 UTST  626  EAC    3   M  48   47  38/6  3.8
##                             Notes Recap     Date
## 47 2 REGENS ONE AT 38 AND ONE AT 6     Y 2/7/2013
```

```
which(liz_df$Regen == "38/6")  #tells us the row:
```

```
## [1] 47
```

```
liz_df[47, ]  #you can use that actual row OR
```

```
##      ID Mark Grid Trap Sex SVL Tail Regen Mass
## 47 UTST  626  EAC    3   M  48   47  38/6  3.8
##                             Notes Recap     Date
## 47 2 REGENS ONE AT 38 AND ONE AT 6     Y 2/7/2013
```

```
liz_df[which(liz_df$Regen == "38/6"), ]
```

```
##      ID Mark Grid Trap Sex SVL Tail Regen Mass
## 47 UTST  626  EAC    3   M  48   47  38/6  3.8
##                             Notes Recap     Date
## 47 2 REGENS ONE AT 38 AND ONE AT 6     Y 2/7/2013
```

```
subset(liz_df, Regen == "38/6")
```

```
##      ID Mark Grid Trap Sex SVL Tail Regen Mass
## 47 UTST  626  EAC    3   M  48   47  38/6  3.8
##                             Notes Recap     Date
## 47 2 REGENS ONE AT 38 AND ONE AT 6     Y 2/7/2013
```

```
liz_df %>% subset(Regen == "38/6")
```

```
##      ID Mark Grid Trap Sex SVL Tail Regen Mass
## 47 UTST  626  EAC    3   M  48   47  38/6  3.8
##                             Notes Recap     Date
## 47 2 REGENS ONE AT 38 AND ONE AT 6     Y 2/7/2013
```

```
# 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_df$Regen <- recode(liz_df$Regen, `38/6` = "38")

# we also need to remove the no's and yes's as just blanks
liz_df$Regen <- recode(liz_df$Regen, No = "", Yes = "")
# and one last table to double check
table(liz_df$Regen)
```

```
##
##       1 121  13 130  14  19  20  21  23  28  32  34  35  36  38  83  86
##  86   1   1   2   1   1   1   1   1   1   1   1   1   2   1   3   1   1
```

```
# so lets make that numeric now.
liz_df$Regen <- as.numeric(liz_df$Regen)

## Let's clean up the sex column, because we do want to use that:
table(liz_df$Sex)
```

```
##
##         F    J    M Male
##    1   35    1   63    4
```

```
# lets change the Male to M - capitilization matters!
liz_df$Sex <- recode(liz_df$Sex, Male = "M", J = "")

# TIP: Recode 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
class(liz_df$Sex)  #this is still a factor, and...
```

```
## [1] "factor"
```

```
levels(liz_df$Sex)  #now we are down to the three levels we wanted
```

```
## [1] ""  "F" "M"
```

```
## 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_df, 1)
```

```
##     ID Mark Grid Trap Sex SVL Tail Regen Mass Notes Recap      Date
## 1 ASMA  101  EFB    4   M  94  221    NA   27           N 5/26/2012
```

```
liz_sub <- liz_df[, c(1:2, 5:9)]
head(liz_sub, 2)
```

```
##     ID Mark Sex SVL Tail Regen Mass
## 1 ASMA  101   M  94  221    NA   27
## 2 ASMA  201   M  92  213    86   25
```

```
# Alternative ways to subset:
liz_sub1 <- liz_df[, c("ID", "Mark", "Sex", "SVL", "Tail", "Regen", "Mass")]
liz_sub2 <- subset(liz_df, select = c("ID", "Mark", "Sex", "SVL", "Tail", "Regen",
    "Mass"))
liz_sub3 <- liz_df %>% select(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?
sum(is.na(liz_sub$Sex))  #sum counts FALSE as 0 and TRUE as 1, letting us get
```

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

```
# a count of 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
```

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

```
sum(is.na(liz_subna$Sex))
```

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

```
dim(liz_sub)
```

```
## [1] 107   7
```

```
dim(liz_subna)
```

```
## [1] 104   7
```

```
# 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)
```

```
## [1] ""  "F" "M"
```

Generally, you don't want to change your input data, but create a new, cleaned data set that you can use for analyses. So, now that we have a clean, subsetted version of our data, we can export a copy to use later.

```
## first I'm going to add a folder to the directory
dir.create("outputs")

write.csv(liz_sub, "outputs/LizardData_Clean.csv")
```

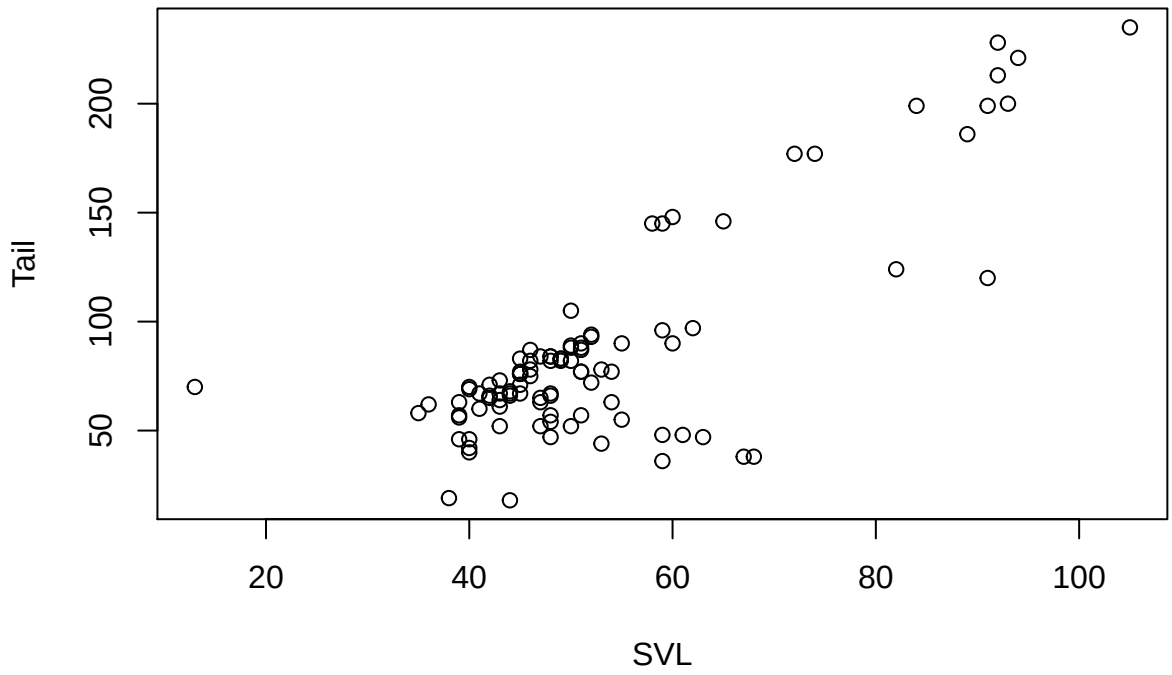
We can also export summary data as files or tables.

```
## 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())

# and again we can save that summary output:
write.csv(liz_summ, "outputs/LizardSummaryData.csv")
```

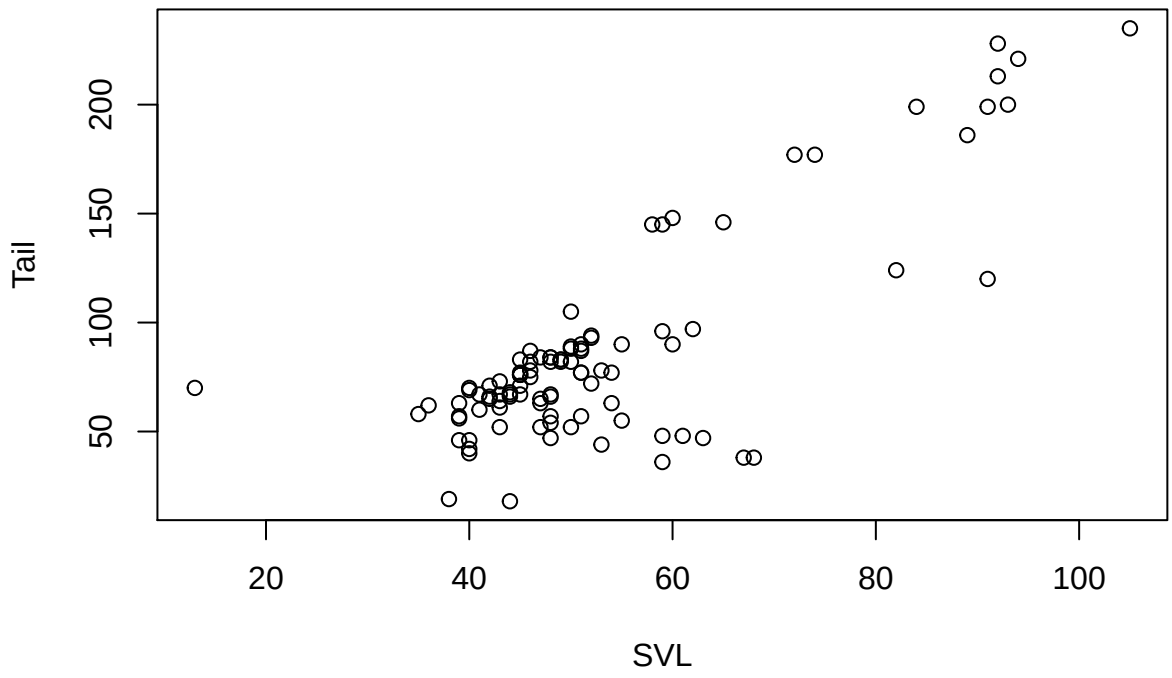
```r
# alternatively, we could print it as a table for word
liz_summ_tbl <- t(liz_summ)
colnames(liz_summ_tbl) <- liz_summ$ID
liz_summ_tbl <- liz_summ_tbl[-1, ]

write2word(kable(liz_summ_tbl, "pandoc"), file = "outputs/Liz_Summ.doc")
```

We can also do some exploratory graphing of the data and export the results. R has a number of plotting options, you'll learn more about them tomorrow.

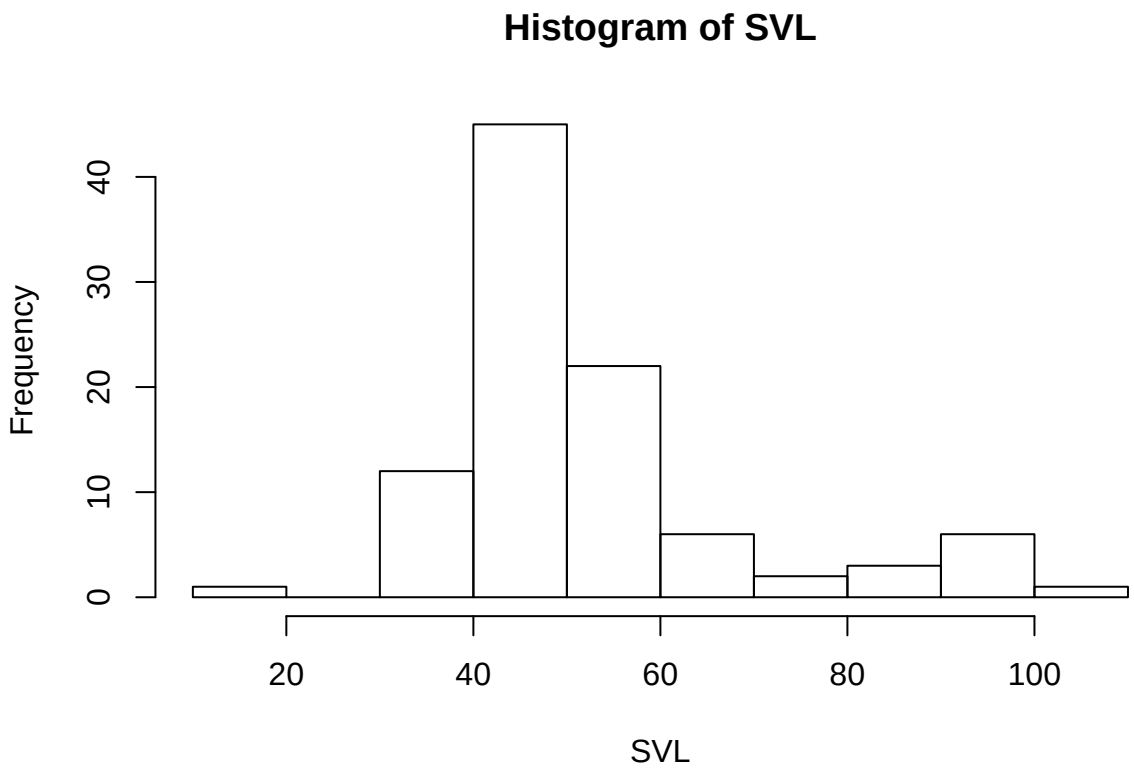
```r
## R has a number of plotting functions, you'll learn more about them tomorrow.

## NOTE: I'm going to attach liz_sub here 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
```



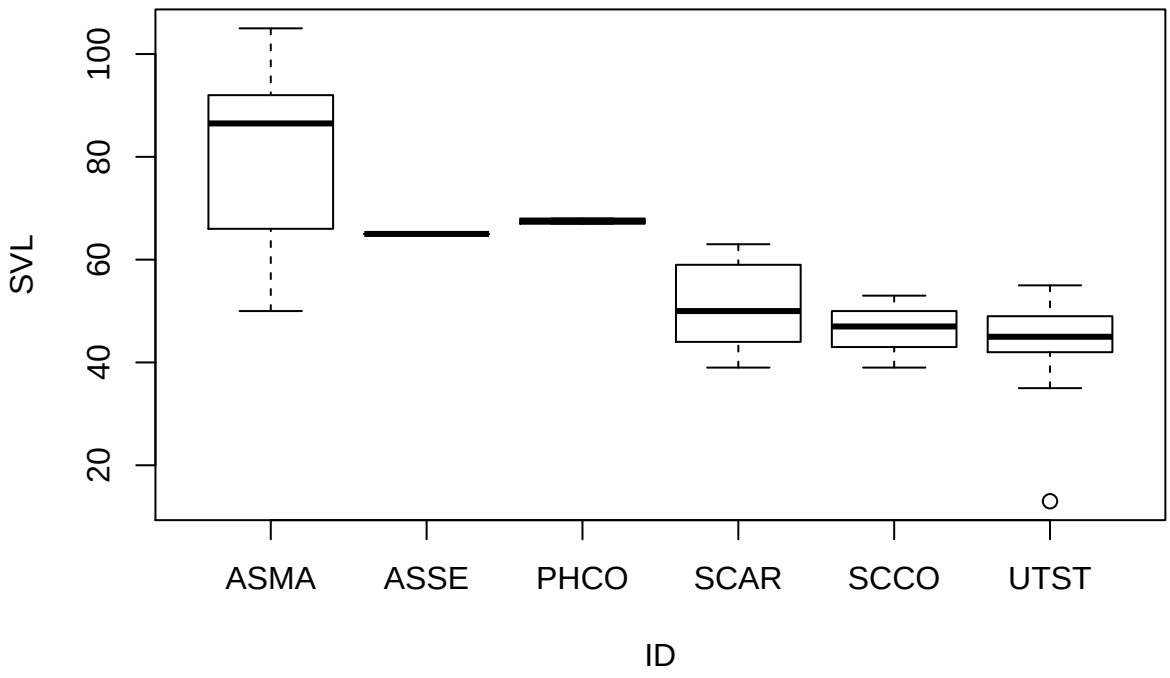

```r
plot(Tail ~ SVL)  #we can also use the '~', here the format is y ~ x though
```




```
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
```




```
## Before we go let's export a plot:
`?`(jpeg)
dir.create("figures")
jpeg(filename = "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 = "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 plotting inspiration: **R Graph Gallery**

4 Creating Documents in R and RStudio

4.1 RMarkdown

For this section, you can open the OSOS-2019.Rmd file and we'll look through it.

4.2 Latex

For this section, you can open the TAMU_LaTeX_Spring2018 folder, and find the TAMU_ex.RNW file and we'll look through it.

5 Additional Resources

Great resources for learning and programming in R

- **R for Data Science**
- **Software Carpentry - Programming with R**
- **Software Carpentry - Reproducible Research**
- **Quick-R Data Camp**